All My X's Come From Texas…Not!!

Matt Weber
Jason Pecor

Silicon Logic Engineering

Matthew.D.Weber@ieee.org
jason@siliconlogic.com

ABSTRACT

Undertaking gate level simulation can put you on the fast track to confusion, frustration, and language more colorful than an X infested simulation waveform. This paper will look at the reasons for doing gate level simulation anyway, describe design and library "features" that can cause gate level simulation problems, and discuss potential solutions to these problems.

Table of Contents

1.0 Introduction 3
2.0 Issues caused by Library Models 4
2.1 Pointing to Library Models 4
2.2 Specify Blocks 5
2.3 #1 Delays (also called #$%*&^ delays) 5
2.4 Pessimistic X propagation 6
3.0 Issues caused by Design 8
3.1 Registers that don't get reset 8
3.2 Clock dividers 11
4.0 Issues caused by Synthesis 15
4.1 Keep resets close to registers 15
4.2 Incomplete Logic Optimization Interferes With Reset 16
4.3 Feedback problems 17
5.0 Issues when running SDF based gate level simulation 18
5.1 Effects of timing failures 18
5.2 Expected Timing Violations 19
5.2.1 Synchronizers for clock domain crossings 19
5.2.2 Multi-cycle paths 20
6.0 Conclusions and Recommendations 20

Table of Figures

Figure 1: Simple Clock Divider 8
Figure 2: Clock Divider with Reset (won't work) 8
Figure 3: Transfer From Fast Clock to Divide By 2 Clock 11
Figure 4: Race Condition in Transfer to Divide By 2 Clock Domain 12
Figure 5: SDF Fixes Race Condition to Divide By 2 Clock Domain 13
Figure 6: Reset Close to Target Register 15
Figure 7: Synthesis Result 16
Figure 8: AND-OR Combination Prevents Reset From Clearing X's 16
Figure 9: Logic Optimization Fixes X Problem 16
Figure 10: Even Better Logic Optimization 17
Figure 11: Simple Flop to Flop Path 17
Figure 12: "Creative" Logic for Simple Flop to Flop Path 18
Figure 13: Synchronizer Circuit 19

1.0 Introduction

With the widespread acceptance static timing and formal verification tools, companies are increasingly asking the question, "Do we still need to do gate level simulations?" The most common reason for asking this question is simple:

Gate level simulation is a great, big pain in the ASIC.

In a recent ESNUG article (http://www.deepchip.com/items/0421-01.html), eighteen engineers shared their view of the current usefulness of gate level simulation. Only one of those engineers has completely removed gate level simulation from their design flow. The other engineers listed many reasons for continuing to do some level of gate level simulation.

1. Since scan and other test structures are added during and after synthesis, they are not checked by the rtl simulations and therefore need to be verified by gate level simulation.
2. Static timing analysis tools do not check asynchronous interfaces, so gate level simulation is required to look at the timing of these interfaces.
3. Careless wildcards in the static timing constraints set false path or mutlicycle path constraints where they don't belong.
4. Design changes, typos, or misunderstanding of the design can lead to incorrect false paths or multicycle paths in the static timing constraints.
5. Using create_clock instead of create_generated_clock leads to incorrect static timing between clock domains.
6. Gate level simulation can be used to collect switching factor data for power estimation.
7. X's in RTL simulation can be optimistic or pessimistic. The best way to verify that the design does not have any unintended dependence on initial conditions is to run gate level simulation.
8. It's a nice "warm fuzzy" that the design has been implemented correctly.

Some design teams use gate level simulation only in a zero-delay, ideal clock mode to check that the design can come out of reset cleanly or that the test structures have been inserted properly. Other teams do fully back annotated simulation as a way to check that the static timing constraints have been set up correctly. In all cases, getting a gate level simulation up and running is generally accompanied by a series of challenges so frustrating that they precipitate a shower of adjectives as caustic as those typically directed at your most unreliable internet service provider.

There are many sources of trouble in gate level simulation. This paper will look at examples of problems that can come from your library vendor, problems that come from the design, and problems that can come from synthesis. It will also look at some of the additional challenges that arise when running gate level simulation with back annotated SDF.

2.0 Issues caused by Library Models

2.1 Pointing to Library Models

One of the first things that needs to be done to set up a gate level simulation is to point to the technology vendor's library models. The following VCS command line arguments are used to accomplish this task:

- -v <path to models file>
  The –v option is used to point to a single file which may include verilog modules for one or more library cells.
- -y <path to models directory>
  +libext+<filename extensions used by files in –y directories>
  The –y option is used to point to a directory. This directory contains a separate verilog file for each library cell. For VCS to find the code, the filename must match the module name and use one of the filename extensions defined by the +libext+ argument.
- +incdir+<directories to look for `include files>
  Sometimes (although rarely) a technology vendor's library files may `include other files. The +incdir+ option is used to define the directories where VCS will look for the `included files.

When using the –y option to point to a directory of library cell models, the module name and filename must match exactly. Sometimes you may find that they do not match; for example, the module name is uppercase and the filename is lowercase. In these situations you must work around the problem by using the –v option to point directly to the desired file(s). You should also fix the problem at its source by informing your technology vendor of the troubles they are causing you.

Some technology libraries are set up with separate verilog files for each library cell, but the modules depend on user defined primitives which are collected in another file. In these cases, the –y option can be used to get the library cell models, but the –v option will also be needed to point to the file containing the user defined primitives.

Instead of directly including these –y and –v options on the VCS command line, they are often put together into a technology specific file that can be referenced with VCS' –f option.

```
+libext+.v
-y /techlibs/your_technology/v6.0/verilog/
-v /techlibs/your_technology/v6.0/verilog/cell_udps.v
-y /techlibs/your_technology/v6.0_rams/verilog/
-y /techlibs/your_technology/v6.0_custom/verilog/
```

2.2 Specify Blocks

Many library models include specify blocks which define a delay between a module's inputs and outputs. When trying to do a non-SDF, zero-delay simulation, these specify delays can get in your way. The verilog for a buffer with a specify block may look like this:

```
module BUFX2 (Z, A);
output Z;
input A;
  buf U0(Z, A);
  specify
    (A *> Z) = (1.0, 1.0);
  endspecify
endmodule
```

Typically, the specify blocks of the modules are set up in one of two ways:

1. All library cells are given specify block delays of 1ns as shown for the buffer above. This arrangement will almost certainly cause simulation failures. If your simulation is running with a 250MHz clock (4ns cycle time), any logic path with more than four gates will not finish its updates in time for the next clock cycle to begin. Simulations with this problem can often be difficult to debug as the logic at first appears to be engaged in all kinds of interesting, creative, and nonsensical behavior. Adding **+nospecify** to the VCS command line (or the technology specific options file described in section 2.1) is the correct way to work around this problem.
2. The flops in the library may be given specify delays of 1ns or 0.1ns, but all combinational cells are given specify delays of 0ns. This type of library setup can usually be run without using the +nospecify option. The +nospecify option may still be needed if your clock is faster than the specify time. Even if the +nospecify option is not required, you may want to use it anyway because it does improve simulation performance.

In some cases, the specify blocks are included inside of `ifdef statements and can be avoided by using the appropriate `define. The define can be set in either your testbench, or on the VCS command line (or technology specific options file) with the +define+ command line option.

2.3 #1 Delays (also called #$%*&^ delays)

Some library models include # delays instead of, or in addition to, the specify delays described above. Similar to the problems seen with specify delays, #1 delays primarily cause troubles when they are used on combinational cells. VCS will igore the # delays when you set **`delay_mode_zero** in your testbench code or use the **+delay_mode_zero** option on the VCS command line. Using delay_mode_zero has the potential of causing other problems since testbench code almost always includes some # delays. Fortunately, delay_mode_zero only affects the delay specifications on gates, switches, continuous assignments and module paths. Delays that are specified within initial or always blocks are unaffected. If all of the # delays in your testbench code are inside of initial or always blocks, delay_mode_zero should allow you to work with the library's #$%*&^ delays.

2.4 Pessimistic X propagation

The way that the library cell models are coded can have a significant impact on how effective your reset signal is at clearing the initial X's from your design. A classic example is a simple multiplexer. Consider a multiplexer model coded like this:

```
primitive MUX2 (Z,D0,D1,SD);
  output Z;
  input  D0,D1,SD;
   table

// D0 D1  SD :  Z
//
   1  ?   0  :  1 ;
   0  ?   0  :  0 ;
   ?  1   1  :  1 ;
   ?  0   1  :  0 ;
   endtable
endprimitive
```

While this model is functionally correct, it can be pessimistic in its propagation of X's. When the D0 and D1 inputs are both 1'b1, the output of the mux should be 1'b1, regardless of the state of the SD input. However, with the above code, a 1'bx on the SD will cause a 1'bx on the Z output, even if D0 and D1 are both 1'b1. The following two lines are typically added to the UDP table for a mux:

```
// D0 D1  SD :  Z
   0  0   x  :  0 ;
   1  1   x  :  1 ;
```

While we have never seen a vendor's library with pessimistic X propagation on a multiplexer, we have seen this problem on more complex gates. Sometimes an element needed to be dropped from the logic equation as in this example:

```
// Original : Z = A&!B | A&B&C
// Fixed    : Z = A&!B | A&C
//  When A=1'b1 and C=1'b1, the output should be 1'b1,
//   regardless of the value of the B input. In the
//   original equation B=1'bx will cause Z=1'bx. In the
//   fixed equation, B=1'bx will correctly result in
//   Z=1'b1.
//  Note : The fixed equation is logically equivalent
//         to the original equation
```

Sometimes an element needed to be added to the logic equation:

```
// Original : Z = A&!B&C | A&B&D
// Fixed    : Z = A&!B&C | A&B&D | A&C&D
//  When A=1'b1, C=1'b1, and D=1'b1, the output should
//   be 1'b1, regardless of the value of the B input.
//   In the original equation B=1'bx will cause Z=1'bx.
//   In the fixed equation, B=1'bx will correctly
//   result in Z=1'b1.
//  Note : The fixed equation is logically equivalent
//         to the original equation
```

In all cases, the ultimate fix was obtained by having the technology vendor modify their library models.

3.0 Issues caused by Design

3.1 Registers that don't get reset

Although resetting every register in the design can be an effective way of clearing startup X's from a simulation, all those resets can affect not only the area, but also the performance and routability of the design. Frequently, many designers will leave some registers without a direct reset. Sometimes, the un-reset registers are expected to run without difficulty, and sometimes the testbench needs to provide a mechanism to remove the initial X's from these registers.

One example of a register that may need help to remove its initial X's is a clock divider. A clock divider is created by having a register feed back into itself through an inverter.

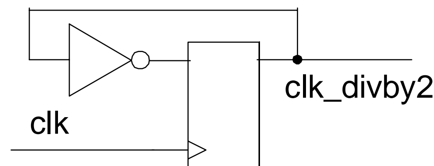

Figure 1: Simple Clock Divider

The feedback path, however, will prevent the clock divider register from getting out of its initial X state. While the clock divider logic can include a reset, this can cause other reset problems as seen in the following example. In this example, the clock divider will be successfully reset. However, as long as rst is active, clk_divby2 is not toggling. Therefore, none of the flops in the divby2 domain will be successfully reset.

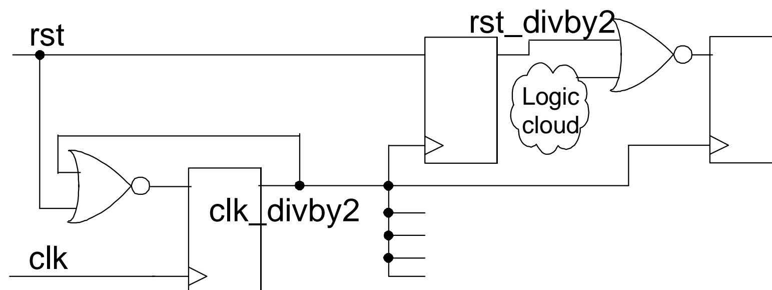

Figure 2: Clock Divider with Reset (won't work)

One solution to this problem is to have two resets. One resets the clock dividers in the design and the other is used to reset all of the other flops. Another solution is to omit the reset signal from the clock divider as shown in figure 1. This solution will require some extra code to clear the X out of the register at startup . Prior to synthesis, this code can reside in the RTL. However, it is usually non-synthesizable code, and it is contained in an `ifdef or synopsys_on/off structure.

```
module SLE_CLKDIV_FLOP (Q,CLK);
   output Q;
   input  CLK;
   reg    Q;

   always @(posedge CLK) Q = ~Q;

   `ifdef EN_START_VALS
      initial Q = 1'b0;
   `endif

endmodule // SLE_CLKDIV_FLOP
```

After synthesis, the initial block will no longer exist, and some other method is required to force an initial condition into the register. If the library model for the flop uses a reg construct, the verilog assign statement can be used at time zero to set an initial condition on the register. Most technology libraries, however, define the functionality with a user defined primitive (UDP) that does not have a reg that can be assigned. The following "gate level kicker" code can be used to initialize such registers.

```
`define KICK_REG0 testbench.tx_phy. \clkdiv_reg/U1
initial begin : kicker0
   reg kick_val;
   kick_val = $random(random_seed) % 2 ;
   $display("Initializing KICK_REG0 to %b", kick_val);
   force `KICK_REG0 .Q = kick_val;
   //  If flop has inverted output, kickstart that also
   //force `KICK_REG0 .QN = ~kick_val;
   @(posedge `KICK_REG0 .CLK);
   release `KICK_REG0 .Q;
   //release `KICK_REG0 .QN;
end
```

With this code, the actual internal state of the register has not been changed. However, by overriding the Q (and/or the QN) output, the initial X in this register is not allowed to propagate to downstream logic. The value is chosen randomly to ensure that the downstream logic is not making any assumptions about the initial state of this register. Because of the clock divider's feedback loop, the D input to the register now has a non-X value on it, and this will clear the register when the clock starts running.

You can also "kick" the D input of the register instead of the Q and/or QN outputs. This will initialize the register when the clock starts. Until then, however, it will still allow the initial X to

propagate to downstream logic and show you how tolerant that logic is to unknown initial conditions.

In Verilog a posedge occurs on any of the following transitions:

1. 0 -> 1
2. 0 -> X
3. 0 -> Z
4. X -> 1
5. Z -> 1

This can cause some complications in the gate level kicker code above. One common case is for the first clock transition to be from X to 1. Many vendors' library models will not register the D input on an X to 1 clock transition. In this case simply waiting for two positive clock edges allows the register to initialize correctly.

```
    repeat (2) @(posedge `KICK_REG0 .CLK);
```

The gate level kicker code above also includes a hint that could save you a few hours of frustrating trial and error debug time. VCS uses a "." character to separate levels of hierarchy in a design. When flattening a design using Design Compiler and other tools, the names of the gates in the flattened design often include the previously hierarchical instance names and a "/" character is often used between these hierarchical instance names (for example "clkdiv_reg/U1" above). In verilog, a "/" character is only valid in an escaped identifier. An escaped identifier starts with a "\" character and ends with a space, tab, or newline. VCS further requires a space in front of the "\" character that starts the escaped identifier.

In the gate level kicker code above, the testbench instantiates a design as tx_phy. This tx_phy design has been flattened, so the name of the clock divider flop is now "clkdiv_reg/U1". There are several methods to incorrectly reference pins on this flop, and only one correct method.

```
force testbench.tx_phy.clkdiv_reg/U1.Q = kick_val;
// 1. Incorrect, with "/" in name, needs to be escaped identifier
force \testbench.tx_phy.clkdiv_reg/U1.Q = kick_val;
// 2. Incorrect, escape just the instance name, not the whole path
force testbench.tx_phy.\clkdiv_reg/U1.Q = kick_val;
// 3. Incorrect, escaped identifier needs to end with white space
force testbench.tx_phy.\clkdiv_reg/U1 .Q = kick_val;
// 4. Incorrect, VCS needs a space before the "\" character
force testbench.tx_phy. \clkdiv_reg/U1 .Q = kick_val;
// 5. Correct reference (finally!!)
```

3.2 Clock dividers

In addition to the problem of clearing the initial X value described above, clock dividers can also cause problems with race conditions in gate level netlists. As with other sources of race conditions, these problems can be very difficult to debug. Different simulators can simulate the circuit differently; different versions of the same simulator may give you different results, even different options such as enabling or disabling dumping can expose or hide the problem. The clock divider induced race condition typically occurs when data is being transferred from the fast clock domain to the divided clock domain.

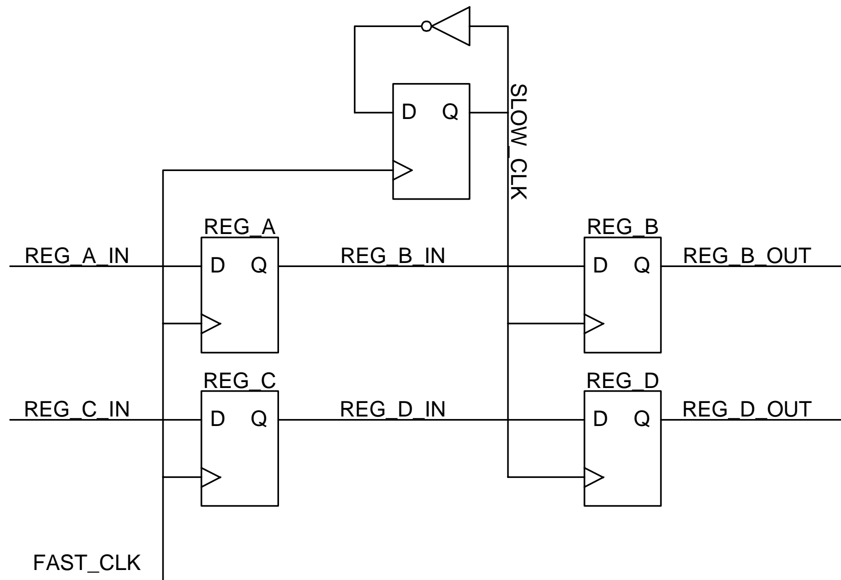

Figure 3: Transfer From Fast Clock to Divide By 2 Clock

In RTL simulations, regA and regC are typically described behaviorally, with non blocking assignments. The clock divider is either described with a blocking assignment, or with the instantiation of a particular gate from the target library. In either case the simulator will update the value of the clock divider before updating the regA and regC outputs. In gate level simulation however, regA, regC, and the clock divider register can be executed in any order. If the simulator chooses to execute regA, then the clock divider, then regC, the data flowing through this simple pipeline will be corrupted. In the following example, the data coming out of {regA,regC} is 00,11,00. However, regA executes before the clock divider and regC executes after the clock divider, and the data coming out of {regB,regD} is 00,10,01. Debugging a problem like this can take many hours if you have not seen it before.

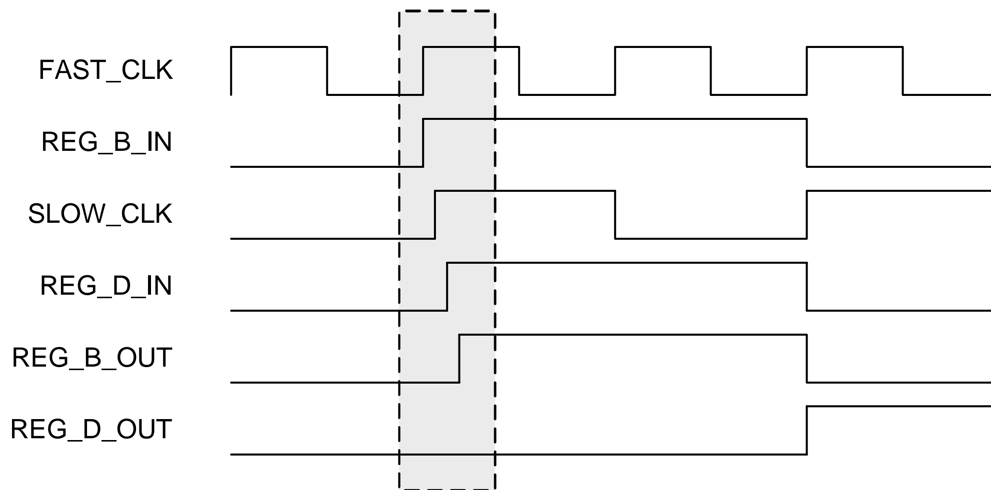

Figure 4: Race Condition in Transfer to Divide By 2 Clock Domain

The signal transitions in the shaded area of the waveform above all happen in the same time step. With typical waveform viewing they would be lined up with each other. Here the time step has been expanded to highlight the order of execution. This time step expansion can also be done in a simulation by using VCS' vcsplusdeltacycleon() feature to record the order of execution of items within a time step and Virsim's "Expand Time" feature to show this order of execution.

There are a few different methods for dealing with this problem. The first possibility is to avoid the problem through careful design. If the fast_clk registers update on the cycles where slow_clk is falling and hold their values on the cycles when slow_clk is rising, the problem will be avoided. This constraint does not need to be applied to all fast_clk registers, just those fast_clk registers that are sending their data into the slow_clk clock domain. Achieving this kind of synchronization between the clock domains is sometimes difficult and requires advance planning during rtl design.

Another design change that can be used to avoid the problem is to use a falling edge triggered flip flop for the clock divider. This guarantees that the rising edges of the original and divided clocks will be at different times and will therefore eliminate the race condition. While this solution can make testability and timing closure more difficult, it is a relatively easy change to make in RTL.

If the problem has not been avoided through the design, there are a few possible methods for working around it in gate level simulation. Each of these methods seeks to ensure that the registers in the slow_clk domain will not see a rising clock edge until after all of the fast_clk domain registers have been updated. Since there is no way to guarantee when the clock divider register will update relative to the other fast_clk registers, some delay needs to be added between the clock divider register and the clock inputs of the slow_clk registers. One way to do this is to use an SDF file to create a CLK->Q delay on the clock divider register. The SDF file may look like this:

```
(DELAYFILE
  (SDFVERSION  "3.0" )
  (DESIGN "sle_spi4_tx_phy")
  (TIMESCALE 1ns)

  (CELL
    (CELLTYPE "DFF")
    (INSTANCE clkdiv_reg\/U1)
    (DELAY
      (ABSOLUTE
        (IOPATH CLK Q (0.2))
      )
    )
  )
)
```

VCS uses the $sdf_annotate system task to apply the SDF to the netlist.

```
initial $sdf_annotate ("clk_div.sdf",testbench.tx_phy);
```

When used by the testbench, this SDF file will create 200ps of CLK->Q delay on the divider flop, ensuring that the fast_clk registers are all updated before the slow_clk registers see a clock edge.

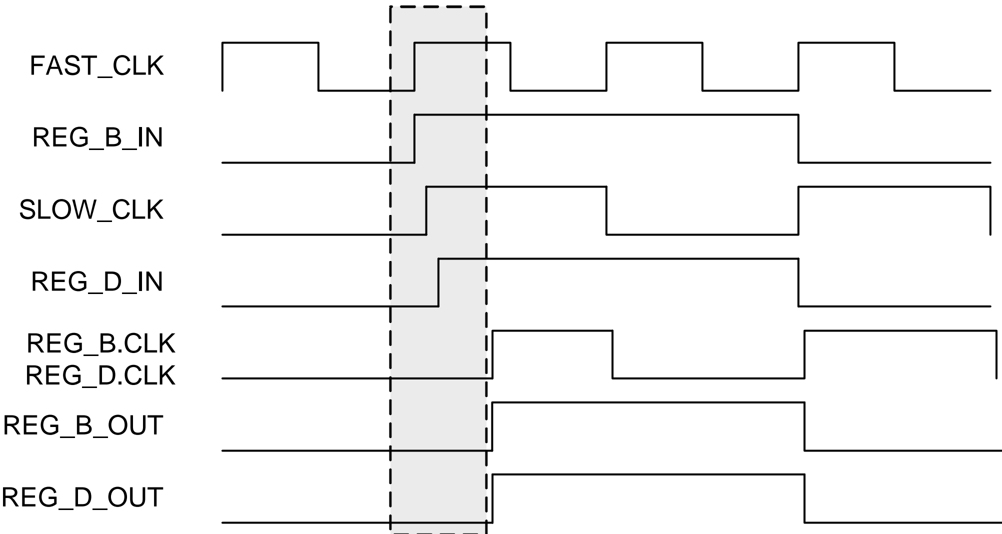

Figure 5: SDF Fixes Race Condition to Divide By 2 Clock Domain

Another potential workaround is typically available after the clock trees have been built. If the slow_clk clock tree starts with a single clock buffer at the root of the tree, that clock buffer can be given a nominal delay without the overhead of using an SDF file. In the gate level kicker code of section 3.1, a constant value was used in a Verilog force statement. The force statement, however, can also reference a wire or other variable, and when that variable changes, the force statement will be reevaluated. Adding delay to the slow_clk path can then be as simple as forcing some delay on the input of the root clock buffer.

```
initial force slow_clk_buf0_0.A = #0.2 testbench.tx_phy.
\clkdiv_reg/U1 .Q;
```

The disadvantage of this method is the care that must be taken when creating this force statement. If there is logic between the clock divider flop and the root of the clock tree (such as a bypass mux for testability), the force statement must account for that functionality, otherwise the functionality of the design will have been changed and the simulation results may not be accurate.

4.0 Issues caused by Synthesis

Even after all of the library and design issues have been addressed, the synthesis process can still create logic structures that cause trouble for gate level simulation.

4.1 Keep resets close to registers.

When using a synchronous reset methodology, the reset signal becomes part of the logic cloud feeding the flip flop's D input. When the reset signal comes in close to the register, it is able to effectively clear the initial X state from that register, even if the logic cloud is generating an X.

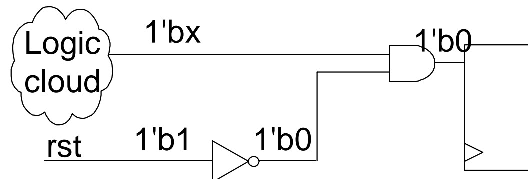

Figure 6: Reset Close to Target Register

During synthesis, however, logic optimization may pull the reset back further into the middle of the logic cloud. In some cases the resulting logic prevents the reset from being able to clear the initial X state from the register. Sometimes, the set_ideal_net synthesis constraint that is typically used on high fanout nets such as reset is sufficient for keeping the reset signal close to the registers. Another effective method for preventing synthesis from moving the reset signal farther back in the logic cloud is to set a late arrival time (large input delay) on the reset signal. With a late arrival time on the reset input, synthesis seeks to meet the required timing constraints by minimizing the amount of logic between the reset input and the registers that it goes to. The synthesis constraint may look something like this:

```
# Allow 500ps for reset logic
set RST_LOGIC 0.5
set RST_IN_DLY [expr $CLK_PERIOD - $CLK_UNCERT - $RST_LOGIC]
set_input_dly $RST_IN_DLY -clock clk [get_ports rst]
```

4.2 Incomplete Logic Optimization Interferes With Reset

Even when the logic feeding a register does include a reset and steps are taken in synthesis to ensure that the reset path is kept close to the register, a netlist can still be created which prevents the reset from clearing the X's. In one case that we've seen recently, we started with RTL which looked like this:

```
always @(posedge clk) begin
   if (rst) begin
      count[4:0] <= 5'h10;
   end else begin
      <Logic cloud>
   end
end
```

And the resulting gates from Design Compiler (v2003.06-SP1) for bit 4 of this register looked like this:

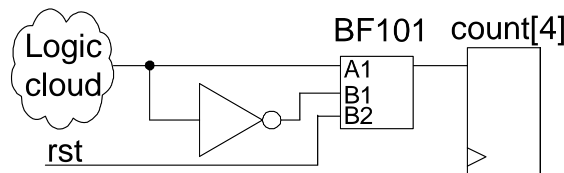

Figure 7: Synthesis Result

At first glance, this looks good, with the reset signal coming into the final gate before the register. However, the BF101 gate is an AND-OR combination, so the function really looks like this:

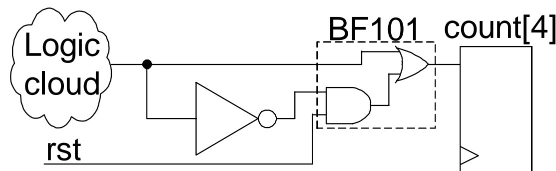

Figure 8: AND-OR Combination Prevents Reset From Clearing X's

With this logic structure, the reset signal is not able to overcome X's in the logic cloud and successfully reset the register. With a little Boolean Algrebra, however, we find that the B1 path is completely unnecessary and could have been tied to 1'b1:

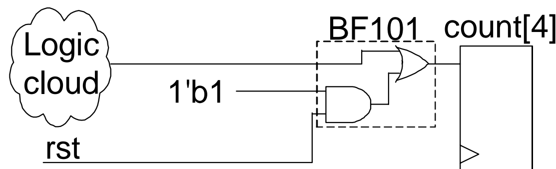

Figure 9: Logic Optimization Fixes X Problem

Even simpler yet would be to use a simple OR gate instead of the BF101:

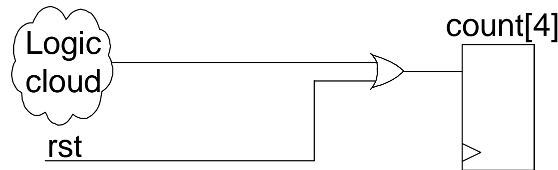

Figure 10: Even Better Logic Optimization

Either of these two modifications will allow the register to reset properly and the gate level simulation to run better. Of course, manual edits of a gate level netlist are not a popular addition to the design flow. Instead, we have reverted to Design Compiler 2003.03 where the problem has not been seen (yet), and we will report the problem to Synopsys Support.

4.3 Feedback problems

This example shows how even simple designs can go horribly wrong. The RTL in one recent design looked like this:

```
always @(posedge clk) begin
  dout_reg <= din_reg;
end
```

It was a simple pipeline stage with no logic between the flops. The expected synthesis output was this:

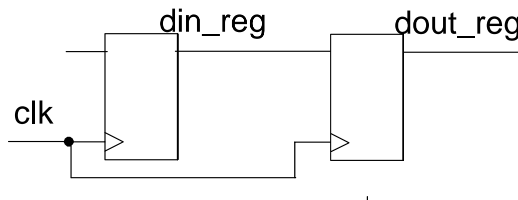

Figure 11: Simple Flop to Flop Path

At simulation startup dout_reg, like all registers, would start with a value of X. Even though dout_reg did not include a reset, din_reg does get reset, and in one clock cycle, this reset value should propagate through dout_reg, clearing the startup X's.

The target technology included both normal and inverting flops, although the inverting flops were noticeably faster than the non-inverting variety. The gates that came out of synthesis used inverting flops that also included an enable input and the circuit looked like this:

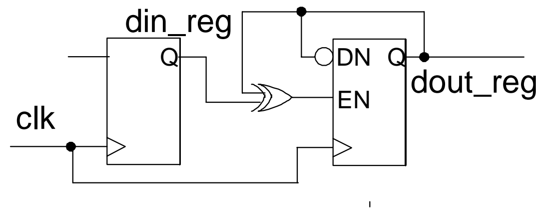

Figure 12: “Creative” Logic for Simple Flop to Flop Path

This is certainly a creative way to implement the functionality. However, it causes serious troubles for gate level simulation. The startup X on dout_reg controls the XOR, preventing the X from ever getting cleared. While we haven’t seen this synthesis result recently, the fixes at the time included adding these lines to our synthesis scripts:

```
set hdlin_keep_inv_feedback {FALSE}
set_dont_use <flops with enable inputs>
```

5.0 Issues when running SDF based gate level simulation

With continuing improvements in the realms of design verification methodologies, static timing analysis, and formal equivalency checking, SDF based gate level simulations are gradually migrating toward obsolescence. However, some companies still choose to perform limited amounts of back-annotated simulations as a way to double-check static timing analysis results. Unfortunately, even if all other X sources have been identified and resolved, moving on to SDF simulations can present a host of new challenges.

The purpose of SDF simulation is to prove that the design will run at the specified operating frequency while modeling expected timing delays in the circuit. When setup and hold check error messages or functional failures generated during simulation are true failures, they indicate areas of your static timing scripts and constraints that need improvement. Unfortunately, timing-based simulations can also generate false errors so egregious that they will have you thinking about what size box you want for your personal effects and where you would like you eat your farewell lunch.

5.1 Effects of timing failures

As one would expect and desire, timing violations occurring in a simulation get appropriately flagged with an often verbose error message. When the messages are caused by real errors in the design, all of the effort invested in getting gate level simulation running is suddenly justified. However, if the error messages are being caused by a false timing violation, especially in a place where they will repeatedly occur such as an asynchronous boundary, the result can be a flood of unnecessary and invalid timing errors in your simulation log file. Interpreting and scrutinizing

these failures can be a long and laborious process, especially when no prior preparation has been made to anticipate timing failures.

Typically, a timing failure will not only result in a message in the error log, but it will also propagate an unknown value on to the register output. Obviously, if a particular register is designed such that it is not intended to meet timing, this behavior will likely introduce terminal cases of X propagation. Some libraries will allow for modification of this behavior by enabling certain defines. In other cases, unique, hand-modified local libraries may be required to handle the problem. In either case, understanding that the issue might develop is a step toward mitigating the time lost debugging this category of problems.

5.2 Expected Timing Violations

There may be some registers in the design that are expected to fail setup and hold timing checks. These registers are designed into the logic to provide a specific function and may not ever be intended to pass flop-to-flop timing checks at operating frequency. While the static timing assertions are likely to account for this, simulation timing checks will catch the setup/hold violations, generate errors, and likely become an X source in the block. The most common circuits that may be expected to have setup/hold violations are synchronizers used for clock domain crossings and multi-cycle paths.

5.2.1 Synchronizers for clock domain crossings

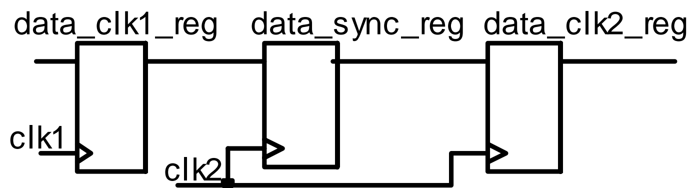

Figure 13: Synchronizer Circuit

Clock domain synchronizers exist for the sole purpose of resynchronizing a signal from one clock domain to another. As such, there is never any guarantee for arrival times at the first capture flop of the synchronizer (data_sync_reg in figure 13). Unfortunately, the timing checks enabled on the library element that models the capture flop will still activate, and upon any setup or hold violation (which are frequent on an asynchronous boundary) will fail timing checks. Enabling successful gate level simulation will typically require removing the synchronizer's setup and hold constraints from the SDF file. This modification to the SDF file can be greatly eased if a consistent naming convention is used for all such asynchronous clock domain crossings in the design. Another powerful methodology for dealing with this issue was outlined in Paul Zimmer's SNUG San Jose 2003 paper, "My Favorite DC/PT TCL Tricks." Paul's methodology uses Primetime scripts to create verilog that a testbench can use to force the notifier that most library flip-flop models use to do the X assertion.

5.2.2 Multi-cycle paths

Multi-cycle paths are another design element that can create false errors in timing based simulations. In this case, the designer understands the need for more than one clock cycle to propagate through a logic cloud, and the STA environment will be appropriately set up to handle the multi-cycle requirement. Unfortunately, the simulation model checks could still recognize and flag timing violations. Similar to the synchronization registers, the endpoints of multicycle paths generally need to have their setup and hold checks removed from the SDF file.

An alternative is to allow the setup/hold check fail and the resulting X to propagate through the design. If the path is truly a mutli-cycle path, the register output should not be used during the cycle when it is at an X value, and allowing the register to go to X during that cycle provides an extra check to ensure that the functionality truly operates this way. When using this approach, some post processing of the log file is typically required to filter out the timing violations that have been accepted on the multi-cycle paths.

6.0 Conclusions and Recommendations

The road to successful gate level simulation can be riddled with potholes and profanity. The obstacles can come from the library models, the design, or the synthesis process. While some of the problems can be avoided through proper influence on your library vendor's models or proper design guidelines and planning, other problems will simply need to be addressed as they are encountered. By knowing the areas where these problems can occur, we hope that you are able to more quickly find appropriate solutions and begin to use some of the *other* four letter words…Pass…Good...Done!