

# **ECE 683**

---

## ***OSU DIGITAL CELL LIBRARY DOCUMENTATION***

MATTHEW BOLIN

REVISION 6.0  
12/05/2005

# ***TIMING CHARACTERIZATION***

# TABLE OF CONTENTS

<b>1. TIMING CHARACTERIZATION</b> -----	<b>3</b>
PURPOSE -----	3
DESIGN FLOW WITH NO TIMING-----	3
DESIGN FLOW WITH TIMING-----	4
<b>2. VERILOG DESIGN MODELS</b> -----	<b>5</b>
DESCRIPTION -----	5
BEHAVIORIAL MODELING-----	5
DATAFLOW MODELING -----	5
GATE LEVEL MODELING -----	5
<b>3. VERILOG DELAY MODELS</b> -----	<b>6</b>
DELAY MODELS OVERVIEW -----	6
VERILOG NOTATION SUMMARY-----	6
TIMESCALE-----	7
<i>DELAY TYPES</i> -----	7
CLASSIFY DELAY BY CONNECTION -----	7
<i>LUMPED DELAY</i> -----	7
<i>DISTRIBUTED DELAY</i> -----	8
<i>PATH DELAY (PIN to PIN DELAY)</i> -----	9
CLASSIFY DELAY BY RISE/FALL AND MIN/MAX-----	10
<i>RISE AND FALL TIMES</i> -----	10
<i>PROCESS DEPENDENT TRIADS</i> -----	11
<i>COMPLEX DELAYS</i> -----	11
<b>4. VERILOG SPECIFY BLOCK</b> -----	<b>12</b>
DESCRIPTION -----	12
SPECPARAMS -----	13
<i>DESCRIPTION</i> -----	13
<i>SCOPE</i> -----	13
<i>NOTATION</i> -----	13
PATH DELAYS -----	14
<i>PARALLEL CONNECTION</i> -----	14
<i>FULLY CONNECTED</i> -----	15
CONDITIONAL PATH DELAYS -----	15
POLARITY OF PATH DELAYS-----	16
TIMING CHECKS -----	16
<i>SETUP TIME</i> -----	17
<i>HOLD TIME</i> -----	17
<i>WIDTH</i> -----	17
<i>RECOVERY</i> -----	18
EXAMPLE OF A DFF -----	18
<b>5. STANDARD DELAY FORMAT</b> -----	<b>19</b>
DESCRIPTION -----	19
GENERATION AND USE -----	19
SDF FILE FORMAT -----	21
HEADER SECTION -----	21
HEADER SECTION DETAILS -----	22
<i>DIVIDER</i> -----	22
<i>TIMESCALE</i> -----	22
CELL SECTION -----	23

CELL SECTION DETAILED-----	24
<i>CELL TYPE</i> -----	24
<i>CELL INSTANCE</i> -----	24
<i>DELAY BLOCK</i> -----	24
<i>DELAY TYPE</i> -----	25
<i>DELAY DEFINITION</i> -----	25
IOPATH DEFINITION -----	25
PORT DEFINITION-----	25
<i>TIMING CHECK BLOCK</i> -----	26
<i>TIMING CHECKS</i> -----	26
SETUP TIME -----	26
HOLD TIME -----	27
SETUP AND HOLD COMBINED -----	27
RECOVERY TIME-----	27
WIDTH-----	28
<b>6. VERILOG BACK ANNOTATION -----</b>	<b>29</b>
DESCRIPTION -----	29
VERILOG SDF_ANNOTATE-----	29
WHERE TO CALL SDF_ANNOTATE -----	29
ADDITIONAL OPTIONS -----	30
<b>STANDARD CELL EXAMPLE: NAND3 -----</b>	<b>31</b>
OVERVIEW-----	31
VERILOG CELL PRIMITIVE-----	32
SDF FILE -----	33
VERILOG TEST BENCH: FUNCTIONAL-----	34
VERILOG TEST BENCH: PATH DELAYS-----	35
<b>STANDARD CELL EXAMPLE: D FLIP FLOP -----</b>	<b>36</b>
OVERVIEW-----	36
VERILOG CELL PRIMITIVE-----	37
USER DEFINED PRIMITIVE -----	38
USER DEFINED PRIMITIVE STATES -----	39
SDF FILE -----	40
<b>STANDARD CELL EXAMPLE: TRISTATE INVERTER -----</b>	<b>41</b>
OVERVIEW-----	41
VERILOG CELL PRIMITIVE-----	42
SDF FILE -----	43
<b>STANDARD CELL EXAMPLE: LATCH-----</b>	<b>44</b>
OVERVIEW-----	44
VERILOG CELL PRIMITIVE-----	45
USER DEFINED PRIMITIVE -----	46
SDF FILE -----	47
<b>REFERENCES AND RESOURCES-----</b>	<b>48</b>
DESCRIPTION -----	48
REFERENCES-----	48

# 1. TIMING CHARACTERIZATION

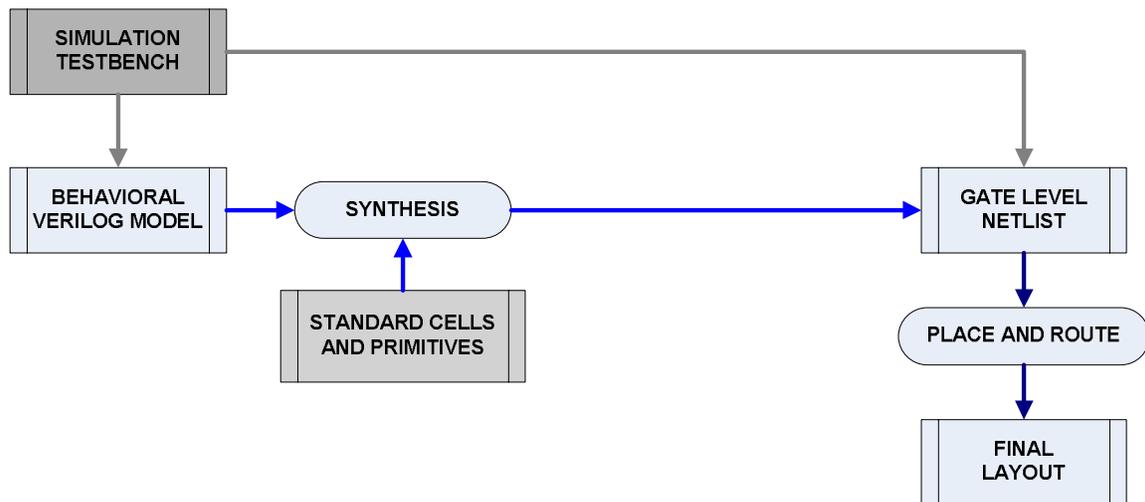
## **PURPOSE**

Timing and delay information is important to the ASIC design process because it allows a level of realism to be incorporated into the circuit model. Typically a Verilog hardware model does not include this timing information; the outputs of modules are in effect resolved instantaneously. This strips the designer of important details such as propagation delay, pulse width, setup and hold times. Incorrect state transitions from setup and hold violations could very easily lead to the design not functioning.

Timing characterization is an often ignored process in a design; it can sometimes be a tedious and time consuming endeavor. This leads most designers to be primarily concerned with the functional correctness of a circuit. In the past, this approach may have been adequate for lower speed designs. However, as technology continues to mature, meeting timing constraints for designs becomes just as important as functional correctness.

## **DESIGN FLOW WITH NO TIMING**

A typical HDL to layout design flow for ASICs is described below. Essentially, there is a Verilog model of a design that is to be built. The synthesis tool utilizes an existing digital cell library and will output a gate level netlist of the design. From here, the gate level netlist will be run through an automatic place and route tool to generate the layout for the design.

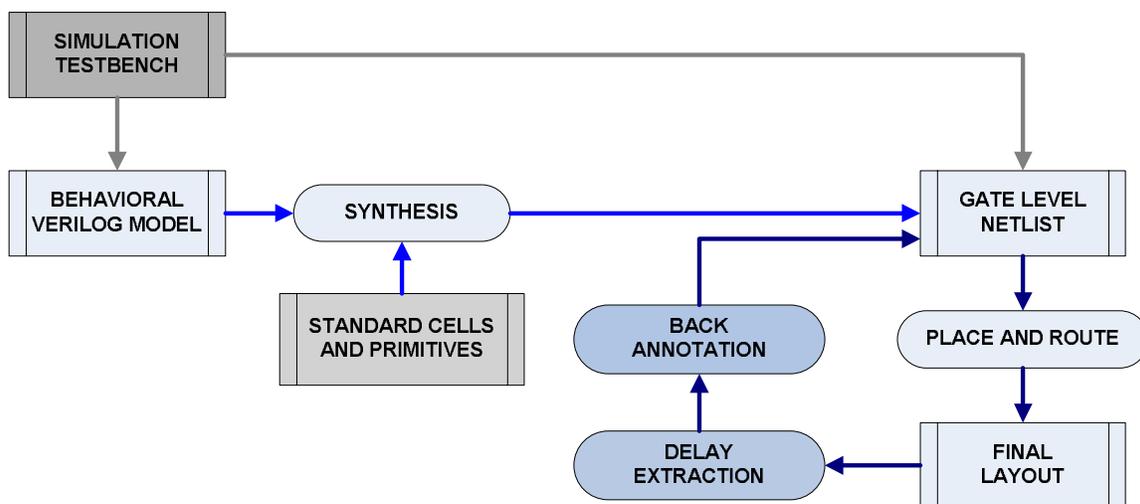


HDL to LAYOUT

At this point the design is well on its way to being finished. The design could be simulated and it could also be functionally verified to be correct. However, up to this point, no timing information has been included. There is no knowledge about whether or not setup and hold times can be met. There is no knowledge that propagation delays are appropriate. The design may be functionally correct, but after getting the chip back from fabrication, it still might fail. The reason would be that no timing information was included in the verification process and critical paths were not properly identified.

### **DESIGN FLOW WITH TIMING**

Additional timing characterization has to be done on the design in order to prove that it will not fail. The design flow should be changed to include delay extraction and back annotation. Delay extraction will allow us to get delays from the placed and routed design. Back annotation will then take this delay information and place it back into the gate level Verilog model.



### **HDL to LAYOUT with BACK ANNOTATION**

At this point, the design can be simulated with timing information to ensure both functional and timing correctness. This is not a complete timing characterization of the circuit, as will be discussed later, but it is a good start. The rest of this document is going to describe the methods in which to add basic timing characterization of a synthesized ASIC design.

## 2. VERILOG DESIGN MODELS

### ***DESCRIPTION***

Verilog has many ways in which to model a design. Each of these methods is appropriate in certain situations. They basically fall nicely into three modeling categories.

*Behavioral Modeling*  
*Dataflow Modeling*  
*Gate Level Modeling*

These different modeling techniques will be briefly discussed to show how the different Verilog model types affect the delays.

### ***BEHAVIORIAL MODELING***

This is a higher level of abstraction. Variables are assigned values in this case. Typically there is no delay is associated with a behavioral model.

### ***DATAFLOW MODELING***

This model has no concept of gates. Signals are used instead. Often times this is called RTL (Register Transfer Language) modeling. Delays in this case will be associated with a net or wire in which a value is transmitted.

### ***GATE LEVEL MODELING***

This is the lowest level of modeling that Verilog allows. The delays being considered are the propagation delays through the gate and the time for the output to change state.

This is the level in which the delays for back annotation will be used. Specific delay information will be specified for each gate level primitive. After we synthesize our behavioral model we are left with a gate level netlist. This gate level netlist will then be used to run the simulation with back annotated timing information.

### 3. VERILOG DELAY MODELS

#### *DELAY MODELS OVERVIEW*

There are a number of ways in which to capture the idea of delay for a Verilog model. There are two common techniques to classify delay. Both are used together in order to achieve accurate timing characterization.

*CLASSIFY DELAY BY CONNECTION  
CLASSIFY DELAY BY RISE/FALL AND MIN/MAX*

#### *VERILOG NOTATION SUMMARY*

This section was included as a reference. It shows the different syntaxes used in Verilog to convey timing information to the simulator.

<i>Type of Delay</i>	<i>Format</i>	<i>Example</i>
1) <i>Absolute Delay</i>	<i>#N</i>	<i>#5</i>
2) <i>Min, Typical, Max</i>	<i> #(N:N:N)</i>	<i> #(4:6:8)</i>
3) <i>Rising, Falling</i>	<i> #(N,N)</i>	<i> #(4,6)</i>
4) <i>Rising, Falling, Turnoff</i>	<i> #(N,N,N)</i>	<i> #(4,6,8)</i>
5) <i>Combined (2) &amp; (4)</i>	<i> #(N:N:N, N:N:N)</i>	<i> #(4:6:8, 5:7:9)</i>

*(Note: N is defined as any positive real number)*

Essentially we can convey very simple delays. We can also convey very complex delays that have process dependent minimum, typical or maximum values.

This allows the designer flexibility to have conservative or aggressive timing characterization in simulation. For instance, we could use minimum delay values assuming we have a very well fabricated chip. This would be used for an aggressive high speed test of the design. Or perhaps we could assume that the fabrication facility might return a lower quality chip and then simulate with maximum delays in mind. The combined method listed above is the most accurate as it includes rise and fall times along with process dependent values.

## ***TIMESCALE***

So far we have been classifying delays but have not mentioned the timescale for the values. The time scale is specified before a module declaration with the following construct.

```
`timescale 1ns/10ps
```

This would imply that a value of 1.0 is equal to a delay of 1.0 nanoseconds and that the possible resolution is 10 picosecond.

For example we could then specify a delay of 1.23 ns, but not a delay of 1.234 ns because we are limited by the 10 ps resolution.

Timescale specification is done on a per module basis. If the timescale is omitted it will be chosen by the simulator. The simulator usually has the ability to override the timescale as well.

## ***DELAY TYPES***

The type of delay can be any valid delay construct of Verilog. It is not limited to simply static delays. For instance, one could specify the rise, fall and turn off delays. See the delay models section for more information on delay types.

## ***CLASSIFY DELAY BY CONNECTION***

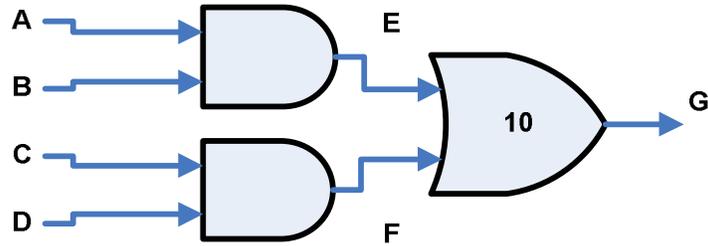
Delay by connection is really as straight forward as the name implies. Delay is calculated by following the connections of the wires and attributing delays. There are three methods of calculating delays by connection. They will be discussed below. The most flexible, and the one used for back annotation, is the path delay.

*Lumped Delay*  
*Distributed Gate Delays*  
*Path Delay (Pin to Pin Delays)*

## ***LUMPED DELAY***

In this case, all the delay is lumped at the output. This is similar to the distributed delay except the modules are assigned delays instead of the specific component

parts. This delay model often uses longest path (or worst case) delay performance.



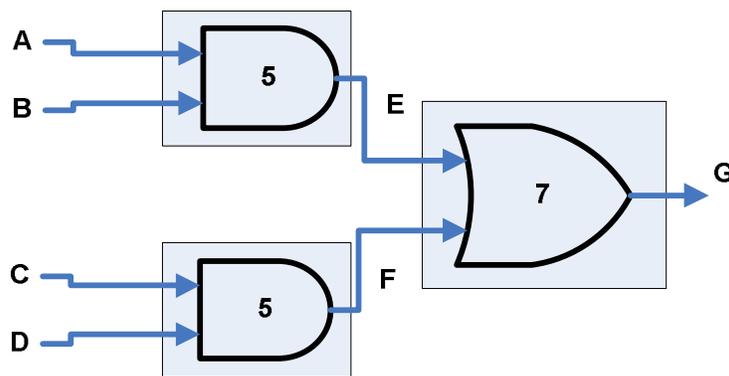
```

and      and1 (E, A, B) ;
and      and2 (F, C, D) ;
or #10   or3 (G, E, F) ;

```

### DISTRIBUTED DELAY

In this case, every element of the circuit has a delay associated with it. The delay between two points is easily calculated by adding together delays of the components through which the signal passes. Basically delays are grouped for each cell element.



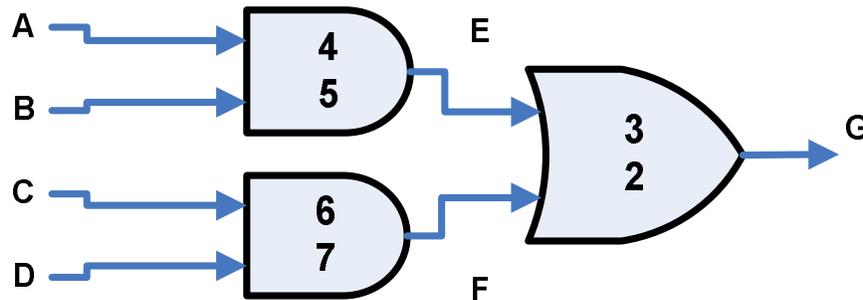
```

and #5   and1 (E, A, B) ;
and #5   and2 (F, C, D) ;
or #7    or3 (G, E, F) ;

```

## PATH DELAY (PIN to PIN DELAY)

Of the three methods, path delay is most flexible. This method is referred to both as path delay and pin to pin delay. Each input to output pin pairing has a delay associated with it. No specific elements have delays associated with them. This technique is useful because it hides the internal details of the module itself. This method is also more complicated than the previous two methods. A special Verilog block needs to be inserted into the code for the cell module in order to handle this timing construct. This block is known as a specify block and will be discussed in later sections of this document. As stated previously, this is the method that will be used when back annotating timing information into a Verilog behavioral model.



```
Module andor(G,A,B,C,D)
...
    specify
        (A => G) = 7; // 4+3 Path
        (B => G) = 8; // 5+3 Path
        (C => G) = 8; // 6+2 Path
        (D => G) = 9; // 7+2 Path
    endspecify
endmodule
```

## CLASSIFY DELAY BY RISE/FALL AND MIN/MAX

In the previous section, it was shown how an absolute delay could be added to a module. Verilog allows us to extend this absolute delay to include more information such as rise and fall times. In addition, it also allows us to define process dependent triads.

### RISE AND FALL TIMES

Rise and fall times are defined as the time that it takes for a signal to change from high to low or low to high. In addition, we could also specify a turnoff time that is a change from any state to high impedance. Turnoff time is mainly used for tri-state devices. Adding in this delay information is as simple as following the Verilog syntax for delays.

DELAY CLASSIFICATION	POSSIBLE START STATES			END STATE
RISE DELAY	0	X	Z	1
FALL DELAY	1	X	Z	0
TURN OFF DELAY	0	1	X	Z

*0 being defined as logic low  
1 being defined as logic high  
X being defined as unknown  
Z being defined as high impedance*

For instance we could create the following situations. Note that a comma separates the division between the rise, fall and turn off delays.

DELAYS	FORMAT	VERILOG CODE
<i>Static_Delay = 5</i>	<i>#D</i>	<i>assign #5 G = A &amp; B;</i>
<i>Rise = 2 Fall = 4 Turn_Off = 8</i>	<i> #(R,F,T)</i>	<i>assign #(2,4,8) G = A &amp; B;</i>

It is important to note that Verilog actually allows us to specify up to 12 styles of state transition! (Instead of just 3) This however is out of the scope of this document. See the Verilog Language Reference Manual for more details.

## PROCESS DEPENDENT TRIADS

Another extension to absolute timing delays would be to incorporate process dependent triads. Before a simulation is run, we could select one of the triads to be used. This allows us to characterize timing in worst case, typical case and best case scenarios. Once again adding this information is as simple as following the Verilog syntax.

For instance we could create the following situation. Note that comma separates the minimum, typical and maximum delays.

<b>DELAYS</b>	<b>FORMAT</b>	<b>VERILOG CODE</b>
<i>Static_Delay = 6</i>	<i>#D</i>	<i>assign #6 G = A &amp; B;</i>
<i>Minimum = 2 Typical = 4 Maximum = 8</i>	<i> #(Min,Typ,Max)</i>	<i>assign #(5:6:7) G = A &amp; B;</i>

## COMPLEX DELAYS

For a complete description of the timing, it is best to use both RISE and FALL times in addition to using the TRIADS. This provides a lot of information to the simulator. Having this information increases the flexibility of the type of simulations a user can run. For instance we could create the following situation. Note how the first triplet is the rise time's minimum, typical and max. Followed by the fall times triplet and then the turn off's triplet.

<b>EXAMPLE</b>		
<i>Minimum</i> <i>(R,F,T) = 1,2,3</i>	<i>Typical</i> <i>(R,F,T) = 4,5,6</i>	<i>Maximum</i> <i>(R,F,T) = 7,8,9</i>
<b>FORMAT</b>		
<i># (RISE(Min,Typ,Max), FALL(Min,Typ,Max), TOFF(Min,Typ,Max))</i>		
<b>VERILOG CODE</b>		
<i>assign #(1:4:7, 2:5:8, 3:6:9) G= A&amp;B;</i>		

## 4. VERILOG SPECIFY BLOCK

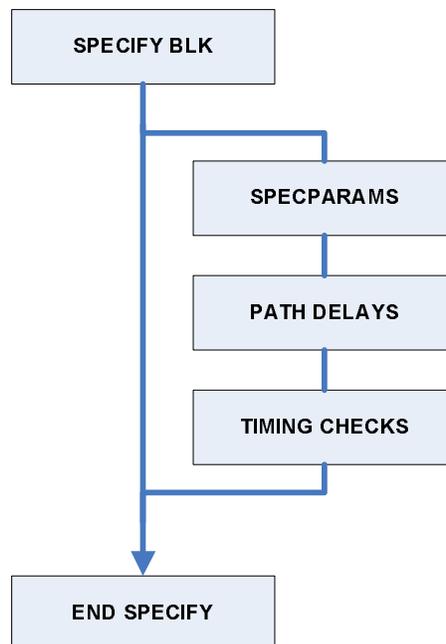
### ***DESCRIPTION***

As previously discussed, Verilog allows us to specify path delays via a special code block called the specify block. A specify block allows us to do the following three things:

*Setup timing checks within the circuit*  
*Define pin to pin (path) delays across the module*  
*Define specparam constants*

Another useful feature is that the simulator will utilize back annotation in conjunction with the specify blocks. The simulator accepts a delay file. Assuming a specify block exists in code, the simulator will then go in and automatically set the appropriate delays inside the specify block. Essentially Verilog and the simulator have a built in ability to handle the changing of delays within a specify block. This is useful because a design is typically coded in Verilog long before it reaches a placed and routed design (where timing information is extracted). The process of back annotation will be discussed later.

The structure of a specify block is shown below. The specparam, path delays and then timing check sections will be detailed. Samples of various specify blocks can be found in the example sections at the end of this document.



## **SPECPARAMS**

### *DESCRIPTION*

Specparams are special parameters that can be defined for use within the delay portions of a specify block. They are declared after a *specparam* statement. The specparams are most commonly used to define delays in one location so that they can be used much like a variable. Another feature is that they can be used with non-simulation tools for tasks such as forward annotation. Since this type of specparam is tool dependent, they will not be discussed.

### *SCOPE*

Specparams can only be used within the specify block. In other words their scope is the specify block they are used within.

### *NOTATION*

There is no hard coded notation for how the specparams are to be named. In the OSU digital cell library the convention that was used is as follows:

```
specparam tplh$A$Y = 1.0;  
  
"Time Propagate Low to High from Input A to Output Y"
```

This would be defined as the time for a change on A to cause the output Y to go from low to high, or in other words the **rise time**.

Another example would be.

```
specparam tplh$A$Y = 1.0;  
  
"Time Propagate High to Low from Input A to Output Y"
```

This would be defined as the time for a change on A to cause the output Y to go from high to low, or in other words the **fall time**.

## PATH DELAYS

Delays can be specified in two fashions. Input output pairings can be specified in a parallel fashion or in fully connected fashion. Path delays also require a source and destination operand. The official Verilog syntax is as follows:

```
(<source_field> <connection_type> <destination_field>) = <delay_value>
```

<code>&lt;source_field&gt;</code>	<i>Is the source signal</i>
<code>&lt;destination_field&gt;</code>	<i>Is the destination signal</i>
<code>&lt;connection_type&gt;</code>	<i>Is one of the following:</i>
	<i>=&gt; for parallel</i>
	<i>*&gt; for fully connected</i>
<code>&lt;delay_value&gt;</code>	<i>Any of the Verilog allowed delays</i>

## PARALLEL CONNECTION

For parallel connections, each bit of the source signal is associated with its equivalent bit in the destination signal. The bitwidth must be exactly the same. For example, the following two are equivalent.

```
// 2-Bit Vector ← equivalent → (A[0] => Y[0]) = 5  
(A => Y) = 5; (A[1] => Y[1]) = 5
```



PARRALLEL CONNECTION GRAPHICAL

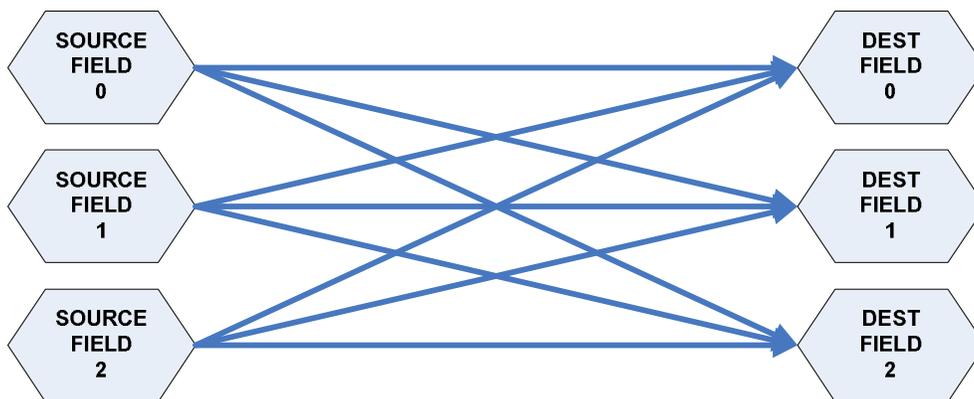
## FULLY CONNECTED

In a full connection, each bit of the source will affect each destination bit. This implies that a bit mismatch is allowed. This is used for specifying delay for each input to each output.

For example, say we had a 32 bit input register A[31:0] and a 4 bit output register Y[3:0]. This would require  $(32 * 4) = 128$  parallel connections for each input bit to each output bit. (<http://www.see.ed.ac.uk/~gerard/Teach/Verilog/me5cds/>)

However this could be specified in one line of code using the fully connected type.

```
(A *> Q) = 5; // Same as 128 parallel connections
```



*FULL CONNECTION GRAPHICAL*

## CONDITIONAL PATH DELAYS

Verilog also has a method for handling state dependent path delays. This is possible by using an IF statement. The conditional statement can contain any bitwise, logical, concatenation, conditional or reduction operator. It is important to note that the ELSE construct can NOT be used.

In effect this is used to set different timings based on the level of a single or multiple inputs. For example:

```

if (A)           (A => Y) = 5;
if ~(A)          (A => Y) = 6;
if (A & C)       (A => Y) = 7;

```

## POLARITY OF PATH DELAYS

One other little quirk of specifying path delays is that the polarity of the output signal can be specified. For instance, we could specify the following for QP and QN of a D flip flop.

```

(posedge EN *> (QP +: D)) = (tplh$EN$QP, tphl$EN$QP);
(posedge EN *> (QN -: D)) = (tplh$EN$QN, tphl$EN$QN);

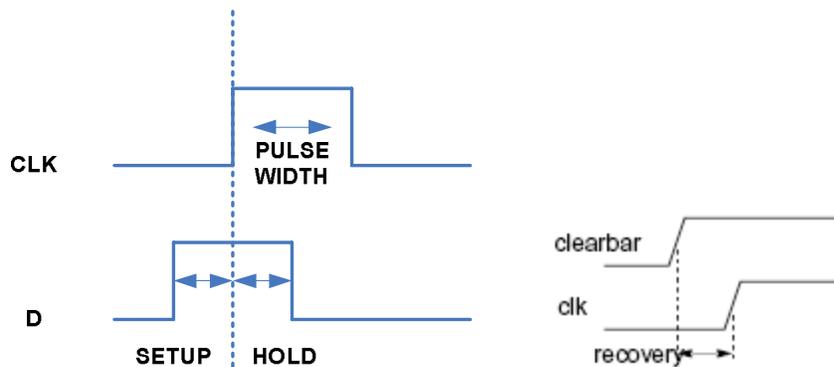
```

Notice how even though QN is opposite in polarity to QP we can still utilize the `tplh$EN$QN` specparm as the rise time.

## TIMING CHECKS

This is the final type of delay that Verilog specify blocks can check. It is important to note that these are simply checks, not delays. All the timing checks are Verilog defined system tasks. There are a variety of system tasks defined in the language standard, the most important are the `$setup`, `$hold`, `$recovery` and `$width` system tasks. All timing checks must be within the `specify` block. Timing checks are most useful for sequential digital elements such as flip flops.

The following diagram details how the clock, input and clear bar for a D flip flop relate to the four timing checks defined above.



## *SETUP TIME*

The setup time is the time that D must be stable before the clock edge. The system task is defined as follows.

```
$setup(<data_event>, <reference_event>, <limit>);  
  
<data_event>      Signal monitored for violations  
<reference_event> Establishes reference for monitoring  
<limit>          Minimum time for setup
```

For example, the following would specify the setup time for an input D with the positive edged signal CLK to be 2.5 nanoseconds. (Assuming timescale of 1ns)

```
setup(D, posedge CLK, 2.5);
```

## *HOLD TIME*

The hold time is the time that D must be stable after the edge of the clock. The system task is defined as follows.

```
$hold(<data_event>, <reference_event>, <limit>);
```

For example, the following would specify the hold time for an input D with the positive edged signal CLK to be 3.2 nanoseconds. (Assuming timescale of 1ns)

```
$hold(D, posedge CLK, 3.2);
```

## *WIDTH*

The pulse width is the minimum time between two changes on a signal for a pulse. This is mainly used to monitor the clocks to make sure they do not transition too fast.

```
$width(<reference_event>, <limit>);
```

For example, the following would specify the minimum width of a signal named CLK to be 5 nanoseconds.

```
$width(CLK, 5.0);
```

## RECOVERY

The recovery time is defined as the limit of the time between the release of an asynchronous control signal from the active state and the next active clock edge.

```
$recovery(<data_event>, <reference_event>, <limit>);
```

For example, the following would specify the recovery time of a CLR signal to be 5 nanoseconds.

```
$recovery(posedge CLR, posedge CLK, 5.0);
```

## EXAMPLE OF A DFF

The following example of a D flip flop is simply provided as a reference. It shows many of the different components of a specify block that were just outlined.

A complete list of examples can be found in Appendix A. Notice the three sections highlighted in grey.

```
'celldefine
module dff (QP, QN, D, CLK);
  output QP, QN;
  input  D, CLK;
  reg    NOTIFIER;
  supply1 RN, SN;

  buf    I3 (clk, CLK);
  udp_dff I0 (n0, D, clk, RN, SN, NOTIFIER);
  and    I4 (flag, RN, SN);
  buf    I1 (QP, n0);
  not    I2 (QN, n0);

  specify
  specparam
    tplh$CLK$QP    = 1.0, tsetup$D$CLK    = 1.0,
    tphl$CLK$QP    = 1.0, thold$D$CLK    = 1.0,
    tplh$CLK$QN    = 1.0, tminpwl$CLK    = 1.0,
    tphl$CLK$QN    = 1.0, tminpwh$CLK    = 1.0;

  if (flag)
    (posedge CLK *> (QP  +: D)) = (tplh$CLK$QP, tphl$CLK$QP);
  if (flag)
    (posedge CLK *> (QN  -: D)) = (tplh$CLK$QN, tphl$CLK$QN);
  $setuphold(posedge CLK &&& (flag == 1), posedge D, tsetup$D$CLK, thold$D$CLK, NOTIFIER);
  $setuphold(posedge CLK &&& (flag == 1), negedge D, tsetup$D$CLK, thold$D$CLK, NOTIFIER);
  $width(negedge CLK, tminpwl$CLK, 0, NOTIFIER);
  $width(posedge CLK, tminpwh$CLK, 0, NOTIFIER);

  endspecify
endmodule
'endcelldefine
```

## 5. Standard Delay Format

### ***DESCRIPTION***

This section is an overview of the Standard Delay Format. (SDF)

The best description of what the SDF is can be found in the opening paragraph of the IEEE Draft Standard.

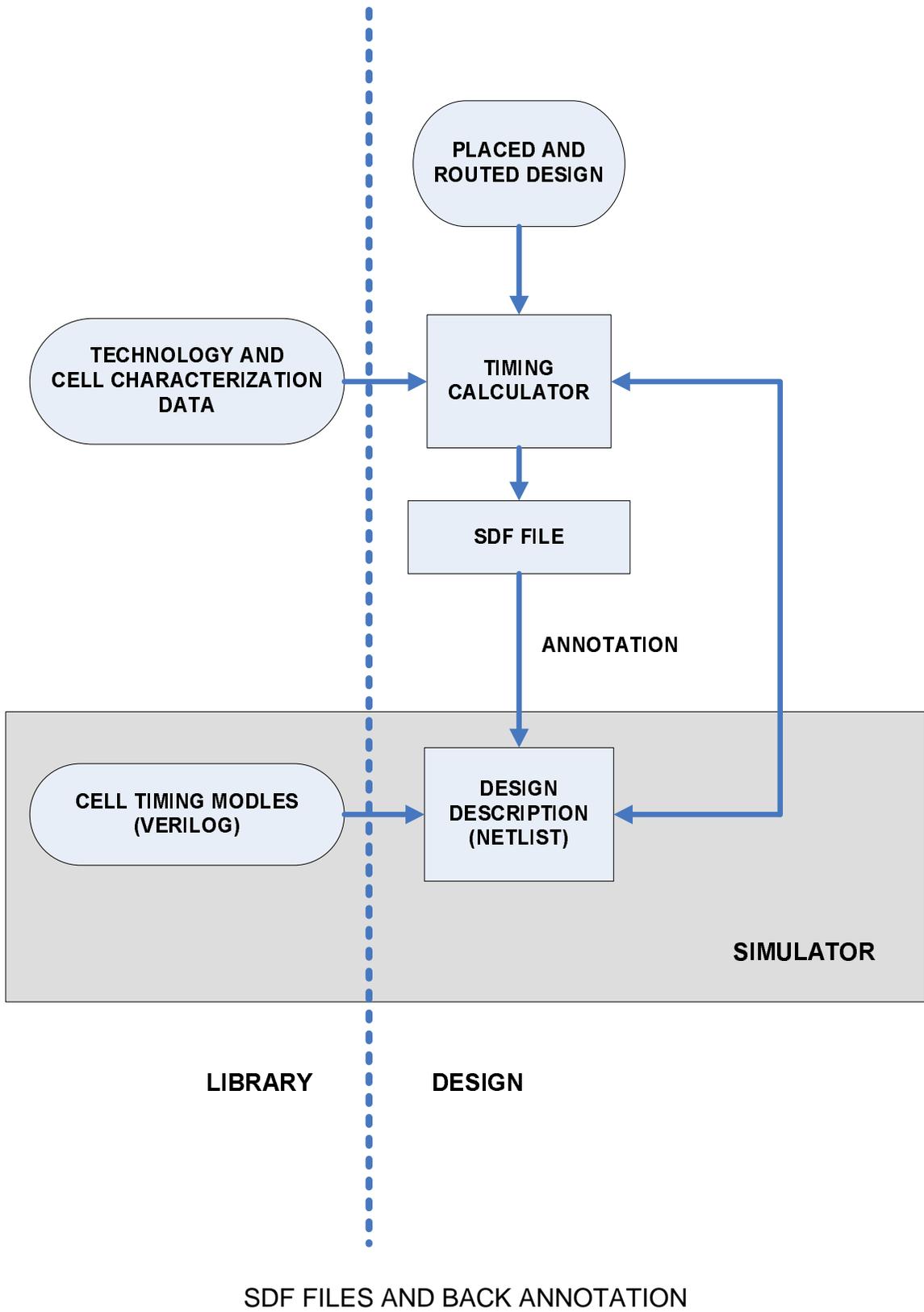
*“ The Standard Delay Format (SDF) was designed to serve as a simple textual medium for communicating timing information and constraints between electronic design automation tools. The original version was designed by Rajit C. Chandra in 1990 while at Cadence Design Systems, and was intended as a means of communicating macrocell and interconnect delays from Gate Ensemble to Verilog-XL, Veritime and other stand-alone tools requiring timing data. “*

The SDF was designed from the ground up to be an easy way to convey timing information to a simulator. The SDF file can furthermore be utilized by other design tools. It can be leveraged to convey design constraints identified during timing analysis to layout tools (forward annotation) and it can also be used for post layout timing analysis and simulation (back annotation).

### ***GENERATION AND USE***

The SDF files are most often generated by a delay calculator. This delay calculator uses information from a placed and routed design. After the SDF file is generated by the timing calculator, simulator will be used to back annotate this delay information into the design description. Timing characteristics of ASICs are strongly influenced by interconnect affects. This is why back annotation is most often done post layout. The SDF imposes no restrictions on the precision of the timing data being represented. This implies that the accuracy of the timing data is dependent on the accuracy of the timing calculator

The figure on the following page depicts the general flow of how to use SDF files in an ASIC design. A similar diagram and further information can be found in the IEEE Standard.



## **SDF FILE FORMAT**

An SDF file is specified as an ASCII formatted text file. The file has two primary sections, a header section and a cell section.

### **HEADER SECTION**

The header section will contain information relevant to the entire file as a whole. Things such as design names, versions, program descriptions, process descriptions and timescales will be included in the header section. Whether or not this information is used in the annotation process is usually left up to the tool.

For our purposes, one can think of the header section as an information section. Very little in the header section will affect the simulation. In effect the header section is for documentation purposes only. Things such as dividers and timescales will, however, affect the way the SDF file is construed. An example header section is below. It was taken from the IEEE Draft Standard.

```
(DELAYFILE
  // HEADER SECTION
  (SDFVERSION "4.0")
  (DESIGN "BIGCHIP")
  (DATE "March 12, 1995 09:46")
  (VENDOR "Southwestern ASIC")
  (PROGRAM "Fast program")
  (VERSION "1.2a")
  (DIVIDER /)
  (VOLTAGE 5.5:5.0:4.5)
  (PROCESS "best:nom:worst")
  (TEMPERATURE -40:25:125)
  (TIMESCALE 100 ps)

  // CELL SECTION
  .....
  .....
)
```

## HEADER SECTION DETAILS

As was stated previously, the header section is mainly concerned with documentation, except for the following two sections. More information on the syntax can be found in the IEEE Draft Standard.

### DIVIDER

The hierarchy divider specifies the permissible characters that will be used to separate different elements in a design hierarchy. The default divider is a period. Therefore, if we specify the divider to be a “/” instead of the default, then hierarchy names would be separated with a “/” instead of a “.”

<b>DIVIDER “/”</b> (INSTANCE adder/x1)	<b>DIVIDER DEFAULT</b> (INSTANCE adder.x1)
---	---

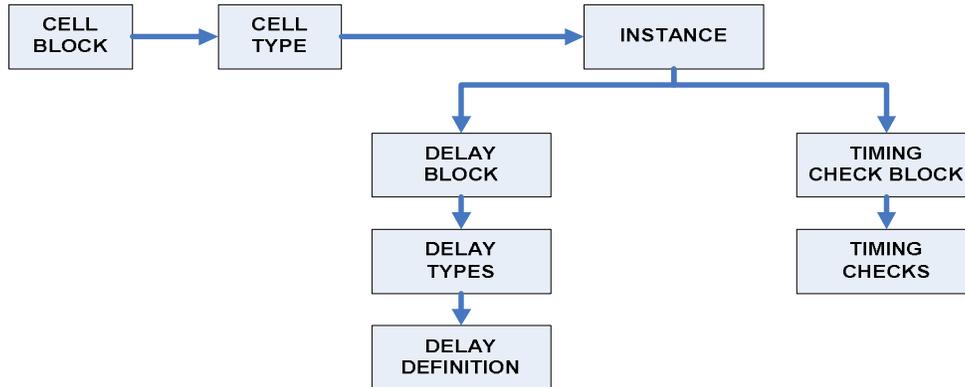
### TIMESCALE

The timescale is an optional field that specifies units for the time values that are specified in the SDF file. The format for the timescale is as follows:

<b>SYNTAX:</b>  (TIMESCALE T_NUMBER T_UNITS)  <i>Allowable timescale numbers are:</i> 1, 10, 100, 1.0, 10.0, 100.0  <i>Allowable timescale units are:</i> s, ms, us, ns, ps, fs
<b>EXAMPLE:</b>  (TIMESCALE 100 ps)

## CELL SECTION

The cell section identifies a specific region within a design in which timing information will be applied. The cell section has the following general format and an example is below.



```
(DELAYFILE
  // HEADER SECTION
  .....
  // CELL SECTION
  // CELL 1
  (CELL
    (CELLTYPE "AND2")
    (INSTANCE top/b/d)
    (DELAY
      // (RISE TRIAD) (FALL TRIAD)
      (ABSOLUTE
        (IOPATH a y (1.5:2.5:3.4) (2.5:3.6:4.7))
        (IOPATH b y (1.4:2.3:3.2) (2.3:3.4:4.3)))
      )
    )
  // CELL 2
  (CELL
    (CELLTYPE "DFF")
    (INSTANCE top/b/c)
    (DELAY
      (ABSOLUTE
        (IOPATH (posedge clk) q (2:3:4) (5:6:7))
        (PORT clr (2:3:4) (5:6:7))
      )
    )
    (TIMINGCHECK
      (SETUPHOLD d (posedge clk) (3:4:5) (-1:-1:-1))
      (WIDTH clk (4.4:7.5:11.3))
    )
  )
)
```

## ***CELL SECTION DETAILED***

This section will detail the relevant parts of the cell section of the SDF file. There is a lot of flexibility within the cell section. However, only selected portions will be discussed, see the IEEE Draft Standard for more information.

### ***CELL TYPE***

The cell type simply indicates the name of the cell. The format is the keyword `CELLTYPE` following by a character string.

```
EXAMPLE
  (CELLTYPE "NAND3")
```

### ***CELL INSTANCE***

The cell instance identifies the region of the design for which the cell contains timing information. It should be consistent with the cell type and design hierarchy. The format is the keyword `INSTANCE` followed by the hierarchal identifier. Assuming a default divider (as discussed in the header section) we have the following example.

```
EXAMPLE
  (INSTANCE adder2.x1)
```

### ***DELAY BLOCK***

This is simply a keyword to identify that we are going to enter delay information. The format is the keyword `DELAY` followed by the various delay types allowed by the SDF.

```
EXAMPLE
  (DELAY
    (DELAY_TYPE . . .)
  )
```

## *DELAY TYPE*

The delay type can be one of the following: ABSOLUTE, INCREMENT, PATHPULSE or PATHPULSEPERCENT. We are only going to be concerned with the ABSOLUTE delay type. The ABSOLUTE delay type simply applies delays to specified regions within a cell. The format is one of the above delay types followed by a delay definition.

```
EXAMPLE
(ABSOLUTE
  (DELAY_DEFINITION1 ...)
  (DELAY_DEFINITION2 ...))
```

## *DELAY DEFINITION*

There are quite a few delay definition constructs available. Only the IOPATH and PORT definitions will be discussed here.

### IOPATH DEFINITION

The IOPATH definition specifies the delays on a legal path from an input port to an output port. The delay values themselves can be any valid Verilog style delay. The IOPATH has no state dependencies. It will annotate independent of the conditions between two ports. The exception to this rule is when a port specifier has an edge identifier associated with it. The format for the IOPATH definition is as follows. The keyword IOPATH followed by input port specification followed by output port specification followed by a list of Verilog style delays. In this case rise time then fall time.

### EXAMPLES

```
(IOPATH a y (1.5:2.5:3.4) (2.5:3.6:4.7))
(IOPATH (posedge clk) q (2:3:4) (5:6:7)) // Edge Identifier
```

### PORT DEFINITION

The PORT definition specifies the interconnect delay of an input port. This can be an estimated or actual delay. The driving output port (start point for the delay path) is not specified. The format is the keyword PORT followed

by the port specification followed by the delay value. The port instance must be an input or bidirectional port.

#### EXAMPLE

```
(PORT clr (2:3:4) (5:6:7))
```

### TIMING CHECK BLOCK

The timing check block is simply a keyword to begin specifying the various timing checks in the SDF. The format is the keyword TIMINGCHECK followed by a timing check type.

#### EXAMPLE

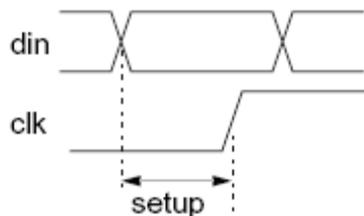
```
(TIMINGCHECK  
  (TIMINGCHECK_TYPE1 ...)  
  (TIMINGCHECK_TYPE1 ...))
```

### TIMING CHECKS

The SDF has many different timing checks available. The important ones are the setup, hold, recovery and width timing checks. They will be discussed below. The format for the timing checks is a timing check definition followed by the appropriate delay information. This delay information varies for each timing check.

#### SETUP TIME

This is the setup timing check. It is defined as the time before a clock that the signal must remain stable in order for that signal to successfully be stored into the device. The format is the keyword SETUP followed by an input port specification followed by an output port specification followed by the delay values.

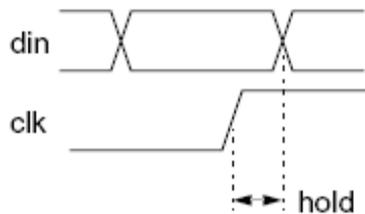


### EXAMPLE

```
(SETUP D (posedge CLK) (2.5:3.6:4.7))
```

### HOLD TIME

This is the hold timing check. It is defined as the time after a clock edge in which a signal must remain stable. The format is the keyword HOLD followed by an input port specification followed by an output port specification followed by the delay values.



### EXAMPLE

```
(HOLD D (posedge CLK) (2.5:3.6:4.7))
```

### SETUP AND HOLD COMBINED

This is a combination of setup and hold. Its format is the keyword SETUPHOLD followed by input port followed by output port followed by setup delay values followed by hold delay values.

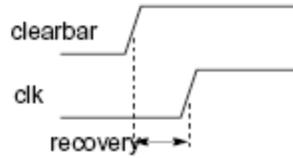
### EXAMPLE

```
(SETPHOLD D (posedge CLK) (2:3:4) (2:3:4))
```

### RECOVERY TIME

The recovery time is defined as the limit of the time between the release of an asynchronous control signal from the active state and the next active

clock edge. The format is the keyword RECOVERY followed by an input port followed by an output port followed by the delay values.

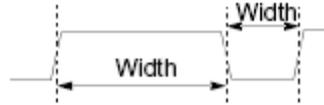


#### EXAMPLE

```
(RECOVERY (posedge CLR) (posedge CLK) (2:3:4))
```

#### WIDTH

The width timing check specifies a limit for a minimum pulse width. If the signal has unequal phases, two pulse widths can be specified.



#### EXAMPLE

```
(WIDTH (posedge clk) (5))
```

## 6. VERILOG BACK ANNOTATION

### *DESCRIPTION*

Back annotation is the process by which timing information is added into a design so that the design can be simulated with realistic delays. For this section back annotation requires that a SDF file has been generated for a design and that specify blocks with path delays have been defined for cells. Different languages have different methods of including the timing information into a simulation. This section is going to focus on Verilog's technique of annotating delays.

### *VERILOG SDF\_ANNOTATE*

As previously shown, Verilog provides a specify block which allows the user to define path delays. Verilog also has a system task that conveniently allows the user to read in an SDF file. This system task is named \$sdf\_annotate. This system task reads in an SDF file and annotates timing info into the design. When the simulation continues all those path delays and timing checks in the specify block will be updated with the timing information from the SDF file. There are additional options for the system task other than a single input file, the basic ones will be discussed below.

### *WHERE TO CALL SDF\_ANNOTATE*

The \$sdf\_annotate system task should be called in the test bench for the design. It should be called between an initial begin and end block before the test vectors.

```
module NOR3_testfixture_min ;
  reg A,B,C;
  wire Y;
  NOR3 test(Y,A,B,C); // Cell Instantiation
  initial
  begin
    // SDF Annotation
    $sdf_annotate("/usr/bolinm/cad/nor3.sdf");

    // Test Vectors
    ...
  end
endmodule
```

In this case, we simply feed the \$sdf\_annotate system task the appropriate SDF file. At the beginning of simulation, all the delay information specified in the SDF file will be back annotated into the appropriate cell instances of the gate level netlist. Please note that in this case, no other options were specified for the SDF file. If there were process dependent triads defined in the SDF file, the "TYPICAL" values would be used as default.

### **ADDITIONAL OPTIONS**

As mentioned previously, there are numerous options for the \$sdf\_annotate system task. For more information on what those options are, please reference the Verilog Language Reference Manual. The most important option is how to specify different process dependent triads. This is done by utilizing three keywords in the system task call.

*minimum*  
*typical*  
*maximum*

From this, we can now specify which process dependent triad we wish to use in the simulation. For example, say we wanted a best case scenario to test how a better chip produced on an ideal run. In the test bench we would call the following.

```
sdf_annotate ("/usr/bolinm/nor3.sdf", , , "minimum");
```

Now say that we wished to run a simulation with worst a worst case scenario because we know that the better chip was produced by a worst case production run. We would call the following in the test bench.

```
sdf_annotate ("/usr/bolinm/nor3.sdf", , , "maximum");
```

## STANDARD CELL EXAMPLE: NAND3

### OVERVIEW

INPUTS

**A, B, C**

OUTPUTS

**Y**

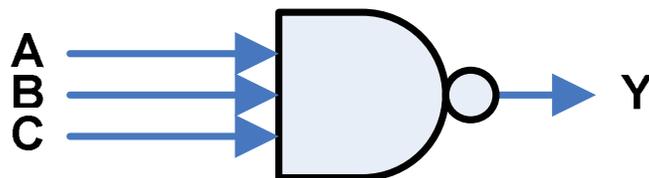
LOGIC FUNCTION

**$Y = \text{NOT} (A \text{ AND } B \text{ AND } C)$**

FUNCTION TABLE

A	B	C	Y
0	X	X	1
X	0	X	1
X	X	0	1
1	1	1	0

SYMBOL



## VERILOG CELL PRIMITIVE

This is a cell definition for a 3 input NAND gate. The first portion sets up input and outputs for the module. The second portion uses a Verilog defined primitive NAND. It feeds the three inputs A, B and C into the NAND and the corresponding output goes to Y. The last portion of the standard cell primitive is the specify block. The specify block is where the path (pin to pin) delays are defined. When SDF back annotation is performed in the test bench, these path delays will be set to the timing information contained within the SDF file.

```
`timescale 1ns/1ps
`celldefine
module NAND3 (Y, A, B, C);
  // Setup Input and Outputs
  output Y;
  input  A, B, C;

  // Use Verilog NAND primitive
  nand (Y, A, B, C);

  specify
    // delay parameters
    specparam
    tplh$A$Y = 1.0,
    tphl$A$Y = 1.0,
    tplh$B$Y = 1.0,
    tphl$B$Y = 1.0,
    tplh$C$Y = 1.0,
    tphl$C$Y = 1.0;

    // path delays
    (A *> Y) = (tplh$A$Y, tphl$A$Y);
    (B *> Y) = (tplh$B$Y, tphl$B$Y);
    (C *> Y) = (tplh$C$Y, tphl$C$Y);
  endspecify

endmodule
`endcelldefine
```

## **SDF FILE**

This is a sample SDF file that can be used to test the back annotation of the NAND3 gate. Note the path delays from A to Y, B to Y and C to Y. These will be back annotated into the path delays in specify block for the NAND3 cell. The timescale in this case is specified to be 1 nanosecond. The instance references the NAND3 called test instantiated in NAND3\_testfixture on the next page. Also note how both rise and fall delays are specified in process dependent triads.

```
(DELAYFILE
  (DESIGN "TESTING")
  (TIMESCALE 1ns)
  (CELL
    (CELLTYPE "NAND3")
    (INSTANCE *) // Any NAND3
    (DELAY
      (ABSOLUTE
        (IOPATH A Y (0.1965:0.2863:0.4268) (0.2101:0.3294:0.5626))
        (IOPATH B Y (0.1935:0.2783:0.4079) (0.2281:0.3477:0.5740))
        (IOPATH C Y (0.1648:0.3614:0.4763) (0.2795:0.4372:0.7556))
      )
    )
  )
)
```

## VERILOG TEST BENCH: FUNCTIONAL

This is an example test bench. It consists of a fully exhaustive functional test and a checking function to verify logical correctness. This has nothing to do with the timing characterization and is simply included as reference. This will be the only cell that a test bench is shown for.

```
module NAND3_testfixture ;
  reg A,B,C;
  wire Y;

  NAND3 test(Y,A,B,C);

  // EXHAUSTIVE FUNCTIONAL TEST
  initial
  begin
    $sdf_annotate("/student/nand3.sdf");

    A = 1'b0;
    B = 1'b0;
    C = 1'b0;

    // Exhaustively test all input combinations
    // Delay each change by 5 time units
    {A,B,C}=3'b000;
    #5 {A,B,C}=3'b001;
    #5 {A,B,C}=3'b010;
    #5 {A,B,C}=3'b011;
    #5 {A,B,C}=3'b100;
    #5 {A,B,C}=3'b101;
    #5 {A,B,C}=3'b110;
    #5 {A,B,C}=3'b111;
    #5 {A,B,C}=3'b111; // HOLD LAST FOR ONE MORE
  end

  // SELF CHECKING FUNCTION
  always @(Y)
  begin
    // Only look when Y changes to HIGH
    if(Y != ~(A&B&C))
      begin
        $display("ERROR_INCORRECT_FUNCTION: Y != ~(A&B&C)");
        $display("Y,A,B,C = ",Y,A,B,C);
      end
  end
end
--
```

## VERILOG TEST BENCH: PATH DELAYS

This is an example test bench that looks at the various pin to pin delays. When run with an appropriate SDF file, it will show all pin to pin delays that are in the specify block.

```
module NAND3_testspecify_fixture ;
  reg A,B,C;
  wire Y;

  NAND3 test(Y,A,B,C);

  // TEST ALL PATH DELAYS
  initial
  begin
    // SDF Annotation
    $sdf_annotate("/student/nand3.sdf");

    // INITIAL WAVEFORM
    {A,B,C}=3'b000;
    #5 {A,B,C}=3'b000;

    // Y FALL TIME FROM A
    #5 {A,B,C}=3'b111;
    #5 {A,B,C}=3'b011;

    // Y RISE TIME FROM A
    #5 {A,B,C}=3'b011;
    #5 {A,B,C}=3'b111;

    // Y FALL TIME FROM B
    #5 {A,B,C}=3'b111;
    #5 {A,B,C}=3'b101;

    // Y FALL TIME FROM B
    #5 {A,B,C}=3'b101;
    #5 {A,B,C}=3'b111;

    // Y FALL TIME FROM C
    #5 {A,B,C}=3'b111;
    #5 {A,B,C}=3'b110;

    // Y FALL TIME FROM C
    #5 {A,B,C}=3'b110;
    #5 {A,B,C}=3'b111;
    #5 {A,B,C}=3'b111;
  end
endmodule
```

## STANDARD CELL EXAMPLE: D FLIP FLOP

### OVERVIEW

#### INPUTS

**D, CLK**

#### OUTPUTS

**QP, QN**

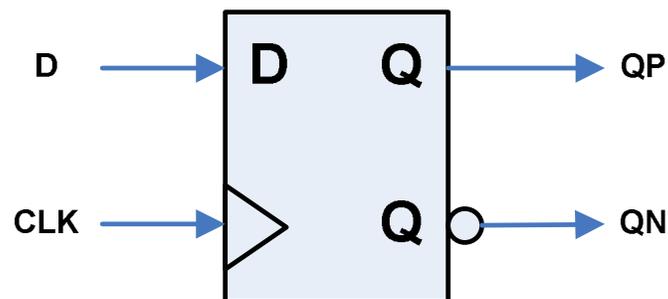
#### LOGIC FUNCTION

D = INPUT  
CLK = CLOCK  
QP = OUTPUT  
QN = INVERTED OUTPUT

#### FUNCTION TABLE

D	CLK	QP[n+1]	QN[n+1]
0	POS EDGE	0	1
1	POS EDGE	1	0
X	NEG EDGE	QP[n]	QN[n]

#### SYMBOL



## VERILOG CELL PRIMITIVE

This is the cell primitive for a D flip flop. The first portion sets up the inputs and outputs for the module. The next section feeds appropriate signals into the user defined primitive for the DFF. Notice how this flip flop has no preset or clear therefore they are simply tied off. The last section is the specify section. There are the path delays, the setup and hold timing checks as well as the pulse width timing checks.

```
`timescale 1ns/1ps
`celldefine
module DFF (QP, QN, D, CLK);
  output QP, QN;
  input  D, CLK;
  reg    NOTIFIER;
  supply1 RN, SN;

  buf    I3 (clk, CLK);
  udp_dff I0 (n0, D, clk, RN, SN, NOTIFIER);
  and    I4 (flag, RN, SN);
  buf    I1 (QP, n0);
  not    I2 (QN, n0);

  specify
  specparam
    tphl$CLK$QP = 1.0,
    tphl$CLK$QP = 1.0,
    tphl$CLK$QN = 1.0,
    tphl$CLK$QN = 1.0,
    tsetup$D$CLK = 1.0,
    thold$D$CLK = 1.0,
    tminpwl$CLK = 1.0,
    tminpwh$CLK = 1.0;

  // PATH DELAYS
  if (flag)
    (posedge CLK *> (QP  +: D)) = (tphl$CLK$QP, tphl$CLK$QP);
    // Polarity of QP is positive referenced to D
  if (flag)
    (posedge CLK *> (QN  -: D)) = (tphl$CLK$QN, tphl$CLK$QN);
    // Polarity of QN is negative referenced to D

  // SETUP AND HOLD CHECKS
  $setuphold(posedge CLK &&& (flag == 1), posedge D,
    tsetup$D$CLK, thold$D$CLK, NOTIFIER);

  $setuphold(posedge CLK &&& (flag == 1), negedge D,
    tsetup$D$CLK, thold$D$CLK, NOTIFIER);

  // MINIMUM WIDTH CHECKING
  $width(negedge CLK, tminpwl$CLK, 0, NOTIFIER);
  $width(posedge CLK, tminpwh$CLK, 0, NOTIFIER);

  endspecify
endmodule
```



## ***USER DEFINED PRIMITIVE STATES***

This section is going to describe what the various symbols mean in the user defined primitive. It is assumed the user has a basic understanding of Verilog and what user defined primitives do.

<b>STATE</b>	<b>Description</b>
0	Logic low
1	Logic high
X	Unknown
Z	High impedance
?	0 or 1 or X
b	0 or 1
f	(1-0), Falling Edge on an input
r	(0-1), Rising Edge on an input
p	(0-1) or (0-x) or (x-1) or (1-z) or (z-1)
n	(1-0) or (1-x) or (x-0) or (0-z) or (z-0)
*	(??), All transitions
-	No Change

## SDF FILE

This is a sample SDF file that corresponds to the D flip flop. Notice that there are path delays for both QP and QN. Since the output Q only changes on the positive edge of a clock, we only need to put in a check for that. The timing checks are all relatively easy to decipher as well. All the timing checks are bound to the input D and the clock and they check for SETUP, HOLD and PULSE WIDTH violations. This information will get fed back into the simulator when the \$sdf\_annotate function is called.

```
(DELAYFILE
  (DESIGN "TESTING")
  (TIMESCALE 1ns)
  (CELL
    (CELLTYPE "DFF")
    (INSTANCE *) // Any DFF
    (DELAY
      (ABSOLUTE
        (IOPATH (posedge CLK) QP (0.32:0.48:0.73) (0.2:0.43:0.67))
        (IOPATH (posedge CLK) QN (0.32:0.48:0.73) (0.2:0.4:0.6))
        (IOPATH D QP (0.1:0.2:0.3) (0.1:0.2:0.4))
        (IOPATH D QN (0.1:0.2:0.3) (0.1:0.2:0.4))))
      (TIMINGCHECK
        (HOLD (posedge D) (posedge CLK) (0.1433:0.1818:0.2228))
        (HOLD (negedge D) (posedge CLK) (0.0580:0.0646:0.0800))
        (SETUP (posedge D) (posedge CLK) (0.0705:0.0411:0.0020))
        (SETUP (negedge D) (posedge CLK) (0.1556:0.1525:0.1464))
        (WIDTH (negedge CLK) (0.1230:0.1780:0.2700))
        (WIDTH (posedge CLK) (0.1650:0.2460:0.3890))
      )
    )
  )
)
```

## STANDARD CELL EXAMPLE: TRISTATE INVERTER

### OVERVIEW

#### INPUTS

**A, OE**

#### OUTPUTS

**Y**

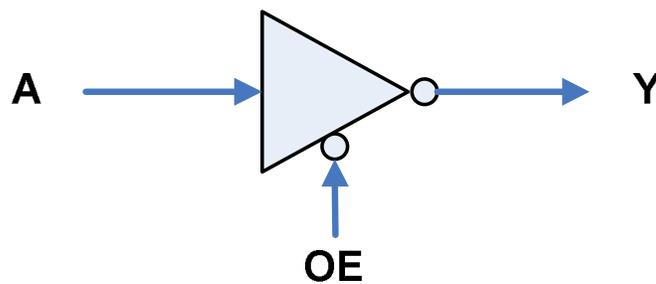
#### LOGIC FUNCTION

**Y = NOT A when OE = 1**  
**Y = Z when OE = 0**

#### FUNCTION TABLE

OE	A	Y
0	X	Z
1	0	1
1	1	0

#### SYMBOL



## VERILOG CELL PRIMITIVE

This is the cell primitive for the tristate inverter. The first section sets up the inputs and outputs for the module. The next section uses the built in Verilog primitive notif1. This primitive is essentially a conditional inverter, which makes designing this cell very simple. The last section is the specify block that contains the path delays for the cell.

```
`timescale 1ns/1ps
`celldefine
module invzx1 (Y, A, OE);
    output Y;
    input  A,OE;

    // notif1 is a built in Verilog primitive
    // The third argument is a control for the gate
    // It works very nicely as a tri-state device

    notif1 I0(Y, A, OE);
    specify
        // delay parameters
        specparam
            tplh$A$Y = 1.0,
            tphl$A$Y = 1.0,
            tplh$OE$Y = 1.0,
            tphl$OE$Y = 1.0;

        // path delays
        (A *> Y) = (tplh$A$Y, tphl$A$Y);
        (OE *> Y) = (tplh$OE$Y, tphl$OE$Y);
    endspecify
endmodule
`endcelldefine
```

## **SDF FILE**

This is a sample SDF file for the tristate inverter. It is very basic because there are only two path delays. One from A to Y and one from OE to Y.

```
(DELAYFILE
  (DESIGN "TESTING")
  (TIMESCALE 1ns)
  (CELL
    (CELLTYPE "invzx1")
    (INSTANCE *) // Any Tristate Inverter Instance
    (DELAY
      (ABSOLUTE
        (IOPATH A Y (0.19:0.28:0.4268) (0.2101:0.3294:0.5626))
        (IOPATH OE Y (0.19:0.27:0.4079) (0.2281:0.3477:0.5740))
      )
    )
  )
)
```

## STANDARD CELL EXAMPLE: LATCH

### OVERVIEW

#### INPUTS

**D, EN**

#### OUTPUTS

**QP, QN**

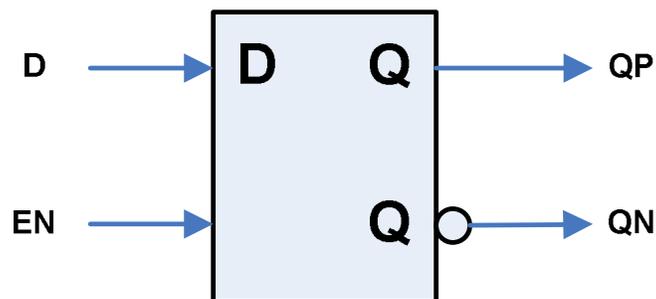
#### LOGIC FUNCTION

D = INPUT  
EN = ENABLE  
QP = OUTPUT  
QN = INVERTED OUTPUT

#### FUNCTION TABLE

D	EN	QP[n+1]	QN[n+1]
1	0	0	1
1	1	1	0
0	X	QP[n]	QN[n]

#### SYMBOL



## VERILOG CELL PRIMITIVE

This is the cell primitive for the transparent D-type latch. The first section sets up the inputs and outputs for the module. The next section uses a user defined primitive of type latch. This user defined primitive will be used for all latches. Its definition is shown on the next page. The last section is the specify block that contains the path delays and timing checks for the cell. Note that since this latch doesn't have a SET or CLEAR (reset) that they are simply tied high. This way they are not enabled affecting the operation of the latch.

```
`timescale 1ns/1ps
`celldefine
module LAT (QP,QN,D,EN);
  output QP,QN;
  input D,EN;
  reg NOTIFIER;
  supply1 RN,SN;

  udp_lat I0 (n0, D, clk, RN, SN, NOTIFIER);
  buf I1 (QP, n0);
  not I2 (QN, n0);
  not I3 (clk,EN);
  buf I4 (flgclk,EN);
  and I5 (SandR,SN,RN);
  and I6 (SandRandCLK,SN,RN,flgclk);

  specify
  specparam
    //timing parameters
    tplh$D$QP = 1.0,
    tphl$D$QP = 1.0,
    tplh$D$QN = 1.0,
    tphl$D$QN = 1.0,
    tplh$EN$QP = 1.0,
    tphl$EN$QP = 1.0,
    tplh$EN$QN = 1.0,
    tphl$EN$QN = 1.0,
    tsetup$D$EN = 1.0,
    thold$D$EN = 1.0,
    tminpwh$EN = 1.0;

    // path delays
    if (SandR)
      (posedge EN *> (QP +: D)) = (tplh$EN$QP, tphl$EN$QP);
      // Polarity of QP is positive referenced to D
    if (SandR)
      (posedge EN *> (QN -: D)) = (tplh$EN$QN, tphl$EN$QN);
      // Polarity of QN is negative referenced to D
    if (SandRandCLK)
      ( D *> QP ) = (tplh$D$QP, tphl$D$QP );
    if (SandRandCLK)
      ( D *> QN ) = (tplh$D$QN, tphl$D$QN );

    // timing checks
    $setuphold(negedge EN &&& (SandR == 1), posedge D,
              tsetup$D$EN,thold$D$EN, NOTIFIER);
    $setuphold(negedge EN &&& (SandR == 1), negedge D,
              tsetup$D$EN,thold$D$EN, NOTIFIER);
    $width(posedge EN &&& (SandR == 1), tminpwh$EN, 0, NOTIFIER);

  endspecify
endmodule
```



## SDF FILE

This is a sample SDF file for the transparent D-type latch. Notice that there are two sections, one section for path delays and one section for the timing checks. There is nothing terribly complicated about this SDF file and it is very similar to the one for the DFF example. Note that we have no clock in this example and that the output of the latch is determined from the enable signal.

```
(DELAYFILE
(SDFVERSION "2.1")
(DESIGN "MYTESTING_DOESNTMATTER")
(PROGRAM "WHOKNOWS_DOESNTMATTER")
(VERSION "v0_DOESNTMATTER")
(TIMESCALE 1ns)
(CELL
(CELLTYPE "LAT")
(INSTANCE *)
(DELAY
  (ABSOLUTE
    (IOPATH (posedge EN) QP (0.1:0.2:0.3) (0.1:0.2:0.4))
    (IOPATH D QP (0.1:0.2:0.3) (0.1:0.2:0.4))
    (IOPATH D QN (0.1:0.2:0.3) (0.1:0.2:0.4))
    (IOPATH (posedge EN) QP (0.1:0.3:0.5) (0.1:0.3:0.4))))
  (TIMINGCHECK
    (HOLD (posedge D) (posedge EN) (0.0194:0.0000:0.0000))
    (HOLD (negedge D) (posedge EN) (0.0000:0.0000:0.0000))
    (SETUP (posedge D) (posedge EN) (0.1988:0.2567:0.3486))
    (SETUP (negedge D) (posedge EN) (0.2735:0.3796:0.5291))
    (WIDTH (negedge EN) (0.1350:0.2030:0.3190))
  )
)
)
}
```

## REFERENCES AND RESOURCES

### *DESCRIPTION*

This section is going to detail where I got the information for this guide. The information contained in this guide is a collection of information from the below websites and the information from Brian Dupaix.

### *REFERENCES*

- 1) Delay Modeling Specify Block  
<http://www.see.ed.ac.uk/~gerard/Teach/Verilog/me5cds/>
- 2) ASIC's Information  
<http://www-ee.eng.hawaii.edu/~msmith/ASICs/HTML/Book/CH13/CH13.5.htm>
- 3) The Verilog Language Reference Online  
[http://www.sutherland-hdl.com/on-line\\_ref\\_guide/vlog\\_ref\\_top.html](http://www.sutherland-hdl.com/on-line_ref_guide/vlog_ref_top.html)
- 4) The Verilog Language  
<http://www1.cs.columbia.edu/~sedwards/classes/2005/languages-summer/verilog.pdf>
- 5) Advanced Verilog HDL  
[http://mufasa.informatik.uni-mannheim.de/lsra/lectures/ws98\\_99/vl\\_simu/vorlesung/vl\\_hw\\_verilogadvanced.pdf](http://mufasa.informatik.uni-mannheim.de/lsra/lectures/ws98_99/vl_simu/vorlesung/vl_hw_verilogadvanced.pdf)
- 6) Digital Circuits, Verilog and other miscellaneous Slides From JKnight  
<http://web.doe.carleton.ca/~jknight/97.478/>
- 7) Modelsim Tutorial  
[http://web.mit.edu/6.111/www/s2004/guides/ModelSim\\_tutorial.pdf](http://web.mit.edu/6.111/www/s2004/guides/ModelSim_tutorial.pdf)
- 8) Index of Cadence Docs  
<http://vlsi.bu.edu/cadence-docs/>
- 9) Simulation of HDL  
<http://www.cs.utah.edu/classes/cs5830/tutorials/simulation.htm>
- 10) Standard Delay Format Reference Standard  
[http://www.eda.org/sdf/sdf\\_3.0.pdf](http://www.eda.org/sdf/sdf_3.0.pdf)