SIMULATION

Engineers used to prototype systems to check their designs, often using a breadboard with connector holes, allowing them to plug in ICs and wires. Breadboarding was feasible when it was possible to construct systems from a few off-the-shelf TTL parts. It is impractical for prototyping an ASIC. Instead most ASIC design engineers turn to simulation as the modern equivalent of breadboarding.

13.1 Types of Simulation

13.2 The Comparator/MUX Example

13.3 Logic Systems

13.4 How Logic Simulation Works

13.5 Cell Models

13.6 Delay Models

13.7 Static Timing Analysis

13.8 Formal Verification

13.9 Switch-Level Simulation

13.10 Transistor-Level Simulation

13.11 Summary

13.12 Problems

13.13 Bibliography

13.14 References

13.1 Types of Simulation

Simulators are usually divided into the following categories or simulation modes :

- Behavioral simulation
- Functional simulation
- Static timing analysis
- Gate-level simulation
- Switch-level simulation
- Transistor-level or circuit-level simulation

This list is ordered from high-level to low-level simulation (high-level being more abstract, and low-level being more detailed). Proceeding from high-level to low-level simulation, the simulations become more accurate, but they also become progressively more complex and take longer to run. While it is just possible to perform a behavioral-level simulation of a complete system, it is impossible to perform a circuit-level simulation of more than a few hundred transistors.

There are several ways to create an imaginary simulation model of a system. One method models large pieces of a system as black boxes with inputs and outputs. This type of simulation (often using VHDL or Verilog) is called behavioral simulation . Functional simulation ignores timing and includes unit-delay simulation , which sets delays to a fixed value (for example, 1 ns). Once a behavioral or functional simulation predicts that a system works correctly, the next step is to check the timing performance. At this point a system is partitioned into ASICs and a timing simulation is performed for each ASIC separately (otherwise the simulation run times become too long). One class of timing simulators employs timing analysis that analyzes logic in a static manner, computing the delay times for each path. This is called static timing analysis because it does not require the creation of a set of test (or stimulus) vectors (an enormous job for a large ASIC). Timing analysis works best with synchronous systems whose maximum operating frequency is determined by the longest path delay between successive flip-flops. The path with the longest delay is the critical path .

Logic simulation or gate-level simulation can also be used to check the timing performance of an ASIC. In a gate-level simulator a logic gate or logic cell (NAND, NOR, and so on) is treated as a black box modeled by a function whose variables are the input signals. The function may also model the delay through the logic cell. Setting all the delays to unit value is the equivalent of functional simulation. If the timing simulation provided by a black-box model of a logic gate is not accurate enough, the next, more detailed, level of simulation is switch-level simulation which models transistors as switches-on or off. Switch-level simulation can provide more accurate timing predictions than gate-level simulation, but without the ability to use logic-cell delays as parameters of the models. The most accurate, but also the most complex and time-consuming, form of simulation is transistor-level simulation . A transistor-level simulator requires models of transistors, describing their nonlinear voltage and current characteristics.

Each type of simulation normally uses a different software tool. A mixed-mode simulator permits different parts of an ASIC simulation to use different simulation modes. For example, a critical part of an ASIC might be simulated at the transistor level while another part is simulated at the functional level.

Be careful not to confuse mixed-level simulation with a mixed analog/digital simulator, these are mixed-level simulators .

Simulation is used at many stages during ASIC design. Initial prelayout simulations include logic-cell delays but no interconnect delays. Estimates of capacitance may be included after completing logic synthesis, but only after physical design is it possible to perform an accurate postlayout simulation

13.2 The Comparator/MUX Example

As an example we borrow the model from Section 12.2, "A Comparator/MUX,"

```
// comp_mux.v

module comp_mux(a, b, outp); input [2:0] a, b; output [2:0] outp;

function [2:0] compare; input [2:0] ina, inb;

begin if (ina <= inb) compare = ina; else compare = inb; end

endfunction

assign outp = compare(a, b);

endmodule
```

We can use the following testbench to generate a sequence of input values (we call these input vectors) that test or exercise the behavioral model, comp_mux.v :

```
// testbench.v

module comp_mux_testbench;

integer i, j;

reg [2:0] x, y, smaller; wire [2:0] z;

always @(x) $display("t x y actual calculated");

initial $monitor("%4g",$time,,x,,y,,z,,,,,,,smaller);

initial $dumpvars; initial #1000 $finish;

initial

begin

for (i = 0; i <= 7; i = i + 1)
```

```
begin

for (j = 0; j <= 7; j = j + 1)

begin

x = i; y = j; smaller = (x <= y) ? x : y;

#1 if (z != smaller) $display("error");

end

end

end

comp_mux v_1 (x, y, z);

endmodule
```

The results from the behavioral simulation are as follows:

t x y actual calculated

0 0 0 0 0

1 0 1 0 0

... 60 lines omitted...

62 7 6 6 6

63 7 7 7 7

We included a delay of one Verilog time unit in line 15 of the testbench model (allowing time to progress), but we did not specify the units-they could be nanoseconds or days. Thus, behavioral simulation can only tell us if our design does not work; it cannot tell us that real hardware will work.

13.2.1 Structural Simulation

We use logic synthesis to produce a structural model from a behavioral model. The following comparator/MUX model is adapted from the example in Section 12.11 , " Performance-Driven Synthesis " (optimized for a 0.6 m m standard-cell library):

```
‘timescale 1ns / 10ps // comp_mux_o2.v
```

```
module comp_mux_o (a, b, outp);

input [2:0] a; input [2:0] b;

output [2:0] outp;

supply1 VDD; supply0 VSS;

mx21d1 b1_i1 (.i0(a[0]), .i1(b[0]), .s(b1_i6_zn), .z(outp[0]));

oa03d1 b1_i2 (.a1(b1_i9_zn), .a2(a[2]), .b1(a[0]), .b2(a[1]),

.c(b1_i4_zn), .zn(b1_i2_zn));

nd02d0 b1_i3 (.a1(a[1]), .a2(a[0]), .zn(b1_i3_zn));

nd02d0 b1_i4 (.a1(b[1]), .a2(b1_i3_zn), .zn(b1_i4_zn));

mx21d1 b1_i5 (.i0(a[1]), .i1(b[1]), .s(b1_i6_zn), .z(outp[1]));

oa04d1 b1_i6 (.a1(b[2]), .a2(b1_i7_zn), .b(b1_i2_zn),

.zn(b1_i6_zn));

in01d0 b1_i7 (.i(a[2]), .zn(b1_i7_zn));

an02d1 b1_i8 (.a1(b[2]), .a2(a[2]), .z(outp[2]));

in01d0 b1_i9 (.i(b[2]), .zn(b1_i9_zn));

endmodule
```

Logic simulation requires Verilog models for the following six logic cells: mx21d1 (2:1 MUX), oa03d1 (OAI221), nd02d0 (two-input NAND), oa04d1 (OAI21), in01d0 (inverter), and an02d1 (two-input AND). These models are part of an ASIC library (often encoded so that they cannot be seen) and thus, from this point on, the designer is dependent on a particular ASIC library company. As an example of this dependence, notice that some of the names in the preceding code have changed from uppercase (in Figure 12.8 on p. 624) to lowercase. Verilog is case sensitive and we are using a cell library that uses lowercase. Most unfortunately, there are no standards for names, cell functions, or the use of case in ASIC libraries.

The following code (a simplified model from a 0.8 m m standard-cell library) models a 2:1 MUX and uses fixed delays:

```
‘timescale 1 ns / 10 ps

module mx21d1 (z, i0, i1, s); input i0, i1, s; output z;
```

```
not G3(N3, s);

and G4(N4, i0, N3), G5(N5, s, i1), G6(N6, i0, i1);

or G7(z, N4, N5, N6);

specify

(i0*>z) = (0.279:0.504:0.900, 0.276:0.498:0.890);

(i1*>z) = (0.248:0.448:0.800, 0.264:0.476:0.850);

(s*>z)  = (0.285:0.515:0.920, 0.298:0.538:0.960);

endspecify

endmodule
```

This code uses Verilog primitive models (not , and , or) to describe the behavior of a MUX, but this is not how the logic cell is implemented.

To simulate the optimized structural model, module comp_mux_o2.v , we use the library cell models (module mx21d1 and the other five that are not shown here) together with the following new testbench model:

```
‘timescale 1 ps / 1 ps // comp_mux_testbench2.v

module comp_mux_testbench2;

integer i, j; integer error;

reg [2:0] x, y, smaller; wire [2:0] z, ref;

always @(x) $display("t x y derived reference");

// initial $monitor("%8.2f",$time/1e3,,x,,y,,z,,,,,,,,ref);

initial $dumpvars;

initial begin

error = 0; #1e6 $display("%4g", error, " errors");

$finish;

end

initial begin
```

```
for (i = 0; i <= 7; i = i + 1) begin

for (j = 0; j <= 7; j = j + 1) begin

x = i; y = j; #10e3;

$display("%8.2f",$time/1e3,,x,,y,,z,,,,,,,,ref);

if (z != ref)

begin $display("error"); error = error + 1; end

end

end

end

comp_mux_o v_1 (x, y, z); // comp_mux_o2.v

reference v_2 (x, y, ref);

endmodule

// reference.v

module reference(a, b, outp);

input [2:0] a, b; output [2:0] outp;

assign outp = (a <= b) ? a : b; // different from comp_mux

endmodule
```

In this testbench we have instantiated two models: a reference model (module reference) and a derived model (module comp_mux_o , the optimized structural model). The high-level behavioral model that represents the initial system specification (module reference) may be different from the model that we use as input to the logic-synthesis tool (module comp_mux). Which is the real reference model? We postpone this question until we discuss formal verification in Section 13.8 . For the moment, we shall simply perform simulations to check the reference model against the derived model. The simulation results are as follows:

```
t x y derived reference

10.00 0 0 0 0

20.00 0 1 0 0
```

... 60 lines omitted...

630.00 7 6 6 6

640.00 7 7 7 7

0 errors

(A summary is printed at the end of the simulation to catch any errors.) The next step is to examine the timing of the structural model (by switching the leading '//' from line 6 to 16 in module comp_mux_testbench2). It is important to simulate using the worst-case delays by using a command-line switch as follows: verilog +maxdelays . We can then find the longest path delay by searching through the simulator output, part of which follows:

t x y derived reference

... lines omitted...

260.00 3 2 1 2

260.80 3 2 3 2

260.85 3 2 2 2

270.00 3 3 2 3

270.80 3 3 3 3

280.00 3 4 3 3

280.85 3 4 0 3

283.17 3 4 3 3

... lines omitted...

At time 280 ns, the input vectors, x and y , switch from (x = 3 , y = 3) to (x = 3 , y = 4). The output of the derived model (which should be equal to the smaller of x and y) is the same for both of these input vectors and should remain unchanged. In fact there is a glitch at the output of the derived model, as it changes from 3 to 0 and back to 3 again, taking 3.17 ns to settle to its final value (this is the longest delay that occurs using this testbench). The glitch occurs because one of the input vectors (input y) changes from '011' (3 in decimal) to '100' (decimal 4). Changing several input bits simultaneously causes the output to vacillate.

Notice that the nominal and worst-case simulations will not necessarily give the same longest path delay. In addition the longest path delay found using this testbench is not necessarily the critical path delay. For example, the longest, and therefore critical, path delay might result from a transition from x = 3 , y = 4 to x = 4 , y = 3 (to choose a random but possible candidate set of input vectors). This testbench

does not include tests with such transitions. To find the critical path using logic simulation requires simulating all possible input transitions (64 ¥ 64 = 4096) and then sifting through the output to find the critical path.

Vector-based simulation (or dynamic simulation) can show us that our design functions correctly-hence the name functional simulation. However, functional simulation does not work well if we wish to find the critical path. For this we turn to a different type of simulation-static simulation or static timing analysis.

TABLE 13.1 Timing analysis of the comparator/MUX structural model, comp_mux_o2.v , from Figure 12.8 .

Command	Timing analyzer/logic synthesizer output 1 1
	instance name
	inPin --> outPin incr arrival trs rampDel cap cell
	(ns) (ns) (ns) (pf)
	---
	a[0] .00 .00 R .00 .12 comp_m...
	b1_i3
	A2 --> ZN .31 .31 F .23 .08 nd02d0
	b1_i4
> report timing	
	A2 --> ZN .41 .72 R .26 .07 nd02d0
	b1_i2
	C --> ZN 1.36 2.08 F .13 .07 oa03d1
	b1_i6
	B --> ZN .94 3.01 R .24 .14 oa04d1
	b1_i5
	S --> Z 1.04 4.06 F .08 .04 mx21d1
	outp[0] .00 4.06 F .00 .00 comp_m...

13.2.2 Static Timing Analysis

A timing analyzer answers the question: "What is the longest delay in my circuit?" Table 13.1 shows the

timing analysis of the comparator/MUX structural model, module comp_mux_o2.v . The longest or critical path delay is 4.06 ns under the following worst-case operating conditions: worst-case process, V_{DD} = 4.75 V, and T = 70 ? C (the same conditions as used for the library data book delay values). The timing analyzer gives us only the critical path and its delay. A timing analyzer does not give us the input vectors that will activate the critical path. In fact input vectors may not exist to activate the critical path. For example, it may be that the decimal values of the input vectors to the comparator/MUX may never differ by more than four, but the timing-analysis tool cannot use this information. Future timing-analysis tools may consider such factors, called Boolean relations , but at present they do not.

Section 13.2.1 explained why dynamic functional simulation does not necessarily find the critical path delay. Nevertheless, the difference between the longest path delay found using functional simulation, 3.17 ns, and the critical path delay reported by the static timing-analysis tool, 4.06 ns, is surprising. This difference occurs because the timing analysis accounts for the loading of each logic cell by the input capacitance of the logic cells that follow, but the simplified Verilog models used for functional simulation in Section 13.2.1 did not include the effects of capacitive loading. For example, in the model for the logic cell mx21d1 , the (rising) delay from the i0 input to the output z , was fixed at 0.900 ns worst case (the maximum delay value is the third number in the first triplet in line 7 of module mx21d1). Normally library models include another portion that adjusts the timing of each logic cell-this portion was removed to simplify the model mx21d1 shown in Section 13.2.1 .

Most timing analyzers do not consider the function of the logic when they search for the critical path. Thus, for example, the following code models z = NAND(a, NOT(a)) , which means that the output, z , is always ’1’ .

```
module check_critical_path_1 (a, z);

input a; output z; supply1 VDD; supply0 VSS;

nd02d0 b1_i3 (.a1(a), .a2(b), .zn(z)); // 2-input NAND

in01d0 b1_i7 (.i(a), .zn(b)); // inverter

endmodule
```

A timing-analyzer report for this model might show the following critical path:

```
inPin --> outPin incr arrival trs rampDel cap cell

                (ns) (ns) (ns) (pf)

-------------------------------------------------------------------

a .00 .00 R .00 .08    check_...

b1_i7

I --> ZN .38 .38 F .30 .07 in01d0
```

b1_i3

A2 --> ZN .28 .66 R .13 .04 nd02d0

z .00 .66 R .00 .00 check_...

Paths such as this, which are impossible to activate, are known as false paths . Timing analysis is essential to ASIC design but has limitations. A timing-analysis tool is more logic calculator than logic simulator.

13.2.3 Gate-Level Simulation

To illustrate the differences between functional simulation, timing analysis, and gate-level simulation, we shall simulate the comparator/MUX critical path (the path is shown in Table 13.1). We start by trying to find vectors that activate this critical path by working forward from the beginning of the critical path, the input a[0] , toward the end of the critical path, output outp[0] , as follows:

1. Input a[0] to the two-input NAND, nd02d0 , cell instance b1_i3 , changes from a ’0’ to a ’1’ . We know this because there is an ’R’ (for rising) under the trs (for transition) heading on the first line of the critical path timing analysis report in Table 13.1 .
2. Input a[1] to the two-input NAND, nd02d0 , cell instance b1_i3 , must be a ’1’ . This allows the change on a[0] to propagate toward the output, outp[0] .
3. Similarly, input b[1] to the two-input NAND, cell instance b1_i4 , must be a ’1’ .
4. We skip over the required inputs to cells b1_i2 and b1_i6 for the moment.
5. From the last line of Table 13.1 we know the output of MUX, mx21d1 , cell instance b1_i5 , changes from ’1’ to a ’0’ . From the previous line in Table 13.1 we know that the select input of this MUX changes from ’0’ to a ’1’ . This means that the final value of input b[0] (the i1 input, selected when the select input is ’1’) must be ’0’ (since this is the final value that must appear at the MUX output). Similarly, the initial value of a[0] must be a ’1’ .

We have now contradicted ourselves. In step 1 we saw that the initial value of a[0] must be a ’0’ . The critical path is thus a false path. Nevertheless we shall proceed. We set the initial input vector to (a = ’110’ , b = ’111’) and then to (a = ’111’ , b = ’110’). These vectors allow the change on a[0] to propagate to the select signal of the MUX, mx21d1 , cell instance b1_i5 . In decimal we are changing a from 6 to 7, and b from 7 to 6; the output should remain unchanged at 6. The simulation results from the gate-level simulator we shall use (CompassSim) can be displayed graphically or in the text form that follows:

...

The calibration was done at Vdd=4.65V, Vss=0.1V, T=70 degrees C

Time = 0:0 [0 ns]

a = ’D6 [0] (input)(display)

b = ’D7 [0] (input)(display)

outp = ’Buuu (’Du) [0] (display)

outp --> ’B1uu (’Du) [.47]

outp --> ’B11u (’Du) [.97]

outp --> ’D6 [4.08]

a --> ’D7 [10]

b --> ’D6 [10]

outp --> ’D7 [10.97]

outp --> ’D6 [14.15]

Time = 0:0 +20ns [20 ns]

The code ’Buuu denotes that the output is initially, at t = 0 ns, a binary vector of three unknown or unsettled signals. The output bits become valid as follows: outp[2] at 0.47 ns, outp[1] at 0.97 ns, and outp[0] at 4.08 ns. The output is stable at ’D6 (decimal 6) or ’110’ at t = 10 ns when the input vectors are changed in an attempt to activate the critical path. The output glitches from ’D6 (’110’) to ’D7 (’111’) at t = 10.97 ns and back to ’D6 again at t = 14.15 ns. Thus, the output bit, outp[0] , takes a total of 4.15 ns to settle.

Can we explain this behavior? The data book entry for the mx21d1 logic cell gives the following equation for the rising delay as a function of Cld (the load capacitance, excluding the output capacitance of the logic cell itself, expressed in picofarads):

tI0Z (IO->Z) = 0.90 + 0.07 + (1.76 ¥ Cld) ns (13.1)

tI0Z (IO->Z) = 0.90 + 0.07 + (1.76 ¥ Cld) ns (13.1)

The capacitance, Cld , at the output of each MUX is zero (because nothing is connected to the outputs). From Eq. 13.1 , the path delay from the input, a[0] , to the output, outp[0] , is thus 0.97 ns. This explains why the output, outp[0] , changes from ’0’ to ’1’ at t = 10.97 ns, 0.97 ns after a change occurs on a[0] .

The gate-level simulation predicts that the input, a[0] , to the MUX will change before the changes on the inputs have time to propagate to the MUX select. Finally, at t = 14.15 ns, the MUX select will change and switch the output, outp[0] , back to ’0’ again. The total delay for this input vector stimulus is thus 4.15 ns. Even though this path is a false path (as far as timing analysis is concerned), it is a critical path. It is indeed necessary to wait for 4.15 ns before using the output signal of this circuit. A timing analyzer can only offer us a guarantee that there is no other path that is slower than the critical path.

13.2.4 Net Capacitance

The timing analyzer predicted a critical path delay of 4.06 ns compared to the gate-level simulation prediction of 4.15 ns. We can check our results by using another gate-level simulator (QSim) which uses a slightly different algorithm. Here is the output (with the same input vectors as before):

```
@nodes

a R10 W1; a[2] a[1] a[0]

b R10 W1; b[2] b[1] b[0]

outp R10 W1; outp[2] outp[1] outp[0]

@data

.00 a -> ’D6

.00 b -> ’D7

.00 outp -> ’Du

.53 outp -> ’Du

.93 outp -> ’Du

4.42 outp -> ’D6

10.00 a -> ’D7

10.00 b -> ’D6

11.03 outp -> ’D7

14.43 outp -> ’D6

### END OF SIMULATION TIME = 20 ns

@end
```

The output is similar but gives yet another value, 4.43 ns, for the path delay. Can this be explained? The simulator prints the following messages as a clue:

defCapacitance = .1E-01 pF

incCapacitance = .1E-01 pF/pin

The simulator is adding capacitance to the outputs of each of the logic cells to model the parasitic net capacitance (interconnect capacitance or wire capacitance) that will be present in the physical layout. The simulator adds 0.01 pF (defCapacitance) on each node and another 0.01 pF (incCapacitance) for

each pin (logic cell input) attached to a node. The model that predicts these values is known as a wire-load model , wire-delay model , or interconnect model . Changing the wire-load model parameters to zero and repeating the simulation changes the critical-path delay to 4.06 ns, which agrees exactly with the logic-synthesizer timing analysis. This emphasizes that the net capacitance may contribute a significant delay.

The library data book (VLSI Technology, vsc450) lists the cell input and output capacitances. For example, the values for the nd02d0 logic cell are as follows:

Cin (inputs, a1 and a2) = 0.042 pF Cout (output, zn) = 0.038 pF (13.2)

Cin (inputs, a1 and a2) = 0.042 pF Cout (output, zn) = 0.038 pF
(13.2)

Armed with this information, let us return to the timing analysis report of Table 13.1 (the part of this table we shall focus on follows) and examine how a timing analyzer handles net capacitance.

inPin --> outPin incr arrival trs rampDel cap cell

(ns) (ns) (ns) (pf)

---

a[0] .00 .00 R .00 .12 comp_m...

b1_i3

A2 --> ZN .31 .31 F .23 .08 nd02d0

...

The total capacitance at the output node of logic cell instance b1_i3 is 0.08 pF. This figure is the sum of the logic cell (nd02d0) output capacitance of cell instance b1_i3 (equal to 0.038 pF) and Cld , the input capacitance of the next cell, b1_i2 (also an nd02d0), equal to 0.042 pF.

The capacitance at the input node, a[0] , is equal to the sum of the input capacitances of the logic cells connected to that node. These capacitances (and their sources) are as follows:

1. 0.042 pF (the a2 input of the two-input NAND, instance b1_i3 , cell nd02d0)
2. 0.038 pF (the i0 input of the 2:1 MUX, instance b1_i1 , cell mx21d1)
3. 0.038 pF (the b1 input of the OAI221, instance b1_i2 , cell oa03d1)

The sum of these capacitances is the 0.12 pF shown in the timing-analysis report.

Having explained the capacitance figures in the timing-analysis report, let us turn to the delay figures. The fall-time delay equation for a nd02d0 logic cell (again from the vsc450 library data book) is as follows:

tD (AX->ZN) = 0.08 + 0.11 + (2.89 ¥ Cld) ns (13.3)

tD (AX->ZN) = 0.08 + 0.11 + (2.89 ¥ Cld) ns (13.3)

Notice 0.11 ns = 2.89 nspF-1 ¥ 0.038 pF, and this figure in Eq. 13.3 is the part of the cell delay attributed to the cell output capacitance. The ramp delay in the timing analysis (under the heading rampDel in Table 13.1) is the sum of the last two terms in Eq. 13.3 . Thus, the ramp delay is 0.11 + (2.89 ¥ 0.042) = 0.231 ns (since Cld is 0.042 pF). The total delay (under incr in Table 13.1) is 0.08 + 0.231 = 0.31 ns.

There are thus the following four figures for the critical path delay:

1. 4.06 ns from a static timing analysis using the logic-synthesizer timing engine (worst-case process, V_{DD} = 4.50 V, and T = 70 ? C). No wire capacitance.
2. 4.15 ns from a gate-level functional simulation (worst-case process, V_{SS} = 0.1 V, V_{DD} = 4.65 V, and T = 70 ? C). No wire capacitance.
3. 4.43 ns from a gate-level functional simulation. Default wire-capacitance model (0.01 pF + 0.01 pF / pin).
4. 4.06 ns from a gate-level functional simulation. No wire capacitance.

Normally we do not check our simulation results this thoroughly. However, we can only trust the tools if we understand what they are doing, how they work, their limitations, and we are able to check that the results are reasonable.

---

1. 1Using a 0.8 m m standard-cell library, VLSI Technology vsc450. Worst-case environment: worst-case process, V_{DD} = 4.75 V, and T = 70 °C. No wire capacitance, no input or output capacitance, prop-ramp timing model. The structural model was synthesized and optimized using a 0.6 m m library, but this timing analysis was performed using the 0.8 m m library. This is because the library models are simpler for the 0.8 m m library and thus easier to explain in the text.

13.3 Logic Systems

Digital signals are actually analog voltage (or current) levels that vary continuously as they change. Digital simulation assumes that digital signals may only take on a set of logic values (or logic states -here we will consider the two terms equivalent) from a logic system . A logic system must be chosen carefully. Too many values will make the simulation complicated and slow. With too few values the simulation may not accurately reflect the hardware performance.

A two-value logic system (or two-state logic system) has a logic value ’0’ corresponding to a logic level ’zero’ and a logic value ’1’ corresponding to a logic level ’one’. However, when the power to a system is initially turned on, we do not immediately know whether the logic value of a flip-flop output is ’1’ or ’0’ (it will be one or the other, but we do not know which). To model this situation we introduce a logic value ’X’ , with an unknown logic level, or unknown . An unknown can propagate through a circuit. For example, if the inputs to a two-input NAND gate are logic values ’1’ and ’X’ , the output is logic value

’X’ or unknown. Next, in order to model a three-state bus, we need a high-impedance state . A high-impedance state may have a logic level of ’zero’ or ’one’, but it is not being driven-we say it is floating. This will occur if none of the gates connected to a three-state bus is driving the bus. A four-value logic system is shown in Table 13.2 .

TABLE 13.2 A four-value logic system.

Logic state	Logic level	Logic value
0	zero	zero
1	one	one
X	zero or one	unknown
Z	zero, one, or neither	high impedance

13.3.1 Signal Resolution

What happens if multiple drivers try to drive different logic values onto a bus? Table 13.3 shows a signal-resolution function for a four-value logic system that will predict the result.

TABLE 13.3 A resolution function R {A, B} that predicts the result of two drivers simultaneously attempting to drive signals with values A and B onto a bus.

R {A, B}	B = 0	B = 1	B = X	B = Z
A = 0	0	X	X	0
A = 1	X	1	X	1
A = X	X	X	X	X
A = Z	0	1	X	Z

A resolution function, R {A, B}, must be commutative and associative . That is,

R {A, B} = R {B, A} and R {R {A, B}, C} = R {A, R {B, C}}. (13.4)

R {A, B} = R {B, A} and R {R {A, B}, C} = R {A, R {B, C}}.(13.4)

Equation 13.4 ensures that, if we have three (or more) signals to resolve, it does not matter in which order we resolve them. Suppose we have four drivers on a bus driving values ’0’ , ’1’ , ’X’ , and ’Z’ . If we use Table 13.3 three times to resolve these signals, the answer is always ’X’ whatever order we use.

13.3.2 Logic Strength

In CMOS logic we use n -channel transistors to produce a logic level ’zero’ (with a forcing strength) and we use p -channel transistors to force a logic level ’one’. An n -channel transistor provides a weak logic level ’one’. This is a new logic value, a resistive ’one’ , which has a logic level of ’one’, but with resistive strength . Similarly, a p -channel transistor produces a resistive ’zero’ . A resistive strength is not as strong as a forcing strength. At a high-impedance node there is nothing to keep the node at any

logic level. We say that the logic strength is high impedance . A high-impedance strength is the weakest strength and we can treat it as either a very high-resistance connection to a power supply or no connection at all.

TABLE 13.4 A 12-state logic system.

	Logic level		
Logic strength	zero	unknown	one
strong	S0	SX	S1
weak	W0	WX	W1
high impedance	Z0	ZX	Z1
unknown	U0	UX	U1

With the introduction of logic strength, a logic value may now have two properties: level and strength. Suppose we were to measure a voltage at a node N with a digital voltmeter (with a very high input impedance). Suppose the measured voltage at node N was 4.98 V (and the measured positive supply, V_{DD} = 5.00 V). We can say that node N is a logic level ’one’, but we do not know the logic strength. Now suppose you connect one end of a 1 k W resistor to node N , the other to GND, and the voltage at N changes to 4.95 V. Now we can say that whatever is driving node N has a strong forcing strength. In fact, we know that whatever is driving N is capable of supplying a current of at least 4.95 V / 1 k W ? 5 mA. Depending on the logic-value system we are using, we can assign a logic value to N . If we allow all possible combinations of logic level with logic strength, we end up with a matrix of logic values and logic states. Table 13.4 shows the 12 states that result with three logic levels (zero, one, unknown) and four logic strengths (strong, weak, high-impedance, and unknown). In this logic system, node N has logic value S1 -a logic level of ’one’ with a logic strength of ’strong’.

The Verilog logic system has three logic levels that are called ’1’ , ’0’ , and ’x’ ; and the eight logic strengths shown in Table 13.5 . The designer does not normally see the logic values that result-only the three logic levels.

TABLE 13.5 Verilog logic strengths.

Logic strength	Strength number	Models	Abbreviation	
supply drive	7	power supply	supply	Su
strong drive	6	default gate and assign output strength	strong	St
pull drive	5	gate and assign output strength	pull	Pu
large capacitor	4	size of trireg net capacitor	large	La
weak drive	3	gate and assign output strength	weak	We
medium capacitor	2	size of trireg net capacitor	medium	Me
small capacitor	1	size of trireg net capacitor	small	Sm
high impedance	0	not applicable	highz	Hi

The IEEE Std 1164-1993 logic system defines a variable type, std_ulogic , with the nine logic values shown in Table 13.6 . When we wish to simulate logic cells using this logic system, we must define the

primitive-gate operations. We also need to define the process of VHDL signal resolution using VHDL signal-resolution functions . For example, the function in the IEEE Std_Logic_1164 package that defines the and operation is as follows 1 :

TABLE 13.6 The nine-value logic system, IEEE Std 1164-1993.

Logic state	Logic value	Logic state	Logic value
’0’	strong low	’X’	strong unknown
’1’	strong high	’W’	weak unknown
’L’	weak low	’Z’	high impedance
’H’	weak high	’-’	don’t care
		’U’	uninitialized

```
function "and"(l,r : std_ulogic_vector) return std_ulogic_vector is

alias lv : std_ulogic_vector (1 to l’LENGTH ) is l;

alias rv : std_ulogic_vector (1 to r’LENGTH ) is r;

variable result : std_ulogic_vector (1 to l’LENGTH );

constant and_table : stdlogic_table := (

------------------------------------------------------------

--| U X 0 1 Z W L H - | |

------------------------------------------------------------

  ( ’U’, ’U’, ’0’, ’U’, ’U’, ’U’, ’0’, ’U’, ’U’ ), -- | U |

  ( ’U’, ’X’, ’0’, ’X’, ’X’, ’X’, ’0’, ’X’, ’X’ ), -- | X |

  ( ’0’, ’0’, ’0’, ’0’, ’0’, ’0’, ’0’, ’U’, ’0’ ), -- | 0 |

  ( ’U’, ’X’, ’0’, ’1’, ’X’, ’X’, ’0’, ’1’, ’X’ ), -- | 1 |

  ( ’U’, ’X’, ’0’, ’X’, ’X’, ’X’, ’0’, ’X’, ’X’ ), -- | Z |

  ( ’U’, ’X’, ’0’, ’X’, ’X’, ’X’, ’0’, ’X’, ’X’ ), -- | W |

  ( ’0’, ’0’, ’0’, ’0’, ’0’, ’0’, ’0’, ’0’, ’0’ ), -- | L |

  ( ’U’, ’X’, ’0’, ’1’, ’X’, ’X’, ’0’, ’1’, ’X’ ), -- | H |

  ( ’U’, ’X’, ’0’, ’X’, ’X’, ’X’, ’0’, ’X’, ’X’ ), -- | - |);
```

```
begin

if (l'LENGTH /= r'LENGTH) then assert false report

"arguments of overloaded 'and' operator are not of the same

length"

severity failure;

else

for i in result'RANGE loop

result(i) := and_table ( lv(i), rv(i) );

end loop;

end if;

return result;

end "and";
```

If a = 'X' and b = '0' , then (a and b) is '0' no matter whether a is, in fact, '0' or '1' .

---

1. IEEE Std 1164-1993, © Copyright 1993 IEEE. All rights reserved.

13.4 How Logic Simulation Works

The most common type of digital simulator is an event-driven simulator . When a circuit node changes in value the time, the node, and the new value are collectively known as an event . The event is scheduled by putting it in an event queue or event list . When the specified time is reached, the logic value of the node is changed. The change affects logic cells that have this node as an input. All of the affected logic cells must be evaluated , which may add more events to the event list. The simulator keeps track of the current time, the current time step , and the event list that holds future events. For each circuit node the simulator keeps a record of the logic state and the strength of the source or sources driving the node. When a node changes logic state, whether as an input or as an output of a logic cell, this causes an event.

An interpreted-code simulator uses the HDL model as data, compiling an executable model as part of the simulator structure, and then executes the model. This type of simulator usually has a short compile time but a longer execution time compared to other types of simulator. An example is Verilog-XL. A compiled-code simulator converts the HDL model to an intermediate form (usually C) and then uses a separate compiler to create executable binary code (an executable). This results in a longer compile time

but shorter execution time than an interpreted-code simulator. A native-code simulator converts the HDL directly to an executable and offers the shortest execution time.

The logic cells for each of these types of event-driven simulator are modeled using a primitive modeling language (primitive in the sense of "fundamental"). There are no standards for this primitive modeling language. For example, the following code is a primitive model of a two-input NAND logic cell:

```
model nd01d1 (a, b, zn)

function (a, b) !(a & b); function end

model end
```

The model has three ports: a , b , and zn . These ports are connected to nodes when a NAND gate is instantiated in an input structural netlist,

```
nand nd01d1(a2, b3, r7)
```

An event occurs when one of the circuit nodes a2 or b3 changes, and the function defined in the primitive model is called. For example, when a2 changes, it affects the port a of the model. The function will be called to set zn to the logical NAND of a and b . The implementation of the primitive functions is unique to each simulator and carefully coded to reduce execution time.

The data associated with an event consists of the affected node, a new logic value for the node, a time for the change to take effect, and the node that caused the event. Written in C, the data structure for an event might look like the following:

```
struct Event {

event_ptr fwd_link, back_link; /* event list */

event_ptr node_link; /* list of node events */

node_ptr event_node; /* node for the event */

node_ptr cause; /* node causing event */

port_ptr port; /* port which caused this event */

long event_time; /* event time, in units of delta */

char new_value; /* new value: ’1’ ’0’ etc. */

};
```

The event list keeps track of logic cells whose outputs are changing and the new values for each output. The evaluation list keeps track of logic cells whose inputs have changed. Using separate event and evaluation lists avoids any dependence on the order in which events are processed, since the evaluations

occur only after all nodes have been updated. The sequence of event-list processing followed by the evaluation-list processing is called a simulation cycle , or an event-evaluation cycle (or event-eval cycle for short).

Delays are tracked using a time wheel divided into ticks or slots, with each slot representing a unit of time. A software pointer marks the current time on the timing wheel. As simulation progresses, the pointer moves forward by one slot for each time step. The event list tracks the events pending and, as the pointer moves, the simulator processes the event list for the current time.

13.4.1 VHDL Simulation Cycle

We shall use VHDL as an example to illustrate the steps in a simulation cycle (which is precisely defined in the LRM). In VHDL, before simulation begins, the design hierarchy is first elaborated . This means all the pieces of the model code (entities, architectures, and configurations) are put together. Then the nets in the model are initialized just before simulation starts. The simulation cycle is then continuously repeated during which processes are executed and signals are updated. A VHDL simulation cycle consists of the following steps:

1. The current time, t_c is set equal to t_n .
2. Each active signal in the model is updated and events may occur as a result.
3. For each process P, if P is currently sensitive to a signal S, and an event has occurred on signal S in this simulation cycle, then process P resumes.
4. Each resumed process is executed until it suspends.
5. The time of the next simulation cycle, t_n , is set to the earliest of:

a. the next time at which a driver becomes active or

b. the next time at which a process resumes

1. If $t_n = t_c$, then the next simulation cycle is a delta cycle .

Simulation is complete when we run out of time (t_n = TIME'HIGH) and there are no active drivers or process resumptions at t_n (there are some slight modifications to these rules involving postponed processes-which we rarely use in ASIC design).

Time in an event-driven simulator has two dimensions. A delta cycle takes delta time , which does not result in a change in real time. Each event that occurs at the same time step executes in delta time. Only when all events have been completed and signals updated does real time advance to the next time step.

13.4.2 Delay

In VHDL you may assign a delay mechanism to an assignment statement. Transport delay is characteristic of wires and transmission lines that exhibit nearly infinite frequency response and will transmit any pulse, no matter how short. Inertial delay more closely models the real behavior of logic cells. Typically, a logic cell will not transmit a pulse that is shorter than the switching time of the circuit, and this is the default pulse-rejection limit . If we explicitly specify a pulse-rejection limit, the

assignment will not transmit a pulse shorter than the limit. As an example, the following three assignments are equivalent to each other:

```
Op <= Ip after 10 ns;

Op <= inertial Ip after 10 ns;

Op <= reject 10 ns inertial Ip after 10 ns;
```

Every assignment that uses transport delay can be written using inertial delay with a pulse-rejection limit, as the following examples illustrate.

```
-- Assignments using transport delay:

Op <= transport Ip after 10 ns;

Op <= transport Ip after 10 ns, not Ip after 20 ns;

-- Their equivalent assignments:

Op <= reject 0 ns inertial Ip after 10 ns;

Op <= reject 0 ns inertial Ip after 10 ns, not Ip after 10 ns;
```

13.5 Cell Models

There are several different kinds of logic cell models:

- Primitive models , which are produced by the ASIC library company and describe the function and properties of each logic cell (NAND, D flip-flop, and so on) using primitive functions.
- Verilog and VHDL models that are produced by an ASIC library company from the primitive models.
- Proprietary models produced by library companies that describe either small logic cells or larger functions such as microprocessors.

A logic cell model is different from the cell delay model, which is used to calculate the delay of the logic cell, from the power model , which is used to calculate power dissipation of the logic cell, and from the interconnect timing model , which is used to calculate the delays between logic cells (we return to these in Section 13.6).

13.5.1 Primitive Models

The following is an example of a primitive model from an ASIC library company (Compass Design Automation). This particular model (for a two-input NAND cell) is complex because it is intended for a 0.35 m m process and has some advanced delay modeling features. The contents are not important to an ASIC designer, but almost all of the information about a logic cell is derived from the primitive model. The designer does not normally see this primitive model; it may only be used by an ASIC library

company to generate other models-Verilog or VHDL, for example.

```
Function
(timingModel = oneOf("ism","pr"); powerModel = oneOf("pin"); )
Rec
Logic = Function (A1; A2; )Rec ZN = not (A1 AND A2); End; End;
miscInfo = Rec Title = "2-Input NAND, 1X Drive"; freq_fact = 0.5;
tml = "nd02d1 nand 2 * zn a1 a2";
MaxParallel = 1; Transistors = 4; power = 0.179018;
Width = 4.2; Height = 12.6; productName = "stdcell35"; libraryName = "cb35sc"; End;
Pin = Rec
A1 = Rec input; cap = 0.010; doc = "Data Input"; End;
A2 = Rec input; cap = 0.010; doc = "Data Input"; End;
ZN = Rec output; cap = 0.009; doc = "Data Output"; End; End;
Symbol = Select
timingModel
On pr Do Rec
tA1D_fr = |( Rec prop = 0.078; ramp = 2.749; End);
tA1D_rf = |( Rec prop = 0.047; ramp = 2.506; End);
tA2D_fr = |( Rec prop = 0.063; ramp = 2.750; End);
tA2D_rf = |( Rec prop = 0.052; ramp = 2.507; End); End
On ism Do Rec
tA1D_fr = |( Rec A0 = 0.0015; dA = 0.0789; D0 = -0.2828;
dD = 4.6642; B = 0.6879; Z = 0.5630; End );
tA1D_rf = |( Rec A0 = 0.0185; dA = 0.0477; D0 = -0.1380;
```

dD = 4.0678; B = 0.5329; Z = 0.3785; End);

tA2D_fr = |(Rec A0 = 0.0079; dA = 0.0462; D0 = -0.2819;

dD = 4.6646; B = 0.6856; Z = 0.5282; End);

tA2D_rf = |(Rec A0 = 0.0060; dA = 0.0464; D0 = -0.1408;

dD = 4.0731; B = 0.6152; Z = 0.4064; End); End; End;

Delay = |(Rec from = pin.A1; to = pin.ZN;

edges = Rec fr = Symbol.tA1D_fr; rf = Symbol.tA1D_rf; End; End, Rec from = pin.A2; to = pin.ZN; edges = Rec fr = Symbol.tA2D_fr; rf = Symbol.tA2D_rf; End; End);

MaxRampTime = |(Rec check = pin.A1; riseTime = 3.000; fallTime = 3.000; End, Rec check = pin.A2; riseTime = 3.000; fallTime = 3.000; End, Rec check = pin.ZN; riseTime = 3.000; fallTime = 3.000; End);

DynamicPower = |(Rec rise = { ZN }; val = 0.003; End); End; End

This primitive model contains the following information:

- The logic cell name, the logic cell function expressed using primitive functions, and port names.
- A list of supported delay models (ism stands for input-slope delay model, and pr for prop-ramp delay model-see Section 13.6).
- Miscellaneous data on the logic cell size, the number of transistors and so on-primarily for use by logic-synthesis tools and for data book generation.
- Information for power dissipation models and timing analysis.

13.5.2 Synopsys Models

The ASIC library company may provide vendor models in formats unique to each CAD tool company. The following is an example of a Synopsys model derived from a primitive model similar to the example in Section 13.5.1 . In a Synopsys library, each logic cell is part of a large file that also contains wire-load models and other characterization information for the cell library.

cell (nd02d1) {

/* title : 2-Input NAND, 1X Drive */

/* pmd checksum : ’HBA7EB26C */

area : 1;

pin(a1) { direction : input; capacitance : 0.088;

fanout_load : 0.088; }

```
pin(a2) { direction : input; capacitance : 0.087;

fanout_load : 0.087; }

pin(zn) { direction : output; max_fanout : 1.786;

max_transition : 3; function : "(a1 a2)’";

timing() {

timing_sense : "negative_unate"

intrinsic_rise : 0.24 intrinsic_fall : 0.17

rise_resistance : 1.68 fall_resistance : 1.13

related_pin : "a1" }

timing() { timing_sense : "negative_unate"

intrinsic_rise : 0.32 intrinsic_fall : 0.18

rise_resistance : 1.68 fall_resistance : 1.13

related_pin : "a2"

} } } /* end of cell */
```

This file contains the only information the Synopsys logic synthesizer, simulator, and other design tools use. If the information is not in this model, the tools cannot produce it. You can see that not all of the information from a primitive model is necessarily present in a vendor model.

13.5.3 Verilog Models

The following is a Verilog model for an inverter (derived from a primitive model):

```
‘celldefine

‘delay_mode_path

‘suppress_faults

‘enable_portfaults

‘timescale 1 ns / 1 ps

module in01d1 (zn, i); input i; output zn; not G2(zn, i);
```

specify specparam

InCap$i = 0.060, OutCap$zn = 0.038, MaxLoad$zn = 1.538,

R_Ramp$i$zn = 0.542:0.980:1.750, F_Ramp$i$zn = 0.605:1.092:1.950;

specparam cell_count = 1.000000; **specparam** Transistors = 4 ;

specparam Power = 1.400000; **specparam** MaxLoadedRamp = 3 ;

(i => zn) = (0.031:0.056:0.100, 0.028:0.050:0.090);

endspecify

endmodule

‘nosuppress_faults

‘disable_portfaults

‘endcelldefine

This is very similar in form to the model for the MUX of Section 13.2.1 , except that this model includes additional timing parameters (at the beginning of the specify block). These timing parameters were omitted to simplify the model of Section 13.2.1 (see Section 13.6 for an explanation of their function).

There are no standards on writing Verilog logic cell models. In the Verilog model, in01d1 , fixed delays (corresponding to zero load capacitance) are embedded in a specify block. The parameters describing the delay equations for the timing model and other logic cell parameters (area, power-model parameters, and so on) are specified using the Verilog specparam feature. Writing the model in this way allows the model information to be accessed using the Verilog PLI routines. It also allows us to back-annotate timing information by overriding the data in the specify block.

The following Verilog code tests the model for logic cell in01d1 :

‘timescale 1 ns / 1 ps

module SDF_b; **reg** A; in01d1 i1 (B, A);

initial begin A = 0; #5; A = 1; #5; A = 0; end

initial $monitor("T=%6g",$realtime," A=",A," B=",B);

endmodule

T= 0 A=0 B=x

T= 0.056 A=0 B=1

T= 5 A=1 B=1

T= 5.05 A=1 B=0

T= 10 A=0 B=0

T=10.056 A=0 B=1

In this case the simulator has used the fixed, typical timing delays (0.056 ns for the rising delay, and 0.05 ns for the falling delay-both from line 12 in module in01d1). Here is an example SDF file (filename SDF_b.sdf) containing back-annotation timing delays:

```
(DELAYFILE

(SDFVERSION "3.0") (DESIGN "SDF.v") (DATE "Aug-13-96")

(VENDOR "MJSS") (PROGRAM "MJSS") (VERSION "v0")

(DIVIDER .) (TIMESCALE 1 ns)

(CELL (CELLTYPE "in01d1")

(INSTANCE SDF_b.i1)

(DELAY (ABSOLUTE

(IOPATH i zn (1.151:1.151:1.151) (1.363:1.363:1.363))

))

)

)
```

(Notice that since Verilog is case sensitive, the instance names and node names in the SDF file are also case sensitive.) This SDF file describes the path delay between input (pin i) and output (pin zn) as 1.151 ns (rising delay-minimum, typical, and maximum are identical in this simple example) and 1.363 ns (falling delay). These delays are calculated by a delay calculator . The delay calculator may be a stand-alone tool or part of the simulator. This tool calculates the delay values by using the delay parameters in the logic cell model (lines 8 - 9 in module in01d1).

We call a system task, $sdf_annotate , to perform back-annotation,

‘timescale 1 ns / 1 ps

module SDF_b; **reg** A; in01d1 i1 (B, A);

initial begin

$sdf_annotate ("SDF_b.sdf", SDF_b, , "sdf_b.log", "minimum", ,);

A = 0; #5; A = 1; #5; A = 0; end

initial $monitor("T=%6g",$realtime," A=",A," B=",B);

endmodule

Here is the output (from MTI V-System/Plus) including back-annotated timing:

T= 0 A=0 B=x

T= 1.151 A=0 B=1

T= 5 A=1 B=1

T= 6.363 A=1 B=0

T= 10 A=0 B=0

T=11.151 A=0 B=1

The delay information from the SDF file has been passed to the simulator.

Back-annotation is not part of the IEEE 1364 Verilog standard, although many Verilog-compatible simulators do support the $sdf_annotate system task. Many ASIC vendors require the use of Verilog to complete a back-annotated timing simulation before they will accept a design for manufacture. Used in this way Verilog is referred to as a golden simulator , since an ASIC vendor uses Verilog to judge whether an ASIC design fabricated using its process will work.

13.5.4 VHDL Models

Initially VHDL did not offer a standard way to perform back-annotation. Here is an example of a VHDL model for an inverter used to perform a back-annotated timing simulation using an Altera programmable ASIC:

library IEEE; **use** IEEE.STD_LOGIC_1164. **all** ;

library COMPASS_LIB; **use** COMPASS_LIB.COMPASS_ETC. **all** ;

entity bknot **is**

generic (derating : REAL := 1.0; Z1_cap : REAL := 0.000;

INSTANCE_NAME : STRING := "bknot");

```
port (Z2 : in Std_Logic; Z1 : out STD_LOGIC);

end bknot;

architecture bknot of bknot is

constant tplh_Z2_Z1 : TIME := (1.00 ns + (0.01 ns * Z1_Cap)) * derating;

constant tphl_Z2_Z1 : TIME := (1.00 ns + (0.01 ns * Z1_Cap)) * derating;

begin

process (Z2)

variable int_Z1 : Std_Logic := 'U';

variable tplh_Z1, tphl_Z1, Z1_delay : time := 0 ns;

variable CHANGED : BOOLEAN;

begin

int_Z1 := not (Z2);

if Z2'EVENT then

tplh_Z1 := tplh_Z2_Z1; tphl_Z1 := tphl_Z2_Z1;

end if ;

Z1_delay := F_Delay(int_Z1, tplh_Z1, tphl_Z1);

Z1 <= int_Z1 after Z1_delay;

end process ;

end bknot;

configuration bknot_CON of bknot is for bknot end for ;

end bknot_CON;
```

This model accepts two generic parameters: load capacitance, Z1_cap , and a derating factor, derating , used to adjust postlayout timing delays. The proliferation of different VHDL back-annotation techniques drove the VHDL community to develop a standard method to complete back-annotation-VITAL.

13.5.5 VITAL Models

VITAL is the VHDL Initiative Toward ASIC Libraries, IEEE Std 1076.4 [1995]. 1 VITAL allows the use of sign-off quality ASIC libraries with VHDL simulators. Sign-off is the transfer of a design from a customer to an ASIC vendor. If the customer has completed simulation of a design using sign-off quality models from an approved cell library and a golden simulator, the customer and ASIC vendor will sign off the design (by signing a contract) and the vendor guarantees that the silicon will match the simulation.

VITAL models, like Verilog models, may be generated from primitive models. Here is an example of a VITAL-compliant model for an inverter,

```
library IEEE; use IEEE.STD_LOGIC_1164. all ;

use IEEE.VITAL_timing. all ; use IEEE.VITAL_primitives. all ;

entity IN01D1 is

generic (

tipd_I : VitalDelayType01 := (0 ns, 0 ns);

tpd_I_ZN : VitalDelayType01 := (0 ns, 0 ns) );

port (

I : in STD_LOGIC := 'U';

ZN : out STD_LOGIC := 'U' );

attribute VITAL_LEVEL0 of IN01D1 : entity is TRUE;

end IN01D1;

architecture IN01D1 of IN01D1 is

attribute VITAL_LEVEL1 of IN01D1 : architecture is TRUE;

signal I_ipd : STD_LOGIC := 'X';

begin

WIREDELAY: block

begin VitalWireDelay(I_ipd, I, tipd_I); end block ;

VITALbehavior : process (I_ipd)

variable ZN_zd : STD_LOGIC;
```

```
variable ZN_GlitchData : VitalGlitchDataType;

begin

ZN_zd := VitalINV(I_ipd);

VitalPathDelay01(

OutSignal => ZN,

OutSignalName => "ZN",

OutTemp => ZN_zd,

Paths => (0 => (I_ipd'LAST_EVENT, tpd_I_ZN, TRUE)),

GlitchData => ZN_GlitchData,

DefaultDelay => VitalZeroDelay01,

Mode => OnEvent,

MsgOn => FALSE,

XOn => TRUE,

MsgSeverity => ERROR);

end process ;

end IN01D1;
```

The following testbench, SDF_testbench , contains an entity, SDF , that in turn instantiates a copy of an inverter, in01d1 :

```
library IEEE; use IEEE.STD_LOGIC_1164. all ;

entity SDF is port ( A : in STD_LOGIC; B : out STD_LOGIC );

end SDF;

architecture SDF of SDF is

component in01d1 port ( I : in STD_LOGIC; ZN : out STD_LOGIC );

end component ;
```

```
begin i1: in01d1 port map ( I => A, ZN => B);

end SDF;

library STD; use STD.TEXTIO. all ;

library IEEE; use IEEE.STD_LOGIC_1164. all ;

entity SDF_testbench is end SDF_testbench;

architecture SDF_testbench of SDF_testbench is

component SDF port ( A : in STD_LOGIC; B : out STD_LOGIC );

end component ;

signal A, B : STD_LOGIC := ’0’;

begin

SDF_b : SDF port map ( A => A, B => B);

process begin

A <= ’0’; wait for 5 ns; A <= ’1’;

wait for 5 ns; A <= ’0’; wait ;

end process ;

process (A, B) variable L: LINE; begin

write(L, now, right, 10, TIME’(ps));

write(L, STRING’(" A=")); write(L, TO_BIT(A));

write(L, STRING’(" B=")); write(L, TO_BIT(B));

writeline(output, L);

end process ;

end SDF_testbench;
```

Here is an SDF file (SDF_b.sdf) that contains back-annotation timing information (min/typ/max timing values are identical in this example):

```
(DELAYFILE
```

```
(SDFVERSION "3.0") (DESIGN "SDF.vhd") (DATE "Aug-13-96")

(VENDOR "MJSS") (PROGRAM "MJSS") (VERSION "v0")

(DIVIDER .) (TIMESCALE 1 ns)

(CELL (CELLTYPE "in01d1")

(INSTANCE i1)

(DELAY (ABSOLUTE

(IOPATH i zn (1.151:1.151:1.151) (1.363:1.363:1.363))

(PORT i (0.021:0.021:0.021) (0.025:0.025:0.025))

))

)

)
```

(VHDL is case insensitive, but to allow the use of an SDF file with both Verilog and VHDL we must maintain case.) As in the Verilog example in Section 13.5.3 the logic cell delay (from the input pin of the inverter, i , to the output pin, zn) follows the IOPATH keyword. In this example there is also an interconnect delay that follows the PORT keyword. The interconnect delay has been placed, or lumped, at the input of the inverter. In order to include back-annotation timing using the SDF file, SDF_b.sdf , we use a command-line switch to the simulator. In the case of MTI V-System/Plus the command is as follows:

```
<msmith/MTI/vital> vsim -c -sdfmax /sdf_b=SDF_b.sdf sdf_testbench

...

# 0 ps A=0 B=0

# 0 ps A=0 B=0

# 1176 ps A=0 B=1

# 5000 ps A=1 B=1

# 6384 ps A=1 B=0

# 10000 ps A=0 B=0

# 11176 ps A=0 B=1
```

We have to explain to the simulator where in the design hierarchy to apply the timing information in the SDF file. The situation is like giving someone directions "Go North on the M1 and turn left at the third intersection," but where do we start? London or Birmingham? VHDL needs much more precise directions. Using VITAL we say we back-annotate to a region . The switch /sdf_b=SDF_b.sdf specifies that all instance names in the SDF file, SDF_b.sdf , are relative to the region /sdf_b . The region refers to instance name sdf_b (line 9 in entity SDF_testbench), which is an instance of component SDF . Component SDF in turn contains an instance of a component, in01d1 , with instance name i1 (line 7 in architecture SDF). Through this rather (for us) difficult-to-follow set of directions, the simulator knows that

... (CELL (CELLTYPE "in01d1") (INSTANCE i1) ...

refers to (SDF) cell or (VHDL) component in01d1 with instance name i1 in instance SDF_b of the compiled model sdf_testbench .

Notice that we cannot use an SDF file of the following form (as we did for the Verilog version of this example):

... (CELL (CELLTYPE "in01d1") (INSTANCE SDF_b.i1) ...

There is no instance in the VHDL model "higher" than instance name SDF_b that we can use as a starting point for VITAL back-annotation. In the Verilog SDF file we can refer to the name of the top-level module (SDF_b in line 2 in module SDF_b). We cannot do this in VHDL; we must name an instance. The result is that, unless you are careful in constructing the hierarchy of your VHDL design, you may not be able to use the same SDF file for back-annotating both VHDL and Verilog.

13.5.6 SDF in Simulation

SDF was developed to handle back-annotation, but it is also used to describe forward-annotation of timing constraints from logic synthesis. Here is an example of an SDF file that contains the timing information for the halfgate ASIC design:

(DELAYFILE

(SDFVERSION "1.0")

(DESIGN "halfgate_ASIC_u")

(DATE "Aug-13-96")

(VENDOR "Compass")

(PROGRAM "HDL Asst")

(VERSION "v9r1.2")

(DIVIDER .)

```
(TIMESCALE 1 ns)

(CELL (CELLTYPE "in01d0")

(INSTANCE v_1.B1_i1)

(DELAY (ABSOLUTE

(IOPATH I ZN (1.151:1.151:1.151) (1.363:1.363:1.363))

))

)

(CELL (CELLTYPE "pc5o06")

(INSTANCE u1_2)

(DELAY (ABSOLUTE

(IOPATH I PAD (1.216:1.216:1.216) (1.249:1.249:1.249))

))

)

(CELL (CELLTYPE "pc5d01r")

(INSTANCE u0_2)

(DELAY (ABSOLUTE

(IOPATH PAD CIN (.169:.169:.169) (.199:.199:.199))

))

)

)
```

This SDF file describes the delay due to the input pad (cell pc5d01r , instance name u0_2), our inverter (cell in01d0 , instance name v_1.B1_i1), and the output pad (cell pc5o06 , instance name u1_2). Since this SDF file was produced before any physical layout, there are no estimates for interconnect delay. The following partial SDF file illustrates how interconnect delay can be specified in SDF.

```
(DELAYFILE

...
```

```
(PROCESS "FAST-FAST")

(TEMPERATURE 0:55:100)

(TIMESCALE 100ps)

(CELL (CELLTYPE "CHIP")

(INSTANCE TOP)

(DELAY (ABSOLUTE

( INTERCONNECT A.INV8.OUT B.DFF1.Q (:0.6:) (:0.6:))

)))
```

This SDF file specifies an interconnect delay (using the keyword INTERCONNECT) of 60 ps (0.6 units with a timescale of 100 ps per unit) between the output port of an inverter with instance name A.INV8 (note that ’.’ is the hierarchy divider) in block A and the Q input port of a D flip-flop (instance name B.DFF1) in block B.

The triplet notation (min : typ : max) in SDF corresponds to minimum, typical, and maximum values of a parameter. Specifying two triplets corresponds to rising (the first triplet) and falling delays. A single triplet corresponds to both. A third triplet corresponds to turn-off delay (transitions to or from ’Z’). You can also specify six triplets (rising, falling, ’0’ to ’Z’ , ’Z’ to ’1’ , ’1’ to ’Z’ , and ’Z’ to ’0’). When only the typical value is specified, the minimum and maximum are set equal to the typical value.

Logic cell delays can use several models in SDF. Here is one example:

```
(INSTANCE B.DFF1)

(DELAY (ABSOLUTE

( IOPATH (POSEDGE CLK) Q (12:14:15) (11:13:15))))
```

The IOPATH construct specifies a delay between the input pin and the output pin of a cell. In this example the delay is between the positive edge of the clock (input port) and the flip-flop output.

The following example SDF file is for an AO221 logic cell:

```
(DELAYFILE

(DESIGN "MYDESIGN")

(DATE "26 AUG 1996")

(VENDOR "ASICS_INC")
```

```
(PROGRAM "SDF_GEN")

(VERSION "3.0")

(DIVIDER .)

(VOLTAGE 3.6:3.3:3.0)

(PROCESS "-3.0:0.0:3.0")

(TEMPERATURE 0.0:25.0:115.0)

(TIMESCALE )

(CELL

(CELLTYPE "AOI221")

(INSTANCE X0)

(DELAY (ABSOLUTE

(IOPATH A1 Y (1.11:1.42:2.47) (1.39:1.78:3.19))

(IOPATH A2 Y (0.97:1.30:2.34) (1.53:1.94:3.50))

(IOPATH B1 Y (1.26:1.59:2.72) (1.52:2.01:3.79))

(IOPATH B2 Y (1.10:1.45:2.56) (1.66:2.18:4.10))

(IOPATH C1 Y (0.79:1.04:1.91) (1.36:1.62:2.61))

))))
```

---

1. IEEE Std 1076.4-1995, © 1995 IEEE. All rights reserved.

13.6 Delay Models

We shall use the term timing model to describe delays outside logic cells and the term delay model to describe delays inside logic cells. These terms are not standard and often people use them interchangeably. There are also different terms for various types of delay:

- A pin-to-pin delay is a delay between an input pin and an output pin of a logic cell. This usually represents the delay of the logic cell excluding any delay contributed by interconnect.
- A pin delay is a delay lumped to a certain pin of a logic cell (usually an input). This usually

represents the delay of the interconnect, but may also represent the delay of the logic cell.

- A net delay or wire delay is a delay outside a logic cell. This always represents the delay of interconnect.

In this section we shall focus on delay models and logic cell delays. In Chapter 3 we modeled logic cell delay as follows (Eq. 3.10):

$t_{PD} = R (C_{out} + C_p) + t_q$. (13.5)

A linear delay model is also known as a prop-ramp delay model , because the delay comprises a fixed propagation delay (the intrinsic delay) and a ramp delay (the extrinsic delay). As an example, the data book entry for the inverter, cell in01d0 , in a 0.8 m m standard-cell library gives the following delay information (with delay measured in nanoseconds and capacitance in picofarads):

RISE = 0.10 + 0.07 + (1.75 ¥ Cld) FALL = 0.09 + 0.07 + (1.95 ¥ Cld) (13.6)

RISE = 0.10 + 0.07 + (1.75 ¥ Cld) FALL = 0.09 + 0.07 + (1.95 ¥ Cld) (13.5)

The first two terms in each of these equations represents the intrinsic delay, with the last term in each equation representing the extrinsic delay. We see that the Cld corresponds to C_{out} , R_{pu} = 1.75 k W , and R_{pd} = 1.95 k W (R_{pu} is the pull-up resistance and R_{pd} is the pull-down resistance) .

From the data book the pin capacitances for this logic cell are as follows:

pin I (input) = 0.060 pF pin ZN (output) = 0.038 pF (13.7)

pin I (input) = 0.060 pF pin ZN (output) = 0.038 pF (13.6)

Thus, C_p = 0.038 pF and we can calculate the component of the intrinsic delay due to the output pin capacitance as follows:

C_p ¥ R_{pu} = 0.038 ¥ 1.75 = 0.0665 ns and C_p ¥ R_{pd} = 0.038 ¥ 1.95 = 0.0741 ns (13.8)

C_p ¥ R_{pu} = 0.038 ¥ 1.75 = 0.0665 ns and C_p ¥ R_{pd} = 0.038 ¥ 1.95 = 0.0741 ns(13.7)

Suppose t_{qr} and t_{qf} are the parasitic delays for the rising and falling waveforms respectively. By comparing the data book equations for the rise and fall delays with Eq. and 13.7 , we can identify t_{qr} = 0.10 ns and t_{qf} = 0.09 ns.

Now we can explain the timing section of the in01d0 model (Section 13.5.3),

specify specparam

InCap$i = 0.060, OutCap$zn = 0.038, MaxLoad$zn = 1.538,

R_Ramp$i$zn = 0.542:0.980:1.750, F_Ramp$i$zn = 0.605:1.092:1.950;

specparam cell_count = 1.000000; **specparam** Transistors = 4 ;

specparam Power = 1.400000; **specparam** MaxLoadedRamp = 3 ;

(i=>zn)=(0.031:0.056:0.100, 0.028:0.050:0.090);

The parameter OutCap$zn is C_p . The maximum value of the parameter R_Ramp$i$zn is R_{pu} , and the maximum value of parameter F_Ramp$i$zn is R_{pd} . Finally, the maximum values of the fixed-delay triplets correspond to t_{qr} and t_{qf} .

13.6.1 Using a Library Data Book

ASIC library data books typically contain two types of information for each cell in the library-capacitance loading and delay. Table 13.7 shows the input capacitances for the inverter family for both an area-optimized library (small) and a performance-optimized library (fast).

From Table 13.7 , the input capacitance of the small library version of the inv1 (a 1X inverter gate) is 0.034 pF. Any logic cell that is driving an inv1 from the small library sees this as a load capacitance. This capacitance consists of the gate capacitance of a p -channel transistor, the gate capacitance of an n -channel transistor, and the internal cell routing. Similarly, 0.145 pF is the input capacitance of a fast inv1 . We can deduce that the transistors in the fast library are approximately 0.145 / 0.034 ? 4 times larger than those in the small version. The small library and fast library may not have the same cell height (they usually do not), so that we cannot mix cells from different libraries in the same standard-cell area.

TABLE 13.7 Input capacitances for an inverter family (pF).1

Library	inv1	invh 1	invs	inv8	inv12
Area	0.034	0.067	0.133	0.265	0.397
Performance	0.145	0.292	0.584	1.169	1.753

The delay table for a 2:1 MUX is shown in Table 13.8 . For example, DO/ to Z/ , indicates the path delay from the DO input rising to the Z output rising. Rising delay is denoted by ’/’ and falling delay by ’\’ .

TABLE 13.8 Delay information for a 2:1 MUX.

		Propagation delay			
		Area		Performance	
From input 2	To output	Extrinsic / $nspF^{-1}$	Intrinsic / ns	Extrinsic / ns	Intrinsic / ns
D0\	Z\	2.10	1.42	0.5	0.8

D0/	Z/	3.66	1.23	0.68	0.70
D1\	Z\	2.10	1.42	0.50	0.80
D1/	Z/	3.66	1.23	0.68	0.70
SD\	Z\	2.10	1.42	0.50	0.80
SD\	Z/	3.66	1.09	0.70	0.73
SD/	Z\	2.10	2.09	0.5	1.09
SD/	Z/	3.66	1.23	0.68	0.70

Both intrinsic delay and extrinsic delay values are given in Table 13.8 . For example, the delay t_{PD} (from DO\ to Z \) of a 2:1 MUX from the small library is

t_{PD} = 1.42 ns + (2.10 ns/pF) ¥ C_L (pF) . (13.9)

ASIC cell libraries may be characterized and the delay information presented in several ways in a data book. Some manufacturers simulate under worst-case slow conditions (4.5 V, 100 °C, and slow process conditions, for example) and then derate each delay value to convert delays to nominal conditions (5.0 V, 25 °C, and nominal process). This allows nominal delays to be used in the data book while maintaining accurate predictions for worst-case behavior. Other manufacturers characterize using nominal conditions and include worst-case values in the data book. In either case, we always design with worst-case values. Data books normally include process, voltage, and temperature derating factors as tables or graphs such as those shown in Tables 13.9 and 13.10 .

For example, suppose we are measuring the performance of an ASIC on the bench and the lab temperature (25 °C) and the power supply voltage (5 V) correspond to nominal operating conditions. We shall assume, in the absence of other information, that we have an ASIC from a nominal process lot. We have data book values given as worst case (worst-case temperature, 100 °C; worst-case voltage, 4.5 V; slow process) and we wish to find nominal values for delay to compare them with our measured results. From Table 13.9 the derating factor from nominal process to slow process is 1.31. From Table 13.10 the derating factor from 100 °C and 4.5 V to nominal (25 °C and 5 V) is 1.60. The derating factor from nominal to worst-case (data book values) is thus:

worst-case = nominal ¥ 1.31 (slow process) ¥ 1.60 (4.5 V, 100 °C). (13.10)

worst-case = nominal ¥ 1.31 (slow process) ¥ 1.60 (4.5 V, 100 °C).(13.8)

To get from the data book values to nominal operating conditions we use the following equation:

nominal = worst-case/(1.31 ¥ 1.60) = 0.477 ¥ worst-case. (13.11)

nominal = worst-case/(1.31 ¥ 1.60) = 0.477 ¥ worst-case. (13.9)

TABLE 13.9 Process derating factors. TABLE 13.10 Temperature and voltage derating factors.

Process	Derating factor	Supply voltage

Slow	1.31
Nominal	1.0
Fast	0.75

Temperature/°C	4.5V	4.75V	5.00V	5.25V	5.50V
-40	0.77	0.73	0.68	0.64	0.61
0	1.00	0.93	0.87	0.82	0.78
25	1.14	1.07	1.00	0.94	0.90
85	1.50	1.40	1.33	1.26	1.20
100	1.60	1.49	1.41	1.34	1.28
125	1.76	1.65	1.56	1.47	1.41

13.6.2 Input-Slope Delay Model

It is increasingly important for submicron technologies to account for the effects of the rise (and fall) time of the input waveforms to a logic cell. The nonlinear delay model described in this section was developed by Mike Misheloff at VLSI Technology and then at Compass. There are, however, no standards in this area-each ASIC company has its own, often proprietary, model.

We begin with some definitions:

- D_{t0} is the time from the beginning of the input to beginning of the output.
- D_{t1} is the time from the beginning of the input to the end of the output.
- I_R is the time from the beginning to the end of the input ramp.

In these definitions "beginning" and "end" refer to the projected intersections of the input waveform or the output waveform with V_{DD} and V_{SS} as appropriate. Then we can calculate the delay, D (measured with 0.5 trip points at input and output), and output ramp, O_R, as follows:

$$D = (D_{t1} + D_{t0} - I_R) / 2 \quad (13.12)$$

and $$O_R = D_{t1} - D_{t0} . \quad (13.13)$$

Experimentally we find that the times, D_{t0} and D_{t1}, are accurately modeled by the following equations:

$$D_{t0} = A_0 + D_0 C_L + B ¥ \min (I_R , C_R) + Z ¥ \max (0, I_R - C_R) \quad (13.14)$$

and

$$D_{t1} = A_1 + B I_R + D_1 C_L . \quad (13.15)$$

C_R is the critical ramp that separates two regions of operation, we call these slow ramp and fast ramp. A sensible definition for C_R is the point at which the end of the input ramp occurs at the same time the output reaches the 0.5 trip point. This leads to the following equation for C_R :

$$C_R = \frac{A_0 + A_1 + (D_0 + D_1) C_L}{2(1 - B)} \qquad (13.16)$$

It is convenient to define two more parameters:

$$d_A = A_1 - A_0 \text{ and } d_D = D_1 - D_0 . \qquad (13.17)$$

In the region that $C_R > I_R$, we can simplify Eqs. 13.14 and by using the definitions in Eq. 13.17 , as follows:

$$D = (D_{t1} + D_{t0} - I_R)/2 = A_0 + D_0 C_L + d_A/2 + d_D C_L/2 \qquad (13.18)$$

and

$$O_R = D_{t1} - D_{t0} = d_A + d_D C_L . \qquad (13.19)$$

Now we can understand the timing parameters in the primitive model in Section 13.5.1 . For example, the following parameter, tA1D_fr , models the falling input to rising output waveform delay for the logic cell (the units are a consistent set: all times are measured in nanoseconds and capacitances in picofarads):

A0 = 0.0015;dA = 0.0789;D0 = -0.2828;dD = 4.6642;B = 0.6879;Z = 0.5630;

The input-slope model predicts delay in the fast-ramp region, D_{ISM} (50 %, FR), as follows (0.5 trip points):

$$\begin{aligned} &D_{ISM}(50\ \%, FR) \\ &= A_0 + D_0 C_L + 0.5\, O_R = A_0 + D_0 C_L + d_A/2 + d_D C_L/2 \\ &= 0.0015 + 0.5 ¥ 0.0789 + (-0.2828 + 0.5 ¥ 4.6642) C_L \\ &= 0.041 + 2.05\, C_L . \end{aligned} \qquad (13.20)$$

We can adjust this delay to 0.35/0.65 trip points as follows:

$$\begin{aligned} &D_{ISM}(65\ \%, FR) \\ &= A_0 + D_0 C_L + 0.65\, O_R \\ &= 0.0015 + 0.65 ¥ 0.0789 + (-0.2828\, C_L + 0.65 ¥ 4.6642) C_L \\ &= 0.053 + 2.749\, C_L . \end{aligned} \qquad (13.21)$$

We can now compare Eq. 13.21 with the prop-ramp model. The prop-ramp parameters for this logic cell

(from the primitive model in Section 13.5.1) are:

tA1D_fr = |(Rec prop = 0.078; ramp = 2.749; End);

These parameters predict the following prop-ramp delay (0.35/0.65 trip points):

$$D_{PR}(65\ \%) = 0.078 + 2.749\ C_L \cdot \quad (13.22)$$

The input-slope delay model and the prop-ramp delay model predict similar delays in the fast-ramp region, but for slower inputs the differences can become significant.

13.6.3 Limitations of Logic Simulation

Table 13.11 shows the switching characteristics of a two-input NAND gate (1X drive) from a commercial 1 m m gate-array family. The difference in propagation delay (with FO = 0) between the inputs A and B is

(0.25 - 0.17) ¥ 2 / (0.25 + 0.17) = 38 %.

This difference is taken into account only by a pin-to-pin delay model.

TABLE 13.11 Switching characteristics of a two-input NAND gate.

Symbol	Parameter	Fanout 3					
		FO = 0	FO = 1	FO = 2	FO = 4	FO = 8	K
		/ns	/ns	/ns	/ns	/ns	/nspF $^{-1}$
t_{PLH}	Propagation delay, A to X	0.25	0.35	0.45	0.65	1.05	1.25
t_{PHL}	Propagation delay, B to X	0.17	0.24	0.30	0.42	0.68	0.79
t_r	Output rise time, X	1.01	1.28	1.56	2.10	3.19	3.40
t_f	Output fall time, X	0.54	0.69	0.84	1.13	1.71	1.83

Timing information for most gate-level simulators is calculated once, before simulation, using a delay calculator. This works as long as the logic cell delays and signal ramps do not change. There are some cases in which this is not true. Table 13.12 shows the switching characteristics of a half adder. In addition to pin-to-pin timing differences there is a timing difference depending on state. For example, the pin-to-pin timing from input pin A to the output pin S depends on the state of the input pin B. Depending on whether B = ’0’ or B = ’1’ the difference in propagation delay (at FO = 0) is

(0.93 - 0.58) ¥ 2 / (0.93 + 0.58) = 46 %.

This state-dependent timing is not taken into account by simple pin-to-pin delay models and is not

accounted for by most gate-level simulators.

TABLE 13.12 Switching characteristics of a half adder.

Symbol	Parameter	Fanout 4					
		FO = 0	FO = 1	FO = 2	FO = 4	FO = 8	K
		/ns	/ns	/ns	/ns	/ns	/nspF $^{-1}$
t_{PLH}	Delay, A to S (B = ’0’)	0.58	0.68	0.78	0.98	1.38	1.25
t_{PHL}	Delay, A to S (B = ’1’)	0.93	0.97	1.00	1.08	1.24	0.48
t_{PLH}	Delay, B to S (B = ’0’)	0.89	0.99	1.09	1.29	1.69	1.25
t_{PHL}	Delay, B to S (B = ’1’)	1.00	1.04	1.08	1.15	1.31	0.48
t_{PLH}	Delay, A to CO	0.43	0.53	0.63	0.83	1.23	1.25
t_{PHL}	Delay, A to CO	0.59	0.63	0.67	0.75	0.90	0.48
t_r	Output rise time, X	1.01	1.28	1.56	2.10	3.19	3.40
t_f	Output fall time, X	0.54	0.69	0.84	1.13	1.71	1.83

1. 1Suffix ’1’ denotes normal drive strength, suffix ’h’ denotes high-power drive strength (approximately ¥ 2) , suffix ’s’ denotes superpower drive strength (approximately ¥ 4), and a suffix ’ m ’ (m =8 or 12) denotes inverter blocks containing m inverters.

2. / = rising and \ = falling.

3. FO = fanout in standard loads (one standard load = 0.08 pF). Nominal conditions: V_{DD} = 5 V, T_A = 25 °C.

4. FO = fanout in standard loads (one standard load = 0.08 pF). Nominal conditions: V_{DD} = 5 V, T_A = 25 °C.

13.7 Static Timing Analysis

We return to the comparator/MUX example to see how timing analysis is applied to sequential logic. We shall use the same input code (comp_mux.v in Section 13.2), but this time we shall target the design to an Actel FPGA.

Before routing we obtain the following static timing analysis:

Instance name in pin-->out pin tr total incr cell

END_OF_PATH

outp_2_ R 27.26

OUT1 : D--->PAD R 27.26 7.55 OUTBUF

I_1_CM8 : S11--->Y R 19.71 4.40 CM8

I_2_CM8 : S11--->Y R 15.31 5.20 CM8

I_3_CM8 : S11--->Y R 10.11 4.80 CM8

IN1 : PAD--->Y R 5.32 5.32 INBUF

a_2_ R 0.00 0.00

BEGIN_OF_PATH

The estimated prelayout critical path delay is nearly 30 ns including the I/O-cell delays (ACT 3, worst-case, standard speed grade). This limits the operating frequency to 33 MHz (assuming we can get the signals to and from the chip pins with no further delays-highly unlikely). The operating frequency can be increased by pipelining the design as follows (by including three register stages: at the inputs, the outputs, and between the comparison and the select functions):

```
// comp_mux_rrr.v

module comp_mux_rrr(a, b, clock, outp);

input [2:0] a, b; output [2:0] outp; input clock;

reg [2:0] a_r, a_rr, b_r, b_rr, outp; reg sel_r;

wire sel = ( a_r <= b_r ) ? 0 : 1;

always @ ( posedge clock) begin a_r <= a; b_r <= b; end

always @ ( posedge clock) begin a_rr <= a_r; b_rr <= b_r; end

always @ ( posedge clock) outp <= sel_r ? b_rr : a_rr;

always @ ( posedge clock) sel_r <= sel;

endmodule
```

Following synthesis we optimize module comp_mux_rrr for maximum speed. Static timing analysis gives the following preroute critical paths:

----------INPAD to SETUP longest path----------

Rise delay, Worst case

Instance name in pin-->out pin tr total incr cell

END_OF_PATH

D.a_r_ff_b2 R 4.52 0.00 DF1

INBUF_24 : PAD--->Y R 4.52 4.52 INBUF

a_2_ R 0.00 0.00

BEGIN_OF_PATH

----------CLOCK to SETUP longest path----------

Rise delay, Worst case

Instance name in pin-->out pin tr total incr cell

END_OF_PATH

D.sel_r_ff R 9.99 0.00 DF1

I_1_CM8 : S10--->Y R 9.99 0.00 CM8

I_3_CM8 : S00--->Y R 9.99 4.40 CM8

a_r_ff_b1 : CLK--->Q R 5.60 5.60 DF1

BEGIN_OF_PATH

----------CLOCK to OUTPAD longest path----------

Rise delay, Worst case

Instance name in pin-->out pin tr total incr cell

END_OF_PATH

outp_2_ R 11.95

OUTBUF_31 : D--->PAD R 11.95 7.55 OUTBUF

outp_ff_b2 : CLK--->Q R 4.40 4.40 DF1

BEGIN_OF_PATH

The timing analyzer has examined the following:

1. Paths that start at an input pad and end on the data input of a sequential logic cell (the D input to a D flip-flop, for example). We might call this an entry path (or input-to-D path) to a pipelined design. The longest entry delay (or input-to-setup delay) is 4.52 ns.
2. Paths that start at a clock input to a sequential logic cell and end at the data input of a sequential logic cell. This is a stage path (register-to-register path or clock-to-D path) in a pipeline stage. The longest stage delay (clock-to-D delay) is 9.99 ns.
3. Paths that start at a sequential logic cell output and end at an output pad. This is an exit path (clock-to-output path) from the pipeline. The longest exit delay (clock-to-output delay) is 11.95 ns.

By pipelining the design we added three clock periods of latency, but we increased the estimated operating speed. The longest prelayout critical path is now an exit delay, approximately 12 ns-more than doubling the maximum operating frequency. Next, we route the registered version of the design. The Actel software informs us that the postroute maximum stage delay is 11.3 ns (close to the preroute estimate of 9.99 ns). To check this figure we can perform another timing analysis. This time we shall measure the stage delays (the start points are all clock pins, and the end points are all inputs to sequential cells, in our case the D input to a D flip-flop). We need to define the sets of nodes at which to start and end the timing analysis (similar to the path clusters we used to specify timing constraints in logic synthesis). In the Actel timing analyzer we can use predefined sets ’clock’ (flip-flop clock pins) and ’gated’ (flip-flop inputs) as follows:

timer> startset clock

timer> endset gated

timer> longest

1st longest path to all endpins

Rank Total Start pin First Net End Net End pin

0 11.3 a_r_ff_b2:CLK a_r_2_ block_0_OUT1 sel_r_ff:D

1 6.6 sel_r_ff:CLK sel_r DEF_NET_50 outp_ff_b0:D

... 8 similar lines omitted ...

We could try to reduce the long stage delay (11.3 ns), but we have already seen from the preroute timing estimates that an exit delay may be the critical path. Next, we check some other important timing parameters.

13.7.1 Hold Time

Hold-time problems can occur if there is clock skew between adjacent flip-flops, for example. We first need to check for the shortest exit delays using the same sets that we used to check stage delays,

timer> shortest

1st shortest path to all endpins

Rank Total Start pin First Net End Net End pin

0 4.0 b_rr_ff_b1:CLK b_rr_1_ DEF_NET_48 outp_ff_b1:D

1 4.1 a_rr_ff_b2:CLK a_rr_2_ DEF_NET_46 outp_ff_b2:D

... 8 similar lines omitted ...

The shortest path delay, 4 ns, is between the clock input of a D flip-flop with instance name b_rr_ff_b1 (call this X) and the D input of flip-flop instance name outp_ff_b1 (Y). Due to clock skew, the clock signal may not arrive at both flip-flops simultaneously. Suppose the clock arrives at flip-flop Y 3 ns earlier than at flip-flop X . The D input to flip-flop Y is only stable for (4 - 3) = 1 ns after the clock edge. To check for hold-time violations we thus need to find the clock skew corresponding to each clock-to-D path. This is tedious and normally timing-analysis tools check hold-time requirements automatically, but we shall show the steps to illustrate the process.

13.7.2 Entry Delay

Before we can measure clock skew, we need to analyze the entry delays, including the clock tree. The synthesis tools automatically add I/O pads and the clock cells. This means that extra nodes are automatically added to the netlist with automatically generated names. The EDIF conversion tools may then modify these names. Before we can perform an analysis of entry delays and the clock network delay, we need to find the input node names. By looking for the EDIF ’rename’ construct in the EDIF netlist we can associate the input and output node names in the behavioral Verilog model, comp_mux_rrr , and the EDIF names,

```
piron% grep rename comp_mux_rrr_o.edn

(port (rename a_2_ "a[2]") (direction INPUT))

... 8 similar lines renaming ports omitted ...

(net (rename a_rr_0_ "a_rr[0]") (joined

... 9 similar lines renaming nets omitted ...

piron%
```

Thus, for example, the EDIF conversion program has renamed input port a[2] to a_2_ because the design tools do not like the Verilog bus notation using square brackets. Next we find the connections between the ports and the added I/O cells by looking for ’PAD’ in the Actel format netlist, which indicates a connection to a pad and the pins of the chip, as follows:

```
piron% grep PAD comp_mux_rrr_o.adl

NET DEF_NET_148; outp_2_, OUTBUF_31:PAD.

NET DEF_NET_151; outp_1_, OUTBUF_32:PAD.

NET DEF_NET_154; outp_0_, OUTBUF_33:PAD.

NET DEF_NET_127; a_2_, INBUF_24:PAD.

NET DEF_NET_130; a_1_, INBUF_25:PAD.

NET DEF_NET_133; a_0_, INBUF_26:PAD.

NET DEF_NET_136; b_2_, INBUF_27:PAD.

NET DEF_NET_139; b_1_, INBUF_28:PAD.

NET DEF_NET_142; b_0_, INBUF_29:PAD.

NET DEF_NET_145; clock, CLKBUF_30:PAD.

piron%
```

This tells us, for example, that the node we called clock in our behavioral model has been joined to a node (with automatically generated name) called CLKBUF_30:PAD , using a net (connection) named DEF_NET_145 (again automatically generated). This net is the connection between the node clock that is dangling in the behavioral model and the clock-buffer pad cell that the synthesis tools automatically added.

13.7.3 Exit Delay

We now know that the clock-pad input is CLKBUF_30:PAD , so we can find the exit delays (the longest path between clock-pad input and an output) as follows (using the clock-pad input as the start set):

timer> startset clockpad

Working startset ’clockpad’ contains 0 pins.

timer> addstart CLKBUF_30:PAD

Working startset ’clockpad’ contains 2 pins.

I shall explain why this set contains two pins and not just one presently. Next, we define the end set and trace the longest exit paths as follows:

timer> endset outpad

Working endset ’outpad’ contains 3 pins.

timer> longest

1st longest path to all endpins

Rank Total Start pin First Net End Net End pin

0 16.1 CLKBUF_30/U0:PAD DEF_NET_144 DEF_NET_154 OUTBUF_33:PAD

1 16.0 CLKBUF_30/U0:PAD DEF_NET_144 DEF_NET_151 OUTBUF_32:PAD

2 16.0 CLKBUF_30/U0:PAD DEF_NET_144 DEF_NET_148 OUTBUF_31:PAD

3 pins

This tells us we have three paths from the clock-pad input to the three output pins (outp[0] , outp[1] , and outp[2]). We can examine the longest exit delay in more detail as follows:

timer> expand 0

1st longest path to OUTBUF_33:PAD (rising) (Rank: 0)

Total Delay Typ Load Macro Start pin Net name

16.1 3.7 Tpd 0 OUTBUF OUTBUF_33:D DEF_NET_154

12.4 4.5 Tpd 1 DF1 outp_ff_b0:CLK DEF_NET_1530

7.9 7.9 Tpd 16 CLKEXT_0 CLKBUF_30/U0:PAD DEF_NET_144

The input-to-clock delay, t_{IC} , due to the clock-buffer cell (or macro) CLKEXT_0 , instance name CLKBUF_30/U0 , is 7.9 ns. The clock-to-Q delay, t_{CQ} , of flip-flop cell DF1 , instance name outp_ff_b0 , is 4.5 ns. The delay, t_{QO} , due to the output buffer cell OUTBUF , instance name OUTBUF_33 , is 3.7 ns. The longest path between clock-pad input and the output, t_{CO} , is thus

$$t_{CO} = t_{IC} + t_{CQ} + t_{QO} = 16.1 \text{ ns} . \quad (13.23)$$

This is the critical path and limits the operating frequency to (1 / 16.1 ns) ª 62 MHz.

When we created a start set using CLKBUF_30:PAD , the timing analyzer told us that this set consisted of two pins. We can list the names of the two pins as follows:

timer> showset clockpad

Pin name Net name Macro name

CLKBUF_30/U0:PAD <no net> CLKEXT_0

CLKBUF_30/U1:PAD DEF_NET_145 CLKTRI_0

2 pins

The clock-buffer instance name, CLKBUF_30/U0 , is hierarchical (with a ’/’ hierarchy separator). This indicates that there is more than one instance inside the clock-buffer cell, CLKBUF_30 . Instance CLKBUF_30/U0 is the input driver, instance CLKBUF_30/U1 is the output driver (which is disabled and unused in this case).

13.7.4 External Setup Time

Each of the six chip data inputs must satisfy the following set-up equation:

$$t_{SU} \text{ (external)} > t_{SU} \text{ (internal) - (clock delay) + (data delay} \quad (13.24)$$

(where both clock and data delays end at the same flip-flop instance). We find the clock delays in Eq. 13.24 using the clock input pin as the start set and the end set ’clock’ . The timing analyzer tells us all 16 clock path delays are the same at 7.9 ns in our design, and the clock skew is thus zero. Actel’s clock distribution system minimizes clock skew, but clock skew will not always be zero. From the discussion in Section 13.7.1 , we see there is no possibility of internal hold-time violations with a clock skew of zero.

Next, we find the data delays in Eq, 13.24 using a start set of all input pads and an end set of ’gated’ ,

timer> longest

... lines omitted ...

1st longest path to all endpins

Rank Total Start pin First Net End Net End pin

10 10.0 INBUF_26:PAD DEF_NET_1320 DEF_NET_1320 a_r_ff_b0:D

11 9.7 INBUF_28:PAD DEF_NET_1380 DEF_NET_1380 b_r_ff_b1:D

12 9.4 INBUF_25:PAD DEF_NET_1290 DEF_NET_1290 a_r_ff_b1:D

13 9.3 INBUF_27:PAD DEF_NET_1350 DEF_NET_1350 b_r_ff_b2:D

14 9.2 INBUF_29:PAD DEF_NET_1410 DEF_NET_1410 b_r_ff_b0:D

15 9.1 INBUF_24:PAD DEF_NET_1260 DEF_NET_1260 a_r_ff_b2:D

16 pins

We are only interested in the last six paths of this analysis (rank 10-15) that describe the delays from each data input pad (a[0] , a[1] , a[2] , b[0] , b[1] , b[2]) to the D input of a flip-flop. The maximum data delay, 10 ns, occurs on input buffer instance name INBUF_26 (pad 26); pin INBUF_26:PAD is node a_0_ in the EDIF file or input a[0] in our behavioral model. The six t_{SU} (external) equations corresponding to Eq, 13.24 may be reduced to the following worst-case relation:

$$t_{SU}(\text{external})_{max} > t_{SU}(\text{internal}) - 7.9\text{ ns} + \max(9.1\text{ ns}, 10.0\text{ ns})$$
$$> t_{SU}(\text{internal}) + 2.1\text{ ns} \qquad (13.25)$$

We calculated the clock and data delay terms in Eq. 13.24 separately, but timing analyzers can normally perform a single analysis as follows:

$$t_{SU}(\text{external})_{max} > t_{SU}(\text{internal}) - (\text{clock delay} - \text{data delay})_{min} . \quad (13.26)$$

Finally, we check that there is no external hold-time requirement. That is to say, we must check that t_{SU} (external) is never negative or

$$t_{SU}(\text{external})_{min} > t_{SU}(\text{internal}) - (\text{clock delay} - \text{data delay})_{max} > 0$$
$$> t_{SU}(\text{internal}) + 1.2\text{ ns} > 0 . \qquad (13.27)$$

Since t_{SU} (internal) is always positive on Actel FPGAs, $t_{SU}(\text{external})_{min}$ is always positive for this

design. In large ASICs, with large clock delays, it is possible to have external hold-time requirements on inputs. This is the reason that some FPGAs (Xilinx, for example) have programmable delay elements that deliberately increase the data delay and eliminate irksome external hold-time requirements.

13.8 Formal Verification

Using logic synthesis we move from a behavioral model to a structural model. How are we to know (other than by trusting the logic synthesizer) that the two representations are the same? We have already seen that we may have to alter the original reference model because the HDL acceptable to a synthesis tool is a subset of HDL acceptable to simulators. Formal verification can prove, in the mathematical sense, that two representations are equivalent. If they are not, the software can tell us why and how two representations differ.

13.8.1 An Example

We shall use the following VHDL entity with two architectures as an example: 1

entity Alarm **is**

port (Clock, Key, Trip : **in** bit; Ring : **out** bit);

end Alarm;

The following behavioral architecture is the reference model :

architecture RTL **of** Alarm **is**

type States **is** (Armed, Off, Ringing); **signal** State : States;

begin

process (Clock) **begin**

if Clock = ’1’ **and** Clock’EVENT **then**

case State **is**

when Off => **if** Key = ’1’ **then** State <= Armed; **end if** ;

when Armed => **if** Key = ’0’ **then** State <= Off;

elsif Trip = ’1’ **then** State <= Ringing;

end if ;

when Ringing => **if** Key = ’0’ **then** State <= Off; **end if** ;

```
end case ;

end if ;

end process ;

Ring <= '1' when State = Ringing else '0';

end RTL;
```

The following synthesized structural architecture is the derived model :

```
library cells; use cells. all ; // ...contains logic cell models

architecture Gates of Alarm is

component Inverter port (i : in BIT;z : out BIT) ; end component ;

component NAnd2 port (a,b : in BIT;z : out BIT) ; end component ;

component NAnd3 port (a,b,c : in BIT;z : out BIT) ; end component ;

component DFF port(d,c : in BIT; q,qn : out BIT) ; end component ;

signal State, NextState : BIT_VECTOR(1 downto 0);

signal s0, s1, s2, s3 : BIT;

begin

g2: Inverter port map ( i => State(0), z => s1 );

g3: NAnd2 port map ( a => s1, b => State(1), z => s2 );

g4: Inverter port map ( i => s2, z => Ring );

g5: NAnd2 port map ( a => State(1), b => Key, z => s0 );

g6: NAnd3 port map ( a => Trip, b => s1, c => Key, z => s3 );

g7: NAnd2 port map ( a => s0, b => s3, z => NextState(1) );

g8: Inverter port map ( i => Key, z => NextState(0) );

state_ff_b0: DFF port map

( d => NextState(0), c => Clock, q => State(0), qn => open );
```

state_ff_b1: DFF **port map**

(d => NextState(1), c => Clock, q => State(1), qn => **open**);

end Gates;

To compare the reference and the derived models (two representations), formal verification performs the following steps: (1) the HDL is parsed, (2) a finite-state machine compiler extracts the states present in any sequential logic, (3) a proof generator automatically generates formulas to be proved, (4) the theorem prover attempts to prove the formulas. The results from the last step are as follows:

formulas to be proved: 8

formulas proved VALID: 8

By constructing and then proving formulas the software tells us that architecture RTL implies architecture Gates (implication is the default proof mechanism-we could also have asked if the architectures are exactly equivalent). Next, we shall explore what this means and how formal verification works.

13.8.2 Understanding Formal Verification

The formulas to be proved are generated in a separate file of proof statements :

axioms

Let Axiom_ref = Axioms Of alarm-rtl

Let Axiom_der = Axioms Of alarm-gates

ProveNotAlwaysFalse (Axiom_ref)

Prove (Axiom_ref => Axiom_der)

assertions

Let Assert_ref = Asserts Of alarm-rtl

Let Assert_der = Asserts Of alarm-gates

Prove (Axiom_ref => (Assert_ref => Assert_der))

clocks

Let ClockEvents_ref = Clocks Of alarm-rtl

Let ClockEvents_der = Clocks Of alarm-gates

```
Let Master__clock_event_ref =

Value (master__clock'event Of alarm-rtl)

Prove (Axiom_ref => (ClockEvents_ref <=> ClockEvents_der))

# next state of memories

Prove ((Axiom_ref And Master__clock_event_ref) =>

(Transition (state(1) Of alarm-rtl) <=>

Transition (state_ff_b1.t Of alarm-gates)))

Prove ((Axiom_ref And Master__clock_event_ref) =>

(Transition (state(0) Of alarm-rtl) <=>

Transition (state_ff_b0.t Of alarm-gates)))

# validity value of outbuses

Prove (Axiom_ref => (Domain (ring Of alarm-rtl) <=>

Domain (ring Of alarm-gates)))

Prove (Axiom_ref => (Domain (ring Of alarm-rtl) =>

(Value (ring Of alarm-rtl) <=>

Value (ring Of alarm-gates))))
```

Formal verification makes strict use of the terms axiom and assertion . An axiom is an explicit or implicit fact. For example, if a VHDL signal is declared to be type BIT , an implicit axiom is that this signal may only take the logic values '0' and '1' . An assertion is derived from a statement placed in the HDL code. For example, the following VHDL statement is an assertion:

```
assert Key /= '1' or Trip /= '1' or NextState = Ringing

report "Alarm on and tripped but not ringing";
```

A VHDL assert statement prints only if the condition is FALSE . We know from de Morgan's theorem that (A + B + C)' = A'B'C' . Thus, this statement checks for a burglar alarm that does not ring when it is on and we are burgled.

In the proof statements the symbol '=>' means implies . In logic calculus we write A ? B to mean A implies B . The symbol '<=>' means equivalence , and this is stricter than implication. We write A ¤ B to mean: A is equivalent to B . Table 13.13 show the truth tables for these two logic operators.

TABLE 13.13 Implication and equivalence.

A	B	A ? B	A ¤ B
F	F	T	T
F	T	T	F
T	F	F	F
T	T	T	T

13.8.3 Adding an Assertion

If we include the assert statement from the previous section in architecture RTL and repeat formal verification, we get the following message from the FSM compiler:

<E> Assertion may be violated

SEVERITY: ERROR

REPORT: Alarm on and tripped but not ringing

FILE: .../alarm-rtl3.vhdl

FSM: alarm-rtl3

STATEMENT or DECLARATION: line8

.../alarm-rtl3.vhdl (line 8)

Context of the message is:

(key And trip And memoryofdriver__state(0))

This message tells us that the assert statement that we included may be triggered under a certain condition: (key And trip And state(0)) . The prefix ’memoryofdriver__’ is used by the theorem prover to refer to the memory element used for state(0) . The state ’off’ in the reference model corresponds to state(0) in the encoding that the finite-state machine compiler has used (and also to state(0) in the derived model). From this message we can isolate the problem to the following case statement (the line numbers follow the original code in architecture RTL):

case State **is**

when Off => **if** Key = ’1’ **then** State <= Armed; **end if** ;

when Armed => **if** Key = ’0’ **then** State <= Off;

elsif Trip = ’1’ **then** State <= Ringing;

end if ;

when Ringing => **if** Key = ’0’ **then** State <= Off; **end if** ;

end case ;

When we start in state Off and the two inputs are Trip = ’1’ and Key = ’1’ , we go to state Armed , and not to state Ringing . On the subsequent clock cycle we will go state Ringing , but only if Trip does not change. Since we have all seen "Mission Impossible" and the burglar who exits the top-secret computer room at the Pentagon at the exact moment the alarm is set, we know this is perfectly possible and the software is warning us of this fact. Continuing on, we get the following results from the theorem prover:

Prove (Axiom_ref => (Assert_ref => Assert_der))

Formula is NOT VALID

But is VALID under Assert Context of alarm-rtl3

We included the assert statement in the reference model (architecture RTL) but not in the derived model (architecture Gates). Now we are really mixed up: The assertion statement in the reference model says one thing, but the case statement in the reference model describes another. The theorem prover retorts: "The axioms of the reference model do not imply that the assertions of the reference model imply the assertions of the derived model." Translation: "These two architectures differ in some way." However, if we assume that the assertion is true (despite what the case statement says) then the formula is true. The prover is also saying: "Make up your mind, you cannot have it both ways." The prover goes on to explain the differences between the two representations:

***Difference is:

(Not state(1) And key And state(0) And trip)

There are 1 cubes and 4 literals in the complete equation

***Local Variable Assert_der is:

Not key Or Not state(0) Or Not trip

There are 3 cubes and 3 literals in the complete equation

***Local Variable Assert_ref is: 1

***Local Variable Axiom_ref is:

Not state(1) Or Not state(0)

There are 2 cubes and 2 literals in the complete equation

formulas to be proved: 8

formulas proved VALID: 7

formulas VALID under assert context of der.model: 1

Study these messages hard and you will see that the differences between the two models are consistent with our explanation.

13.8.4 Completing a Proof

To fix the problem we change the code as follows:

```
...

case State is

when Off => if Key = '1' then

if Trip = '1' then NextState <= Ringing;

else NextState <= Armed;

end if ;

end if ;

when Armed => if Key = '0' then NextState <= Off;

elsif Trip = '1' then NextState <= Ringing;

end if ;

when Ringing => if Key = '0' then NextState <= Off; end if ;

end case ;

...
```

This results in a minor change in the synthesized netlist,

```
g2: Inverter port map ( i => State(0), z => s1 );

g3: NAnd2 port map ( a => s1, b => State(1), z => s2 );

g4: Inverter port map ( i => s2, z => Ring );

g5: NAnd2 port map ( a => State(1), b => Key, z => s0 );

g6: NAnd3 port map ( a => Trip, b => s1, c => Key, z => s3 );

g7: NAnd2 port map ( a => s0, b => s3, z => NextState(1) );

g8: Inverter port map ( i => Key, z => NextState(0) );

state_ff_b0: DFF port map ( d => NextState(0), c => Clock, q => State(0), qn => open );

state_ff_b1: DFF port map ( d => NextState(1), c => Clock, q => State(1), qn => open );
```

Repeating the formal verification confirms and formally proves that the derived model will operate correctly. Strictly, we say that the operation of the derived model is implied by the reference model.

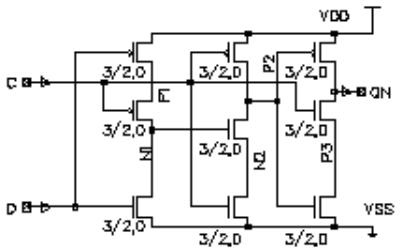
---

1. By one of the architects of the Compass VFormal software, Erich Marschner.

13.9 Switch-Level Simulation

The switch-level simulator is a more detailed level of simulation than we have discussed so far. Figure 13.1 shows the circuit schematic of a true single-phase flip-flop using true single-phase clocking (TSPC). TSPC has been used in some full-custom ICs to attempt to save area and power.

(a) (b)



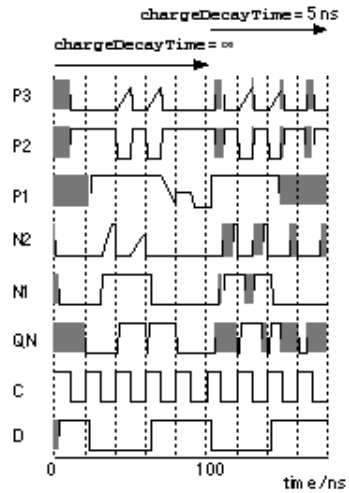


FIGURE 13.1 A TSPC (true single-phase clock) flip-flop. (a) The schematic (all devices are W/L = 3/2) created using a Compass schematic-entry tool. (b) The switch-level simulation results (Compass MixSim). The parameter chargeDecayTime sets the time after which the simulator sets an undriven node to an invalid logic level (shown shaded).

In a CMOS logic cell every node is driven to a strong ’1’ or a strong ’0’ . This is not true in TSPC, some nodes are left floating, so we ask the switch-level simulator to model charge leakage or charge decay (normally we need not worry about this low-level device issue). Figure 13.1 shows the waveform results. After five clock cycles, or 100 ns, we set the charge decay time to 5 ns. We notice two things. First, some of the node waveforms have values that are between logic ’0’ and ’1’ . Second, there are shaded areas on some node waveforms that represent the fact that, during the period of time marked, the logic value of the node is unknown. We can see that initially, before t = 100 ns (while we neglect the effects of charge decay), the circuit functions as a flip-flop. After t = 100 ns (when we begin including the effects of charge decay), the simulator tells us that this circuit may not function correctly. It is unlikely that all the charge would leak from a node in 5 ns, but we could not stop the clock in a design that uses a TSPC flip-flop. In ASIC design we do not use dangerous techniques such as TSPC and therefore do not normally need to use switch-level simulation.