DesignCon 2005

Automatic Verification of Timing Constraints

Ramin Hojati, Ph.D., Averant, Inc.
rhojati@averant.com

Yen-min Chiu, Averant, Inc.
ychiu@averant.com

Abstract

False paths and multi-cycle paths present an enormous problem when trying to achieve timing closure in contemporary, high-speed designs. Chip design teams typically do not specify these kinds of constraints at the start of a design, but rather apply them in response to timing problems encountered late in the design cycle.

In this paper, false paths are explained as a path through a circuit that is not responsible for the circuit delay due to the nature of the logic. A sample false path is shown for reference purposes. A circuit is then shown for which a false-path constraint has been generated by an automatic analysis tool. This constraint is shown to be incorrect, using formal verification techniques.

A multi-cycle path is a register-to-register path in a circuit where if the source changes, the destination should not be responsible for the delay over the next N cycles. A sample circuit is shown to explain more clearly what is meant by a multi-cycle path. In addition, multi-cycle path constraints are shown for the accompanying example, to show how these are specified. A different example is then offered showing multi-cycle path constraints that have been generated from another analysis tool. Using formal verification techniques, these constraints are shown to be incorrect for the particular design example.

The paper concludes showing how properties (assertions) are written to verify correctness of timing constraints and may be used to verify constraints that are either generated by hand or by using automatic tools. This formal verification process is easy to setup, quick to run, exhaustive in its analysis, and provides a measure of confidence that the right constraints are being used to verify a design.

Authors Biography

Dr. Ramin Hojati is the President and founder of Averant. He obtained his B.S. from Massachusetts Institute of Technology in 1988 in math and computer science. From 1988 to 1990, he worked on layout and logic synthesis tools at Cadence Design Systems. He obtained his M.S. and Ph.D. in computer science in 1992 and 1996, respectively, from the University of California, Berkeley. Dr. Hojati has been the main architect of several large scale software developments, has written over 20 research papers in verification and delivered many talks in CAD, has served in the program committees of trade conferences, and is a leading figure in computer-aided verification. At U.C. Berkeley, he was the main driving force behind Berkeley's first generation verification system, HSIS. Before that, he was the author of the first BDD-based engine for Lucent Technologies FormalCheck. Dr. Hojati has significant background in design verification, and has been a design verification consultant to Cisco, Compaq, IBM, SGI, and SUN Microsystems

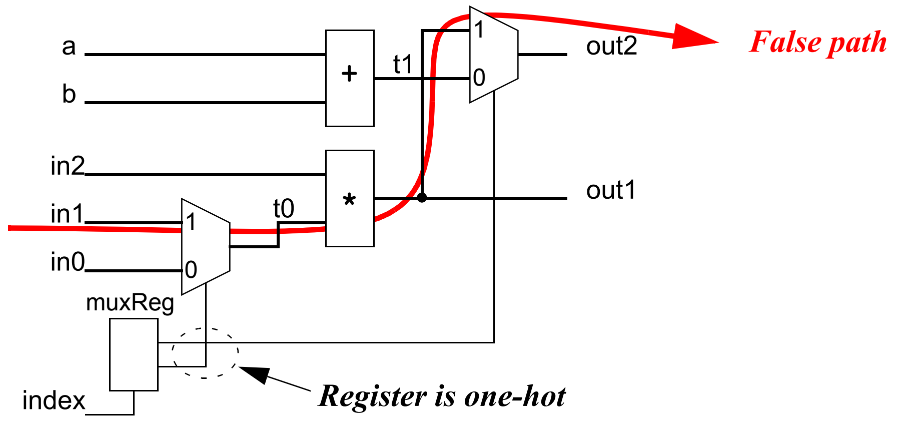
Yen-min Chiu is a principal developer at Averant, Inc. He obtained a Bachelors in Electronic Engineering from the National Taiwan Institute of Technology, and a Masters in Computer Engineering from the University of Massachusetts, Lowell. Mr. Chiu has held numerous developer positions in the EDA industry, including Principal Software Engineer for VIEWlogic Systems, where he developed the Speedwave VHDL simulator. Mr. Chiu has also held key engineering positions with such firms as Racal-Redac, Silc Technologies, and Texas Instruments.

Introduction

False paths and multi-cycle-paths (MCP) are timing exceptions that present a particularly difficult problem when trying to achieve timing closure in modern, high-performance designs. Typically, these exceptions, as well as all timing constraints, are considered late in the design cycle and are specified in response to timing problems during verification. For optimum timing results, all timing exceptions must be guaranteed to be correct. SolidTC is a formal tool that reads the RTL description of the design (Verilog or VHDL) as well as the timing constraints (SDC format) and proves conclusively that all timing exceptions are valid. Otherwise, a waveform is generated illustrating why a path is not a false-path or a MCP.

What are False Paths?

A false path is a path through a circuit that cannot be responsible for the circuit delay and no sequence of vectors result in the propagation of an event along the path. An example of a false path is shown below. Notice the register **`muxReg`** that controls the two muxes in the design is one-hot, and as result there is no path from the input **`in1`** to the output **`out2`**. Therefore this path may be ignored when performing timing analysis.



The source code for this example is included below.

```
module fp2( index, in0, in1, in2, a, b, out1, out2);
   input          index;
   input   [15:0] in0, in1, in2;
   input   [31:0] a, b;
   output  [31:0] out1, out2;
   reg     [1:0]  muxReg;
   wire    [15:0] t0;
   wire    [31:0] t1;

always @* begin
   muxReg        = 2'b0;
   muxReg[index] = 1'b1;
end

assign t0    = (muxReg[0] ? in1 : in0);
assign out1  = t0 * in2;
assign t1    = a + b;
assign out2  = (muxReg[1] ? out1 : t1);

endmodule
```

This says the register "muxReg" is onehot

The dataflow part of the design, controlled by muxes

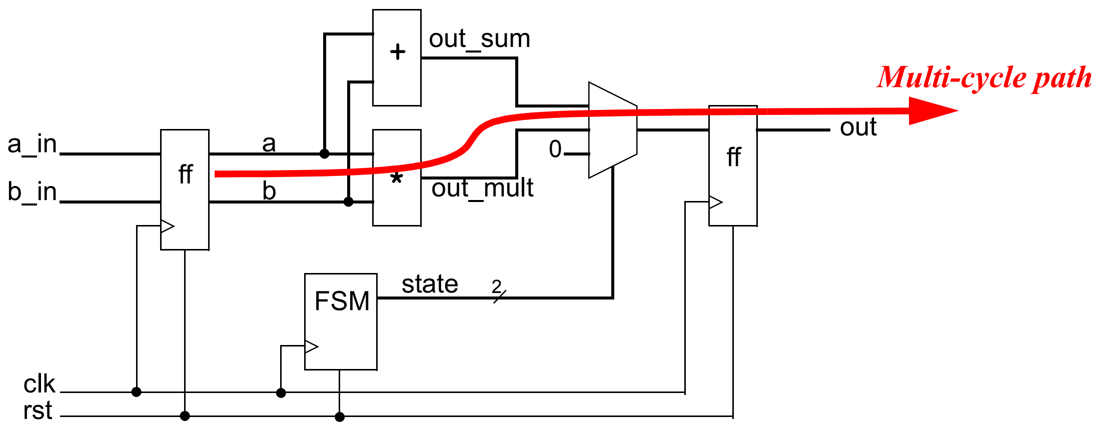
The corresponding constraints file would specify this false path as follows:

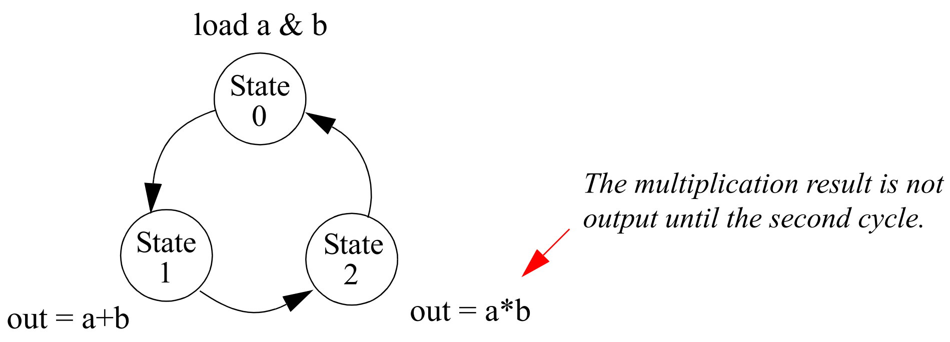
```
set_false_path -from in1[*] -to out2[*]
```

What are Multi-Cycle Paths?

A multi-cycle path in a design is a register-to-register path through some combinational logic where if the source-register changes, the path will require ***N*** cycles (where ***N*** > 1) before the computation is propagated to the destination register. In other words, if the source register changes, then the destination register should not change in the next ***N*** cycles under any delay condition.

The circuit below takes the signals **`a_in`** and **`b_in`** and inputs them to both an adder and multiplier. A finite-state machine controls the outputs such that the addition is driven out in one clock cycle, while the multiplication is driven out a cycle later. In other words, it takes two cycles to produce the multiplication result.




load a & b
State 0
State 1
State 2
out = a+b
out = a*b
The multiplication result is not output until the second cycle.

In this case, the user might simply specify the multi-cycle path constraint in this manner:

```
set_multicycle_path 2 -from {a[*]} -to {out[*]}
```

However, this is too pessimistic, as it also specifies the **`out_sum`** path as a two-cycle path. The following timing constraint is the correct one:

```
set_multicycle_path 2 -from {a[*]} -through {out_mul[*]} -to
{out[*]}
```

The Verilog source code for this example is included below.

```
module mcp_chg( clk, rst, a_in, b_in, out);
   input   clk, rst;
   input   [31:0]  a_in, b_in;
   output  [31:0]  out;
   reg     [31:0]  a, b, out;
   reg     [1:0]   state;
   wire    [31:0]  out_sum, out_mul;

assign out_sum = (a + b);
assign out_mul = (a * b);

always @(posedge clk) begin
   if (rst) out = 0;
   else     out = (state == 1 ? out_sum : state == 2 ? out_mul : 0);
end

always @(posedge clk) begin
   if (rst) begin
     a = a_in; b = a_in;
     state = 2'd1;
   end
   else begin
     if (state == 2'd0) begin
          a = a_in;
          b = b_in;
          state = state + 1'b1;
     end
     else if (state == 2'd1)
          state = state + 1'b1;
     else if (state == 2'd2)
          state = 2'd0;
   end
end

endmodule
```

Load up a and b in cycle t0.
- In t1, out gets a + b.
*- In t2, out gets a * b.*
The multiplication path is a 2 cycle MCP.

Automatic Verification of Timing Constraints

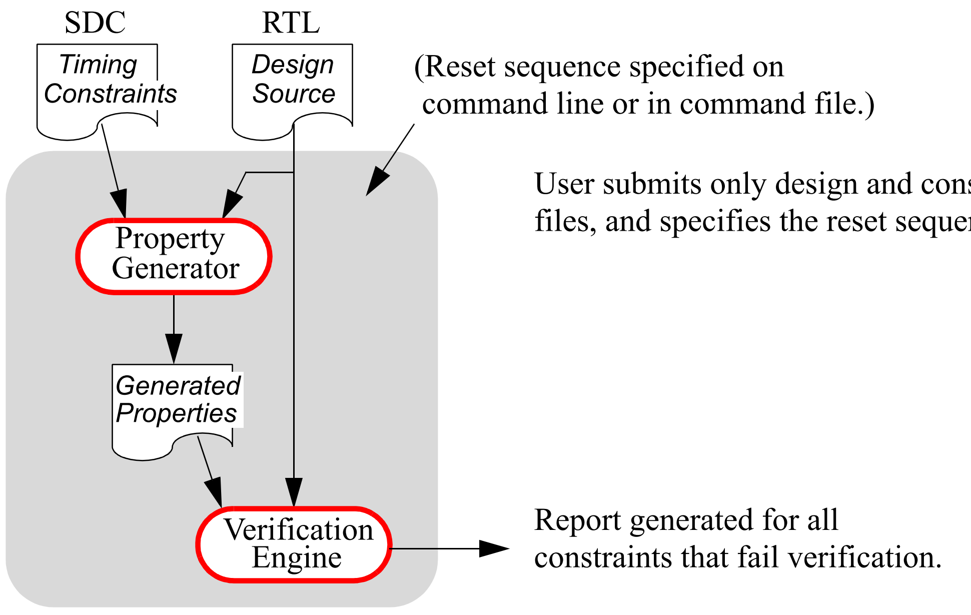
Currently false paths and multi-cycle paths are specified manually, which can be a tedious and haphazard process. Even in the case where timing constraints are generated automatically, verifying these constraints is done by visual inspection which is time consuming and error prone. Circuits verified with incorrect constraints run substantial risk of fatal bugs making it all the way to silicon.

Fortunately, formal verification can be used to analyze false path and multi-cycle path constraints and verify their correctness. SolidTC, a timing constraint verifier from Averant, applies formal verification technology to the problem of verifying timing constraints in complex, multi-million gate designs.

Flow Through the Tool

Automatic verification of timing constraints is performed by first analyzing the circuit and generating verification properties for the design. This is handled automatically and does not require any user input or intervention. All that is needed is to specify the design source, the constraint file, and the reset sequence (whatever sequence is needed to reset the logic to its initial, stable state. Eg.: hold the signal `reset` low for 2 clock cycles.)

The generated properties are then passed to the formal verification engine. Analysis is performed, and failures associated with a given constraint are reported to the user.


SDC
Timing Constraints
RTL
Design Source
(Reset sequence specified on command line or in command file.)
User submits only design and constraint files, and specifies the reset sequence.
Property Generator
Generated Properties
Verification Engine
Report generated for all constraints that fail verification.

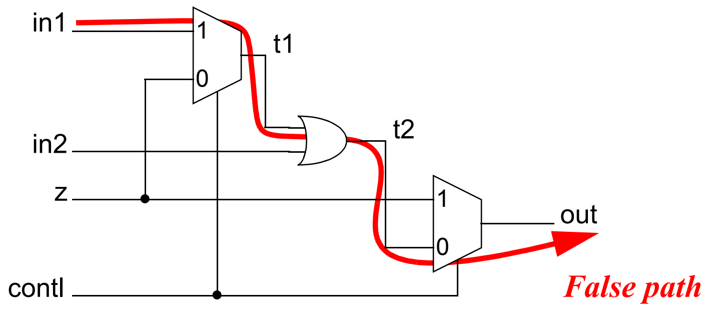
It's important to note this is a RTL tool. When a path is reported as false, it is false under *all* delay assignments to gates and wires.

To better appreciate the value of such an analysis, we'll take a look at a more complex circuit and see if we can determine the correct timing constraints.

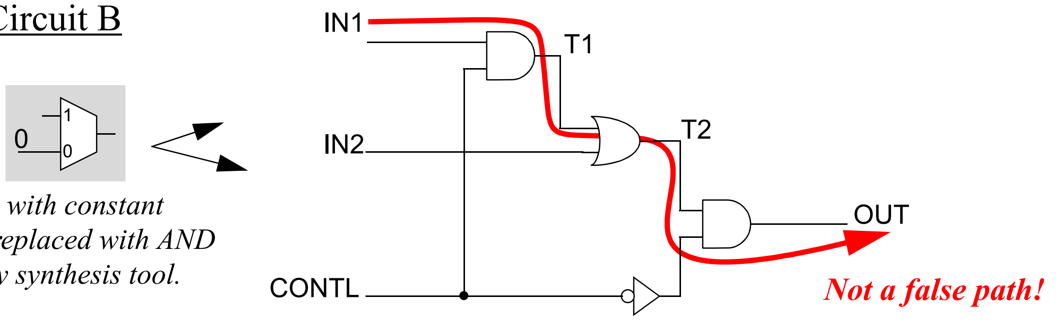
Example of Incorrect False-path Constraints

Below are two design examples for which constraints have been created. In the first case, the false-path constraints are correct, and the path may be ignored during timing analysis. However in the second case, the constraints written for the designs are incorrect, that is, they specify conditions that should *not* be ignored during timing analysis, as the path can be responsible for delay.

Circuit A



Circuit B


0
1
0
Muxes with constant input replaced with AND gate by synthesis tool.
IN1
T1
IN2
T2
OUT
CONTL
Not a false path!

The following constraint for Circuit A is correct:

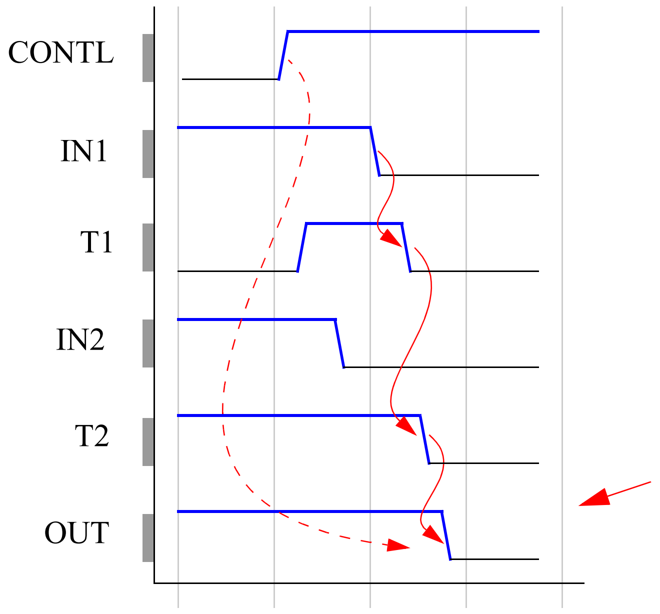
```
set_false_path -from in1 -to out
```

However, the following constraint for Circuit B is *incorrect*:

```
set_false_path -from IN1 -to OUT
```

The second circuit is derived from the first, except that the **z** input has been tied to a constant 1. As result, the synthesis tool replaced the **MUXes** with **AND** gates, with the correct phase for the inputs. To begin our analysis, assume the following values **{CONTL=0, IN1=1, T1=0, IN2=1, T2=1, OUT=1}**. Assume in the new cycle, **CONTL** goes to 1 at the input of the gate driving **T1**. **T1** becomes 1 as the result of this. Assume after this change, **IN2** goes to 0. We still have **T2=1** and **OUT=1**. Now, assume **IN1** goes to 0. This will make **T1** to go to 0, and after that **T2** and **OUT** to go to 0. Assume, eventually, **CONTL** at the input of gate **OUT** goes to 1. This however has no impact on the value of the gate driving **OUT** as it has take its final value of 0. The path responsible for delay was **{IN1, T1, T2, OUT}**. Hence, it is not a false path. (Note: we needed to make appropriate assignment to wire delays to create a situation in which the path is not false.)

The timing of these events is shown in the diagram below.



Because of wire delays, CONTL is late de-gating the signal OUT. Therefore, IN1 contributes to the delay seen at OUT.

The source code for these circuits is included below.

```
module constant_mux( in1, in2, in3, contl, z, out,
                     IN1, IN2, IN3, CONTL, OUT);

  input   in1, in2, in3, contl, z;
  output  out;
  input   IN1, IN2, IN3, CONTL;
  output  OUT;
  wire    t1, t2;
  wire    T1, T2;


assign t1  = (contl ? in1 : z);
assign t2  = (t1 | in2);
assign out = (contl ? z : t2);


assign T1  = (CONTL ? IN1 : 1'b0);
assign T2  = (T1 | IN2);
assign OUT = (CONTL ? 1'b0 : T2);

endmodule
```

The first circuit with path in1, t1, t2, out being false

The second circuit with path IN1, T1, T2, OUT <u>not</u> being false

Excluding True Critical Paths

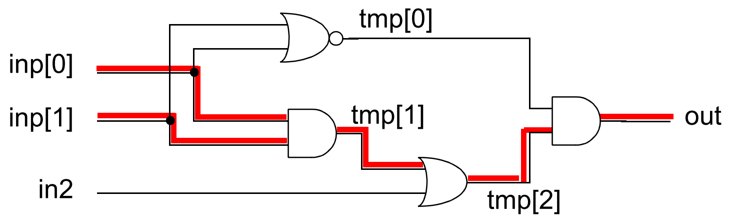
In the above example, even though the path {IN1, T1, T2, OUT} can be a critical delay-carrying path depending on the values of gate and wire, it is never a unique critical path. In other words, whenever this path carries delay, there is another path which is no shorter. In the example given above, the path {CONTL, OUT} is this path of equal or greater length. Therefore, if the path {IN1, T1, T2, OUT} is excluded in Static Timing Analysis by marking it as "false", we would be guaranteed that the analysis will not return a lower delay for the design.

However, caution is required when applying this technique so as not to exclude too many such paths. For the next example, we must review the concept of "static sensitization". A path in a circuit is considered statically sensitizable if there exists an input vector such that all the "side inputs" to the path are set to "non-controlling" values (Ref 1). A non-controlling value is one that allows changes to propagate through the path (i.e.: a non-controlling value for an AND gate is a 1, allowing changes on other inputs to propagate through.)

Consider the design below. This design contains two critical paths (assuming gate delays to be 1 and wire delays to be 0). The two critical paths are:

```
{inp[0], tmp[1], tmp[2], out}
{inp[1], tmp[1], tmp[2], out}
```

Both of these paths are "non statically-sensitizable". If only one of these paths is marked as "false" and is excluded from Static Timing Analysis, then the analysis will return the correct value of 3. However, the danger here is to assume *all* non statically-sensitizable paths can be considered as false. In this example, if *both* of these paths are marked as false, the analysis will return the incorrect value of 2.



Both paths are "non statically-sensitizable. Yet if they are both treated as false paths, the analysis will be wrong.

This example shows that when true critical paths are excluded, the addition of an extra "false" path constraint can make Static Timing Analysis return an incorrect value. Hence, when a new constraint is added, all previous constraints need to be verified to ensure analysis still returns the correct value.

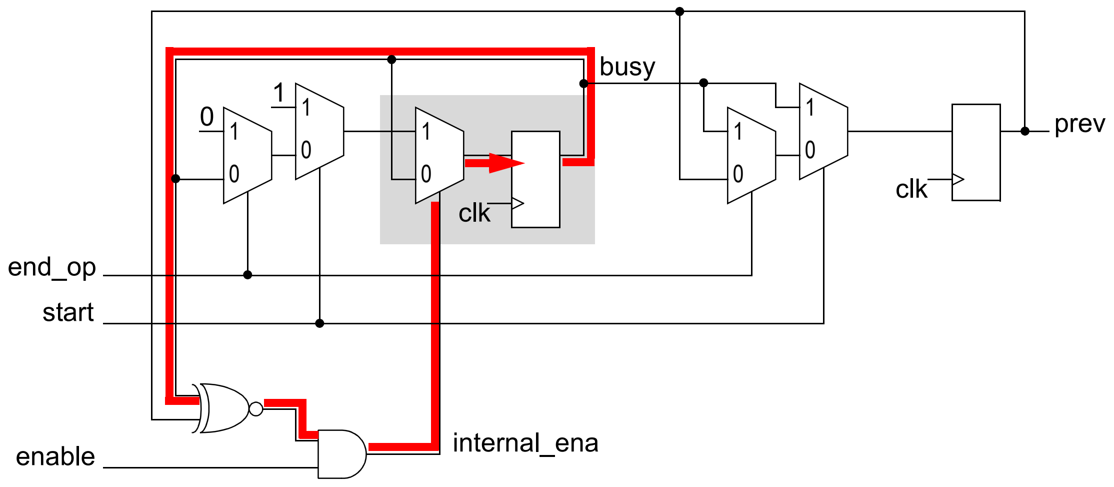
The source code for this example is included below.

.

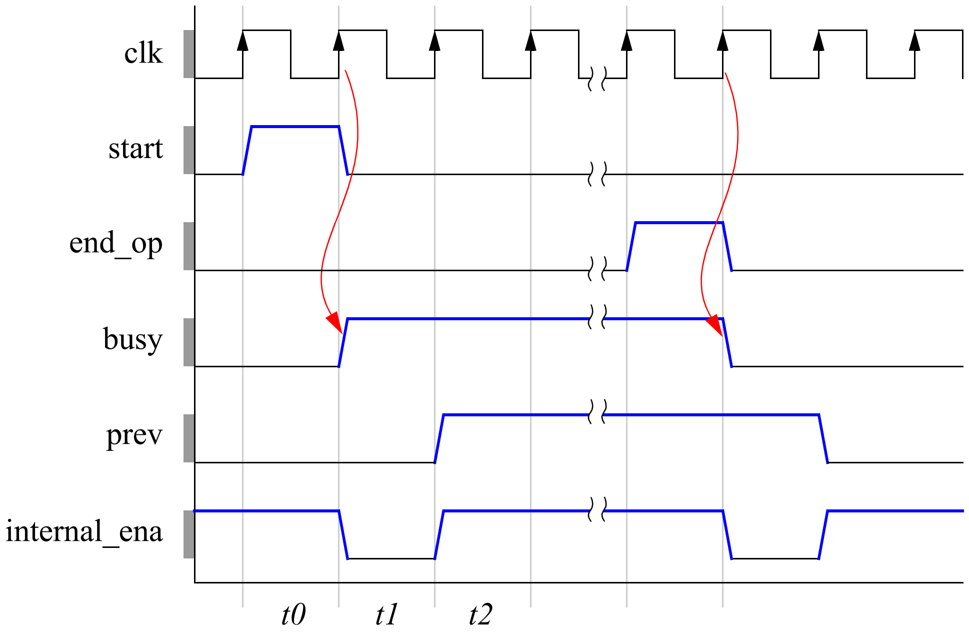
```
module excluded_fp(inp, in2, out);
  input [1:0] inp;
  input       in2;
  output      out;

  wire [2:0] tmp;

assign tmp[0] = !(inp[0] | inp[1]);
assign tmp[1] =  (inp[0] & inp[1]);
assign tmp[2] =  (tmp[1] | in2);
assign out    =  (tmp[0] & tmp[2]);

endmodule
```

Example of Incorrect Multi-Cycle Path Constraints

The following is an example of a design that appears to contain a multi-cycle path. As result, a timing constraint was generated for this design to exclude the path from resolving in a single cycle. However, this particular path is *not* multi-cycle, and must not be verified using relaxed conditions if the design is to perform at highest speed.



The following is the associated timing diagram.



The source code for this example is included below.

```
module busyFlag (start, end_op, enable, busy, clk, reset);
   input   start, end_op, enable, clk, reset;
   output  busy;
   reg     prev, busy;
   wire    internal_ena;

assign internal_ena = (prev!=busy) ? 1'b0 : enable;

always @ (posedge clk )begin
   if(reset) begin
      busy <= 1'b0;
      prev <= 1'b0;
   end
   else begin
      prev <= busy;
      if(internal_ena) begin
         if(start) begin
            busy <= 1'b1;
         end
         else begin
            if ( end_op ) begin
               busy <= 1'b0;
            end
         end
      end
   end
end
endmodule
```

The constraint below would seem to indicate the path from **`busy`** to **`busy`** is a MCP:

```
set_multicycle_path 2 -from busy -to busy
```

However, for the following reason, this is incorrect:

Assume **`busy`** changes from 0 to 1 between *t0* and *t1* (where *t0* is the initial time, and *t1* is right after the first clock cycle.) Then, we have that **`internal_ena`** must be 1 in *t0*. In *t1*, until the signals from the equation **`(prev != busy) & enable`** reaches **`internal_ena`**, we have that **`internal_ena`** is 1.

In that case, the next value of **`busy`** in *t1* can go to 0 temporarily if **`end_op==1`** and **`start==1`**. Once the signal from the equation **`(prev != busy) & enable`** reaches **`internal_ena`** (which will change it to 0), then we will latch in the previous value of **`busy`**, which will be 1. We've determined that after **`busy`** was unstable, the path from **`busy`** to (next)**`busy`** was responsible for delay in *t1*, which means the path is *<u>not</u>* a MCP.

Conclusion

Achieving timing closure is a critical factor in producing reliable, bug-free, high-performance designs. The key to this is thorough design verification, being sure not to inadvertently relax the design tests through the application of incorrect timing constraints.

SolidTC provides a valuable solution to check the validity of multi-cycle and false path constraints, helping to insure that designs are subjected to correct and thorough verification. Using SolidTC the user can not only exhaustively validate their constraints but also perform validation in a fraction of the time it would take to do it manually.

References

1. Y. Kukimoto, B. Chappell. "Functional Timing Analysis - How to Detect False Paths" Lecture at UC Berkeley, 1999.

2. S. Devadas, K. Keutzer, and S. Malik. Computation of floating mode delay in combinational circuits: Theory and algorithms. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 12(12):1913-1922, December 1993. The definitions and basic theorems are given in this paper. Draws heavily on earlier works, especially, that of Chen and Du in Tau 90.