3.1 INTRODUCTION

The number of ASICs designed increases every year. Advances in technology allow more transistors to be packed onto a single die which expands the applications where they can be used and accelerates development. Successful development of an ASIC depends on accurate modeling of its operation. Designing a circuit to be logically correct is simple. Producing an accurate timing model is critical to successful development. Current methodologies for generating accurate timing models for ASIC designs are described here.

Integrated circuits start as computer representations of a physical device. The designer’s goal is to model the device characteristics with sufficient accuracy that actual silicon behaves as the model predicts, assuming the computer simulations exercise the model in the same way the device is expected to operate in the real world. Modeling a device’s logical operation is relatively simple, and the translation from the model to the physical would be easy if it were not for the major difference introduced during fabrication: timing delays. The conversion of a logic statement to a model of its physical implementation is shown in Figure 3.1. The operation of the circuit
in Figure 3.1 is affected by the charging and discharging of the parasitic capacitor through resistors, both of which are inherent to silicon physical implementation. The stray capacitance and resistance can have such a great and deleterious effect that the physical operation is nothing like the simulated logical model.

A circuit's correct operation can be assured only if the timing of the simulated model is a close approximation of the final device. The accurate modeling of delay is of major importance. As process geometry shrinks and the number of transistors per die increases, the task of modeling the effects of parasitic capacitance and resistance makes it more challenging to correlate prelayout to postfabrication timing. Fortunately, CAD tools exist to accurately estimate delays before layout and extract the capacitance and resistance once layout is complete. Modeling estimated and extracted delays plays an important part in guaranteeing the timing and operation.

Any delay value used before the device is fabricated is merely an estimate. The four sources of delay are shown in Figure 3.2. Gate delay is determined by input slew rate and the inherent RC loading of the gate. Delay through a line depends on the RC load the gate drives. The fanout load simply increases the capacitance the driver must charge and discharge. Methodologies for predicting delay are well established. Gate delay is measured from fabricated test struc-
3.1 INTRODUCTION

Fig. 3.2 Components of Circuit Delay: Input Slew Rate, Inherent Gate Delay, Line Propagation Delay, Fanout Load

...tures tested at specific operating points. A transistor’s speed, and therefore the inherent delay in a gate, is affected by its dimensions, the supply voltage, doping levels, input slew rate, operating temperature, and fanout load. The data measured from the test structure provides a device model that extrapolates to estimate delay under all operating and fabrication conditions.

The delay due to signal lines may be modeled in two stages: prelayout and postlayout. In either case, the physical characteristics of fabricated traces are known, having been measured from test structures. In the absence of layout, the unknown elements that affect timing are the trace’s length, width, and surrounding signals. Figure 3.3 shows the parasitic capacitors seen by a metal trace. Parasitic capacitance is explored in detail in section 3.3. Before the layout is completed, any delay attributed to a signal line is an estimate based on probable length and width of the trace. Since the actual path is not known, the length is simply a guess based on the size of the overall circuit and the probability of placing the output of one gate close to the input terminals of the gates it drives. Another unknown aspect of the trace is the topology over which it passes. Once the layout is finished, the trace lines, and therefore their...
Delay, can be accurately modeled. The layout fixes their length and reveals what lies under the trace, whether it is substrate, transistors, or other layers.

Delay estimations are made in all stages of design: prelayout, synthesis, and postlayout. The most common methods used to estimate delays at all stages of the design cycle are explored.

3.2 PRELAYOUT TIMING

The design environment and methodology determine the accuracy and ease of modeling delays before the layout is finished. HDL languages, such as Verilog and VHDL, make it possible to add both gate and interconnect delays; however, except in situations where the layout is regular and known, such as in memories or decoders, the effects of delays due to interconnect are ignored until after synthesis or layout. The ease of including gate delays also depends on the type of model used. HDL modeling can also be done at two different levels: RTL and gate level.
3.2 PRELAYOUT TIMING

3.2.1 RTL vs. Gate-Level Timing

RTL code models a logic function without regard to its implementation, whereas gate-level code specifies the exact gates required. Both the RTL and gate-level code for the logic function shown in Figure 3.4 are given in Example 3.1.

Example 3.1

RTL Code
\[ out = ((a \& b) \mid !a); \]

Gate-Level Code
\[
\begin{align*}
\text{and}(a, b, s2); \\
\text{not}(a, s1); \\
\text{or}(s1, s2, out);
\end{align*}
\]

Modeling delay at the gate level is straightforward. The delay of each gate is found in the technology library. The appropriate delay can be assigned to every gate in the code and the propagation delay of signals estimated to provide a fairly accurate representation. However, manually implementing HDL code with delays for each gate is time consuming. At the prelayout stage, most design methodologies use synthesis to provide a gate model with delays while RTL code is used to model the circuit's behavior.

Fig. 3.4 Circuits Can Be Represented as RTL Code or Instantiated Gates
A clear method of accounting for delay is to determine the delay through each gate. The technology library already has delay information for every gate. Accurate modeling requires the assignment of the appropriate delay to each gate as described below. Estimating the delay of the RTL code is more difficult because of its level of abstraction. Until synthesis is complete, there is no straightforward way to correlate RTL code to actual gate delays.

The level of coding used affects the delays that can be modeled. Generally, RTL code is used to determine correct logical operation without regard for delays. A design at the gate level not only checks for correct operation, it also ensures that delays meet the required timing. Most designs start with an RTL code, then use synthesis to generate the gates needed to verify timing. Furthermore, few designs start at the gate level because the simulations, especially when timing is included, are very slow. Design at the RTL level offers a fast method to ensure that the logic is properly implemented. Synthesis then converts the design to gates that include delays from gates, estimated routing, and fanout.

### 3.2.2 Timing in RTL Code

Although it is impractical to assign delays to individual lines of RTL code, it is feasible to assign delays to entire modules. In RTL code, timing should be applied to any module or port that has a known response such as:

- Bus models
- Memories
- I/O ports
- Setup and hold times

A high-level system is shown in Figure 3.5. Each block is implemented as RTL. The RAM and the EPROM will not be synthesized. They are both modeled as an array of memory indexed by the address. The processor comes from a vendor’s library. Its model reflects only the bus transactions that take place. The address-
decode and low-speed I/O port will be synthesized and include any logic and flip-flops needed to perform their functions. Timing is important in the system simulation. At the RTL level, it is possible to see if the processor bus timing matches the RAM and EPROM timing. It can be determined if the decoder has too much delay or if the read/write timing of the I/O port meshes with the processor’s requirements. The timing response of each block can be added to the model.

The read timing of the RAM is given in Figure 3.6. When the RTL model detects a read cycle, it can instantaneously get the data from its memory array and present it on the bus, but a fast response does not correspond to reality. The delay, shown in Figure 3.6 as \( T_{\text{read}} \) must be implemented in the model to reflect the time actually needed for the RAM to access and present valid data. The
response time of the address decode cannot be instantaneous, but should reflect a delay based on the maximum delay it can have and still work in the system. The I/O port also needs bus timing to match the processor’s characteristics. The processor model comes from the vendor with timing that matches the processor’s real operation. The processor cycle time provides a check of the timing of all the other blocks. If a block meets the bus cycle time, it will work when fabricated.

A snippet of Verilog code, shown in Example 3.2, demonstrates how to implement the \( T_{vavd} \) and \( T_{s2z} \) delays in the memory model.

**Example 3.2**

```verilog
1. `define Tavd 10 // data delay out of memory
2. `define Ts2z 5   // delay of deselect to tristate
3. module RAM (addr, data, sel, rw);
4.   input [15:0] addr;
5.   inout [15:0] data;
6.   input   sel, rw;
7.   reg [15:0] mem_array [0:65536], data_internal;
8.   // data bus tristate. Bi-directional.
9.   assign #Ts2z data = (sel) ? data_internal : 16'bz;
```
3.2 PRELAYOUT TIMING

Example 3.2 (Continued)

Example 3.3

Assign #5 sel = address15 | address16 | address17;

The output, sel, is simply the OR of the inputs address15, address16, and address17. Logically, whenever one of the address signals goes high, sel goes high; however, delay changes that fundamental assumption slightly. The relationship between the input and the output signals is shown in Figure 3.7.
At 20ns, each input sequentially goes high for 3ns. Each input stays high for less time than the specified delay of 5. The output does not change because the delay is inertial and no input is high longer than the delay. At 50ns, address15 goes high for 6ns. After the input signal has been high for 5ns, the output responds and produces a pulse 6ns wide. At 70ns, both address16 and address17 go high for 3ns, but they are coincident and do not satisfy the inertial delay requirement, so the output does not change. At 90ns, a 3ns-wide pulse on address16 overlaps a 4ns-wide pulse from address17. The simulator interprets the overlap as meeting the delay requirement and a 7ns pulse occurs on the output.

Both the continuous assignment statement and the regular delay operate like combinatorial logic. Just as the delay through a gate suppresses glitches, so does the regular delay when used with a continuous assignment statement.

### 3.2.4 Delay in a Process Statement

Process statements, such as always or initial, support two types of assignment statements: blocking and nonblocking. The effects of
regular and intra-assignment delays on both types of statements are shown below.

A regular delay with a blocking assignment is given in Example 3.4.

**Example 3.4**

```verilog
always @(posedge clk)
begin
    #2 q1 = d;
    #2 q2 = d;
    #3 q3 = d;
end
```

A blocking statement means the simulator is blocked from moving on to any subsequent statement until the present one is complete. A regular delay delays evaluation of the inputs. The output signals that correspond to the process in Example 3.4 reveal exactly how a regular delay in a blocking statement works. Refer to Figure 3.8.

![Figure 3.8](image-url) **Fig. 3.8** Signals from the Blocking Assignment Statements with Regular Delays from Example 3.4
At 15ns, when the clock goes high, the simulator begins to execute the first statement, \#2 q1 = d. It interprets it to mean: after a delay of 2 time units, assign the current value of d to q1. At 17ns, d is zero, so q1 becomes zero. The simulator waits at the first statement until it is completely finished; then it moves to the second statement. The second statement means the same as the first: wait 2 time units, then assign the present value of d to q2. Waiting an additional 2 time units means the value of d at 19ns is assigned to q2. At 19ns, d is zero, so q2 is assigned a zero. The simulator stays at the second line until the assignment to q2 takes place; then it moves to execute the third statement. The last statement has a delay of 3 time units. Like the previous regular delays, the simulator waits the specified time, 3 time units, then assigns d to q3. In this case, d changed to a one at 21ns, so when the simulator evaluates d at 22ns, it assigns a one to q3. The important concepts to remember about regular delays and blocking assignments are:

- **Blocking Assignments**: Finish executing the current, including the delay, before moving to the next line.
- **Regular Delays**: Wait the specified delay before evaluating the input signals and determining the output signal.

### 3.2.5 Intra-Assignment Delays

The intra-assignment delay is defined as follows.

- **Intra-Assignment Delay**: Upon execution, immediately evaluate the input signals and determine the value of the output signal. Wait the specified delay before assigning the value to the output.

The regular delay waits, evaluates, then assigns. The intra-assignment delay evaluates, waits, then assigns. An intra-assignment delay with blocking assignment statements is given in Example 3.5 along with the process statement that generates the input signal d.
Example 3.5

The waveforms in Figure 3.9 show how the input is evaluated immediately upon execution. At 15ns when the clock goes high, the first statement immediately grabs the value of d. The positive edge of clk triggers both the evaluation of d and its transition. At clk's positive edge, d has not yet changed and does not change until after it is grabbed by the q3 = #2 d assignment statement. As a result, the value assigned to q3 is d's value just before the clock's rising edge. At 15ns, d's value is one, so a one is grabbed and 2 time units later, at 17ns, a one is assigned to q3. The execution of the first statement is done, so the execution of the second assignment statement begins. At 17ns, the value of d is zero, so a zero value is grabbed by the second assignment statement and is assigned 2 time units later to q4.

```vhdl
always @(posedge clk)
begin
    q3 = #2 d;
end

always @(posedge clk)
begin
    q4 = #2 d;
end

always @(posedge clk)
begin
    d <= ~d;
end
```

![Figure 3.9](image.png)

Fig. 3.9 Signals from Blocking Assignment Statements with Intra-Assignment Delays from Example 3.5
The operation of the intra-assignment delay is the same with nonblocking assignment statements, but the operation of a nonblocking statement does affect the output.

**Nonblocking assignment:** Execute all nonblocking statements simultaneously. Do not execute them serially.

Nonblocking assignment statements with intra-assignment delays are given in Example 3.6.

The waveforms in Figure 3.10 show the value of d at the rising edge at 15ns to be a one. Two nanoseconds later, the value of one is assigned to both q7 and q8. Since the delay is the same in both statements, both outputs change at the same time.

Of the delay and assignment types described above, continuous assignment statements with regular delays closely model combinatorial logic. However, nonblocking assignment statements with intra-assignment delays in an always block, controlled by the clock, exactly model a flop-flop or sequential logic.

The regular and intra-assignment delays with continuous blocking and nonblocking assignment statements allow the designer to put delays anyplace in the circuit; however, assigning delays to possibly every line of RTL code takes a lot of time. As discussed in the section on synthesis, detailed timing should wait until synthesis or layout is complete. At the RTL level, it is sufficient to describe delays the boundaries and not lower. A higher level of granularity saves time developing code and also provides enough timing information to do meaningful analysis until the synthesis is complete. All HDL languages can express delays between module inputs and outputs. The approach taken in Verilog is presented below.

### 3.2.6 The Verilog Specify Block

Delays between module input and output ports in Verilog are described in a specify block. Delay is not the only timing parameter that can be expressed in the specify block, but all timing checks occur only between module input or ioput ports and output or ioput.
Example 3.6

At the rising edge of clk, both assignment statements start execution. As shown in Figure 3.10, the positive edge of clk at 15ns causes both assignment statements to grab d’s value. With blocking statements, the second statement was not executed until the first was completed, but with nonblocking statements both immediately start execution. Since the delay is intra-assignment, the input signal is immediately evaluated; then both statements wait 2 time units before assigning the evaluated result to the outputs.

```verilog
always @(posedge clk)
begin
  q7 <= #2 d;
  q8 <= #2 d;
end
```

![Fig. 3.10](image)

Fig. 3.10  Signals from Nonblocking Assignment Statements with Intra-Assignment Delays from Example 3.6

ports. A RAM memory module is again used to show how timing delays and verification are easily implemented in the specify block. A synchronous memory easily displays what types of checks can be done. The memory has the following timing requirements as shown
in Table 3.1. Although numerous other parameters are needed to specify correct operation, these are sufficient to show how timing checks are defined. A diagram of the timing given in Table 3.1 is shown in Figure 3.11.

**Table 3.1 Synchronous Memory Timing Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Time (ns)</th>
<th>Parameter</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tclk_period</td>
<td>20</td>
<td>Tclk_data_valid</td>
<td>9</td>
</tr>
<tr>
<td>Tclk_high_min</td>
<td>9</td>
<td>T0_to_z</td>
<td>0.1</td>
</tr>
<tr>
<td>Tclk_low_min</td>
<td>7</td>
<td>Tz_to_1</td>
<td>0.3</td>
</tr>
<tr>
<td>Taddr_clk_setup</td>
<td>4</td>
<td>T1_to_z</td>
<td>0.1</td>
</tr>
<tr>
<td>Taddr_clk_hold</td>
<td>3</td>
<td>Tz_to_0</td>
<td>0.2</td>
</tr>
<tr>
<td>Tsel_clk_setup</td>
<td>4</td>
<td>Trise</td>
<td>0.5</td>
</tr>
<tr>
<td>Tsel_clk_hold</td>
<td>12</td>
<td>Tfall</td>
<td>0.3</td>
</tr>
</tbody>
</table>

![Figure 3.11 Partial Timing Diagram for a Synchronous RAM](image)
The RAM Verilog model is given below in Example 3.7.

**Example 3.7**

All the timing parameters listed in Table 3.1 are codified in the specify section. Each parameter is listed as a specparam. The parameter names of Table 3.1 directly correspond to the specparam names for easy correlation. The specparam statements span lines 30 through 43. The paths through the module are declared and described in terms of the timing parameters.

The memory model has only two paths with defined delays: clock to data and sel to data. The statement that defines the delay from the rising edge of the clock to valid data out is on lines 46 and 47 of Example 3.7. If the input signal, sel, is active, the delay from the rising edge of the clock to valid data out is defined in the parentheses following the equal sign. The delay from clock to valid data is Tclk_data_valid and the rise and fall times of internal signals are Trise and Tfall. The value for clk_data_valid is combined with the rise and fall times to provide more accurate delays.
The meaning of the numbers in parentheses, lines 46 through 50, is summarized in Example 3.8. The first term defines the time it takes for a signal to transition from zero to one, the second is the time to transition from one to zero, the third is zero to high impedance, high impedance to one, one to high impedance, and high impedance to zero.
3.2 PRELAYOUT TIMING

Example 3.8

(0 -> 1, 1 -> 0, 0 -> z, z -> 1, 1 -> z, z -> 0)

For the memory module, the delay from the positive edge of the clock to valid data only needs to have the zero to one and one to zero transitions defined because clock transitions do not cause the data bus to tristate.

The sel signal does cause the data bus to tristate, so the delay statements that define the relationship between the sel input and the data bus, lines 49 and 50 in Example 3.7, do not specify 0->1 or 1->0 delays. The statement on line 49 defines the time it takes to tristate the bus when sel goes inactive. Line 50 defines the time for the bus to leave tristate when sel becomes active.

Periods, pulse widths, and setup and hold times are also checked to see if they are in the specification. Lines 53 through 55 check the clock period, the time it is high and the time it is low. The setup and hold times of the address with respect to the clock’s rising edge are checked in lines 57 and 58. The setup and hold times of sel to the rising edge of clock are checked in line 60. The formats of the verification statements are explained in Example 3.9. The notifier toggles every time a violation is found. The always statement after the specify block, lines 65 through 68, is activated when the notifier toggles to report the time of the violation.

Example 3.9

$period (ref_event, limit, notifier);
$width (ref_event, limit, threshold, notifier);
$setup (data_event, ref_event, limit, notifier);
$hold (ref_event, data_event, limit, notifier);
$setuphold (ref_event, data_event, s_limit, h_limit, notifier);

The specify block in Verilog HDL provides a convenient and powerful way to add timing to modules. It offers the right level of timing for RTL code. More specific and involved timing is available after synthesis or layout, automatically through the use of CAD tools. Do not spend time at the RTL level adding too much detail. Simply take advantage of any model-level timing offered by the simulator.
3.2.7 Timing in Gate-Level Code

HDL languages can simulate a design on the gate level where every gate is instantiated in the net list. Most designs do not start at the gate level but as RTL code and then go through synthesis to get gates. Gate-level simulations are important when the gate and interconnect delays are included because they provide insight into how the fabricated chip will work. Gate-level simulations are discussed in the synthesis section in the context of the standard delay format (SDF) file.

3.2.8 Synthesis and Timing Constraints

The object of synthesis is to produce a logically correct circuit from the RTL that meets the timing requirements. The synthesized logic gates should be correct by construction; however, the timing may not meet specification. The key to synthesis and obtaining correct timing is to provide reasonable timing constraints. Various approaches to I/O constraints, accounting for routing estimates, feeding timing information to a floorplanner or simulator, and synthesis strategies are discussed in this chapter. Any mention of the synthesis tool or commands refers to Synopsys Design Compiler and its related modules. All other synthesis tools have similar capabilities.

3.2.8.1 Synthesis Priorities

A designer may have certain priorities when designing a circuit like low-power, high-speed, small-area, first-time manufacturability, etc. The synthesis tool also has priorities that may not coincide with the designer’s goal. Synthesis walks a balance between what are called design rule constraints (DRC) and optimization constraints. Design rules are imposed on synthesis by the physical limitations of the technology library chosen to implement the design. Design rules deal with the following elements:

1. Maximum fanout per gate
2. Maximum transition time of a signal
3. Maximum allowable capacitance per net
The designer specifies optimization constraints to control these elements:

1. Speed
2. Area

### 3.2.9 Design Rule Constraints

As the synthesis tool translates the RTL into gates, it tries to meet the speed and area constraints requested by the designer. If the library is pushed to its limits and the tool must choose between meeting an optimization goal or a design rule priority, it satisfies the DRCs first. DRCs must take precedence over optimization constraints because if the gates of a library cannot meet the designer’s requirements there is nothing that can be done except get a higher performance library (in terms of speed) or a library with smaller cells (in terms of area).

Although the library limitations play a role in forming the DRCs, the designer can also set limits on the library by specifying maximum fanout, transition, and capacitance to provide margin in the design. The designer must be sure that the DRCs are consistent for the entire design by propagating all user-set limits to all levels through appropriate use of .synopsys_dc.setup files. When setting DRCs, first consult the library to understand its limitations. Even if the designer chooses to use the same limits specified by the library, put them in the synopsys_dc.setup file that pertains to the design, so there are no questions what the limits are and where they are applied.

The following Design Compiler commands set DRC limits.

- **set_max_fanout**: Every input pin of every gate of the library has a fanout-load attribute. The sum of all fanout loads connected to an output cannot exceed the max_fanout limit. The command limits only the number of gates driven by any given output. Loading from wire capacitance is not controlled with this command.
**set_max_transition:** The transition time is the amount of time it takes to charge or discharge a node. It is a product of the signal-line capacitance and resistance. The command set_max_transition watches the RC delay on a wire. In an effort to stay below the max_transition limit, the synthesis tool may increase the drive capacity of a gate to better swing the load or limit the capacitance and resistance by setting constraints that can be passed on to the floorplanner. The characteristics of the wire, such as area, capacitance, and resistance, are found in the wire-load model.

**set_max_capacitance:** There are two components to a load on a net: fanout (other gates) and interconnect capacitance. The command set_max_capacitance checks to see that no gate drives more capacitance than the limit whether the source be interconnect or gate capacitance. There is no direct correlation between the command and net delay, simply between the command and capacitance. The wire-load model details the capacitance of a wire.

The three constraints mentioned above must be used in conjunction to ensure that the limits of the library are not exceeded. In the case where the library constraints do not match the limits set by the designer, the synthesis tool will meet the more restrictive value.

### 3.2.10 Optimization Constraints

After the DRCs are met, the synthesis tool works on optimizing the design. The most important optimization constraint is speed. The synthesis tool uses an internal static timing analyzer to determine if a path meets the required time. Static timing analysis (STA) is described more fully in the next section; however, in a nutshell, it sums up the delays of every element in a path to see if the total delay is faster or slower than required. The delay is measured from one sequential element to the next. A sequential element is considered to be a flip-flop or a latch. A more precise definition of a path is from an output pin to an input pin with a setup-and-hold-time
3.2 PRELAYOUT TIMING

requirement. Synthesis and STA work best on synchronous designs. There are techniques to deal with asynchronous circuits; however, if it is possible to design the circuit to be synchronous, it will fit into the modern ASIC flow with fewer exceptions that need to be manually checked.

There is another design practice, in addition to synchronous design, that enhances the use of synthesis and STA. The static timing analyzer in the synthesis tool considers the clock tree to be ideal which means there is no delay between the clock source and the input of any gate. In a design where the clock signal goes directly from the clock tree to the gates, its operation is nearly ideal. Any design technique, such as gated clocks, that places delays in the clock’s path will not work unless the amount of delay in the clock is quantified. It is possible to use the clock skew parameter to account for the delay in the clock, but it must include both the skew of the tree and the delay through gates. The clock delay through gates is not automatically measured, so it may be a difficult figure to arrive at. It is a good design practice to not gate the clock.

The designer can control the synthesized speed of the circuit with commands explained below.

- **create_clock**: At a minimum, the synthesis tool must know the clock’s period and duty cycle. The clock sets the time allowed for signals to propagate between sequential elements. The create_clock command also specifies clock skew.

- **set_input_delay**: The delay of the input of a module is assumed to be zero. The circuit, shown in Figure 3.12, has four inputs and is considered a module. Two inputs go directly to flip-flops while the other two inputs go through gates before they reach a flip-flop. The delay time for input a to reach the flip-flop is input_delay. If the set_input_delay command specifies the input_delay as 2ns, then the synthesis tool measures the delay of a and b as 2ns and if necessary modifies the design appropriately to still work at speed. The delay of input c or d is: input_delay + and-gate delay + or-gate delay. The value of input_delay is added to the gate delays to arrive at the final
speed of the path. If there is a lot of input_delay, the synthesis tool chooses faster gates to maintain the overall speed specified by the designer.

**set_output_delay**: The delay out of a module can be increased by the amount specified by set_output_delay. The module shown in Figure 3.13 has two output signals. The delay of

![Diagram](image1)

**Fig. 3.12** The set_input_delay Command Adds Additional Delay to Module Input Times

![Diagram](image2)

**Fig. 3.13** The set_output_delay Command Adds Additional Delay to Module Output Times
out1 is: flip-flop propagation delay + output_delay. If the set_output_delay command sets out Delay to 5ns, the delay of out1 is 5ns longer than the propagation delay of a flip-flop. The delay of out2 is: flip-flop propagation delay + [maximum of (and-gate or or-gate delay)] + and-gate delay + output_delay. Once again, the output_delay adds to the circuit’s inherent delays.

set_max_delay: Timing constraints can be placed on asynchronous paths with set_max_delay and set_min_delay. The values set by these two commands determine the time allowed to propagate through a path not controlled by a clock.

set_min_delay: Refer to set_max_delay.

set_max_area: The area constraint is set by a single command. If an area is specified, the synthesis tool will try to keep the area of both the gates and the wires under the max_area limit. The area of the wires can only be estimated if it is specified in the wire-load model.

Once the design rule and optimization constraints are specified, the synthesis tool works to find the correct gates to implement the logic functions specified in the RTL code with the timing specified by the designer. Timing in ASIC standard cell circuits cannot be fully understood without knowing the source of gate and wire delays.

3.2.11 Gate and Wire-Load Models

As gates are chosen by the synthesis tool to implement the logic functions described in the RTL, the synthesis tool uses an internal static timing analyzer to add up gate and wire delays to see if their total stays within the timing constraints. The delays are completely dependent on the process technology of the library chosen for fabrication. The vendor provides the delay numbers for both gate and wire-load models.

3.2.11.1 Gate Models  Every gate available to the synthesis tool must be described as a library model. A sample of a library that con-
tains only an AND-gate is given in Example 3.10. The description of the AND2 cell provides all the information the synthesis tool needs to determine if it can meet timing and area requirements. The area of the cell is given on line 3. Each input pin, lines 6 and 11, is described with its associated capacitance, lines 8 and 13, so the synthesis tool can calculate total fanout loads for the driving gates. The output is described in terms of the logic function it performs, line 18, in addition to the response of the output with respect to each input. The timing response of the output with respect to input A is given in lines 20 through 27, and with respect to input B in lines 28 through 36. The most important timing figure is the propagation delay for rising and falling transitions as controlled by each input, which is given in lines 20 and 21 and 29 and 30. The output rise and fall times and slopes are given along with the output resistance.

Example 3.10

```plaintext
1. library (proc_35) {
2.   date: "September 29, 2001"
3.   revision: 1.9
4.   cell(AND2) {
5.     area: 3
6.     pin(A) {
7.       Direction: input
8.       Capacitance: 1.2
9.       fanout_load: 1.0
10.    }
11.   pin(B) {
12.     Direction: input
13.     Capacitance: 1.2
14.     fanout_load: 1.0
15.    }
16.   pin(Z) {
17.     Direction: output
18.     Function: "AB"
19.     Timing(): {
20.       intrinsic_rise: 1.38
21.       intrinsic_fall: 0.97
22.       rise_resistance: 1.00
23.       fall_resistance: 1.00
24.       slope_rise: 1.00
25.       slope_fall: 1.00
26.       related_pin: "A"
27.     }
28.     Timing(): {
29.       intrinsic_rise: 1.38
30.       intrinsic_fall: 0.97
31.       rise_resistance: 1.00
32.       fall_resistance: 1.00
33.       slope_rise: 1.00
34.    }
35. }
```
3.2 PRELAYOUT TIMING

3.2.11.2 Wire-Load Models  The synthesis tool estimates wire delays using a wire-load model that relates a net’s estimated length to estimated capacitance and resistance. The manual calculation of the characteristics of a line is fully described in section 3.3. The synthesis tool uses the same techniques to find the RC delay of each net. There is a statistical aspect of the wire delay calculation. The actual length of each net is unknown to the synthesis tool; however, it makes a guess using statistics of routing from the reference design. Based on the statistical estimate of length, it calculates area, capacitance, and resistance. The delays determined using vendor wire-load models are inaccurate because the model is design dependent. If your design is not similar to the reference design used to make the wire-load model, there is significant error; however, the estimated delay decreases the number of synthesis iterations because estimated delay is better than ignoring it altogether.

Fortunately, more accurate wire-load models can be generated specifically for a given design. As soon as the RTL code is complete, the design can be synthesized and given to a floorplanner, then a place-and-route tool. The information from the preliminary route is fed back into the synthesis tool to make custom wire-load models that are much more accurate than the vendor-supplied models because they are design specific. The most accurate wire-load models are available after the place-and-route procedure once each wire’s exact dimensions are known. A wire-load model is shown in Example 3.11.

Example 3.11

```plaintext
Wire-load ("16x16") {
  Resistance : 0.1 ;
  Capacitance : 1.85;
}
```
Another approach to compensating for inaccurate wire-load models is to synthesize to a faster clock than the design will actually use. The synthesis tool chooses gates capable of driving larger loads, so when the accurate delays are fed back to the simulator after layout, the extra speed is used up in driving the lines. Another technique is to overestimate the capacitance values of the gates in the library so the synthesis tool chooses gates with extra drive capacity. The problem with any approach based on deliberate overdesign is that the area is larger than it may have to be and the amount of overcompensation, whether it be in time or capacitance, is merely a guess. The best approach is to have the most accurate models possible, which for wire-load models means that the data from an early floorplan should be used to develop accurate wire-load models.

### 3.2.12 The Synthesis Flow

By now it is clear that synthesis is a key step to obtaining the correct timing in ASIC standard cell design, but it is not the only step. Synthesis selects the gates used to implement the logic functions, but they are fashioned into the final form for fabrication by a floorplanner, and place-and-route tools. The process of converting RTL code into final layout is an iterative process. The major steps are listed below with emphasis on how the major tools interact.

1. Synthesize using vendor library and statistical wire-load models.
2. Write out timing constraint information (SDF) from the synthesis tool to be used by the floorplanner.
3. Using a timing driven floorplanner, plan the overall placement that meets the timing constraints from synthesis.
4. Place and route the design.
5. Extract cluster values (PDEF), delay values (SDF), and parasitic estimated values (RC). Feed the information to the floorplanner.
6. Create wire-load models using the information from place and route.
7. Back annotate the wire-load models into the synthesis tool (or a stand-alone static timing analyzer). Analyze the design to see if it meets timing requirements.
8. If the timing is close, use the reoptimize_design command to fix the few problems that exist. Generate new constraint information; then go to step 3 when done.
9. If the timing is not close, use the new wire-load models to synthesize again. Generate new constraints and return to step 3.
10. If the timing has plenty of slack, do the final place and route. Go to step 9.
11. If the timing is perfect after the final floorplan place-and-route iteration, the design is done. Otherwise fix the few minor problems that exist with the in-place optimize option of the synthesis tool.
12. Write out final delay and parasitic values for use in a static timing analyzer or in RTL-gate simulations as a final verification that the correct timing was achieved.

There is a tremendous amount of communication between the synthesis tool, the floorplanner, the place-and-route tool, static timing analyzers, and even the RTL simulator. The information sent from each tool helps the next tool in the process do its job better. Each iteration brings the design closer to the correct timing which is verified with either a static timing analyzer or RTL-gate simulations with full-timing back annotation.

Three common file formats pass the information between the tools: physical data exchange format (PDEF), standard delay format (SDF), and resistance/load scripts. Each is described below.
**Physical Data Exchange Format (PDEF):** The PDEF file contains information about the clustering of cells. The synthesis tool determines which cells should be close to each other (in a cluster) based on how the RTL file is organized. Since most designers partition their designs based on logic functions, the synthesis tool also groups logically. Once the floorplanner gets the netlist, it places the cells together based on timing or routing considerations. It generates a PDEF file based on physical placement that may not be anything like the logical groupings generated by the synthesis tool. A typical PDEF file is shown in Example 3.12.

**Example 3.12**

```plaintext
(CLUSTERFILE
 (PDEFVERSION "2.0")
 (DESIGN "top")
 (DATE "October 29, 2001")
 (VENDOR "Intrinsix")
 (DIVIDER /)
 (CLUSTER
   (NAME "MultB1")
   (X_BOUNDS 0.0   150.0)
   (Y_BOUNDS 0.0   163.0)
     (NAME "MultSub1")
     (X_BOUNDS 0.0   25.6)
     (Y_BOUNDS 0.0   89.3)
     (CELL (NAME U24/U78)   (LOC 3.8   16))
     (CELL (NAME U24/U45)   (LOC 16.0   46.5))
   (NAME "MultSub2")
   (X_BOUNDS 25.6   78.8)
   (Y_BOUNDS 0.0   89.3)
   (CELL (NAME U55/U14)   (LOC 30.0   42.7))
   (CELL (NAME U55/U83)   (LOC 51.0   46.5))
 )
 (NAME "MultB2")
 (X_BOUNDS 150   204.0)
 (Y_BOUNDS 0.0   79.0)
   (NAME "Ncs1")
   (X_BOUNDS 154.2   36.0)
   (Y_BOUNDS 185.0   0.0)
   (CELL (NAME U47/U64)   (LOC 153.8   24.0))
   (CELL (NAME U86/U37)   (LOC 167.0   28.2))
 )
)
```
3.2 Prelayout Timing

**Standard Delay Format (SDF):** The standard delay format file specifies delays. It is a case-sensitive format. The synthesis tool uses the SDF file to pass timing constraints to the floorplanner, an action known as forward-annotation. It uses the PATHCONSTRAINT parameter to tell the floorplanner the amount of propagation delay allowed for critical paths. The format of the PATHCONSTRAINT statement is given in Example 3.13.

**Example 3.13**

```
(PATHCONSTRAINT port_start [intermediate_node, ...] port_end (rise time) (fall time))
```

A simple SDF constraint file for the circuit shown in Figure 3.14 is shown in Example 3.14. The three highlighted paths are described.

![Circuit](image)

**Fig. 3.14** Circuit Corresponding to the SDF Constraint File of Example 3.14

**Example 3.14**

```
(DelayFile
 // Start of the sdf header. This file contains all typical data
 (SDFVERSION "1.0")
 (Design "test")
```
The same file format passes delay information from the synthesis tool to the RTL simulator and from the floorplanner/router to synthesis or RTL. The format can define the delays across a module, gates, or interconnect. Timing for setup, hold, setuphold, skew, width, and period are also valid parameters. Delays can also be specified to be absolute or incremental. Most HDL simulators use a subset of the SDF parameters. The designer does not need to do anything with the SDF file. The simulator accepts and assigns the delays using built-in system tasks. For Verilog, the command to read an SDF file is `$sdf_annotate`. The user can specify if minimum, typical, or maximum timing values are extracted from the SDF file and can set a scale factor if desirable.
The SDF file for the circuit shown in Figure 3.15 is given in Example 3.15. It includes the most common parameters used by RTL simulators. Each construct is also described.

**Example 3.15**

```
1. (DELAYFILE
2.  // Start of the sdf header.
3.  // This file contains all typical data.
4.  (SDFVERSION "1.0")
5.  (DESIGN "test")
6.  (DATE "Monday January 30 08:30:33 PST 1999")
7.  (VENDOR "Intrinsix Corp.")
8.  (PROGRAM "delay_find")
9.  (VERSION "3.6")
10. (DIVIDER /)
11. (VOLTAGE 5.0:5.0:5.0)
12. (PROCESS "typical")
13. (TEMPERATURE 85:85:85)
14. (TIMESCALE 1ns)
15. // description of interconnect delays.
```
Example 3.15 (Continued)

16. (CELL
17.    (CELLTYPE "test")
18.    (INSTANCE bk3)
19.    (DELAY
20.    (ABSOLUTE
21.    (INTERCONNECT P1/z s1/c1/i (.163:.163:.163) (.147:.147:.147))
22.    (INTERCONNECT P1/z s2/c4/i2 (.152:.152:.152) (.139:.139:.139))
23.    (INTERCONNECT P2/z s1/c2/clk (.102:.102:.102) (.099:.099:.099))
25.    (INTERCONNECT P3/z s2/c3/i2 (.178:.178:.178) (.165:.165:.165))
26.    (INTERCONNECT P3/z s2/c4/i1 (.176:.176:.176) (.163:.163:.163))
27.    (INTERCONNECT s1/c1/z s1/c2/d (.184:.184:.184) (.175:.175:.175))
28.    (INTERCONNECT s1/c2/q s2/c3/i1 (.171:.171:.171) (.163:.163:.163))
29.    (INTERCONNECT s2/c3/z s2/c5/i1 (.185:.185:.185) (.173:.173:.173))
30.    (INTERCONNECT s2/c4/z s2/c5/i2 (.146:.146:.146) (.137:.137:.137))
31.    (INTERCONNECT s2/c5/z s2/c6/d (.189:.189:.189) (.176:.176:.176))
32.    (INTERCONNECT s2/c6/q P4/i (.169:.169:.169) (.155:.155:.155))
34. )))
35. // The intrinsic delays of each cell used in the design. Equivalent to gate delays.
36. (CELL
37.    (CELLTYPE "INV")
38.    (INSTANCE s1/c1)
39.    (DELAY
40.    (ABSOLUTE
41.    (IOPATH i z (.323:.323:.323) (.311:.311:.311))
42. )))
43. (CELL
44.    (CELLTYPE "DFF")
45.    (INSTANCE s1/c2)
46.    (DELAY
47.    (ABSOLUTE
48.    (IOPATH clk q (.417:.417:.417) (.404:.404:.404))
49. ))
50. (TIMINGCHECK
51.    (SETUP D (posedge clk) (.260))
52.    (HOLD D (posedge clk) (.000))
53.    (WIDTH (negedge clk) (1.60))
54.    (WIDTH (posedge clk) (1.73))
55. )
56. (CELL
Example 3.15 (Continued)

57. (CELLTYPE "DFF1")
58. (INSTANCE s2/c6)
59. (DELAY
60. (ABSOLUTE
61. (IOPATH clk q (.379:.379:.379) (.367:.367:.367))
62. (IOPATH clk qn (.421:.421:.421) (.414:.414:.414))
63. ))
64. (TIMINGCHECK
65.  (SETUP D (posedge clk) (.302))
66.  (HOLD D (posedge clk) (.000))
67.  (WIDTH (negedge clk) (1.64))
68.  (WIDTH (posedge clk) (1.79))
69. ))
70. (CELL
71. (CELLTYPE "OR2")
72. (INSTANCE s2/c3)
73. (DELAY
74. (ABSOLUTE
75. (IOPATH i1 z (.304:.304:.304) (.298:.298:.298))
76. (IOPATH i2 z (.304:.304:.304) (.298:.298:.298))
77. )))
78. (CELL
79. (CELLTYPE "OR2")
80. (INSTANCE s2/c5)
81. (DELAY
82. (ABSOLUTE
83. (IOPATH i1 z (.304:.304:.304) (.298:.298:.298))
84. (IOPATH i2 z (.304:.304:.304) (.298:.298:.298))
85. )))
86. (CELL
87. (CELLTYPE "AND2")
88. (INSTANCE s2/c4)
89. (DELAY
90. (ABSOLUTE
91. (IOPATH i1 z (.337:.337:.337) (.325:.325:.325))
92. (IOPATH i2 z (.337:.337:.337) (.325:.325:.325))
93. )))

The delay times are specified as triplets. The first group of three numbers is the rise time. The second is the fall time. The three numbers, separated by colons, are supposed to represent (minimum:typical:maximum) delays; however, many tools write only one case at a time. In the SDF file example above, all three numbers in the parentheses are the same. The header specifies that they are the typical case. Completely new files would need to be written for the minimum and maximum cases.

The time scale, line 13, is also given in the header as nanoseconds. For this process, most of the gates have a propagation delay of...
between 0.250 and 0.45ns. In all cases for this example, the delay
times are declared as absolute. If the delays had been incremental,
the final delay would be the sum of a base time and the increment
specified in the SDF file. In this case, there are no base time and no
increments. The timing number states how long it takes for a signal
to propagate through a line or gate.

### 3.2.12.1 INTERCONNECT

The interconnect command is used to specify interconnect delay. After synthesis, but before routing, the interconnect delays are estimates based on wire-load models. After place and route, the exact dimensions of each line are extracted and the RC delay calculated. The interconnect statement on line 20 describes the delay from the pad input P1 to the input of the inverter C1. The signal propagation delay is 0.163ns for a rising edge and 0.147ns for a falling edge. The format for the statement is:

```
(INTERCONNECT port_start port_end (rise times) (fall times))
```

### 3.2.12.2 IO PATH

The ioport statement specifies input to output cell delays. The path can be from any input/ioput port to any legal output/ioput port. Lines 35 through 41 describe the propagation delay through the inverter C1. The propagation delay of a rising edge is 0.323ns and a falling edge is 0.311ns. The format for the statement is:

```
(IOPATH input_port output_port (rise times) (fall times))
```

### 3.2.12.3 SETUP, HOLD, SETUPHOLD, WIDTH, PERIOD

These statements are timing checks of sequential devices like flip-flop or latches. The timing checks must be specified in the TIMINGCHECK part of the cell definition. The purpose of each timing check matches its name. The definition of the D flip-flop (DFF), lines 42 through 54, describes the clk to q propagation delay, on line 47, as 0.417ns for a rising edge and 0.404ns for a falling edge. It also provides timing checks for the data setup and hold times on lines 50 and 51. Minimum pulse widths are also specified for the clock on lines 52 and 53. Line 52 requires the clock to be low for at least 1.6ns. The minimum
3.2 Prelayout Timing

Clock high time of 1.73ns is given on line 53. The formats for the timing check statements are:

```plaintext
(TIMINGCHECK
  (SETUP data_signal reference_signal (time))
  (HOLD data_signal reference_signal (time))
  (WIDTH (posedge/negedge signal) (time))
  (SETUPHOLD data_signal reference_signal (setup time) (hold time))
  (PERIOD (posedge/negedge signal) (period))
)
```

3.2.12.4 Resistance/Load Scripts The floorplanner sends line resistance and load scripts to the synthesis tool to make new wire-load models. The format is shown in Example 3.16.

Example 3.16

```plaintext
Set_resistance 0.1569 "aout"
Set_load 0.673 "aout"
```

3.2.13 Synthesis Tips

Some general suggestions can help the synthesis process converge to the correct timing faster.

1. **Are the constraints realistic?** Understand the limitations of the cells in the technology library even before starting RTL coding. Make a short table of common gates with propagation delays for various loads. Does the RTL code expect too much out of the gates? Does pipelining need to be used to make the target speed realistic? Test the synthesis process on small sections of the code to make sure it can meet the speed expectations before the entire design is done and it is discovered that the library is not up to the challenge.

2. **Have false paths and multicycle paths been identified?** A false path is a path that does not need to meet timing requirements. Signals that activate test modes are good examples of false paths. A multicycle path is one that is not expected to complete its computation in a single cycle. Identification of both types of paths eliminates unnecessary warnings from the tools.
3. **Have minimum and maximum delays been set for all asynchronous paths?** As mentioned earlier, synchronous designs are better for the synthesis design methodology. If a path simply cannot be made synchronous, be sure that both minimum and maximum delays are specified to set boundaries on its operation.

4. **Do the timing violations require a change in architecture?** Are the timing violations so egregious they cannot be solved with minor fixes or more iterations through the design cycle? Look at timing violations with an eye on architecture. Always ask if an architectural change would make many timing problems go away.

5. **Can minor timing problems be solved with incremental compiles?** At times, the synthesis, floorplan, place, and route cycle can be like the golpher game at the arcade—after hitting one golpher over the head with the bat, another three pop up. Sometimes doing another synthesis run to solve a minor problem can result in problems in several other paths. When the timing is close, use the incremental compile option to iron out any remaining problems.

### 3.2.14 Back Annotation to Gate-Level RTL

Delays can be applied to the RTL model at two important stages in the design process. After synthesis, it is possible to add precise fanout and gate delays to the RTL model along with estimated interconnect delays. Once a design is through floorplanning, placement, and routing, accurate interconnect delays can be added to the gate and fanout delays known from synthesis. Gate-level timing can be tested at either point: after synthesis or layout. Usually the vendor sign-off procedure requires verification with postlayout, back-annotated timing. Two timing verification approaches are common: static timing analysis and back-annotated gate-level simulations.
3.3 POSTLAYOUT TIMING

The most accurate simulations of a device are those that use delays extracted from the actual layout. Synthesis provides accurate fanout figures, but only estimated propagation delays for signals. Once the layout is complete, the exact dimensions of every line are known and can be used to determine the RC loading and propagation delay.

Even extracted delays are only estimates of the delays of the fabricated device; however, at the postlayout state, the accuracy of the model is sufficient to ensure first-time functional silicon. Two generally accepted methods for verifying the postlayout model have been previously discussed: static timing analysis and gate-level RTL simulations with back-annotation from an SDF file. The delays extracted from layout provide more accuracy than any prelayout estimates; however, there are a few instances where manual verification of the extracted delay merits attention.

3.3.1 Manual Line-Propagation Delay Calculations

Interconnect can be modeled at several different levels each with decreasing amounts of accuracy. Complete 3D modeling is the most accurate, followed by 2.5D, transmission lines, and finally lumped RC analysis [Chiprout98]. Except at ultrahigh frequencies of 1GHz or above, transmission lines and lumped components provide sufficient accuracy. Most standard cell methodologies do not require any manual intervention to correctly determine interconnect delays. There are some situations, however, when it may be necessary. Generally these conditions do not occur in a standard cell methodology. If there is any deviation from the standard design flow sanctioned by the vendors or if the design is known to push the process limits, you may want to check for the following conditions:

1. Signal lines in polysilicon
2. Extremely long signal lines
Such cases will be rare; however, if they exist, it is worth the effort to manually verify the timing of a few lines. If the delay from the manual calculation is no more accurate than the delay already reported, there is no need to individually model any more lines.

There are extraction tools capable of determining the width and length of any line. If such tools are available, use them to get the information necessary to calculate the line resistance and capacitance as shown below. Except at ultrahigh frequencies, internal routing lines do not exhibit inductive characteristics, so inductance can safely be ignored in the model.

### 3.3.2 Signal-Line Capacitance Calculation

There are two components to line capacitance: plate and fringe. Capacitors form between a conductor and whatever lies underneath, whether it be the substrate or another conductor. Two conductors and the capacitors formed are shown in Figure 3.16. Capacitors form where the metal-2 line crosses over the metal-1 trace and where both traces cross over the substrate. The two types of capacitors that affect a trace are shown in Figure 3.17. A plate capacitor forms between the bottom of the trace and the underlying layer. A fringe capacitor exists between the sides of the conductor and whatever is underneath.

Total line capacitance is the sum of the capacitance of the bottom and both edges:

$$C_{\text{line}} = C_{\text{plate}} + 2C_{\text{fringe}}$$  \hspace{1cm} \text{(Eq. 3.1)}

Plate capacitance is calculated as follows:

$$C_{\text{plate}} = \frac{\varepsilon WL}{D}$$  \hspace{1cm} \text{(Eq. 3.2)}

where

$\varepsilon$ is the dielectric constant of the insulator (SiO2) between the trace and the substrate. A value for permittivity of free space, $\varepsilon_0$, is 8.85e-6 pf/um. The permittivity of SiO2 is 3.9*$\varepsilon_0$ or 3.45e-5 pf/um.
The fringe capacitance is more difficult to model, but a close approximation is to convert each edge of the trace to a half cylinder then combine them into a separate, complete cylinder. Figure 3.18 diagrams the conversion. The cylinder represents the capacitance contributed by both edges and replaces the $2C_{\text{fringe}}$ term in equation 3.1. The formula to calculate the capacitance of a cylindrical conductor is:

$$C_{\text{cylinder}} = \frac{2\pi \varepsilon d}{\ln(1 + (2D/d)\{1 + \sqrt{1 + (d/D)}\})}$$  \hspace{1cm} \text{(Eq. 3.3)}$$

where

- $d$ = diameter of cylinder
- $D$ = distance from the substrate
- $\varepsilon$ = dielectric constant of SiO2

\textbf{Fig. 3.16} Parasitic Capacitors Exist Between All Layers of a Process
Notice in Figure 3.18, the width of the rectangle is decreased by T/2 for the calculation of the plate capacitance. It would seem that its width should decrease by T since a T/2 slice was taken off each side; however, the cylinder represents all the edge capacitance plus a little bit of plate capacitance, so the rectangle width is reduced by a lesser amount. Refer to [GD85] for more in-depth information.

Many design rule specifications include values for both plate and fringe capacitance which are expressed as pf/um^2 and pf/um. If the values are not already available, calculate the plate capacitance as pf/um^2 and fringe capacitance as pf/um as follows:

\[
C_p = \frac{\varepsilon}{D} \quad \text{ (Eq. 3.4)}
\]

\[
C_c = \frac{2\pi\varepsilon}{\ln(1 + (2D/T)(1 + \sqrt{1 + (T/D)}))} \quad \text{ (Eq. 3.5)}
\]
The final equation to approximate the capacitance of a trace is:

\[ \text{Cline} = C_{pL}WL + C_{cL} \]  \hspace{1cm} (Eq. 3.6)

Minimum line widths and typical trace thickness are listed in Table 3.2 for an 0.8um process. Typical capacitance values for a
triple-layer metal process are given in Table 3.3. The dielectric constant for SiO2 is also given in Equation 3.7:

\[
\varepsilon_{\text{SiO2}} = 3.45 \times 10^{-5} \text{ pf/um} \quad \text{(Eq. 3.7)}
\]

The capacitances of several long lines for all of the above capacitance combinations are given in Table 3.4 below. The width is assumed to be the minimum width for the material listed.

<table>
<thead>
<tr>
<th>Table 3.2 Typical Metal Widths for 0.8um Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>T (um)</td>
</tr>
<tr>
<td>Poly</td>
</tr>
<tr>
<td>Metal 1</td>
</tr>
<tr>
<td>Metal 2</td>
</tr>
<tr>
<td>Metal 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3.3 Typical Capacitance Values for 0.8um Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>D (um)</td>
</tr>
<tr>
<td>Poly to Substrate</td>
</tr>
<tr>
<td>Metal 1 to Poly</td>
</tr>
<tr>
<td>Metal 1 to Substrate</td>
</tr>
<tr>
<td>Metal 1 to Diffusion</td>
</tr>
<tr>
<td>Metal 2 to Metal 1</td>
</tr>
<tr>
<td>Metal 2 to Poly</td>
</tr>
<tr>
<td>Metal 2 to Substrate</td>
</tr>
<tr>
<td>Metal 2 to Diffusion</td>
</tr>
<tr>
<td>Metal 3 to Metal 2</td>
</tr>
<tr>
<td>Metal 3 to Metal 1</td>
</tr>
<tr>
<td>Metal 3 to Poly</td>
</tr>
<tr>
<td>Metal 3 to Substrate</td>
</tr>
<tr>
<td>Metal 3 to Diffusion</td>
</tr>
</tbody>
</table>
Table 3.4 Capacitance of Long Lines

<table>
<thead>
<tr>
<th>Total Line Capacitance (pf)</th>
<th>Line Length (um)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>500</td>
</tr>
<tr>
<td>Poly to Substrate</td>
<td>0.076</td>
</tr>
<tr>
<td>Metal 1 to Poly</td>
<td>0.095</td>
</tr>
<tr>
<td>Metal 1 to Substrate</td>
<td>0.066</td>
</tr>
<tr>
<td>Metal 1 to Diffusion</td>
<td>0.095</td>
</tr>
<tr>
<td>Metal 2 to Metal 1</td>
<td>0.117</td>
</tr>
<tr>
<td>Metal 2 to Poly</td>
<td>0.072</td>
</tr>
<tr>
<td>Metal 2 to Substrate</td>
<td>0.063</td>
</tr>
<tr>
<td>Metal 2 to Diffusion</td>
<td>0.072</td>
</tr>
<tr>
<td>Metal 3 to Metal 2</td>
<td>0.128</td>
</tr>
<tr>
<td>Metal 3 to Metal 1</td>
<td>0.07</td>
</tr>
<tr>
<td>Metal 3 to Poly</td>
<td>0.058</td>
</tr>
<tr>
<td>Metal 3 to Substrate</td>
<td>0.054</td>
</tr>
<tr>
<td>Metal 3 to Diffusion</td>
<td>0.058</td>
</tr>
</tbody>
</table>

Table 3.4 shows the capacitance of a line over a single material. For example, in the metal-2-to-substrate capacitance, it is assumed that there are no poly or metal-1 lines between the substrate and the metal 2. The amount of capacitance depends on the amount of area overlap and which layers are involved. Assuming that a line runs over a single material is not an accurate assumption. Routing between cells leaves myriad lines crossing one another, over transistors and the substrate. A good extraction tool can determine exactly what lies under a line; however, if the line crosses lots of other layers, making a capacitance model can be difficult. If the tool can report only the width and length of a trace, but not the layers underneath, the capacitance value for the trace to the substrate may have to be used. If a quick visual check reveals the line is over a different layer some part of the time, the model accuracy can be improved.
For example, if the line is a very long metal-2 line with half of it over metal 1, use the metal-2-to-metal-1 capacitance value for 50% of the trace’s area and the metal-2-to-substrate value for the rest.

Also note that the capacitance scales linearly with the line length. A table, like Table 3.4, which contains relatively few entries, can be used to determine the capacitance of a line of any length by extrapolation.

### 3.3.3 Signal Line Resistance Calculation

The resistance of a line is measured in units called squares. The lines shown in Figure 3.19 have width $W$ and length $L$. The length of each line is subdivided into squares $W$ wide and $W$ long. There are ten squares total for the line in 3.19a. Note that the squares are

---

**Fig. 3.19** Line Resistance Is Measured in Squares
marked off along the direction of the flow of current. The line shown in 3.19b has 0.33 squares.

Each type of material used in semiconductors has a characteristic resistance per square. The resistance of a line is calculated by finding the total number of squares and multiplying by the ohms per square value. Resistance values for each layer are given in the process design rules. Typical resistance values for a 0.8um process are given in Table 3.5.

The resistances of the lines drawn in Figure 3.19 for different materials are given in Table 3.6.

A look at the resistance of the different materials reveals why lines laid out in polysilicon need additional attention. The capacitance of poly to substrate is about the same as the capacitance between metal 2 and metal 3 or metal 1 and metal 2; however, its resistance is 160 times greater than the resistance of metal 1 and 400 times that of metal 2 and metal 3. Lines of polysilicon have a high RC delay. If the CAD tools do not account for its RC characteristics, the simulation would be highly inaccurate. Fortunately, all

<table>
<thead>
<tr>
<th>Polysilicon</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metal 1</td>
<td>0.25</td>
</tr>
<tr>
<td>Metal 2</td>
<td>0.1</td>
</tr>
<tr>
<td>Metal 3</td>
<td>0.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Polysilicon</th>
<th>400</th>
<th>13.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metal 1</td>
<td>2.5</td>
<td>0.083</td>
</tr>
<tr>
<td>Metal 2</td>
<td>1.0</td>
<td>0.033</td>
</tr>
<tr>
<td>Metal 3</td>
<td>1.0</td>
<td>0.033</td>
</tr>
</tbody>
</table>
Routing in a standard cell process is done in metal. If any hand edits are made or if there are any deviations from a previously proven flow, check for poly routing. Special cells, that are more custom in nature, such as memory decoders, select lines in memories, or decoded buses may use poly as a routing layer to achieve higher density. If the timing of drop-in cells is not provided to include in the RTL model as described in section 3.2, do not neglect the RC delay of lines when characterizing the block's timing.

### 3.3.4 Signal Trace RC Delay Evaluation

Once the resistance and capacitance for a trace are known, the propagation delay can be determined. The easiest approach is to combine all the line resistance into a single resistor and all the capacitance into a single capacitor. Elementary circuit theory predicts the rise time from the 10% to the 90% voltage points of the lumped RC model to be $2.2RC$; however, simulations of a distributed RC line show much less delay. Breaking the line into at least three, and not more than ten, equal sections, as shown in Figure 3.20, provides a model that is

![Distributed RC Model](image)

**Fig. 3.20** Distributed RC Model Provides Greater Accuracy than Lumped Model
accurate to within 3% of a transmission line model. Simulations show that the 10%-90% delay of the distributed line approaches the RC time constant as the number of RC segments in the simulation increase [Wilnai71] and [Chiprout98].

The RC delay for lines of minimum width and various lengths are listed in Table 3.7. The capacitance in each case is that of the line to the substrate.

Polysilicon is the highest resistance routing material, so it results in the highest delays. Very few lines, if any, will be run in polysilicon, but those that are need a closer look. The delays are easily reduced by increasing the line width which in turn decreases the resistance. Although the capacitance also increases, the significant decrease in resistance results in lower propagation delays. Note that the delays in metal are minimal. They may be a bit higher if the line runs over closer layers like poly or other metal lines; however, for the most part the delay of a metal trace will not affect performance significantly. If the circuit is designed right on the edge of the process capabilities, long metal lines may need closer scrutiny. If there are several long lines on the device, simulate the distributed RC model of one of them to see if all need to be modeled more accurately.

## 3.4 ASIC SIGN-OFF CHECKLIST

Standard cell ASIC design is a partnership between the designer and the vendor who fabricates the device. The designer usually

<table>
<thead>
<tr>
<th>RC Time Constant (ns)</th>
<th>Line Length (um)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>500</td>
</tr>
<tr>
<td>Polysilicon</td>
<td>1.9</td>
</tr>
<tr>
<td>Metal 1</td>
<td>0.006</td>
</tr>
<tr>
<td>Metal 2</td>
<td>0.005</td>
</tr>
<tr>
<td>Metal 3</td>
<td>0.002</td>
</tr>
</tbody>
</table>
takes care of what is known as the front-end part of the design which consists of the functional specification through to postsynthesis gate-level simulations. The vendor’s responsibilities are to supply the technology library and to perform the layout. Once the layout is done, the vendor provides the designer with an extracted SDF file for final verification. If the extracted design passes final verification, there is a high probability that the design will work when fabricated.

A checklist details which tasks are to be done by whom and how to ensure, or verify, that the task has been done properly. Most checklists cover at least the following items.

### 3.4.1 Library Development

The vendor is responsible for development of the standard cell library. The library must be characterized so accurate gate and interconnect models can be used in simulation and STA. One consideration when selecting a library is the variety of gates it offers. The richer the selection of gates, the more flexibility it offers the synthesis tool. Variety refers to the different flavors of any given cell such as low-power, high-speed, small area, etc.

### 3.4.2 Functional Specification

Only the designer can write the functional specification. Only the designer knows what the application requires. The specification may be determined in large part by components of a library if there are special cells or intellectual property (IP) that meets the application needs.

### 3.4.3 RTL Coding

The designer develops the RTL code that implements the functional specification. Some RTL code is much easier to synthesize into stable, workable circuits. If the vendor has any coding guide-
lines that help the synthesis tool better utilize the library, they should be followed.

3.4.4 Simulations of RTL

The designer is responsible to verify that the RTL code correctly implements the functional specification. Simulations at the RTL level should be thorough because this is really the only place where correct function can efficiently be verified. Simulations at the gate level are much too slow to be complete and STA does not verify function, only timing.

3.4.5 Logic Synthesis

Synthesis is the responsibility of the designer; however, the vendor may have to supply information about the library as the synthesis tool tries to make decisions about which cells to use and how to estimate the wire-loads. The synthesis tool generates both forward and backward annotation files. The forward annotation provides constraints to timing-driven layout tools while the back-annotated files provide delay information to either a simulator, for gate-level simulations, or a static timing analyzer.

3.4.6 Test Insertion and ATPG

The insertion of flip-flops for partial or full scan is done by the designer as is automatic test program generation (ATPG).

3.4.7 Postsynthesis Gate-Level Simulation or Static Timing Analysis

The designer is responsible for verifying the synthesized gates for functional correctness and for estimated performance. Whether the verification is done with a simulator or a static timing analyzer, the wire-loads are only estimates. The gate delays come from the tech-
ology library and are accurate. The delays are provided from the synthesis tool via a standard delay format (SDF) file.

### 3.4.8 Floorplanning

Some designers have the tools and capabilities to perform their own floorplanning. Most do not and the task falls to the vendor. Floorplanning takes information from synthesis to group the cells to meet the timing performance. It feeds back more accurate wire-load models to the synthesis tool and it provides the framework for place and route.

### 3.4.9 Place and Route

Most vendors drive the tools that actually place the cells and run the wires between them. The end result is the actual layout that, if it is verified to work properly, is used to make masks for fabrication. The vendor extracts gate and interconnect delays to be used in final verification. The clock tree is also developed at the same time by the vendor. Some tools exist to measure the load on the tree and develop the branches in such a way to minimize skew. The vendor performs simulations on the extracted clock tree to ensure that it meets specification.

### 3.4.10 Final Verification of the Extracted Netlist

The designer is responsible for final verification of the extracted netlist. The extracted netlist offers the most accurate model of the device. The exact fanout of every gate is known along with the delay of every signal line. The method of final verification, whether it be by simulation or STA, is determined by the vendor. Some vendors do not accept STA alone as proof that the device will work and require extensive back annotated simulations. Other vendors use STA as the only final verification tool. Most designers use both: complete STA analysis plus a few strategic simulation vectors.
3.4.11 Mask Generation and Fabrication

Once final verification is complete, the vendor assumes the responsibility of everything connected with fabricating the device.

3.4.12 Testing

Once the device is fabricated, the vendor does some limited tests to ensure the process met specifications. Production uses the vectors developed by the designer.