CHAPTER

2

STA Concepts

This chapter describes the basics of CMOS technology and the terminology involved in performing static timing analysis.

2.1 CMOS Logic Design

2.1.1 Basic MOS Structure

The physical implementation of MOS transistors (NMOS[1] and PMOS[2]) is depicted in Figure 2-1. The separation between the *source* and *drain* regions is the *length* of the MOS transistor. The smallest length used to build a MOS

1. N-channel Metal Oxide Semiconductor.
2. P-channel Metal Oxide Semiconductor.

J. Bhasker and R. Chadha, *Static Timing Analysis for Nanometer Designs: A Practical Approach*,
DOI: 10.1007/978-0-387-93820-2_2,

transistor is normally the smallest feature size for the CMOS technology process. For example, a 0.25µm technology allows MOS transistors with a channel length of 0.25µm or larger to be fabricated. By shrinking the channel geometry, the transistor size becomes smaller, and subsequently more transistors can be packed in a given area. As we shall see later in this chapter, this also allows the designs to operate at a greater speed.

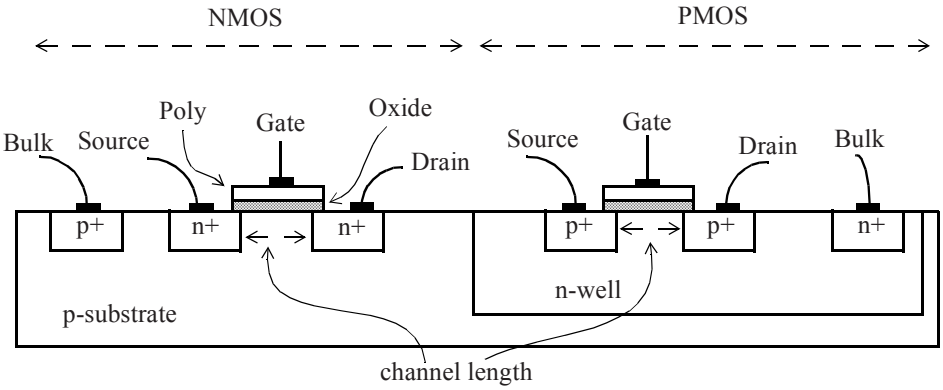


Figure 2-1 *Structure of NMOS and PMOS transistors.*

2.1.2 CMOS Logic Gate

A CMOS logic gate is built using NMOS and PMOS transistors. Figure 2-2 shows an example of a CMOS inverter. There are two stable states of the CMOS inverter depending upon the state of the input. When input *A* is low (at *Vss* or logic-0), the NMOS transistor is *off* and the PMOS transistor is *on*, causing the output *Z* to be pulled to *Vdd*, which is a logic-1. When input *A* is high (at *Vdd* or logic-1), the NMOS transistor is *on* and the PMOS transistor is *off*, causing the output *Z* to be pulled to *Vss*, which is a logic-0. In either of the two states described above, the CMOS inverter is stable and

does not draw any current[1] from the input *A* or from the power supply *Vdd*.

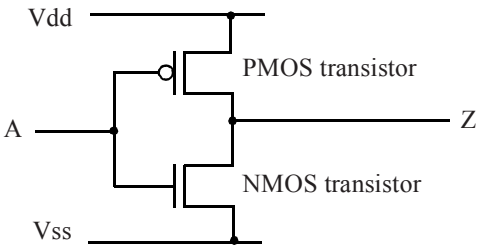


Figure 2-2 *A CMOS inverter.*

The characteristics of the CMOS inverter can be extended to any CMOS logic gate. In a CMOS logic gate, the output node is connected by a pull-up structure (made up of PMOS transistors) to *Vdd* and a pull-down structure (made up of NMOS transistors) to *Vss*. As an example, a two-input CMOS *nand* gate is shown in Figure 2-3. In this example, the pull-up structure is comprised of the two parallel PMOS transistors and the pull-down structure is made up of two series NMOS transistors.

For any CMOS logic gate, the pull-up and pull-down structures are complementary. For inputs at logic-0 or logic-1, this means that if the pull-up stage is turned *on*, the pull-down stage will be *off* and similarly if the pull-up stage is turned *off*, the pull-down stage will be turned *on*. The pull-down and pull-up structures are governed by the logic function implemented by the CMOS gate. For example, in a CMOS *nand* gate, the function controlling the pull-down structure is "*A and B*", that is, the pull-down is turned *on* when *A* and *B* are both at logic-1. Similarly, the function controlling the pull-up structure is "*not A or not B*", that is, the pull-up is turned *on* when either *A* or *B* is at logic-0. These characteristics ensure that the output node logic will be pulled to *Vdd* based upon the function controlling the pull-up structure. Since the pull-down structure is controlled by a comple-

1. Depending upon the specifics of the CMOS technology, there is a small amount of leakage current that is drawn even in steady state.

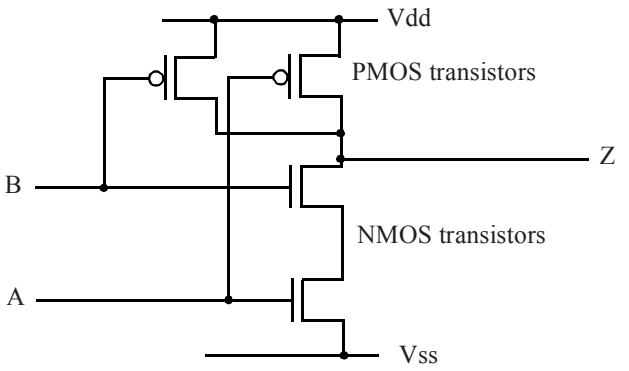


Figure 2-3 *CMOS two-input NAND gate.*

mentary function, the output node is at logic-0 when the pull-up structure function evaluates to 0.

For inputs at logic-0 or at logic-1, the CMOS logic gate does not draw any current from the inputs or from the power supply in steady state since the pull-up and pull-down structures can not both be *on*[1]. Another important aspect of CMOS logic is that the inputs pose only a capacitive load to the previous stage.

The CMOS logic gate is an inverting gate which means that a single switching input (rising or falling) can only cause the output to switch in the opposite direction, that is, the output can not switch in the same direction as the switching input. The CMOS logic gates can however be cascaded to put together a more complex logic function - inverting as well as non-inverting.

2.1.3 Standard Cells

Most of the complex functionality in a chip is normally designed using basic building blocks which implement simple logic functions such as *and*, *or*, *nand*, *nor*, a*nd-or-invert*, *or-and-invert* and *flip-flop*. These basic building

1. The pull-up and pull-down structures are both *on* only during switching.

blocks are pre-designed and referred to as **standard cells**. The functionality and timing of the standard cells is pre-characterized and available to the designer. The designer can then implement the required functionality using the standard cells as the building blocks.

The key characteristics of the CMOS logic gates described in previous subsection are applicable to all CMOS digital designs. All digital CMOS cells are designed such that there is no current drawn from power supply (except for leakage) when the inputs are in a stable logic state. Thus, most of the power dissipation is related to the activity in the design and is caused by the charging and discharging of the inputs of CMOS cells in the design.

What is a logic-1 or a logic-0? In a CMOS cell, two values *VIHmin* and *VILmax* define the limits. That is, any voltage value above *VIHmin* is considered as a **logic-1** and any voltage value below *VILmax* is considered as a **logic-0**. See Figure 2-4. Typical values for a CMOS 0.13µm inverter cell with 1.2V *Vdd* supply are 0.465V for *VILmax* and 0.625V for *VIHmin*. The *VIHmin* and *VILmax* values are derived from the DC transfer characteristics of the cell. The DC transfer characteristics are described in greater detail in Section 6.2.3.

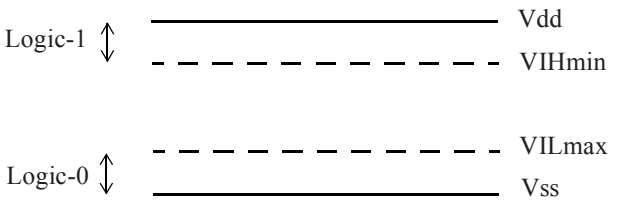


Figure 2-4 *CMOS logic levels.*

For more details on CMOS technology, refer to one of the relevant texts listed in the Bibliography.

2.2 Modeling of CMOS Cells

If a cell output pin drives multiple fanout cells, the total capacitance on the output pin of the cell is the sum of all the input capacitances of the cells that it is driving plus the sum of the capacitance of all the wire segments that comprise the net plus the output capacitance of the driving cell. Note that in a CMOS cell, the inputs to the cell present a capacitive load only.

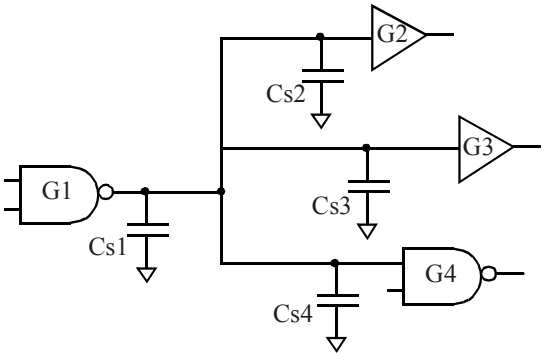


Figure 2-5 *Capacitance on a net.*

Figure 2-5 shows an example of a cell *G1* driving three other cells *G2*, *G3*, and *G4*. *Cs1*, *Cs2*, *Cs3* and *Cs4* are the capacitance values of wire segments that comprise the net. Thus:

```
Total cap (Output G1) = Cout(G1) + Cin(G2) + Cin(G3) +
                        Cin(G4) + Cs1 + Cs2 + Cs3 + Cs4
# Cout is the output pin capacitance of the cell.
# Cin is the input pin capacitance of the cell.
```

This is the capacitance that needs to be charged or discharged when cell *G1* switches and thus this total capacitance value impacts the timing of cell *G1*.

From a timing perspective, we need to model the CMOS cell to aid us in analyzing the timing through the cell. An input pin capacitance is specified

for each of the input pins. There can also be an output pin capacitance though most CMOS logic cells do not include the pin capacitance for the output pins.

When output is a logic-1, the pull-up structure for the output stage is *on*, and it provides a path from the output to *Vdd*. Similarly, when the output is a logic-0, the pull-down structure for the output stage provides a path from the output to *Vss*. When the CMOS cell switches state, the speed of the switching is governed by how fast the capacitance on the output net can be charged or discharged. The capacitance on the output net (Figure 2-5) is charged and discharged through the pull-up and pull-down structures respectively. Note that the channel in the pull-up and pull-down structures poses resistances for the output charging and discharging paths. The charging and discharging path resistances are a major factor in determining the speed of the CMOS cell. The inverse of the pull-up resistance is called the **output high drive** of the cell. The larger the output pull-up structure, the smaller the pull-up resistance and the larger the output high drive of the cell. The larger output structures also mean that the cell is larger in area. The smaller the output pull-up structures, the cell is smaller in area, and its output high drive is also smaller. The same concept for the pull-up structure can be applied for the pull-down structure which determines the resistance of the pull-down path and **output low drive**. In general, the cells are designed to have similar drive strengths (both large or both small) for pull-up and pull-down structures.

The output drive determines the maximum capacitive load that can be driven. The maximum capacitive load determines the maximum number of fanouts, that is, how many other cells it can drive. A higher output drive corresponds to a lower output pull-up and pull-down resistance which allows the cell to charge and discharge a higher load at the output pin.

Figure 2-6 shows an equivalent abstract model for a CMOS cell. The objective of this model is to abstract the timing behavior of the cell, and thus only the input and output stages are modeled. This model does not capture the intrinsic delay or the electrical behavior of the cell.

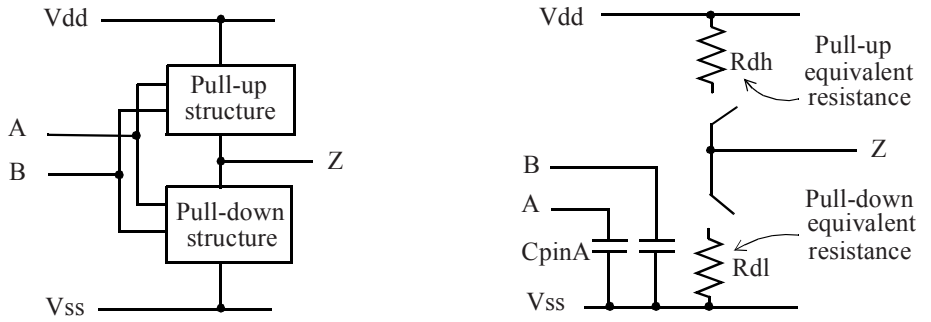


Figure 2-6 *CMOS cell and its electrically equivalent model.*

CpinA is the input pin capacitance of the cell on input *A*. *Rdh* and *Rdl* are the output drive resistances of the cell and determine the rise and fall times of the output pin *Z* based upon the load being driven by the cell. This drive also determines the maximum fanout limit of the cell.

Figure 2-7 shows the same net as in Figure 2-5 but with the equivalent models for the cells.

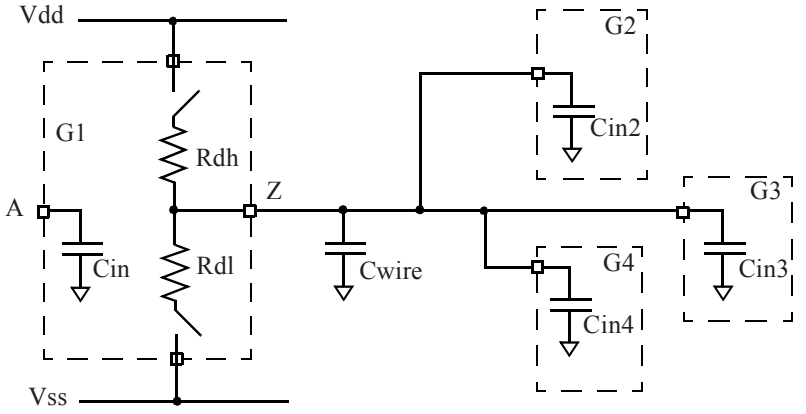


Figure 2-7 *Net with CMOS equivalent models.*

```
Cwire = Cs1 + Cs2 + Cs3 + Cs4
Output charging delay (for high or low) =
  Rout * (Cwire + Cin2 +Cin3 + Cin4)
```

In the above expression, *Rout* is one of *Rdh* or *Rdl* where *Rdh* is the output drive resistance for pull-up and *Rdl* is the output drive resistance for pull-down.

2.3 Switching Waveform

When a voltage is applied to the RC network as shown in Figure 2-8(a) by activating the *SW0* switch, the output goes to a logic-1. Assuming the output is at 0V when *SW0* is activated, the voltage transition at the output is described by the equation:

$$V = Vdd * [1 - e^{-t/(Rdh * Cload)}]$$

The voltage waveform for this rise is shown in Figure 2-8(b). The product *(Rdh * Cload)* is called the **RC time constant** - typically this is also related to the transition time of the output.

When the output goes from logic-1 to logic-0, caused by input changes disconnecting *SW0* and activating *SW1*, the output transition looks like the one shown in Figure 2-8(c). The output capacitance discharges through the *SW1* switch which is *on*. The voltage transition in this case is described by the equation:

$$V = Vdd * e^{-t/(Rdl * Cload)}$$

In a CMOS cell, the output charging and discharging waveforms do not appear like the RC charging and discharging waveforms of Figure 2-8 since the PMOS pull-up and the NMOS pull-down transistors are both *on* simultaneously for a brief amount of time. Figure 2-9 shows the *current* paths within a CMOS inverter cell for various stages of output switching from logic-1 to logic-0. Figure 2-9(a) shows the *current* flow when both the pull-

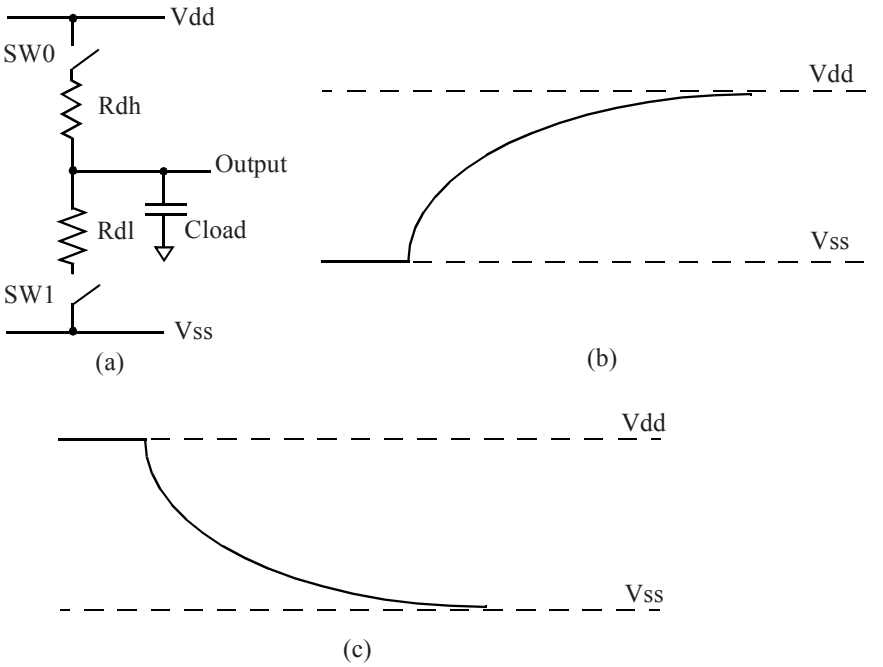
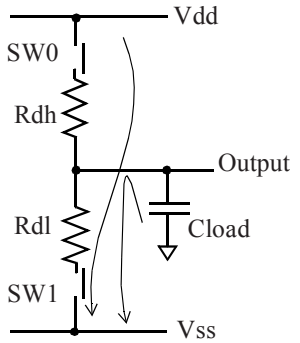


Figure 2-8 *RC charging and discharging waveforms.*

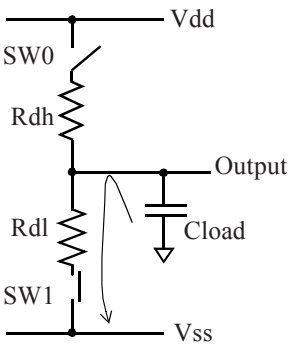
up and pull-down structures are *on*. Later, the pull-up structure turns *off* and the current flow is depicted in Figure 2-9(b). After the output reaches the final state, there is no current flow as the capacitance *Cload* is completely discharged.

Figure 2-10(a) shows a representative waveform at the output of a CMOS cell. Notice how the transition waveforms curve asymptotically towards the *Vss* rail and the *Vdd* rail, and the linear portion of the waveform is in the middle.

In this text, we shall depict some waveforms using a simplistic drawing as shown in Figure 2-10(b). It shows the waveform with some transition time, which is the time needed to transition from one logic state to the other. Fig-



(a) Cell is switching.
(Pull-up, pull-down both *on*)



(b) Cell is discharging to logic-0.
(Pull-up *off*, pull-down *on*)

Figure 2-9 *Current flow for a CMOS cell output stage.*

ure 2-10(c) shows the same waveforms using a transition time of 0, that is, as ideal waveforms. We shall be using both of these forms in this text interchangeably to explain the concepts, though in reality, each waveform has its real edge characteristics as shown in Figure 2-10(a).

2.4 Propagation Delay

Consider a CMOS inverter cell and its input and output waveforms. The **propagation delay** of the cell is defined with respect to some measurement points on the switching waveforms. Such points are defined using the following four variables:

```
# Threshold point of an input falling edge:
input_threshold_pct_fall : 50.0;
# Threshold point of an input rising edge:
input_threshold_pct_rise : 50.0;
# Threshold point of an output falling edge:
output_threshold_pct_fall : 50.0;
```

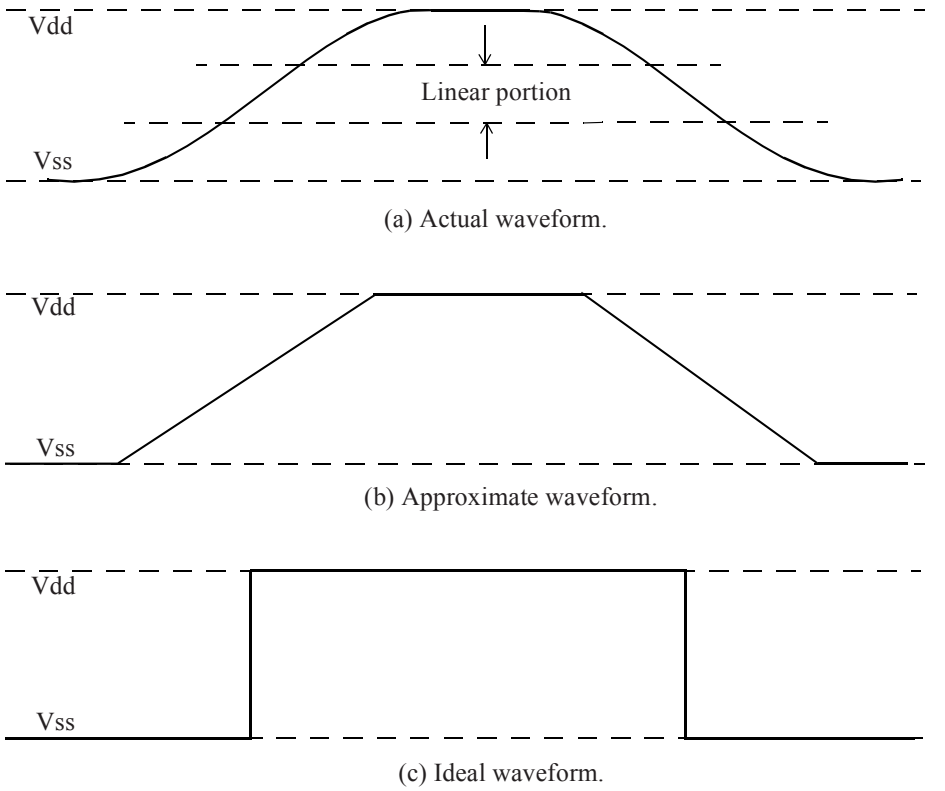


Figure 2-10 *CMOS output waveforms.*

```
# Threshold point of an output rising edge:
output_threshold_pct_rise : 50.0;
```

These variables are part of a command set used to describe a cell library (this command set is described in Liberty[1]). These threshold specifications are in terms of the percent of *Vdd*, or the power supply. Typically 50% threshold is used for delay measurement for most standard cell libraries.

1. See [LIB] in Bibliography.

Rising edge is the transition from logic-0 to logic-1. Falling edge is the transition from logic-1 to logic-0.

Consider the example inverter cell and the waveforms at its pins shown in Figure 2-11. The propagation delays are represented as:

i. Output fall delay (*Tf*)

ii. Output rise delay (*Tr*)

In general, these two values are different. Figure 2-11 shows how these two propagation delays are measured.

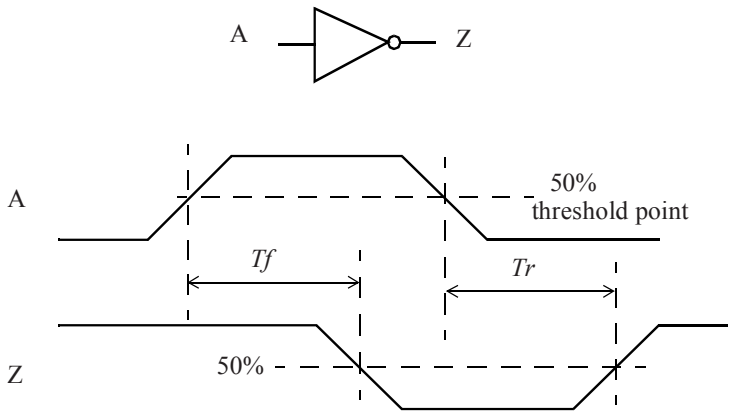


Figure 2-11 *Propagation delays.*

If we were looking at ideal waveforms, propagation delay would simply be the delay between the two edges. This is shown in Figure 2-12.

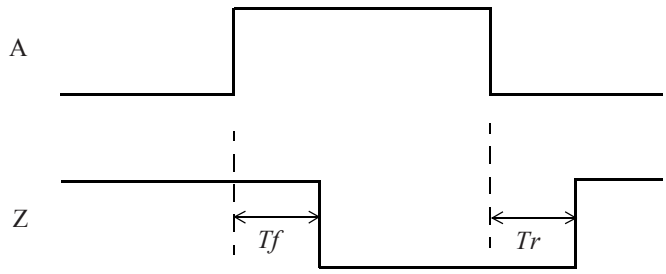


Figure 2-12 *Propagation delay using ideal waveforms.*

2.5 Slew of a Waveform

A slew rate is defined as a rate of change. In static timing analysis, the rising or falling waveforms are measured in terms of whether the transition is slow or fast. The slew is typically measured in terms of the **transition time**, that is, the time it takes for a signal to transition between two specific levels. Note that the transition time is actually inverse of the slew rate - the larger the transition time, the slower the slew, and vice versa. Figure 2-10 illustrates a typical waveform at the output of a CMOS cell. The waveforms at the ends are asymptotic and it is hard to determine the exact start and end points of the transition. Consequently, the transition time is defined with respect to specific threshold levels. For example, the slew threshold settings can be:

```
# Falling edge thresholds:
slew_lower_threshold_pct_fall : 30.0;
slew_upper_threshold_pct_fall : 70.0;
# Rising edge thresholds:
slew_lower_threshold_pct_rise : 30.0;
slew_upper_threshold_pct_rise : 70.0;
```

These values are specified as a percent of *Vdd*. The threshold settings specify that falling slew is the difference between the times that falling edge

reaches 70% and 30% of *Vdd*. Similarly, the settings for rise specify that the rise slew is the difference in times that the rising edge reaches 30% and 70% of *Vdd*. Figure 2-13 shows this pictorially.

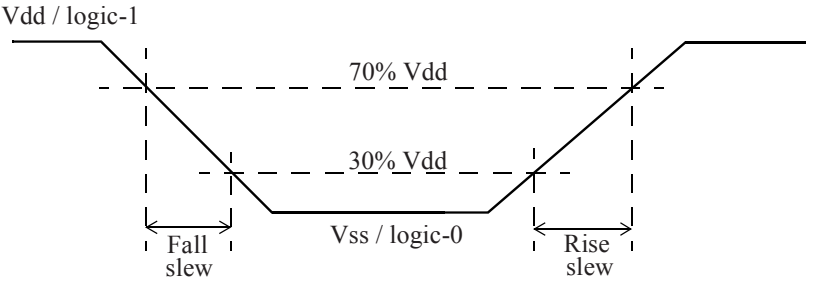


Figure 2-13 *Rise and fall transition times.*

Figure 2-14 shows another example where the slew on a falling edge is measured 20-80 (80% to 20%) and that on the rising edge is measured 10-90 (10% to 90%). Here are the threshold settings for this case.

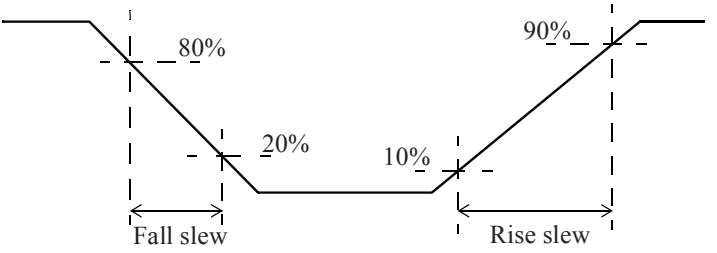


Figure 2-14 *Another example of slew measurement.*

```
# Falling edge thresholds:
slew_lower_threshold_pct_fall : 20.0;
slew_upper_threshold_pct_fall : 80.0;
```

```
# Rising edge thresholds:
slew_lower_threshold_pct_rise : 10.0;
slew_upper_threshold_pct_rise : 90.0;
```

2.6 Skew between Signals

Skew is the difference in timing between two or more signals, maybe data, clock or both. For example, if a clock tree has 500 end points and has a skew of 50ps, it means that the difference in latency between the longest path and the shortest clock path is 50ps. Figure 2-15 shows an example of a clock tree. The beginning point of a clock tree typically is a node where a clock is defined. The end points of a clock tree are typically clock pins of synchronous elements, such as flip-flops. **Clock latency** is the total time it takes from the clock source to an end point. **Clock skew** is the difference in arrival times at the end points of the clock tree.

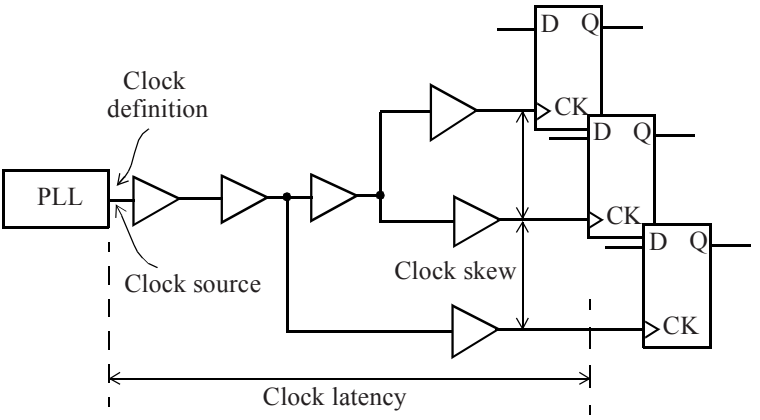


Figure 2-15 *Clock tree, clock latency and clock skew.*

An **ideal clock tree** is one where the clock source is assumed to have an infinite drive, that is, the clock can drive infinite sources with no delay. In addition, any cells present in the clock tree are assumed to have zero delay. In

the early stages of logical design, STA is often performed with ideal clock trees so that the focus of the analysis is on the data paths. In an ideal clock tree, clock skew is 0ps by default. Latency of a clock tree can be explicitly specified using the **set_clock_latency** command. The following example models the latency of a clock tree:

```
set_clock_latency 2.2 [get_clocks BZCLK]
  # Both rise and fall latency is 2.2ns.
  # Use options -rise and -fall if different.
```

Clock skew for a clock tree can also be implied by explicitly specifying its value using the **set_clock_uncertainty** command:

```
set_clock_uncertainty 0.250 -setup [get_clocks BZCLK]
set_clock_uncertainty 0.100 -hold [get_clocks BZCLK]
```

The *set_clock_uncertainty* specifies a window within which a clock edge can occur. The uncertainty in the timing of the clock edge is to account for several factors such as clock period jitter and additional margins used for timing verification. Every real clock source has a finite amount of jitter - a window within which a clock edge can occur. The clock period jitter is determined by the type of clock generator utilized. In reality, there are no ideal clocks, that is, all clocks have a finite amount of jitter and the clock period jitter should be included while specifying the clock uncertainty.

Before the clock tree is implemented, the clock uncertainty must also include the expected clock skew of the implementation.

One can specify different clock uncertainties for setup checks and for hold checks. The hold checks do not require the clock jitter to be included in the uncertainty and thus a smaller value of clock uncertainty is generally specified for hold.

Figure 2-16 shows an example of a clock with a setup uncertainty of 250ps. Figure 2-16(b) shows how the uncertainty takes away from the time avail-

able for the logic to propagate to the next flip-flop stage. This is equivalent to validating the design to run at a higher frequency.

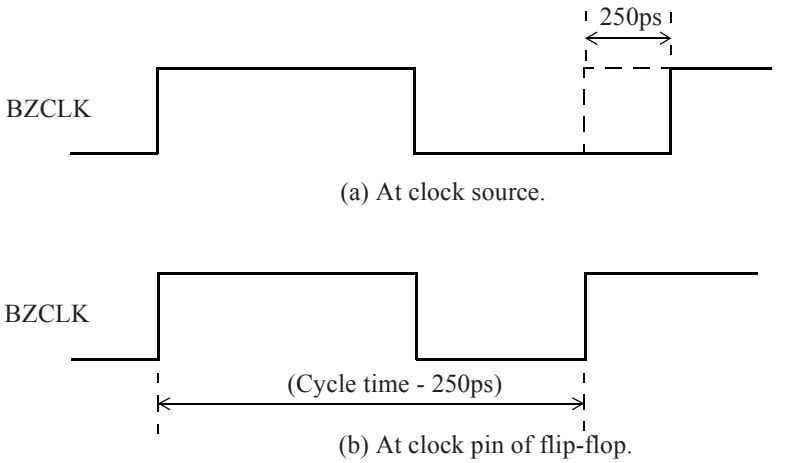


Figure 2-16 *Clock setup uncertainty.*

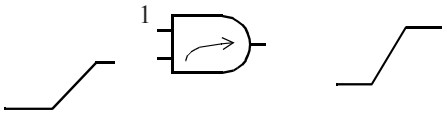
As specified above, the *set_clock_uncertainty* can also be used to model any additional margin. For example, a designer may use a 50ps timing margin as additional pessimism during design. This component can be added and included in the *set_clock_uncertainty* command. In general, before the clock tree is implemented, the *set_clock_uncertainty* command is used to specify a value that includes clock jitter plus estimated clock skew plus additional pessimism.

```
set_clock_latency 2.0 [get_clocks USBCLK]
set_clock_uncertainty 0.2 [get_clocks USBCLK]
  # The 200ps may be composed of 50ps clock jitter,
  # 100ps clock skew and 50ps additional pessimism.
```

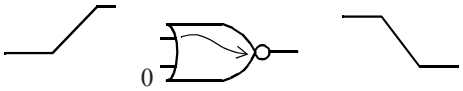
We shall see later how *set_clock_uncertainty* influences setup and hold checks. It is best to think of clock uncertainty as an offset to the final slack calculation.

2.7 Timing Arcs and Unateness

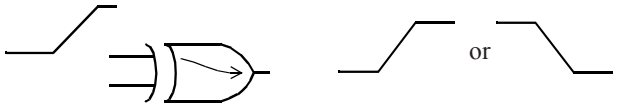
Every cell has multiple timing arcs. For example, a combinational logic cell, such as *and*, *or*, *nand*, *nor*, *adder* cell, has timing arcs from each input to each output of the cell. Sequential cells such as flip-flops have timing arcs from the clock to the outputs and timing constraints for the data pins with respect to the clock. Each timing arc has a timing sense, that is, how the output changes for different types of transitions on input. The timing arc is **positive unate** if a rising transition on an input causes the output to rise (or not to change) and a falling transition on an input causes the output to fall (or not to change). For example, the timing arcs for *and* and *or* type cells are positive unate. See Figure 2-17(a).



(a) Positive unate arc.



(b) Negative unate arc.



(c) Non-unate arc.

Figure 2-17 *Timing sense of arcs.*

A **negative unate** timing arc is one where a rising transition on an input causes the output to have a falling transition (or not to change) and a fall-

ing transition on an input causes the output to have a rising transition (or not to change). For example, the timing arcs for *nand* and *nor* type cells are negative unate. See Figure 2-17(b).

In a **non-unate** timing arc, the output transition cannot be determined solely from the direction of change of an input but also depends upon the state of the other inputs. For example, the timing arcs in an *xor* cell (exclusive-or) are non-unate.[1] See Figure 2-17(c).

Unateness is important for timing as it specifies how the edges (transitions) can propagate through a cell and how they appear at the output of the cell.

One can take advantage of the non-unateness property of a timing arc, such as when an *xor* cell is used, to invert the polarity of a clock. See the example in Figure 2-18. If input *POLCTRL* is a logic-0, the clock *DDRCLK* on output of the cell *UXOR0* has the same polarity as the input clock *MEMCLK*. If *POLCTRL* is a logic-1, the clock on the output of the cell *UXOR0* has the opposite polarity as the input clock *MEMCLK*.

2.8 Min and Max Timing Paths

The total delay for the logic to propagate through a logic path is referred to as the **path delay**. This corresponds to the sum of the delays through the various logic cells and nets along the path. In general, there are multiple paths through which the logic can propagate to the required destination point. The actual path taken depends upon the state of the other inputs along the logic path. An example is illustrated in Figure 2-19. Since there are multiple paths to the destination, the maximum and minimum timing to the destination points can be obtained. The paths corresponding to the maximum timing and minimum timing are referred to as the max path and min path respectively. A **max path** between two end points is the path with the largest delay (also referred to as the **longest path**). Similarly, a **min**

1. It is possible to specify state-dependent timing arcs for an *xor* cell which are positive unate and negative unate. This is described in Chapter 3.

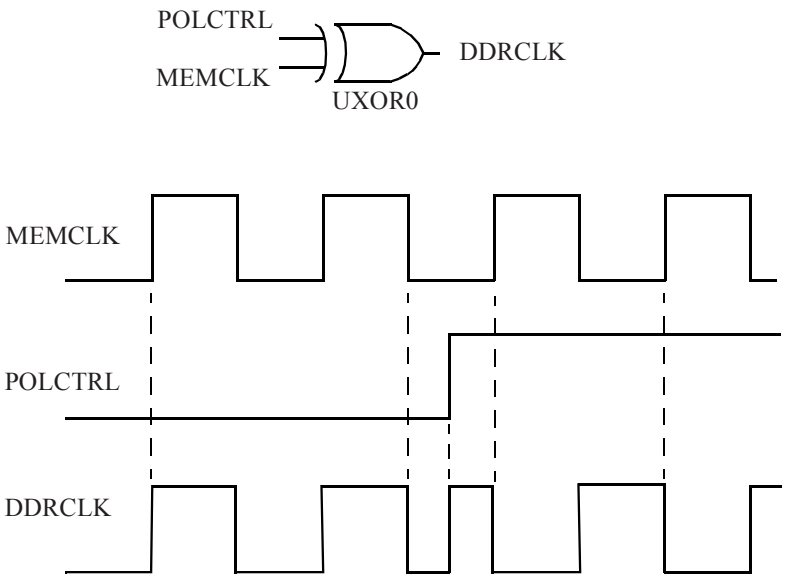


Figure 2-18 *Controlling clock polarity using non-unate cell.*

path is the path with the smallest delay (also referred to as the **shortest path**). Note that the longest and shortest refer to the cumulative delay of the path, not to the number of cells in the path.

Figure 2-19 shows an example of a data path between flip-flops. A max path between flip-flops *UFF1* and *UFF3* is assumed to be the one that goes through *UNAND0*, *UBUF2*, *UOR2* and *UNAND6* cells. A min path between the flip-flops *UFF1* and *UFF3* is assumed to be the one that goes through the *UOR4* and *UNAND6* cells. Note that in this example, the max and min are with reference to the destination point which is the *D* pin of the flip-flop *UFF3*.

A max path is often called a **late path**, while a min path is often called an **early path**.

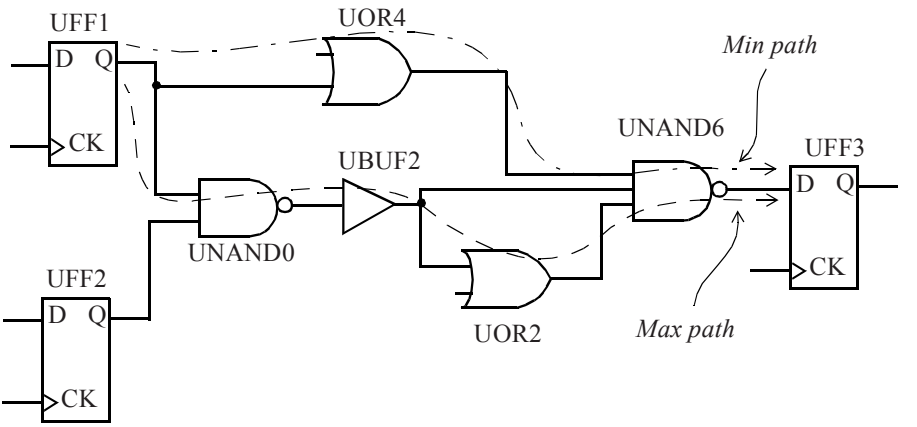


Figure 2-19 *Max and min timing paths.*

When a flip-flop to flip-flop path such as from *UFF1* to *UFF3* is considered, one of the flip-flops launches the data and the other flip-flop captures the data. In this case, since *UFF1* launches the data, *UFF1* is referred to as the **launch** flip-flop. And since *UFF3* captures the data, *UFF3* is referred to as the **capture** flip-flop. Notice that the launch and capture terminology are always with reference to a flip-flop to flip-flop path. For example, *UFF3* would become a launch flip-flop for the path to whatever flip-flop captures the data produced by *UFF3*.

2.9 Clock Domains

In synchronous logic design, a periodic clock signal latches the new data computed into the flip-flops. The new data inputs are based upon the flip-flop values from a previous clock cycle. The latched data thus gets used for computing the data for the next clock cycle.

A clock typically feeds a number of flip-flops. The set of flip-flops being fed by one clock is called its **clock domain**. In a typical design, there may be

more than one clock domain. For example, 200 flip-flops may be clocked by *USBCLK* and 1000 flip-flops may be fed by clock *MEMCLK*. Figure 2-20 depicts the flip-flops along with the clocks. In this example, we say that there are two clock domains.

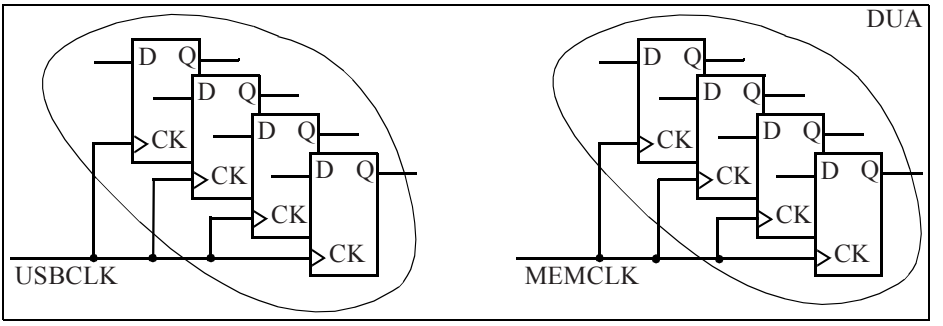


Figure 2-20 *Two clock domains.*

A question of interest is whether the clock domains are related or independent of each other. The answer depends on whether there are any data paths that start from one clock domain and end in the other clock domain. If there are no such paths, we can safely say that the two clock domains are independent of each other. This means that there is no timing path that starts from one clock domain and ends in the other clock domain.

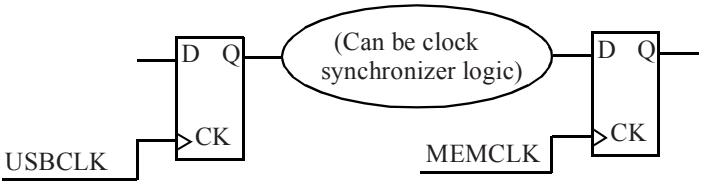


Figure 2-21 *Clock domain crossing.*

If indeed there are data paths that cross between clock domains (see Figure 2-21), a decision has to be made as to whether the paths are real or not. An example of a real path is a flip-flop with a 2x speed clock driving into a flip-flop with a 1x speed clock. An example of a false path is where the designer has explicitly placed clock synchronizer logic between the two clock domains. In this case, even though there appears to be a timing path from one clock domain to the next, it is not a real timing path since the data is not constrained to propagate through the synchronizer logic in one clock cycle. Such a path is referred to as a false path - not real - because the clock synchronizer ensures that the data passes correctly from one domain to the next. False paths between clock domains can be specified using the *set_false_path* specification, such as:

```
set_false_path -from [get_clocks USBCLK] \
  -to [get_clocks MEMCLK]
# This specification is explained in more detail in Chapter 8.
```

Even though it is not depicted in Figure 2-21, a clock domain crossing can occur both ways, from *USBCLK* clock domain to *MEMCLK* clock domain, and from *MEMCLK* clock domain to *USBCLK* clock domain. Both scenarios need to be understood and handled properly in STA.

What is the reason to discuss paths between clock domains? Typically a design has a large number of clocks and there can be a myriad number of paths between the clock domains. Identifying which clock domain crossings are real and which clock crossings are not real is an important part of the timing verification effort. This enables the designer to focus on validating only the real timing paths.

Figure 2-22 shows another example of clock domains. A multiplexer selects a clock source - it is either one or the other depending on the mode of operation of the design. There is only one clock domain, but two clocks, and these two clocks are said to be mutually-exclusive, as only one clock is active at one time. Thus, in this example, it is important to note that there can never be a path between the two clock domains for *USBCLK* and *USBCLKx2* (assuming that the multiplexer control is static and that such paths do not exist elsewhere in the design).

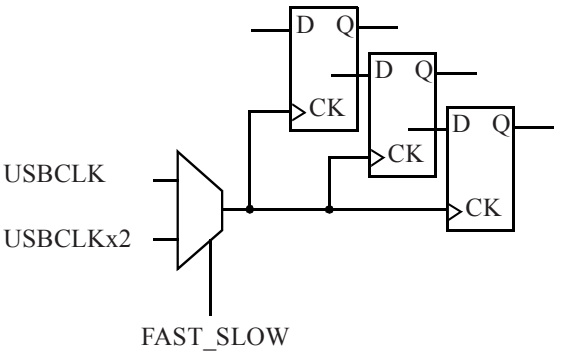


Figure 2-22 *Mutually-exclusive clocks.*

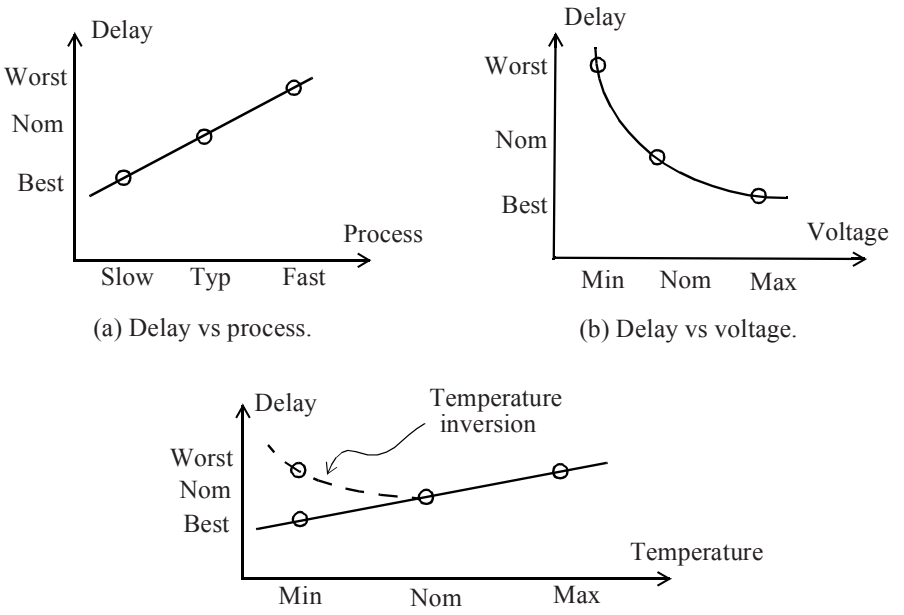
2.10 Operating Conditions

Static timing analysis is typically performed at a specific operating condition[1]. An operating condition is defined as a combination of *Process*, *Voltage* and *Temperature* (PVT). Cell delays and interconnect delays are computed based upon the specified operating condition.

There are three kinds of manufacturing process models that are provided by the semiconductor foundry for digital designs: *slow* process models, *typical* process models, and *fast* process models. The *slow* and *fast* process models represent the extreme corners of the manufacturing process of a foundry. For robust design, the design is validated at the extreme corners of the manufacturing process as well as environment extremes for temperature and power supply. Figure 2-23(a) shows how a cell delay changes with the process corners. Figure 2-23(b) shows how cell delays vary with power supply voltage, and Figure 2-23(c) shows how cell delays can vary

1. STA may be performed on a design with cells that have different voltages. We shall see later how these are handled. STA can also be performed statistically, which is described in Chapter 10.

with temperature. Thus it is important to decide the operating conditions that should be used for various static timing analyses.



(a) Delay vs process.

(b) Delay vs voltage.

(c) Delay vs temperature.

Figure 2-23 *Delay variations with PVT.*

The choice of what operating condition to use for STA is also governed by the operating conditions under which cell libraries are available. Three standard operating conditions are:

i. *WCS (Worst-Case Slow)*: Process is *slow*, temperature is highest (say 125C) and voltage is lowest (say nominal 1.2V minus 10%). For nanometer technologies that use low power supplies, there can be another worst-case slow corner that corresponds to the *slow* process, lowest power supply, and lowest temperature. The delays at low temperatures are not always smaller than the de-

lays at higher temperatures. This is because the device threshold voltage (Vt) margin with respect to the power supply is reduced for nanometer technologies. In such cases, at low power supply, the delay of a lightly loaded cell is higher at low temperatures than at high temperatures. This is especially true of high Vt (higher threshold, larger delay) or even standard Vt (regular threshold, lower delay) cells. This anomalous behavior of delays increasing at lower temperatures is called *temperature inversion*. See Figure 2-23(c).

ii. *TYP (Typical)*: Process is *typical*, temperature is nominal (say 25C) and voltage is nominal (say 1.2V).

iii. *BCF (Best-Case Fast)*: Process is *fast*, temperature is lowest (say -40C) and voltage is highest (say nominal 1.2V plus 10%).

The environment conditions for power analysis are generally different than the ones used for static timing analysis. For power analysis, the operating conditions may be:

i. *ML (Maximal Leakage)*: Process is *fast*, temperature is highest (say 125C) and the voltage is also the highest (say 1.2V plus 10%). This corner corresponds to the maximum leakage power. For most designs, this corner also corresponds to the largest active power.

ii. *TL (Typical Leakage)*: Process is *typical*, temperature is highest (say 125C) and the voltage is nominal (say 1.2V). This refers to the condition where the leakage is representative for most designs since the chip temperature will be higher due to power dissipated in normal operation.

The static timing analysis is based on the libraries that are loaded and linked in for the STA. An operating condition for the design can be explicitly specified using the **set_operating_conditions** command.

```
set_operating_conditions "WCCOM" -library mychip
# Use the operating condition called WCCOM defined in the
# cell library mychip.
```

The cell libraries are available at various operating conditions and the operating condition chosen for analysis depends on what has been loaded for the STA.

❑

CHAPTER

3

Standard Cell Library

This chapter describes timing information present in library cell descriptions. A cell could be a standard cell, an IO buffer, or a complex IP such as a USB core.

In addition to timing information, the library cell description contains several attributes such as cell area and functionality, which are unrelated to timing but are relevant during the RTL synthesis process. In this chapter, we focus only on the attributes relevant to the timing and power calculations.

A library cell can be described using various standard formats. While the content of various formats is essentially similar, we have described the library cell examples using the Liberty syntax.

J. Bhasker and R. Chadha, *Static Timing Analysis for Nanometer Designs: A Practical Approach*,
DOI: 10.1007/978-0-387-93820-2_3, © Springer Science + Business Media, LLC 2009

The initial sections in this chapter describe the linear and the non-linear timing models followed by advanced timing models for nanometer technologies which are described in Section 3.7.

3.1 Pin Capacitance

Every input and output of a cell can specify capacitance at the pin. In most cases, the capacitance is specified only for the cell inputs and not for the outputs, that is, the output pin capacitance in most cell libraries is 0.

```
pin (INP1) {
  capacitance: 0.5;
  rise_capacitance: 0.5;
  rise_capacitance_range: (0.48, 0.52);
  fall_capacitance: 0.45;
  fall_capacitance_range: (0.435, 0.46);
  . . .
}
```

The above example shows the general specification for the pin capacitance values for the input *INP1*. In its most basic form, the pin capacitance is specified as a single value (0.5 units in above example). (The capacitance unit is normally picofarad and is specified in the beginning of the library file). The cell description can also specify separate values for *rise_capacitance* (0.5 units) and *fall_capacitance* (0.45 units) which refer to the values used for rising and falling transitions at the pin *INP1*. The *rise_capacitance* and *fall_capacitance* values can also be specified as a range with the lower and upper bound values being specified in the description.

3.2 Timing Modeling

The cell timing models are intended to provide accurate timing for various instances of the cell in the design environment. The timing models are nor-

mally obtained from detailed circuit simulations of the cell to model the actual scenario of the cell operation. The timing models are specified for each timing arc of the cell.

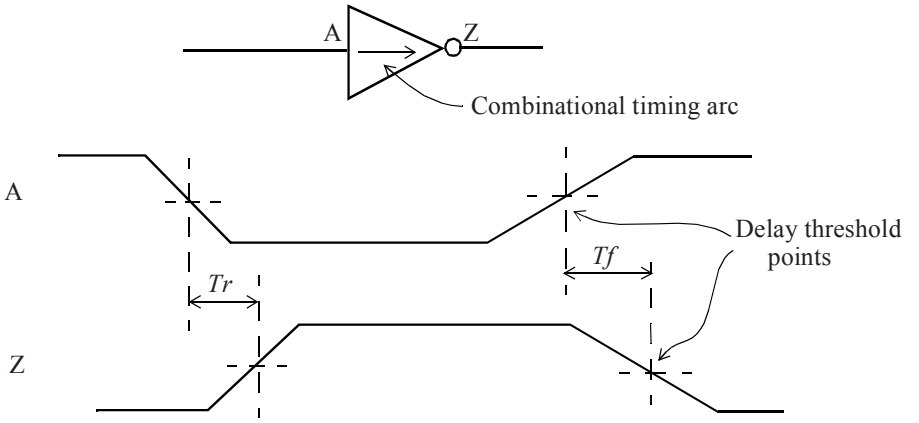


Figure 3-1 *Timing arc delays for an inverter cell.*

Let us first consider timing arcs for a simple inverter logic cell shown in Figure 3-1. Since it is an inverter, a rising (falling) transition at the input causes a falling (rising) transition at the output. The two kinds of delay characterized for the cell are:

- *Tr* : Output rise delay
- *Tf* : Output fall delay

Notice that the delays are measured based upon the threshold points defined in a cell library (see Section 2.4), which is typically 50% *Vdd*. Thus, delays are measured from input crossing its threshold point to the output crossing its threshold point.

The delay for the timing arc through the inverter cell is dependent on two factors:

i. the output load, that is, the capacitance load at the output pin of the inverter, and

ii. the transition time of the signal at the input.

The delay values have a direct correlation with the load capacitance - the larger the load capacitance, the larger the delay. In most cases, the delay increases with increasing input transition time. There are a few scenarios where the input threshold (used for measuring delay) is significantly different from the internal switching point of the cell. In such cases, the delay through the cell may show non-monotonic behavior with respect to the input transition time - a larger input transition time may produce a smaller delay especially if the output is lightly loaded.

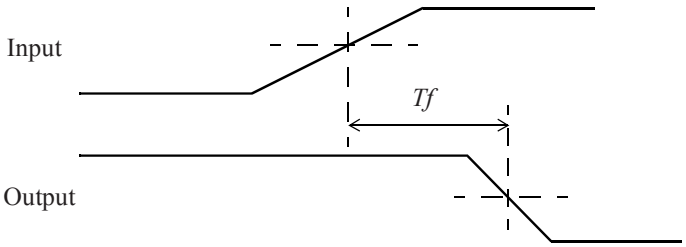
The slew at the output of a cell depends mainly upon the output capacitance - output transition time increases with output load. Thus, a large slew at the input (large transition time) can improve at the output depending upon the cell type and its output load. Figure 3-2 shows cases where the transition time at the output of a cell can improve or deteriorate depending on the load at the output of the cell.

3.2.1 Linear Timing Model

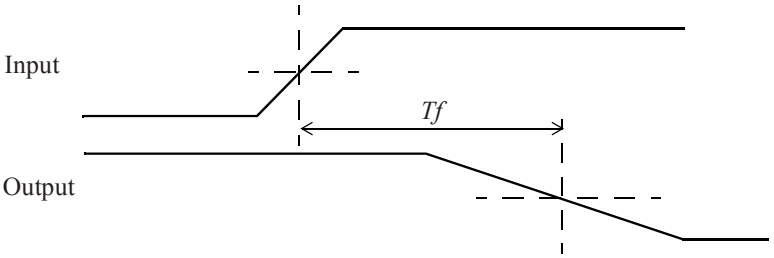
A simple timing model is a *linear delay model*, where the delay and the output transition time of the cell are represented as linear functions of the two parameters: input transition time and the output load capacitance. The general form of the linear model for the delay, *D*, through the cell is illustrated below.

```
D = D0 + D1 * S + D2 * C
```

where *D0*, *D1*, *D2* are constants, *S* is the input transition time, and *C* is the output load capacitance. The linear delay models are not accurate over the range of input transition time and output capacitance for submicron tech-



(a) Slew improves.



(b) Slew degrades (due to large output load).

Figure 3-2 *Slew changes going through a cell.*

nologies, and thus most cell libraries presently use the more complex models such as the non-linear delay model.

3.2.2 Non-Linear Delay Model

Most of the cell libraries include table models to specify the delays and timing checks for various timing arcs of the cell. Some newer timing libraries for nanometer technologies also provide current source based advanced timing models (such as CCS, ECSM, etc.) which are described later in this chapter. The table models are referred to as **NLDM** (**N**on-**L**inear **D**elay **M**odel) and are used for delay, output slew, or other timing checks. The table models capture the delay through the cell for various combinations of

input transition time at the cell input pin and total output capacitance at the cell output.

An NLDM model for delay is presented in a two-dimensional form, with the two independent variables being the input transition time and the output load capacitance, and the entries in the table denoting the delay. Here is an example of such a table for a typical inverter cell:

```
pin (OUT) {
  max_transition : 1.0;
  timing() {
    related_pin : "INP1";
    timing_sense : negative_unate;
    cell_rise(delay_template_3x3) {
      index_1 ("0.1, 0.3, 0.7"); /* Input transition */
      index_2 ("0.16, 0.35, 1.43"); /* Output capacitance */
      values (  /*  0.16      0.35     1.43 */ \
        /* 0.1 */   "0.0513,  0.1537,  0.5280", \
        /* 0.3 */   "0.1018,  0.2327,  0.6476", \
        /* 0.7 */   "0.1334,  0.2973,  0.7252");
      }
    cell_fall(delay_template_3x3) {
      index_1 ("0.1, 0.3, 0.7"); /* Input transition */
      index_2 ("0.16, 0.35, 1.43"); /* Output capacitance */
      values (  /*  0.16      0.35      1.43 */ \
        /* 0.1 */   "0.0617,  0.1537,   0.5280", \
        /* 0.3 */   "0.0918,  0.2027,   0.5676", \
        /* 0.7 */   "0.1034,  0.2273,   0.6452");
    }
```

In the above example, the delays of the output pin *OUT* are described. This portion of the cell description contains the rising and falling delay models for the timing arc from pin *INP1* to pin *OUT*, as well as the *max_transition* allowed time at pin *OUT*. There are separate models for the rise and fall delays (for the output pin) and these are labeled as *cell_rise* and *cell_fall* respectively. The type of indices and the order of table lookup indices are described in the lookup table template *delay_template_3x3*.

```
lu_table_template(delay_template_3x3) {
  variable_1 : input_net_transition;
  variable_2 : total_output_net_capacitance;
  index_1 ("1000, 1001, 1002");
  index_2 ("1000, 1001, 1002");
}/* The input transition and the output capacitance can be
    in either order, that is, variable_1 can be the output
    capacitance. However, these designations are usually
    consistent across all templates in a library. */
```

This lookup table template specifies that the first variable in the table is the input transition time and the second variable is the output capacitance. The table values are specified like a nested loop with the first index (*index_1*) being the outer (or least varying) variable and the second index (*index_2*) being the inner (or most varying) variable and so on. There are three entries for each variable and thus it corresponds to a 3-by-3 table. In most cases, the entries for the table are also formatted like a table and the first index (*index_1*) can then be treated as a row index and the second index (*index_2*) becomes equivalent to the column index. The index values (for example 1000) are dummy placeholders which are overridden by the actual index values in the *cell_fall* and *cell_rise* delay tables. An alternate way of specifying the index values is to specify the index values in the template definition and to not specify them in the *cell_rise* and *cell_fall* tables. Such a template would look like this:

```
lu_table_template(delay_template_3x3) {
  variable_1 : input_net_transition;
  variable_2 : total_output_net_capacitance;
  index_1 ("0.1, 0.3, 0.7");
  index_2 ("0.16, 0.35, 1.43");
}
```

Based upon the delay tables, an input fall transition time of 0.3ns and an output load of 0.16pf will correspond to the rise delay of the inverter of 0.1018ns. Since a falling transition at the input results in the inverter output

rise, the table lookup for the rise delay involves a falling transition at the inverter input.

This form of representing delays in a table as a function of two variables, transition time and capacitance, is called the *non-linear* delay model, since non-linear variations of delay with input transition time and load capacitance are expressed in such tables.

The table models can also be 3-dimensional - an example is a flip-flop with complementary outputs, *Q* and *QN*, which is described in Section 3.8.

The NLDM models are used not only for the delay but also for the transition time at the output of a cell which is characterized by the input transition time and the output load. Thus, there are separate two-dimensional tables for computing the output rise and fall transition times of a cell.

```
pin (OUT) {
  max_transition : 1.0;
  timing() {
    related_pin : "INP";
    timing_sense : negative_unate;
    rise_transition(delay_template_3x3) {
      index_1 ("0.1, 0.3, 0.7"); /* Input transition */
      index_2 ("0.16, 0.35, 1.43"); /* Output capacitance */
      values ( /*   0.16      0.35     1.43 */ \
        /* 0.1 */  "0.0417,  0.1337,  0.4680", \
        /* 0.3 */  "0.0718,  0.1827,  0.5676", \
        /* 0.7 */  "0.1034,  0.2173,  0.6452");
    }
    fall_transition(delay_template_3x3) {
      index_1 ("0.1, 0.3, 0.7"); /* Input transition */
      index_2 ("0.16, 0.35, 1.43"); /* Output capacitance */
      values ( /*    0.16      0.35     1.43 */ \
        /* 0.1 */   "0.0817,  0.1937,  0.7280", \
        /* 0.3 */   "0.1018,  0.2327,  0.7676", \
        /* 0.7 */   "0.1334,  0.2973,  0.8452");
    }
```

```
      . . .
    }
    . . .
  }
```

There are two such tables for transition time: *rise_transition* and *fall_transition*. As described in Chapter 2, the transition times are measured based on the specific slew thresholds, usually 10%-90% of the power supply.

As illustrated above, an inverter cell with an NLDM model has the following tables:

- Rise delay
- Fall delay
- Rise transition
- Fall transition

Given the input transition time and output capacitance of such a cell, as shown in Figure 3-3, the rise delay is obtained from the *cell_rise* table for 15ps input transition time (falling) and 10fF load, and the fall delay is obtained from the *cell_fall* table for 20ps input transition time (rising) and 10fF load.

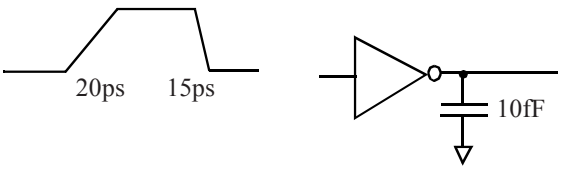


Figure 3-3 *Transition time and capacitance for computing cell delays.*

Where is the information which specifies that the cell is inverting? This information is specified as part of the *timing_sense* field of the timing arc. In some cases, this field is not specified but is expected to be derived from the pin function.

For the example inverter cell, the timing arc is *negative_unate* which implies that the output pin transition direction is opposite (negative) of the input pin transition direction. Thus, the *cell_rise* table lookup corresponds to the falling transition time at the input pin.

Example of Non-Linear Delay Model Lookup

This section illustrates the lookup of the table models through an example. If the input transition time and the output capacitance correspond to a table entry, the table lookup is trivial since the timing value corresponds directly to the value in the table. The example below corresponds to a general case where the lookup does not correspond to any of the entries available in the table. In such cases, two-dimensional interpolation is utilized to provide the resulting timing value. The two nearest table indices in each dimension are chosen for the table interpolation. Consider the table lookup for fall transition (example table specified above) for the input transition time of 0.15ns and an output capacitance of 1.16pF. The corresponding section of the fall transition table relevant for two-dimensional interpolation is reproduced below.

```
fall_transition(delay_template_3x3) {
  index_1 ("0.1, 0.3 . . .");
  index_2 (". . . 0.35, 1.43");
  values ( \
    ". . . 0.1937, 0.7280", \
    ". . . 0.2327, 0.7676"
    . . .
```

In the formulation below, the two *index_1* values are denoted as x_1 and x_2; the two *index_2* values are denoted as y_1 and y_2 and the corresponding table values are denoted as T_{11}, T_{12}, T_{21} and T_{22} respectively.

If the table lookup is required for (x_0, y_0), the lookup value T_{00} is obtained by interpolation and is given by:

$$T_{00} = x_{20} * y_{20} * T_{11} + x_{20} * y_{01} * T_{12} + x_{01} * y_{20} * T_{21} + x_{01} * y_{01} * T_{22}$$

where

$$x_{01} = (x_0 - x_1) / (x_2 - x_1)$$
$$x_{20} = (x_2 - x_0) / (x_2 - x_1)$$
$$y_{01} = (y_0 - y_1) / (y_2 - y_1)$$
$$y_{20} = (y_2 - y_0) / (y_2 - y_1)$$

Substituting 0.15 for *index_1* and 1.16 for *index_2* results in the fall_transition value of:

$$T_{00} = 0.75 * 0.25 * 0.1937 + 0.75 * 0.75 * 0.7280 + 0.25 * 0.25 * 0.2327 + 0.25 * 0.75 * 0.7676 = 0.6043$$

Note that the equations above are valid for interpolation as well as extrapolation - that is when the indices (x_0, y_0) lie outside the characterized range of indices. As an example, for the table lookup with 0.05 for *index_1* and 1.7 for *index_2*, the fall transition value is obtained as:

$$T_{00} = 1.25 * (-0.25) * 0.1937 + 1.25 * 1.25 * 0.7280 + (-0.25) * (-0.25) * 0.2327 + (-0.25) * 1.25 * 0.7676 = 0.8516$$

3.2.3 Threshold Specifications and Slew Derating

The slew[1] values are based upon the measurement thresholds specified in the library. Most of the previous generation libraries (0.25µm or older) used 10% and 90% as measurement thresholds for slew or transition time.

1. Slew is same as transition time.

The slew thresholds are chosen to correspond to the linear portion of the waveform. As technology becomes finer, the portion where the actual waveform is most linear is typically between 30% and 70% points. Thus, most of the newer generation timing libraries specify slew measurement points as 30% and 70% of *Vdd*. However, because the transition times were previously measured between 10% and 90%, the transition times measured between 30% and 70% are usually doubled for populating the library. This is specified by the *slew derate factor* which is typically specified as 0.5. The slew thresholds of 30% and 70% with slew derate as 0.5 results in equivalent measurement points of 10% and 90%. An example settings of threshold is illustrated below.

```
/* Threshold definitions */
slew_lower_threshold_pct_fall : 30.0;
slew_upper_threshold_pct_fall : 70.0;
slew_lower_threshold_pct_rise : 30.0;
slew_upper_threshold_pct_rise : 70.0;
input_threshold_pct_fall : 50.0;
input_threshold_pct_rise : 50.0;
output_threshold_pct_fall : 50.0;
output_threshold_pct_rise : 50.0;
slew_derate_from_library : 0.5;
```

The above settings specify that the transition times in the library tables have to be multiplied by 0.5 to obtain the transition times which correspond to the slew threshold (30-70) settings. This means that the values in the transition tables (as well as corresponding index values) are effectively 10-90 values. During characterization, the transition is measured at 30-70 and the transition data in the library corresponds to extrapolation of measured values to 10% to 90% ((70 - 30)/(90 - 10) = 0.5).

Another example with a different set of slew threshold settings may contain:

```
/* Threshold definitions 20/80/1 */
slew_lower_threshold_pct_fall : 20.0;
```

```
slew_upper_threshold_pct_fall : 80.0;
slew_lower_threshold_pct_rise : 20.0;
slew_upper_threshold_pct_rise : 80.0;
/* slew_derate_from_library not specified */
```

In this example of 20-80 slew threshold settings, there is no *slew_derate_from_library* specified (implies a default of 1.0), which means that the transition time data in the library is not derated. The values in the transition tables correspond directly to the 20-80 characterized slew values. See Figure 3-4.

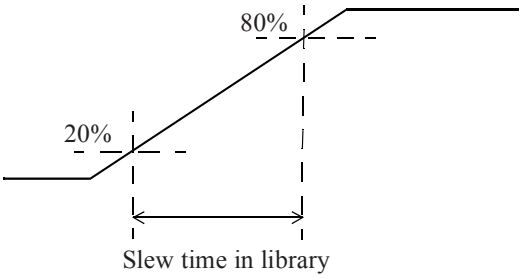


Figure 3-4 *No derating of slew.*

Here is another example of slew threshold settings in a cell library.

```
slew_lower_threshold_pct_rise : 20.00;
slew_upper_threshold_pct_rise : 80.00;
slew_lower_threshold_pct_fall : 20.00;
slew_upper_threshold_pct_fall : 80.00;
slew_derate_from_library : 0.6;
```

In this case, the *slew_derate_from_library* is set to 0.6 and characterization slew trip points are specified as 20% and 80%. This implies that transition table data in the library corresponds to 0% to 100% ((80 - 20)/(100 - 0) = 0.6) extrapolated values. This is shown in Figure 3-5.

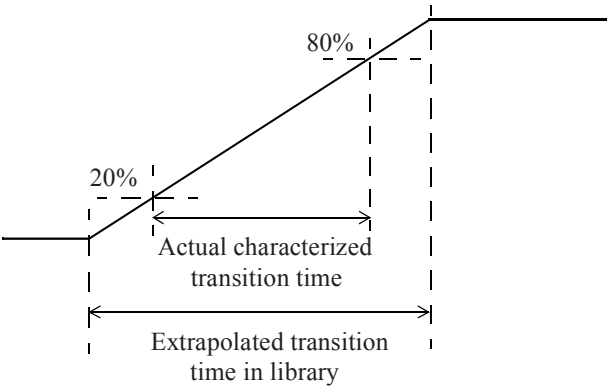


Figure 3-5 *Slew derating applied.*

When slew derating is specified, the slew value internally used during delay calculation is:

```
library_transition_time_value * slew_derate
```

This is the slew used internally by the delay calculation tool and corresponds to the characterized slew threshold measurement points.

3.3 Timing Models - Combinational Cells

Let us consider the timing arcs for a two-input *and* cell. Both the timing arcs for this cell are *positive_unate*; therefore an input pin rise corresponds to an output rise and vice versa.

For the two-input *and* cell, there are four delays:

- *A* -> *Z*: Output rise
- *A* -> *Z*: Output fall
- *B* -> *Z*: Output rise

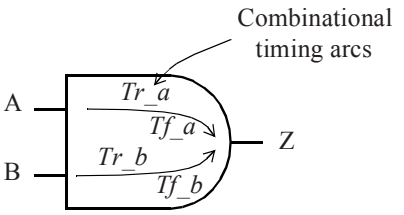


Figure 3-6 *Combinational timing arcs.*

- *B* -> *Z*: Output fall

This implies that for the NLDM model, there would be four table models for specifying delays. Similarly, there would be four such table models for specifying the output transition times as well.

3.3.1 Delay and Slew Models

An example of a timing model for input *INP1* to output *OUT* for a three-input *nand* cell is specified as follows.

```
pin (OUT) {
  max_transition : 1.0;
  timing() {
    related_pin : "INP1";
    timing_sense : negative_unate;
    cell_rise(delay_template_3x3) {
      index_1 ("0.1, 0.3, 0.7");
      index_2 ("0.16, 0.35, 1.43");
      values ( \
        "0.0513, 0.1537, 0.5280", \
        "0.1018, 0.2327, 0.6476", \
        "0.1334, 0.2973, 0.7252");
      }
    rise_transition(delay_template_3x3) {
      index_1 ("0.1, 0.3, 0.7");
```

```
        index_2 ("0.16, 0.35, 1.43");
        values ( \
          "0.0417, 0.1337, 0.4680", \
          "0.0718, 0.1827, 0.5676", \
          "0.1034, 0.2173, 0.6452");
      }
      cell_fall(delay_template_3x3) {
        index_1 ("0.1, 0.3, 0.7");
        index_2 ("0.16, 0.35, 1.43");
        values ( \
          "0.0617, 0.1537, 0.5280", \
          "0.0918, 0.2027, 0.5676", \
          "0.1034, 0.2273, 0.6452");
      }
      fall_transition(delay_template_3x3) {
        index_1 ("0.1, 0.3, 0.7");
        index_2 ("0.16, 0.35, 1.43");
        values ( \
          "0.0817, 0.1937, 0.7280", \
          "0.1018, 0.2327, 0.7676", \
          "0.1334, 0.2973, 0.8452");
      }
      . . .
    }
    . . .
  }
```

In this example, the characteristics of the timing arc from *INP1* to *OUT* are described using two cell delay tables, *cell_rise* and *cell_fall*, and two transition tables, *rise_transition* and *fall_transition*. The output *max_transition* value is also included in the above example.

Positive or Negative Unate

As described in Section 2.7, the timing arc in the *nand* cell example is negative unate which implies that the output pin transition direction is opposite (negative) of the input pin transition direction. Thus, the *cell_rise* table

lookup corresponds to the falling transition time at the input pin. On the other hand, the timing arcs through an *and* cell or *or* cell are positive unate since the output transition is in the same direction as the input transition.

3.3.2 General Combinational Block

Consider a combinational block with three inputs and two outputs.

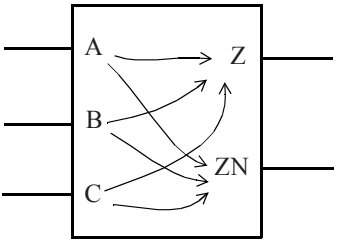


Figure 3-7 *General combinational block.*

A block such as this can have a number of timing arcs. In general, the timing arcs can be from each input to each output of the block. If the logic path from input to output is non-inverting or positive unate, then the output has the same polarity as the input. If it is an inverting logic path or negative unate, the output has an opposite polarity to input; thus, when the input rises, the output falls. These timing arcs represent the propagation delays through the block.

Some timing arcs through a combinational cell can be positive unate as well as negative unate. An example is the timing arc through a two-input *xor* cell. A transition at an input of a two-input *xor* cell can cause an output transition in the same or in the opposite transition direction depending on the logic state of the other input of the cell. The timing for these arcs can be described as non-unate or as two different sets of positive unate and negative unate timing models which are state-dependent. Such state-dependent tables are described in greater detail in Section 3.5.

3.4 Timing Models - Sequential Cells

Consider the timing arcs of a sequential cell shown in Figure 3-8.

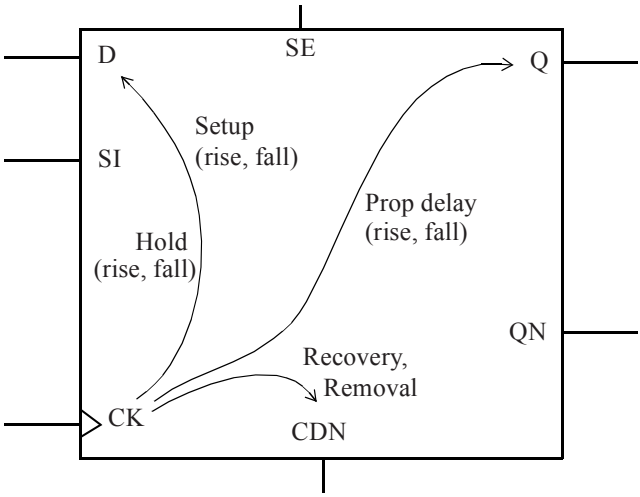


Figure 3-8 *Sequential cell timing arcs.*

For synchronous inputs, such as pin *D* (or *SI, SE*), there are the following timing arcs:

i. Setup check arc (rising and falling)

ii. Hold check arc (rising and falling)

For asynchronous inputs, such as pin *CDN*, there are the following timing arcs:

i. Recovery check arc

ii. Removal check arc

For synchronous outputs of a flip-flop, such as pins *Q* or *QN*, there is the following timing arc:

i. CK-to-output propagation delay arc (rising and falling)

All of the synchronous timing arcs are with respect to the **active edge** of the clock, the edge of the clock that causes the sequential cell to capture the data. In addition, the clock pin and asynchronous pins such as clear, can have pulse width timing checks. Figure 3-9 shows the timing checks using various signal waveforms.

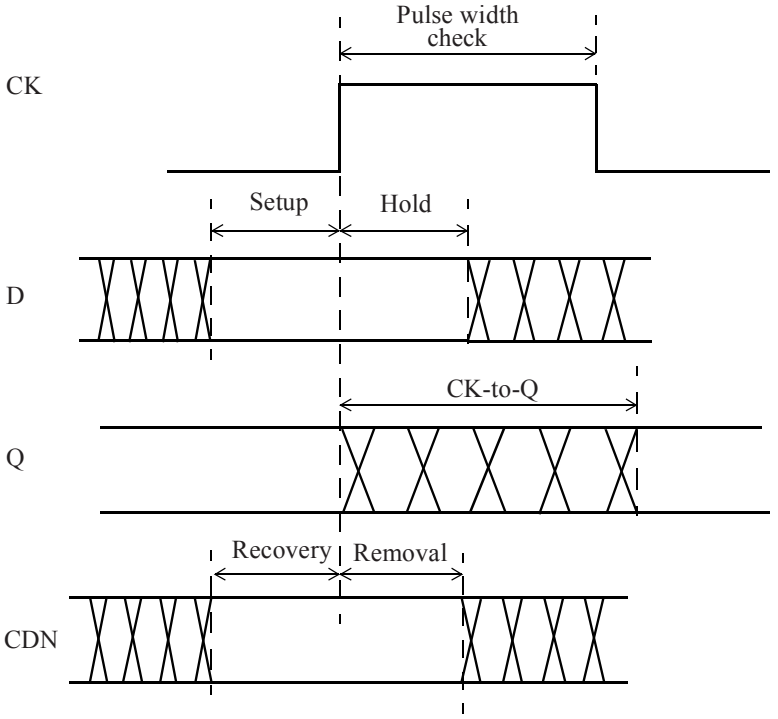


Figure 3-9 *Timing arcs on an active rising clock edge.*

3.4.1 Synchronous Checks: Setup and Hold

The setup and hold synchronous timing checks are needed for proper propagation of data through the sequential cells. These checks verify that the data input is unambiguous at the active edge of the clock and the proper data is latched in at the active edge. These timing checks validate if the data input is stable around the active clock edge. The minimum time before the active clock when the data input must remain stable is called the *setup time*. This is measured as the time interval from the latest data signal crossing its threshold (normally 50% of *Vdd*) to the active clock edge crossing its threshold (normally 50% of *Vdd*). Similarly, the hold time is the minimum time the data input must remain stable just after the active edge of the clock. This is measured as the time interval from the active clock edge crossing its threshold to the earliest data signal crossing its threshold. As mentioned previously, the active edge of the clock for a sequential cell is the rising or falling edge that causes the sequential cell to capture data.

Example of Setup and Hold Checks

The setup and hold constraints for a synchronous pin of a sequential cell are normally described in terms of two-dimensional tables as illustrated below. The example below shows the setup and hold timing information for the data pin of a flip-flop.

```
pin (D) {
  direction : input;
  . . .
  timing () {
    related_pin : "CK";
    timing_type : "setup_rising";
    rise_constraint ("setuphold_template_3x3") {
      index_1("0.4, 0.57, 0.84"); /* Data transition */
      index_2("0.4, 0.57, 0.84"); /* Clock transition */
      values( /*    0.4      0.57     0.84 */ \
        /* 0.4 */   "0.063,  0.093,   0.112", \
        /* 0.57 */  "0.526,  0.644,   0.824", \
        /* 0.84 */  "0.720,  0.839,   0.930");
```

```
    }
    fall_constraint ("setuphold_template_3x3") {
     index_1("0.4, 0.57, 0.84"); /* Data transition */
     index_2("0.4, 0.57, 0.84"); /* Clock transition */
     values( /*    0.4     0.57     0.84 */ \
       /* 0.4 */   "0.762,  0.895,   0.969", \
       /* 0.57 */  "0.804,  0.952,   0.166", \
       /* 0.84 */  "0.159,  0.170,   0.245");
    }
  }
}

timing () {
  related_pin : "CK";
  timing_type : "hold_rising";
  rise_constraint ("setuphold_template_3x3") {
    index_1("0.4, 0.57, 0.84"); /* Data transition */
    index_2("0.4, 0.57, 0.84"); /* Clock transition */
    values(   /*    0.4      0.57     0.84 */ \
      /* 0.4 */   "-0.220,  -0.339,  -0.584", \
      /* 0.57 */  "-0.247,  -0.381,  -0.729", \
      /* 0.84 */  "-0.398,  -0.516,  -0.864");
  }
  fall_constraint ("setuphold_template_3x3") {
    index_1("0.4, 0.57, 0.84"); /* Data transition */
    index_2("0.4, 0.57, 0.84");/* Clock transition */
    values(   /*    0.4         0.57     0.84 */ \
      /* 0.4 */    "-0.028,    -0.397,  -0.489", \
      /* 0.57 */   "-0.408,    -0.527,  -0.649", \
      /* 0.84 */   "-0.705,    -0.839,  -0.580");
  }
}
```

The example above shows setup and hold constraints on the input pin *D* with respect to the rising edge of the clock *CK* of a sequential cell. The two-dimensional models are in terms of the transition times at the *constrained_pin* (*D*) and the *related_pin* (*CK*). The lookup for the two-

dimensional table is based upon the template *setuphold_template_3x3* described in the library. For the above example, the lookup table template *setuphold_template_3x3* is described as:

```
lu_table_template(setuphold_template_3x3) {
  variable_1 : constrained_pin_transition;
  variable_2 : related_pin_transition;
  index_1 ("1000, 1001, 1002");
  index_2 ("1000, 1001, 1002");
}
/* The constrained pin and the related pin can be in either or-
der, that is, variable_1 could be the related pin transition.
However, these designations are usually consistent across all
templates in a library. */
```

Like in previous examples, the setup values in the table are specified like a nested loop with the first index, *index_1*, being the outer (or least varying) variable and the second index, *index_2*, being the inner (or most varying) variable and so on. Thus, with a *D* pin rise transition time of 0.4ns and *CK* pin rise transition time of 0.84ns, the setup constraint for the rising edge of the *D* pin is 0.112ns - the value is read from the *rise_constraint* table. For the falling edge of the *D* pin, the setup constraint will examine the *fall_constraint* table of the setup tables. For lookup of the setup and hold constraint tables where the transition times do not correspond to the index values, the general procedure for *non-linear model* lookup described in Section 3.2 is applicable.

Note that the *rise_constraint* and *fall_constraint* tables of the setup constraint refer to the *constrained_pin*. The clock transition used is determined by the *timing_type* which specifies whether the cell is rising edge-triggered or falling edge-triggered.

Negative Values in Setup and Hold Checks

Notice that some of the hold values in the example above are negative. This is acceptable and normally happens when the path from the pin of the flip-flop to the internal latch point for the data is longer than the corresponding

path for the clock. Thus, a negative hold check implies that the data pin of the flip-flop can change ahead of the clock pin and still meet the hold time check.

The setup values of a flip-flop can also be negative. This means that at the pins of the flip-flop, the data can change after the clock pin and still meet the setup time check.

Can both setup and hold be negative? No; for the setup and hold checks to be consistent, the sum of setup and hold values should be positive. Thus, if the setup (or hold) check contains negative values - the corresponding hold (or setup) should be sufficiently positive so that the setup plus hold value is a positive quantity. See Figure 3-10 for an example with a negative hold value. Since the setup has to occur prior to the hold, setup plus hold is a positive quantity. The setup plus hold time is the width of the region where the data signal is required to be steady.

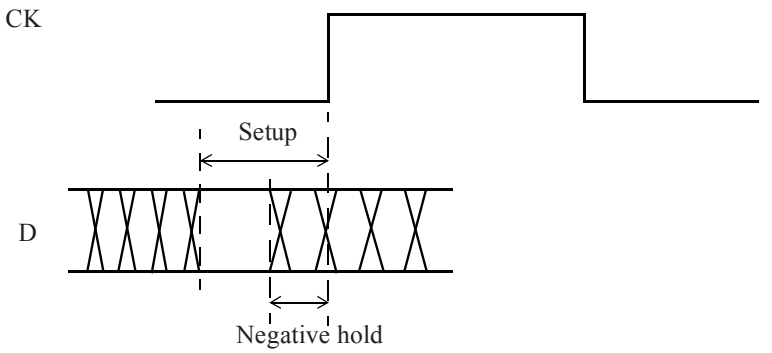


Figure 3-10 *Negative value for hold timing check.*

For flip-flops, it is helpful to have a negative hold time on scan data input pins. This gives flexibility in terms of clock skew and can eliminate the need for almost all buffer insertion for fixing hold violations in scan mode (*scan mode* is the one in which flip-flops are tied serially forming a scan chain - output of flip-flop is typically connected to the scan data input pin of the next flip-flop in series; these connections are for testability).

Similar to the setup or hold check on the synchronous data inputs, there are constraint checks governing the asynchronous pins. These are described next.

3.4.2 Asynchronous Checks

Recovery and Removal Checks

Asynchronous pins such as asynchronous clear or asynchronous set override any synchronous behavior of the cell. When an asynchronous pin is active, the output is governed by the asynchronous pin and not by the clock latching in the data inputs. However, when the asynchronous pin becomes inactive, the active edge of the clock starts latching in the data input. The asynchronous recovery and removal constraint checks verify that the asynchronous pin has returned unambiguously to an inactive state at the next active clock edge.

The **recovery time** is the minimum time that an asynchronous input is stable after being de-asserted before the next active clock edge.

Similarly, the **removal time** is the minimum time after an active clock edge that the asynchronous pin must remain active before it can be de-asserted.

The asynchronous removal and recovery checks are described in Section 8.6 and Section 8.7 respectively.

Pulse Width Checks

In addition to the synchronous and asynchronous timing checks, there is a check which ensures that the pulse width at an input pin of a cell meets the minimum requirement. For example, if the width of pulse at the clock pin is smaller than the specified minimum, the clock may not latch the data properly. The pulse width checks can be specified for relevant synchronous and asynchronous pins also. The minimum pulse width checks can be specified for high pulse and also for low pulse.

Example of Recovery, Removal and Pulse Width Checks

An example of recovery time, removal time, and pulse width check for an asynchronous clear pin *CDN* of a flip-flop is given below. The recovery and removal checks are with respect to the clock pin *CK*. Since the recovery and removal checks are defined for an asynchronous pin being de-asserted, only rise constraints exists in the example below. The minimum pulse width check for the pin *CDN* is for a low pulse. Since the *CDN* pin is active low, there is no constraint for the high pulse width on this pin and is thus not specified.

```
pin(CDN) {
  direction : input;
  capacitance : 0.002236;
  . . .
  timing() {
    related_pin : "CDN";
    timing_type : min_pulse_width;
    fall_constraint(width_template_3x1) { /*low pulse check*/
     index_1 ("0.032, 0.504, 0.788"); /* Input transition */
     values ( /*    0.032     0.504    0.788 */ \
                  "0.034,    0.060,   0.377");
    }
  }
  timing() {
    related_pin : "CK";
    timing_type : recovery_rising;
    rise_constraint(recovery_template_3x3) { /* CDN rising */
     index_1 ("0.032, 0.504, 0.788"); /* Data transition */
     index_2 ("0.032, 0.504, 0.788"); /* Clock transition */
     values( /*     0.032     0.504    0.788 */ \
       /* 0.032 */  "-0.198,   -0.122,  0.187", \
       /* 0.504 */  "-0.268,   -0.157,  0.124", \
       /* 0.788 */  "-0.490,   -0.219,  -0.069");
    }
  }
```

```
  timing() {
    related_pin : "CP";
    timing_type : removal_rising;
    rise_constraint(removal_template_3x3) { /* CDN rising */
      index_1 ("0.032, 0.504, 0.788"); /* Data transition */
      index_2 ("0.032, 0.504, 0.788"); /* Clock transition */
      values( /*     0.032   0.504    0.788 */ \
        /* 0.032 */  "0.106,  0.167,   0.548", \
        /* 0.504 */  "0.221,  0.381,   0.662", \
        /* 0.788 */  "0.381,  0.456,   0.778");
    }
  }
}
```

3.4.3 Propagation Delay

The propagation delay of a sequential cell is from the active edge of the clock to a rising or falling edge on the output. Here is an example of a propagation delay arc for a negative edge-triggered flip-flop, from clock pin *CKN* to output *Q*. This is a non-unate timing arc as the active edge of the clock can cause either a rising or a falling edge on the output *Q*. Here is the delay table:

```
timing() {
  related_pin : "CKN";
  timing_type : falling_edge;
  timing_sense : non_unate;
  cell_rise(delay_template_3x3) {
    index_1 ("0.1, 0.3, 0.7"); /* Clock transition */
    index_2 ("0.16, 0.35, 1.43"); /* Output capacitance */
    values ( /*    0.16    0.35     1.43 */ \
      /* 0.1 */   "0.0513,  0.1537,  0.5280", \
      /* 0.3 */   "0.1018,  0.2327,  0.6476", \
      /* 0.7 */   "0.1334,  0.2973,  0.7252");
  }
```

```
  rise_transition(delay_template_3x3) {
    index_1 ("0.1, 0.3, 0.7");
    index_2 ("0.16, 0.35, 1.43");
    values ( \
      "0.0417, 0.1337, 0.4680", \
      "0.0718, 0.1827, 0.5676", \
      "0.1034, 0.2173, 0.6452");
  }
  cell_fall(delay_template_3x3) {
    index_1 ("0.1, 0.3, 0.7");
    index_2 ("0.16, 0.35, 1.43");
    values ( \
      "0.0617, 0.1537, 0.5280", \
      "0.0918, 0.2027, 0.5676", \
      "0.1034, 0.2273, 0.6452");
  }
  fall_transition(delay_template_3x3) {
    index_1 ("0.1, 0.3, 0.7");
    index_2 ("0.16, 0.35, 1.43");
    values ( \
      "0.0817, 0.1937, 0.7280", \
      "0.1018, 0.2327, 0.7676", \
      "0.1334, 0.2973, 0.8452");
  }
}
```

As in the earlier examples, the delays to the output are expressed as two-dimensional tables in terms of input transition time and the capacitance at the output pin. However in this example, the input transition time to use is the falling transition time at the *CKN* pin since this is a falling edge-triggered flip-flop. This is indicated by the construct *timing_type* in the example above. A rising edge-triggered flip-flop will specify *rising_edge* as its *timing_type*.

```
  timing() {
    related_pin : "CKP";
    timing_type : rising_edge;
```

```
    timing_sense : non_unate;
    cell_rise(delay_template_3x3) {
      . . .
    }
    . . .
  }
```

3.5 State-Dependent Models

In many combinational blocks, the timing arcs between inputs and outputs depend on the state of other pins in the block. These timing arcs between input and output pins can be positive unate, negative unate, or both positive as well as negative unate arcs. An example is the *xor* or *xnor* cell where the timing to the output can be positive unate or negative unate. In such cases, the timing behaviors can be different depending upon the state of other inputs of the block. In general, multiple timing models depending upon the states of the pins are described. Such models are referred to as **state-dependent models**.

XOR, XNOR and Sequential Cells

Consider an example of a two-input *xor* cell. The timing path from an input *A1* to output *Z* is positive unate when the other input *A2* is logic-0. When the input *A2* is logic-1, the path from *A1* to *Z* is negative unate. These two timing models are specified using state-dependent models. The timing model from *A1* to *Z* when *A2* is logic-0 is specified as follows:

```
pin (Z) {
  direction : output;
  max_capacitance : 0.0842;
  function : "(A1^A2)";
  timing() {
    related_pin : "A1";
    when : "!A2";
    sdf_cond : "A2 == 1'b0";
```

```
    timing_sense : positive_unate;
    cell_rise(delay_template_3x3) {
     index_1 ("0.0272, 0.0576, 0.1184"); /* Input slew */
     index_2 ("0.0102, 0.0208, 0.0419"); /* Output load */
     values( \
       "0.0581, 0.0898, 0.2791", \
       "0.0913, 0.1545, 0.2806", \
       "0.0461, 0.0626, 0.2838");
   }
   . . .
}
```

The state-dependent condition is specified using the *when* condition. While the cell model excerpt only illustrates the *cell_rise* delay, other timing models (*cell_fall, rise_transition* and *fall_transition* tables) are also specified with the same *when* condition. A separate timing model is specified for the other *when* condition - for the case when *A2* is logic-1.

```
timing() {
  related_pin : "A1";
  when : "A2";
  sdf_cond : "A2 == 1'b1";
  timing_sense : negative_unate;
  cell_fall(delay_template_3x3) {
    index_1 ("0.0272, 0.0576, 0.1184");
    index_2 ("0.0102, 0.0208, 0.0419");
    values( \
      "0.0784, 0.1019, 0.2269", \
      "0.0943, 0.1177, 0.2428", \
      "0.0997, 0.1796, 0.2620");
  }
  . . .
}
```

The *sdf_cond* is used to specify the condition of the timing arc that is to be used when generating SDF - see the example in Section 3.9 and the *COND* construct described in Appendix B.

State-dependent models are used for various types of timing arcs. Many sequential cells specify the setup or hold timing constraints using state-dependent models. An example of a scan flip-flop using state-dependent models for hold constraint is specified next. In this case, two sets of models are specified - one when the scan enable pin *SE* is active and another when the scan enable pin is inactive.

```
pin (D) {
  . . .
  timing() {
    related_pin : "CK";
    timing_type : hold_rising;
    when : "!SE";
    fall_constraint(hold_template_3x3) {
      index_1("0.08573, 0.2057, 0.3926");
      index_2("0.08573, 0.2057, 0.3926");
      values("-0.05018, -0.02966, -0.00919",\
             "-0.0703, -0.05008, -0.0091",\
             "-0.1407, -0.1206, -0.1096");
    }
    . . .
  }
}
```

The above model is used when the *SE* pin is at logic-0. A similar model is specified with the *when* condition *SE* as logic-1.

Some timing relationships are specified using both state-dependent as well as non-state-dependent models. In such cases, the timing analysis will use the state-dependent model if the state of the cell is known and is included in one of the state-dependent models. If the state-dependent models do not cover the condition of the cell, the timing from the non-state-dependent model is utilized. For example, consider a case where the hold constraint is

specified by only one *when* condition for *SE* at logic-0 and no separate state-dependent model is specified for *SE* at logic-1. In such a scenario, if the *SE* is set to logic-1, the hold constraint from the non-state-dependent model is used. If there is no non-state-dependent model for the hold constraint, there will *not* be any active hold constraint!

State-dependent models can be specified for any of the attributes in the timing library. Thus state-dependent specifications can exist for power, leakage power, transition time, rise and fall delays, timing constraints, and so on. An example of the state dependent leakage power specification is given below:

```
leakage_power() {
  when : "A1 !A2";
  value : 259.8;
}
leakage_power() {
  when : "A1 A2";
  value : 282.7;
}
```

3.6 Interface Timing Model for a Black Box

This section describes the timing arcs for the IO interfaces of a black box (an arbitrary module or block). A timing model captures the timing for the IO interfaces of the black box. The black box interface model can have combinational as well as sequential timing arcs. In general, these arcs can also be state-dependent.

For the example shown in Figure 3-11, the timing arcs can be placed under the following categories:

- *Input to output combinational arc*: This corresponds to a direct combinational path from input to output, such as from the input port *FIN* to the output port *FOUT*.

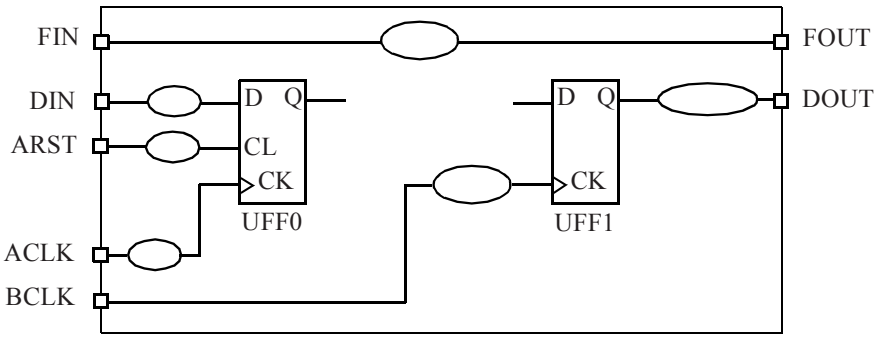


Figure 3-11 *Design modeled with interface timing.*

- *Input sequential arc*: This is described as setup or hold time for the input connected to a *D*-pin of the flip-flop. In general, there can be combinational logic from the input of the block before it is connected to a *D*-pin of the flip-flop. An example of this is a set-up check at port *DIN* with respect to clock *ACLK*.

- *Asynchronous input arc*: This is similar to the recovery or removal timing constraint for the input asynchronous pins of a flip-flop. An example is the input *ARST* to the asynchronous clear pin of flip-flop *UFF0*.

- *Output sequential arc*: This is similar to the output propagation timing for the clock to output connected to *Q* of the flip-flop. In general, there can be combinational logic between the flip-flop output and the output of the module. An example is the path from clock *BCLK* to the output of flip-flop *UFF1* to the output port *DOUT*.

In addition to the timing arcs above, there can also be pulse width checks on the external clock pins of the black box. It is also possible to define internal nodes and to define generated clocks on these internal nodes as well as

specify timing arcs to and from these nodes. In summary, a black box model can have the following timing arcs:

i. Input to output timing arcs for combinational logic paths.

ii. Setup and hold timing arcs from the synchronous inputs to the related clock pins.

iii. Recovery and removal timing arcs for the asynchronous inputs to the related clock pins.

iv. Output propagation delay from clock pins to the output pins.

The interface timing model as described above is not intended to capture the internal timing of the black box, but only the timing of its interfaces.

3.7 Advanced Timing Modeling

The timing models, such as NLDM, represent the delay through the timing arcs based upon output load capacitance and input transition time. In reality, the load seen by the cell output is comprised of capacitance as well as interconnect resistance. The interconnect resistance becomes an issue since the NLDM approach assumes that the output loading is purely capacitive. Even with non-zero interconnect resistance, these NLDM models have been utilized when the effect of interconnect resistance is small. In presence of resistive interconnect, the delay calculation methodologies retrofit the NLDM models by obtaining an equivalent **effective capacitance** at the output of the cell. The "effective" capacitance methodology used within delay calculation tools obtains an equivalent capacitance that has the same delay at the output of the cell as the cell with RC interconnect. The effective capacitance approach is described as part of delay calculation in Section 5.2.

As the feature size shrinks, the effect of interconnect resistance can result in large inaccuracy as the waveforms become highly non-linear. Various modeling approaches provide additional accuracy for the cell output drivers. Broadly, these approaches obtain higher accuracy by modeling the out-

put stage of the driver by an equivalent current source. Examples of these approaches are - **CCS** (**C**omposite **C**urrent **S**ource), or ECSM (**E**ffective **C**urrent **S**ource **M**odel). For example, the CCS timing models provide the additional accuracy for modeling cell output drivers by using a time-varying and voltage-dependent current source. The timing information is provided by specifying detailed models for the receiver pin capacitance[1] and output charging currents under different scenarios. The details of the CCS model are described next.

3.7.1 Receiver Pin Capacitance

The receiver pin capacitance corresponds to the input pin capacitance specified for the NLDM models. Unlike the pin capacitance for the NLDM models, the CCS models allow separate specification of receiver capacitance in different portions of the transitioning waveform. Due to interconnect RC and the equivalent input non-linear capacitance due to the Miller effect from the input devices within the cell, the receiver capacitance value varies at different points on the transitioning waveform. This capacitance is thus modeled differently in the initial (or leading) portion of waveform versus the trailing portion of the waveform.

The receiver pin capacitance can be specified at the pin level (like in NLDM models) where all timing arcs through that pin use that capacitance value. Alternately, the receiver capacitance can be specified at the timing arc level in which case different capacitance models can be specified for different timing arcs. These two methods of specifying the receiver pin capacitance are described next.

1. Input pin capacitance - equivalent to *pin capacitance* in NLDM.

Specifying Capacitance at the Pin Level

When specified at the pin level, an example of a one-dimensional table specification for receiver pin capacitance is given next.

```
pin (IN) {
  . . .
  receiver_capacitance1_rise ("Lookup_table_4") {
  index_1: ("0.1, 0.2, 0.3, 0.4"); /* Input transition */
  values("0.001040, 0.001072, 0.001074, 0.001085");
}
```

The *index_1* specifies the indices for input transition time for this pin. The one-dimensional table in *values* specifies the receiver capacitance for rising waveform at an input pin for the leading portion of the waveform.

Similar to the *receiver_capacitance1_rise* shown above, the *receiver_capacitance2_rise* specifies the rise capacitance for the trailing portion of the input rising waveform. The fall capacitances (pin capacitance for falling input waveform) are specified by the attributes *receiver_capacitance1_fall* and *receiver_capacitance2_fall* respectively.

Specifying Capacitance at the Timing Arc Level

The receiver pin capacitance can also be specified with the timing arc as a two-dimensional table in terms of input transition time and output load. An example of the specification at the timing arc level is given below. This example specifies the receiver pin rise capacitance for the leading portion of the waveform at pin *IN* as a function of transition time at input pin *IN* and load at output pin *OUT*.

```
pin (OUT) {
  . . .
  timing () {
    related_pin : "IN"-;
    . . .
```

```
    receiver_capacitance1_rise ("Lookup_table_4x4") {
      index_1("0.1, 0.2, 0.3, 0.4"); /* Input transition */
      index_2("0.01, 0.2, 0.4, 0.8"); /* Output capacitance */
      values("0.001040-, 0.001072-, 0.001074-, 0.001075", \
             "0.001148-, 0.001150-, 0.001152-, 0.001153", \
             "0.001174-, 0.001172-, 0.001172-, 0.001172", \
             "0.001174-, 0.001171-, 0.001177-, 0.001174");
    }
    . . .
  }
  . . .
}
```

The above example specifies the model for the *receiver_capacitance1_rise*. The library includes similar definitions for the *receiver_capacitance2_rise, receiver_capacitance1_fall,* and *receiver_capacitance2_fall* specifications.

The four different types of receiver capacitance types are summarized in the table below. As illustrated above, these can be specified at the pin level as a one-dimensional table or at the timing level as a two-dimensional table.

Capacitance type	*Edge*	*Transition*
Receiver_capacitance1_rise	Rising	Leading portion of transition
Receiver_capacitance1_fall	Falling	Leading portion of transition
Receiver_capacitance2_rise	Rising	Trailing portion of transition
Receiver_capacitance2_fall	Falling	Trailing portion of transition

Table 3-12 *Receiver capacitance types.*

3.7.2 Output Current

In the CCS model, the non-linear timing is represented in terms of output current. The output current information is specified as a lookup table that is dependent on input transition time and output load.

The output current is specified for different combinations of input transition time and output capacitance. For each of these combinations, the output current waveform is specified. Essentially, the waveform here refers to output current values specified as a function of time. An example of the output current for falling output waveform, specified using *output_current_fall*, is as shown next.

```
pin (OUT) {
  . . .
  timing () {
    related_pin : "IN"-;
    . . .
    output_current_fall () {
      vector ("LOOKUP_TABLE_1x1x5") {
        reference_time : 5.06; /* Time of input crossing
          threshold */
        index_1("0.040"); /* Input transition */
        index_2("0.900"); /* Output capacitance */
        index_3("5.079e+00, 5.093e+00, 5.152e+00,
                 5.170e+00, 5.352e+00");/* Time values */
        /* Output charging current: */
        values("-5.784e-02, -5.980e-02, -5.417e-02,
                 -4.257e-02, -2.184e-03");
      }
      . . .
    }
    . . .
  }
  . . .
}
```

The *reference_time* attribute refers to the time when the input waveform crosses the delay threshold. The *index_1* and *index_2* refer to the input transition time and the output load used and *index_3* is the time. The *index_1* and *index_2* (the input transition time and output capacitance) can have only one value each. The *index_3* refers to the time values and the table values refer to the corresponding output current. Thus, for the given input transition time and output load, the output current waveform as a function of time is available. Additional lookup tables for other combinations of input transition time and output capacitance are also specified.

Output current for rising output waveform, specified using *output_current_rise*, is described similarly.

3.7.3 Models for Crosstalk Noise Analysis

This section describes the CCS models for the crosstalk noise (or glitch) analysis. These are described as **CCSN** (**CCS N**oise) models. The CCS noise models are structural models and are represented for different CCBs (Channel Connected Blocks) within the cell.

What is a CCB? The **CCB** refers to the source-drain channel connected portion of a cell. For example, single stage cells such as an *inverter*, *nand* and *nor* cells contain only one CCB - the entire cell is connected through using one channel connection region. Multi-stage cells such as *and* cells, or *or* cells, contain multiple CCBs.

The CCSN models are usually specified for the first CCB driven by the cell input, and the last CCB driving the cell output. These are specified using steady state current, output voltage and propagated noise models.

For single stage combinational cells such as *nand* and *nor* cells, the CCS noise models are specified for each timing arc. These cells have only one CCB and thus the models are from input pins to the output pin of the cell.

An example model for a *nand* cell is described below:

```
pin (OUT) {
  . . .
  timing () {
    related_pin : "IN1";
    . . .
    ccsn_first_stage() { /* First stage CCB */
      is_needed : true;
      stage_type : both; /*CCB contains pull-up and pull-down*/
      is_inverting : true;
      miller_cap_rise : 0.8;
      miller_cap_fall : 0.5;
      dc_current (ccsn_dc) {
        index_1 ("-0.9, 0, 0.5, 1.35, 1.8"); /* Input voltage */
        index_2 ("-0.9, 0, 0.5, 1.35, 1.8"); /* Output voltage*/
        values ( \
          "1.56, 0.42, . . ."); /* Current at output pin */
      }
      . . .
      output_voltage_rise () {
        vector (ccsn_ovrf) {
          index_1 ("0.01"); /* Rail-to-rail input transition */
          index_2 ("0.001"); /* Output net capacitance */
          index_3 ("0.3, 0.5, 0.8"); /* Time */
          values ("0.27, 0.63, 0.81");
        }
        . . .
      }
      output_voltage_fall () {
        vector (ccsn_ovrf) {
          index_1 ("0.01"); /* Rail-to-rail input transition */
          index_2 ("0.001"); /* Output net capacitance */
          index_3 ("0.2, 0.4, 0.6"); /* Time */
          values ("0.81, 0.63, 0.27");
        }
        . . .
```

```
        }
        propagated_noise_low () {
          vector (ccsn_pnlh) {
          index_1 ("0.5"); /* Input glitch height */
          index_2 ("0.6"); /* Input glitch width */
          index_3 ("0.05"); /* Output net capacitance */
          index_4 ("0.3, 0.4, 0.5, 0.7"); /* Time */
          values ("0.19, 0.23, 0.19, 0.11");
        }
        propagated_noise_high () {
          . . .
      }
    }
  }
```

We now describe the attributes of the CCS noise model. The attribute *ccsn_first_stage* indicates that the model is for the first stage CCB of the *nand* cell. As mentioned before, the *nand* cell has only one CCB. The attribute *is_needed* is almost always true with the exception being that for non-functional cells such as load cells and antenna cells. The *stage_type* with value *both* specifies that this stage has both pull-up and pull-down structures. The *miller_cap_rise* and *miller_cap_fall* represent the Miller capacitances[1] for the rising and falling output transitions respectively.

DC Current

The *dc_current* tables represents the DC current at the output pin for different combinations of input and output pin voltages. The *index_1* specifies the input voltage and *index_2* specifies the output voltage. The *values* in the two-dimensional table specify the DC current at the CCB output. The input voltages and output currents are all specified in library units (normally *Volt* and *mA*). For the example CCS noise model from input *IN1* to *OUT* of the *nand* cell, an input voltage of -0.9V and output voltage of 0V, results in the DC current at the output of 0.42mA.

1. Miller capacitance accounts for the increase in the equivalent input capacitance of an inverting stage due to amplification of capacitance between the input and output terminals.

Output Voltage

The *output_voltage_rise* and *output_voltage_fall* constructs contain the timing information for the CCB output rising and falling respectively. These are specified as multi-dimensional tables for the CCB output node. The multi-dimensional tables are organized as multiple tables specifying the rising and falling output voltages for different input transition time and output net capacitances. Each table has *index_1* specifying the rail-to-rail input transition time rate and *index_2* specifying the output net capacitance. The *index_3* specifies the times when the output voltage crosses specific threshold points (in this case 30%, 70% and 90% of *Vdd* supply of 0.9V). In each multi-dimensional table, the voltage crossing points are fixed and the time-values when the CCB output node crosses the voltage is specified in *index_3*.

Propagated Noise

The *propagated_noise_high* and *propagated_noise_low* models specify multi-dimensional tables which provide noise propagation information through the CCB. These models characterize the crosstalk glitch (or noise) propagation from an input to the output of the CCB. The characterization uses symmetric triangular waveform at the input. The multi-dimensional tables for *propagated_noise* are organized as multiple tables specifying the glitch waveform at the output of the CCB. These multi-dimensional tables contain:

i. input glitch magnitude (in *index_1*),

ii. input glitch width (in *index_2*),

iii. CCB output net capacitance (in *index_3*), and

iv. time (in *index_4*).

The CCB output voltage (or the noise propagated through CCB) is specified in the table values.

Noise Models for Two-Stage Cells

Just like the single stage cells, the CCS noise models for two-stage cells such as *and* cells and *or* cells, are normally described as part of the timing arc. Since these cells contain two separate CCBs, the noise models are specified for *ccsn_first_stage* and another for *ccsn_last_stage* separately. For example, for a two-input *and* cell, the CCS noise model is comprised of separate models for the first stage and for the last stage. This is illustrated next.

```
pin (OUT) {
  . . .
  timing () {
    related_pin : "IN1";
    . . .
    ccsn_first_stage() {
      /* IN1 to internal node between stages */
      . . .
    }
    ccsn_last_stage() { /* Internal node to output */
      . . .
    }
    . . .
  }
  timing () {
    related_pin : "IN2";
    . . .
    ccsn_first_stage() {
      /* IN2 to internal node between stages */
      . . .
    }
    ccsn_last_stage() {
      /* Internal node to output */
      /* Same as from IN1 */
      . . .
    }
    . . .
```

```
    }
    . . .
  }
```

The model within the *ccsn_last_stage* specified for *IN2* is the same as the model in *ccsn_last_stage* described for *IN1*.

Noise Models for Multi-stage and Sequential Cells

The CCS noise models for complex combinational or sequential cells are normally described as part of the pin specification. This is different from the one-stage or two-stage cells such as *nand*, *nor*, *and*, *or* where the CCS noise models are normally specified on a pin-pair basis as part of the timing arc. Complex multi-stage and sequential cells are normally described by a *ccsn_first_stage* model for all input pins and another *ccsn_last_stage* model at the output pins. The CCS noise models for these cells are not part of the timing arc but are normally specified for a pin.

If the internal paths between inputs and outputs are up to two CCB stages, the noise models can also be represented as part of the pin-pair timing arcs. In general, a multi-stage cell description can have some CCS noise models specified as part of the pin-pair timing arc while some other noise models can be specified with the pin description.

An example below has the CCS noise models specified with the pin description as well as part of the timing arc.

```
pin (CDN) {
  . . .
}
pin (CP) {
  . . .
  ccsn_first_stage() {
    . . .
  }
}
pin (D) {
```

```
    . . .
    ccsn_first_stage() {
      . . .
    }
  }
  pin (Q) {
    . . .
     timing() {
      related_pin : "CDN";
      . . .
      ccsn_first_stage() {
        . . .
      }
      ccsn_last_stage() {
        . . .
      }
    }
  }
  pin (QN) {
    . . .
    ccsn_last_stage() {
      . . .
    }
  }
```

Note that some of the CCS models for the flip-flop cell above are defined with the pins. Those defined with the pin specification at input pins are designated as *ccsn_first_stage,* and the CCS model at the output pin *QN* is designated as *ccsn_last_stage*. In addition, two-stage CCS noise models are described as part of the timing arc for *CDN* to *Q*. This example thus shows that a cell can have the CCS models designated as part of pin specification and as part of the timing group.

3.7.4 Other Noise Models

Apart from the CCS noise models described above, some cell libraries can provide other models for characterizing noise. Some of these models were utilized before the advent of the CCS noise models. These additional model are not required if the CCS noise models are available. We describe below some of these earlier noise models for completeness.

Models for DC margin: The DC margin refers to the largest DC variation allowed at the input pin of the cell which would keep the cell in its steady condition, that is, without causing a glitch at the output. For example, DC margin for input low refers to the largest DC voltage value at the input pin without causing any transition at the output.

Models for noise immunity: The noise immunity models specify the glitch magnitude that can be allowed at an input pin. These are normally described in terms of two-dimensional tables with the glitch width and output capacitance as the two indices. The values in the table correspond to the glitch magnitude that can be allowed at the input pin. This means that any glitch smaller than the specified magnitude and width will not propagate through the cell. Different variations of the noise immunity models can be specified such as:

i. *noise_immunity_high*

ii. *noise_immunity_low*

iii. *noise_immunity_above_high* (overshoot)

iv. *noise_immunity_below_low* (undershoot).

3.8 Power Dissipation Modeling

The cell library contains information related to power dissipation in the cells. This includes active power as well as standby or leakage power. As the names imply, the active power is related to the activity in the design whereas the standby power is the power dissipated in the standby mode, which is mainly due to leakage.

3.8.1 Active Power

The active power is related to the activity at the input and output pin of the cell. The active power in the cell is due to charging of the output load as well as internal switching. These two are normally referred to as *output switching power* and *internal switching power* respectively.

The output switching power is independent of the cell type and depends only upon the output capacitive load, frequency of switching and the power supply of the cell. The internal switching power depends upon the type of the cell and this value is thus included in the cell library. The specification of the internal switching power in the library is described next.

The internal switching power is referred to as *internal power* in the cell library. This is the power consumption within the cell when there is activity at the input or the output of the cell. For a combinational cell, an input pin transition can cause the output to switch and this results in internal switching power. For example, an inverter cell consumes power whenever the input switches (has a rise or fall transition at the input). The internal power is described in the library as:

```
pin (Z1) {
  . . .
  power_down_function : "!VDD + VSS";
  related_power_pin : VDD;
  related_ground_pin : VSS;
  internal_power () {
    related_pin : "A";
```

```
    power (template_2x2) {
     index_1 ("0.1, 0.4"); /* Input transition */
     index_2 ("0.05, 0.1"); /* Output capacitance */
     values ( /*   0.05    0.1 */ \
       /* 0.1 */   "0.045,  0.050", \
       /* 0.4 */   "0.055,  0.056");
    }
   }
  }
```

The example above shows the power dissipation from input pin *A* to the output pin *Z1* of the cell. The 2x2 table in the template is in terms of input transition at pin *A* and the output capacitance at pin *Z1*. Note that while the table includes the output capacitance, the table values only correspond to the internal switching and do not include the contribution of the output capacitance. The values represent the internal energy dissipated in the cell for each switching transition (rise or fall). The units are as derived from other units in the library (typically voltage is in volts (V) and capacitance is in picofarads (pF), and this maps to energy in picojoules (pJ)). The internal power in the library thus actually specifies the internal energy dissipated per transition.

In addition to the power tables, the example above also illustrates the specification of the power pins, ground pins, and the power down function which specifies the condition when the cell can be powered off. These constructs allow for multiple power supplies in the design and scenarios where different supplies may be powered down. The following illustration shows the power pin specification for each cell.

```
cell (NAND2) {
  . . .
  pg_pin (VDD) {
    pg_type : primary_power;
    voltage_name : COREVDD1;
    . . .
  }
```

```
    pg_pin (VSS) {
      pg_type : primary_ground;
      voltage_name : COREGND1;
      . . .
    }
  }
```

The power specification syntax allows for separate constructs for rise and fall power (referring to the output sense). Just like the timing arcs, the power specification can also be state-dependent. For example, the state-dependent power dissipation for an *xor* cell can be specified as dependent on the state of various inputs.

For combinational cells, the switching power is specified on an input-output pin pair basis. However for a sequential cell such as a flip-flop with complementary outputs, *Q* and *QN*, the *CLK*->*Q* transition also results in a *CLK*->*QN* transition. Thus, the library can specify the internal switching power as a three-dimensional table, which is shown next. The three dimensions in the example below are the input slew at *CLK* and the output capacitances at *Q* and *QN* respectively.

```
pin (Q) {
  . . .
  internal_power() {
  related_pin : "CLK";
  equal_or_opposite_output : "QN";
  rise_power(energy_template_3x2x2) {
    index_1 ("0.02, 0.2, 1.0"); /* Clock transition */
    index_2 ("0.005, 0.2"); /* Output Q capacitance */
    index_3 ("0.005, 0.2"); /* Output QN capacitance */
    values ( /*  0.005   0.2 */ /*   0.005    0.2 */ \
      /* 0.02 */ "0.060,  0.070",   "0.061,   0.068", \
      /* 0.2 */  "0.061,  0.071",   "0.063,   0.069", \
      /* 1.0 */  "0.062,  0.080",   "0.068,   0.075");
  }
  fall_power(energy_template_3x2x2) {
    index_1 ("0.02, 0.2, 1.0");
```

```
    index_2 ("0.005, 0.2");
    index_3 ("0.005, 0.2");
    values ( \
     "0.070, 0.080", "0.071, 0.078", \
     "0.071, 0.081", "0.073, 0.079", \
     "0.066, 0.082", "0.068, 0.085");
  }
}
```

Switching power can be dissipated even when the outputs or the internal state does not have a transition. A common example is the clock that toggles at the clock pin of a flip-flop. The flip-flop dissipates power with each clock toggle - typically due to switching of an inverter inside of the flip-flop cell. The power due to clock pin toggle is dissipated even if the flip-flop output does not switch. Thus for sequential cells, the input pin power refers to the power dissipation internal to the cell, that is, when the outputs do not transition. An example of the input pin power specification follows.

```
cell (DFF) {
  . . .
  pin (CLK) {
    . . .
    rise_power () {
     power (template_3x1) {
       index_1 ("0.1, 0.25, 0.4"); /* Input transition */
       values ( /*    0.1     0.25     0.4 */ \
                   "0.045,  0.050,   0.090");
     }
    }
    fall_power () {
     power (template_3x1) {
       index_1 ("0.1, 0.25, 0.4");
       values ( \
         "0.045, 0.050, 0.090");
     }
    }
  }
```

```
    . . .
}
```

This example shows the power specification when the *CLK* pin toggles. This represents the power dissipation due to clock switching even when the output does not switch.

Double Counting Clock Pin Power?

Note that a flip-flop also contains the power dissipation due to *CLK*->*Q* transition. It is thus important that the values in the *CLK*->*Q* power specification tables do not include the contribution due to the *CLK* internal power corresponding to the condition when the output *Q* does not switch.

The above guideline refers to consistency of usage of power tables by the application tool and ensures that the internal power specified due to clock input is not double-counted during power calculation.

3.8.2 Leakage Power

Most standard cells are designed such that the power is dissipated only when the output or state changes. Any power dissipated when the cell is powered but there is no activity is due to non-zero leakage current. The leakage can be due to subthreshold current for MOS devices or due to tunneling current through the gate oxide. In the earlier generations of CMOS process technologies, the leakage power has been negligible and has not been a major consideration during the design process. However, as the technology shrinks, the leakage power is becoming significant and is no longer negligible in comparison to active power.

As described above, the leakage power contribution is from two phenomena: subthreshold current in the MOS device and gate oxide tunneling. By using high Vt cells[1], one can reduce the subthreshold current; however,

1. The high Vt cells refer to cells with higher threshold voltage than the standard for the process technology.

there is a trade-off due to reduced speed of the high Vt cells. The high Vt cells have smaller leakage but are slower in speed. Similarly, the low Vt cells have larger leakage but allow greater speed. The contribution due to gate oxide tunneling does not change significantly by switching to high (or low) Vt cells. Thus, a possible way to control the leakage power is to utilize high Vt cells. Similar to the selection between high Vt and standard Vt cells, the strength of cells used in the design is a trade-off between leakage and speed. The higher strength cells have higher leakage power but provide higher speed. The trade-off related to power management are described in detail in Section 10.6.

The subthreshold MOS leakage has a strong non-linear dependence with respect to temperature. In most process technologies, the subthreshold leakage can grow by 10x to 20x as the device junction temperature is increased from 25C to 125C. The contribution due to gate oxide tunneling is relatively invariant with respect to temperature or the Vt of the devices. The gate oxide tunneling which was negligible at process technologies 100nm and above, has become a significant contributor to leakage at lower temperatures for 65nm or finer technologies. For example, gate oxide tunneling leakage may equal the subthreshold leakage at room temperature for 65nm or finer process technologies. At high temperatures, the subthreshold leakage continues to be the dominant contributor to leakage power.

Leakage power is specified for each cell in the library. For example, an inverter cell may contain the following specification:

```
cell_leakage_power : 1.366;
```

This is the leakage power dissipated in the cell - the leakage power units are as specified in the header of the library, typically in nanowatts. In general, the leakage power depends upon the state of the cell and state dependent values can be specified using the *when* condition.

For example, an *INV1* cell can have the following specification:

```
cell_leakage_power : 0.70;
leakage_power() {
  when : "!I";
  value : 1.17;
}

leakage_power() {
  when : "I";
  value : 0.23;
}
```

where *I* is the input pin of the *INV1* cell. It should be noted that the specification includes a default value (outside of the *when* conditions) and that the default value is generally the average of the leakage values specified within the *when* conditions.

3.9 Other Attributes in Cell Library

In addition to the timing information, a cell description in the library specifies area, functionality and the SDF condition of the timing arcs. These are briefly described in this section; for more details the reader is referred to the Liberty manual.

Area Specification

The **area** specification provides the area of a cell or cell group.

```
area : 2.35;
```

The above specifies that the area of the cell is 2.35 area units. This can represent the actual silicon area used by the cell or it can be a relative measure of the area.

Function Specification

The **function** specification specifies the functionality of a pin (or pin group).

```
pin (Z) {
  function: "IN1 & IN2";
  . . .
}
```

The above specifies the functionality of the *Z* pin of a two-input *and* cell.

SDF Condition

The SDF condition attribute supports the Standard Delay Format (SDF) file generation and condition matching during backannotation. Just as the *when* specifies the condition for the state-dependent models for timing analysis, the corresponding specification for state-dependent timing usage for SDF annotation is denoted by *sdf_cond*.

This is illustrated by the following example:

```
timing() {
  related_pin : "A1";
  when : "!A2";
  sdf_cond : "A2 == 1'b0";
  timing_sense : positive_unate;
  cell_rise(delay_template_7x7) {
    . . .
  }
}
```

3.10 Characterization and Operating Conditions

A cell library specifies the characterization and operating conditions under which the library is created. For example, the header of the library may contain the following:

```
nom_process : 1;
nom_temperature : 0;
nom_voltage : 1.1;
voltage_map(COREVDD1, 1.1);
voltage_map(COREGND1, 0.0);
operating_conditions("BCCOM"){
  process : 1;
  temperature : 0;
  voltage : 1.1;
  tree_type : "balanced_tree";
}
```

The nominal environmental conditions (designated as *nom_process*, *nom_temperature* and *nom_voltage*) specify the process, voltage and temperature under which the library was characterized. The operating conditions specify the conditions under which the cells from this library will get used. If the characterization and operating conditions are different, the timing values obtained during delay calculation need to be derated; this is accomplished by using the derating factor (k-factors) specified in the library.

The usage of derating to obtain timing values at a condition other than what was used for characterization introduces inaccuracy in timing calculation. The derating procedure is employed only if it is not feasible to characterize the library at the condition of interest.

What is the Process Variable?

Unlike temperature and voltage which are physical quantities, the process is not a quantifiable quantity. It is likely to be one of *slow*, *typical* or *fast* processes for the purposes of digital characterization and verification. Thus,

what does the process value of 1.0 (or any other value) mean? The answer is given below.

The library characterization is a time-consuming process and characterizing the library for various process corners can take weeks. The process variable setting allows a library characterized at a specific process corner be used for timing calculation for a different process corner. The k-factors for process can be used to derate the delays from the characterized process to the target process. As mentioned above, the use of derating factors introduces inaccuracy during timing calculation. Derating across process conditions is especially inaccurate and is rarely employed. To summarize, the only function of specifying different process values (say 1.0 or any other) is to allow derating across conditions which is rarely (if ever) employed.

3.10.1 Derating using K-factors

As described, the derating factors (referred to as **k-factors**) are used to obtain delays when the operating conditions are different from the characterization conditions. The k-factors are approximate factors. An example of the k-factors in a library are as specified below:

```
/* k-factors */
k_process_cell_fall             : 1;
k_process_cell_leakage_power    : 0;
k_process_cell_rise             : 1;
k_process_fall_transition       : 1;
k_process_hold_fall             : 1;
k_process_hold_rise             : 1;
k_process_internal_power        : 0;
k_process_min_pulse_width_high  : 1;
k_process_min_pulse_width_low   : 1;
k_process_pin_cap               : 0;
k_process_recovery_fall         : 1;
k_process_recovery_rise         : 1;
k_process_rise_transition       : 1;
k_process_setup_fall            : 1;
k_process_setup_rise            : 1;
```

```
k_process_wire_cap              : 0;
k_process_wire_res              : 0;
k_temp_cell_fall                : 0.0012;
k_temp_cell_rise                : 0.0012;
k_temp_fall_transition          : 0;
k_temp_hold_fall                : 0.0012;
k_temp_hold_rise                : 0.0012;
k_temp_min_pulse_width_high     : 0.0012;
k_temp_min_pulse_width_low      : 0.0012;
k_temp_min_period               : 0.0012;
k_temp_rise_propagation         : 0.0012;
k_temp_fall_propagation         : 0.0012;
k_temp_recovery_fall            : 0.0012;
k_temp_recovery_rise            : 0.0012;
k_temp_rise_transition          : 0;
k_temp_setup_fall               : 0.0012;
k_temp_setup_rise               : 0.0012;
k_volt_cell_fall                : -0.42;
k_volt_cell_rise                : -0.42;
k_volt_fall_transition          : 0;
k_volt_hold_fall                : -0.42;
k_volt_hold_rise                : -0.42;
k_volt_min_pulse_width_high     : -0.42;
k_volt_min_pulse_width_low      : -0.42;
k_volt_min_period               : -0.42;
k_volt_rise_propagation         : -0.42;
k_volt_fall_propagation         : -0.42;
k_volt_recovery_fall            : -0.42;
k_volt_recovery_rise            : -0.42;
k_volt_rise_transition          : 0;
k_volt_setup_fall               : -0.42;
k_volt_setup_rise               : -0.42;
```

These factors are used to obtain timing when the process, voltage or temperature for the operating conditions during delay calculation are different from the nominal conditions in the library. Note that *k_volt* factors are negative, implying that the delays reduce with increasing voltage supply,

whereas the *k_temp* factors are positive, implying that the delays normally increase with increasing temperature (except for cells exhibiting temperature inversion phenomenon described in Section 2.10). The k-factors are used as follows:

```
Result with derating = Original_value *
  ( 1 + k_process * DELTA_Process
      + k_volt * DELTA_Volt
      + k_temp * DELTA_Temp)
```

For example, assume that a library is characterized at 1.08V and 125C with *slow* process models. If the delays are to be obtained for 1.14V and 100C, the cell rise delays for the *slow* process models can be obtained as:

```
Derated_delay = Library_delay *
  ( 1 + k_volt_cell_rise * 0.06
      - k_temp_cell_rise * 25)
```

Assuming the k_factors outlined above are used, the previous equation maps to:

```
Derated_delay = Library_delay * (1 - 0.42 * 0.06 - 0.0012 * 25)
              = Library_delay * 0.9448
```

The delay at the derated condition works out to about 94.48% of the original delay.

3.10.2 Library Units

A cell description has all values in terms of library units. The units are declared in the library file using the Liberty command set. Units for voltage, time, capacitance, and resistance are declared as shown in the following example:

```
library("my_cell_library") {
  voltage_unit : "1V";
```

```
    time_unit : "1ns";
    capacitive_load_unit (1.000000, pf);
    current_unit : 1mA;
    pulling_resistance_unit : "1kohm";
    . . .
  }
```

In this text, we shall assume that the library time units are in nanoseconds (ns), voltage is in volts (V), internal power is in picojoules (pJ) per transition, leakage power is in nanowatts (nW), capacitance values are in picofarad (pF), resistance values are in Kohms and area units are square micron (μm^2), unless explicitly specified to help with an explanation.

❑