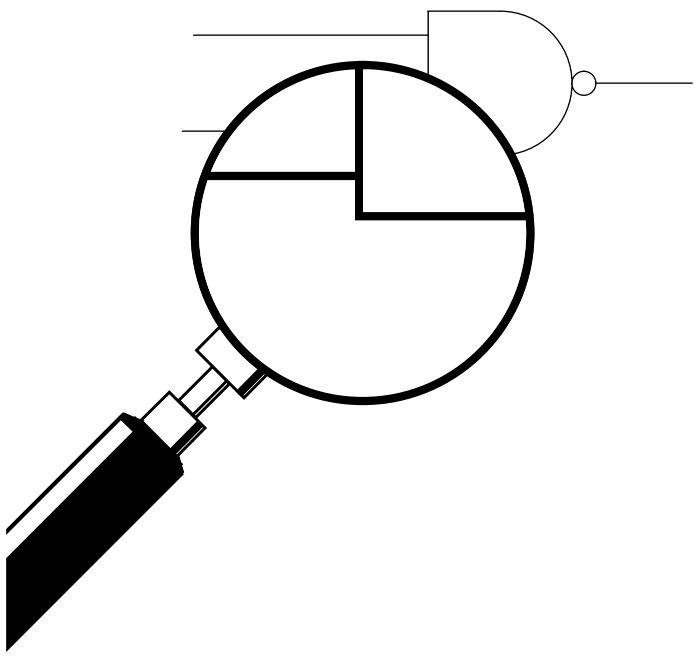
# SIMIC Engineering Guide
# For
# IC Design Verification



Genashor Corp.
9 Piney Woods Drive
Belle Mead, New Jersey 08502
Tel: (908) 281-0164
Fax: (908) 281-9607

Genashor Corp
9 Piney Woods Drive
Belle Mead, NJ 08502
Telephone: (908) 281-0164
Fax: (908) 281-9607

# Table Of Contents

# Introduction

## Overview

This Guide will provide you with criteria to evaluate logic simulators; either the simulator you are currently using, or commercial simulators currently on the market. All simulators are not created equal. They differ in their abilities to model components and component configurations accurately, to locate design problems and provide the debugging facilities to fix them, and to verify functional test vectors for tester program generation. This Guide also contains simple testcases that many simulators mishandle; if you have access to one of them, the results could be amusing. However, if you rely on this simulator to perform final design verification, the results could also be disturbing.

The importance of a simulator's ability to help locate and correct design problems cannot be overemphasized. Obviously, engineering productivity is increased if these problems can be located and corrected quickly. Also, product reliability is increased if the simulator supports a robust set of timing checks, since the corrected design will be more impervious to process and environmental variations, and less sensitive to input arrival times. Lastly, development costs are significantly reduced if problems are found *before* committing a design to silicon, rather than *afterwards*.

The time you spend reading this informative Guide will be well rewarded with the perspective gained on necessary and desirable simulator capabilities and, by contrast, on the shortcomings and idiosyncracies of some commercial simulators. Of course, since we want to convince you that SIMIC is the best simulator available today for verifying *real* designs, many of these capabilities are illustrated by describing SIMIC features. We believe that if you are seeking the most effective design verification tools, you will be convinced, after reading this material, that SIMIC definitely belongs in your design system.

## Organization Of This Guide

The design cycle can be conceptually divided into three phases. Section 1 discusses the first phase, <u>functional verification</u>, which addresses the design's correctness and completeness. The questions addressed in this phase are fundamental: have the functional specifications been met? is the architecture correct? is the polarity of every signal correct (missing inverters)? etc. Simulation is usually performed with TYPICAL delays, but MINIMUM or MAXIMUM delay simulations may also be used when the design's response time is critical. Statistical wiring delays may be incorporated into the circuit description.

Section 2 discusses the second phase, <u>timing verification</u>, which addresses the question of whether the design could have timing problems when implemented in silicon. Post-layout wiring delays are incorporated, and pessimism is introduced to account for possible differences between simulation and physical operation.

Section 3 discusses the third phase, <u>tester program verification</u>. As with timing verification, pessimism and wiring delays are used to detect potential testing problems that might arise, this time because of input skew and small timing margins between output strobes and output transitions.

In all phases, simulation accuracy is obviously essential. Also, the simulator should be capable of "expecting the unexpected", and help the user find problems as quickly as possible. For example, the simulator should:

(a) report unexpected or undesirable situations—oscillation, wire-tie conflicts, simultaneously-driven busses (in busses where only one driver should be active at any time), HIZ at unidirectional gate inputs, etc.,

(b) report the time required for the circuit's state to become stable, and signals that are still active beyond a specified time limit (Section 2 discusses the significance of these requirements),

(c) perform timing checks (setup, hold, etc.),

(d) support interactive debugging—break on specified condition, trace activity, save/restore state, force/release values at selected signals, etc., and,

(e) perform extensive topological and electrical checks (undriven gate inputs, overloaded driver, wire-tied non-tristatable gates, etc.).

## Historical Perspective

Prior to digital simulation, designers would "breadboard", or build a prototype of the circuit, using discrete components, and connect signal generators, logic analyzers, and oscilloscopes to debug the circuit. It soon became apparent that breadboards were inadequate to predict the behavior of integrated circuits; prototypes and manufactured ICs differed greatly in timing, device properties, and interaction with their external environments. For circuits containing a small number of devices, typically under 100 elements, the use of circuit simulation (i.e., analog simulation at the device level) was effective, but required significant post-simulation analysis of the results. With increasing circuit size, this approach became impractical, due to the extensive CPU and analysis time required. Rudimentary logic simulators were developed in the '60s to check design functionality. These early simulators usually performed functional verification in the absence of timing (or used arbitrarily timing, such as unit-delay).

More sophisticated simulators were developed internally by design houses and foundries that reflected the major concerns of their design environments. Simulation accuracy was the primary goal, the *sine qua non* for determining manufacturability. In the late '70s, a number commercial CAE vendors began promoting simulation capabilities within their design system. The figure of merit for logic simulators somehow shifted from accuracy to speed. These vendors battled over performance (whose simulator performed the highest number of events or element evaluations per second). In this battle, accuracy at the gate level suffered, as the electrical properties of gates became more abstracted to obtain additional performance. As these commercial simulators gained popularity in the '80s, design houses began abandoning their internally-developed tools to reduce their overhead and maintain compatibility with their customer base.

Vendors of current commercial simulators have attempted to reconcile these inaccuracies by force-fitting some electrical effects into their abstracted models. However, without building fundamental accuracy into the simulator's kernel, incorrect results can still be obtained that, under certain conditions, can yield steady-state deviation from actual device operation. Many simulators provide only basic capabilities in the simulator's kernel, necessitating workarounds in the models, which could detrimentally impact overall performance.

Many of SIMIC's features have been utilized in proprietary simulators (developed by Genashor personnel as early as 1974) that have received worldwide acclaim. The success of these concepts in finding design problems before manufacture has been verified by a number of large IC design houses. This track record is the basis of the simulation philosophy found in SIMIC. SIMIC builds on these concepts to bring a new level of accuracy and performance to logic simulators. It performs many topological and electrical rules checks as well as sophisticated hazard and timing analysis. Debugging circuits is simplified by SIMIC's unique interactive control and reports.

## Simulator Performance

Simulator vendors usually advertise one of two simulator performance measures: <u>evaluations per second</u> and <u>events per second</u>. These terms appear to be self-explanatory and universal, that is, a property of the simulator that is independent of the circuit being simulated. Unfortunately, actual performance on your *real* design may never come close to the advertised values, since it indeed depends heavily on circuit activity, delay distribution, and the simulator's primitive set. The vendor-supplied numbers are usually based on a best-case circuit that minimizes event scheduling and propagation, and slow activity such as output. The measures can also be misleading because they may not correspond exactly to common usage. For example, "events per second" should indicate the number of signal value transitions calculated per second. At least one vendor, however, defines this as the number of scheduled operations per second, where "operations" include simulator bookkeeping functions and other artifacts, dramatically inflating the performance value. Also, "evaluations per second" should indicate the number gates evaluated per second in order to determine their corresponding output values. However, some simulation algorithms force gates to be evaluated even when no activity occurs at their inputs. Thus, the measure is inflated, since the number of <u>useful</u> evaluations may actually be small.

Be suspicious of vendor's performance claims. For example, while simulating identical (and real) circuits, one simulator reported approximately 1,000,000 events per second in its 'accelerated' mode, while SIMIC reported about 50,000. However, both simulators required virtually the *same* wall clock time (on a single user environment) to complete. The only way to evaluate relative simulator performance is to ignore the vendor supplied performance figure, simulate circuits that represent your typical design style, and measure the actual throughput for each circuit.

Finally, be very careful to select identical simulation options and analysis levels for comparison. For example, most of the per-element processing time in SIMIC is associated with hazard analysis; evaluation time is relatively short. You may have to suppress SIMIC combinatorial and functional timing analysis to make it "compatible" with the other simulator. Also note the system resources needed to simulate your testcases. In the testcase described above, the other simulator required almost 4 times the memory (30 MB) to simulate the same circuit as SIMIC (8 MB).

# Section 1 Functional Verification

In this design phase, the simulator is used to verify the design's completeness and correctness. The issues addressed here are fundamental. For example: have the functional specifications been met? is the architecture correct? is the polarity of every signal correct (missing inverters)? etc. Simulation is performed with nominal timing—TYPICAL, MINIMUM, or MAXIMUM. Information about, and propagation of, transient pulses is suppressed.

## Controlling Timing And Hazard Analysis

While the ultimate goal is to obtain working silicon, functional verification is sufficiently complex that it should be performed as a separate task. It is difficult enough to determine that the logical design is correct; if, in the process, the designer must also track down timing problems that cause signal values to become either incorrect or unknown, this task can become exasperating.

Thus, the simulator should allow you to selectively disable timing checks and hazard analysis for functional verification, and enable them for timing verification. For example, if you do not specify analysis options to SIMIC, its default operation is to perform spike propagation and flip-flop timing checks. A single SIMIC run command can disable all spike propagation; another will disable all timing checks.

It may not be desirable to disable all simulation checks for functional verification, since incorrect logical design can sometimes be detected by "pathological" behavior. Two common situations are wire-tie conflicts (possibly caused by incorrectly decoding control signals for tristating gates, or by improper test stimuli) and oscillations (possibly caused by odd inversion polarity or by pulses in feedback loops). Examples of these situations are given in Testcase #5 and Testcase #6 below. By default, SIMIC checks for both conditions. Although each check can be disabled (again, with a single command), it may never be advisable to do so.

## Circuit Response Time

Design specifications usually require that outputs respond to input stimuli within given time limits, or that the circuit state always stabilize within a given time limit in order to meet a target operating frequency. The simulator should facilitate verifying whether specified design limits have been met.

Most simulators have no built-in concept of circuit stability or response time, so verification often requires analysis of voluminous simulation data describing all simulated events. SIMIC supports a unique simulation mode that is oriented exactly to this task. In fact, it supports <u>three</u> modes:

(a) <u>Pattern Stimuli</u> – each new primary input state is applied only when the circuit state stabilizes in response to the previous input state. SIMIC will report the minimum and maximum response times of each primary output, and the circuit stabilization time required for each input state.

(b) Waveform Stimuli – the time of every primary input event is user-specified; this is the stimulus mode found in almost all simulators.

(c) Tester Emulation Mode – primary input events are described as time-sets, and output strobes are define, to debug tester functional tests during simulation.

## Simulation Accuracy

Most modern commercial simulators provide the basis for creating component models, either by supporting a comprehensive set of built-in primitives or by providing methodologies to define new primitives (e.g., by specifying truth tables or boolean equations, by supporting hardware description languages, or by supporting an interface to models written in programming languages such as C). They also allow specification of component propagation delays and support general event scheduling mechanisms. Since a "design" is the implementation of a circuit in a particular technology, and since most simulators provide the capability of creating standard or custom cell models that are as accurate as the library developer can make them, shouldn't these simulators be capable of simulating an entire design accurately?

Unfortunately, the answer is *no*.

First, even if the library developer does his/her job flawlessly, unexpected behavior could result from the simulator's handling of the model, rather than from the model itself; some simulators have interesting algorithmic "quirks" that produce erroneous results under certain conditions. For example, one widely-used simulator supports a component pin-to-pin delay algorithm that produces the correct response when a single input changes state, but sometimes generates very interesting, and very wrong, responses when multiple inputs change within path delay intervals.

Second, while unexpected model behavior can be a problem, the most common source of inaccuracy is the simulator's inability to handle component *interaction* properly, or to supply modeling constructs for resolving component interaction.

### Component Interaction

Algorithmic Limitations

Even if a simulator can model each component precisely, algorithmic limitations within its event processing routines may cause it to mishandle component interaction. Testcase #1 illustrates a simple, but fundamental, circuit that few, if any, simulators other than SIMIC handle correctly. It contains tristating drivers and a latch-type bus-hold circuit. Usually, algorithmic errors cause false transient pulses at wire-ties, but with the bus-hold, the result is a steady-state error.

Delay vs. Loading

Clearly, if the simulation results are to correspond to reality, the modeled propagation delays must be as accurate as possible. Thus, each signal's propagation delay must be based on the signal's loading. Some commercial simulators do not support automatic delay vs. loading computation. For these simulators, you (or an ASIC vendor) are burdened with the responsibility of writing auxiliary programs to compute propagation delays from the netlist and from cell pin capacitances.

In contrast, if the simulator supports delay vs. loading computation, as SIMIC does, not only is the user freed from this task, but simulation accuracy can be increased as well. Two of the testcases that follow illustrate this: SIMIC automatically computes the equivalent delay of paralleled elements (Testcase #2—other simulators require workarounds to handle paralleled elements) and dynamically adjusts driver delays when switches add or remove loading at the driver outputs (Testcase # 3). Incorporating delay vs. loading into the simulator also facilitates backannotation of wiring delays from the layout and experimentation with statistical wiring delay parameters without having to change the network description or recompile it, since the simulator can recompute the new parameters automatically at run time.

Drive Strengths

Some simulators do not support multiple signal strengths. Lacking this model, circuits that require one driver to overpower another (such as the bus-hold circuit in Testcase #1, RAM cells, etc.) would be difficult or impossible to model structurally (a behavioral model would be required as a workaround—this would destroy the integrity of the netlist). Current standard cell libraries may require at least three levels of gate drive strength; SIMIC provides four.

Typical custom designs contain a large range of device sizes. At least four orders of magnitude in resistive strengths may be required to model these circuits properly at the switch-level. Thus, simulators with switch models having only a few levels of strength cannot be used for verifying networks extracted from layouts. This application requires a simulator specifically oriented toward handling extreme device size variations. For example, SIMIC supports 32K resistive gradations for switch-level simulation.

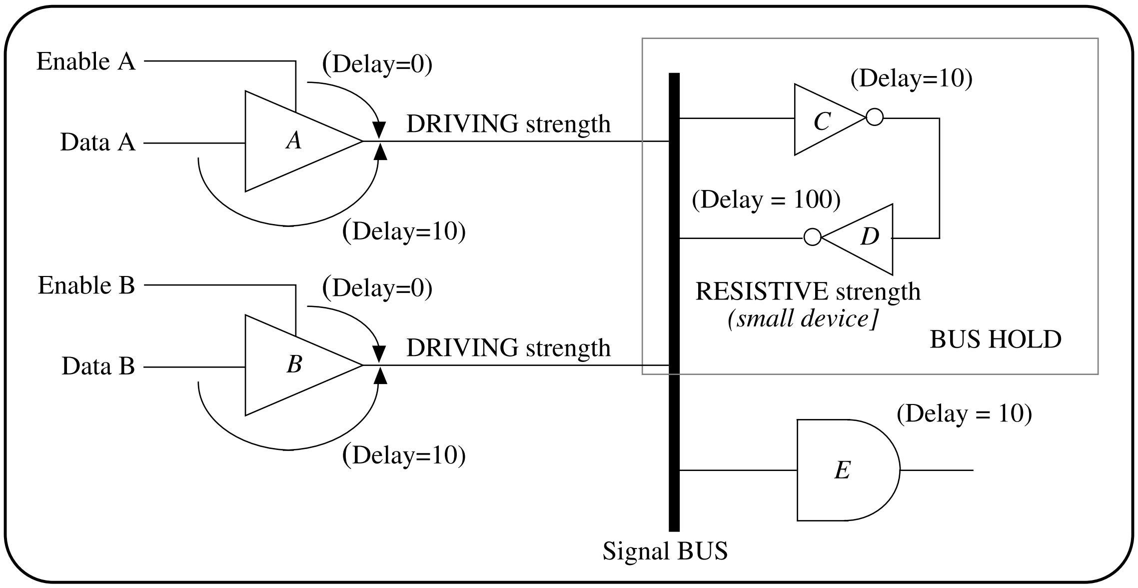
In addition to supporting drive strengths, it is also necessary for the simulator to keep track of *ranges* of drive strength and/or level when unknown values arise. A typical situation is illustrated in Testcase #4. Here, the value of a tristating driver's control input is unknown, but its data input has the same value as the data input of a second driver that is enabled. Thus, the value of the common bus driven by these elements is determinable. In order to handle this situation, the simulator must be capable of representing the output value of the first element as "anything from tristating to a driven logic-0". This range of possible values is called an "interval". Some simulators do not support intervals, and therefore would set the bus of Testcase #4 to the unknown (X) value. In SIMIC, intervals can represent the full range of 32K strength/value pairs (approximately 2.14 billion combinations).

Timing

Perhaps the most discussed and worried-about interaction between components is event timing. Are events at an element's inputs too close for comfort? Has a setup/hold/pulse-width constraint been violated? Is the circuit sensitive to minor variations in delay and/or input arrival times? These issues are the subject of Section 2.

## Testcase # 1 – Bus Hold Circuit

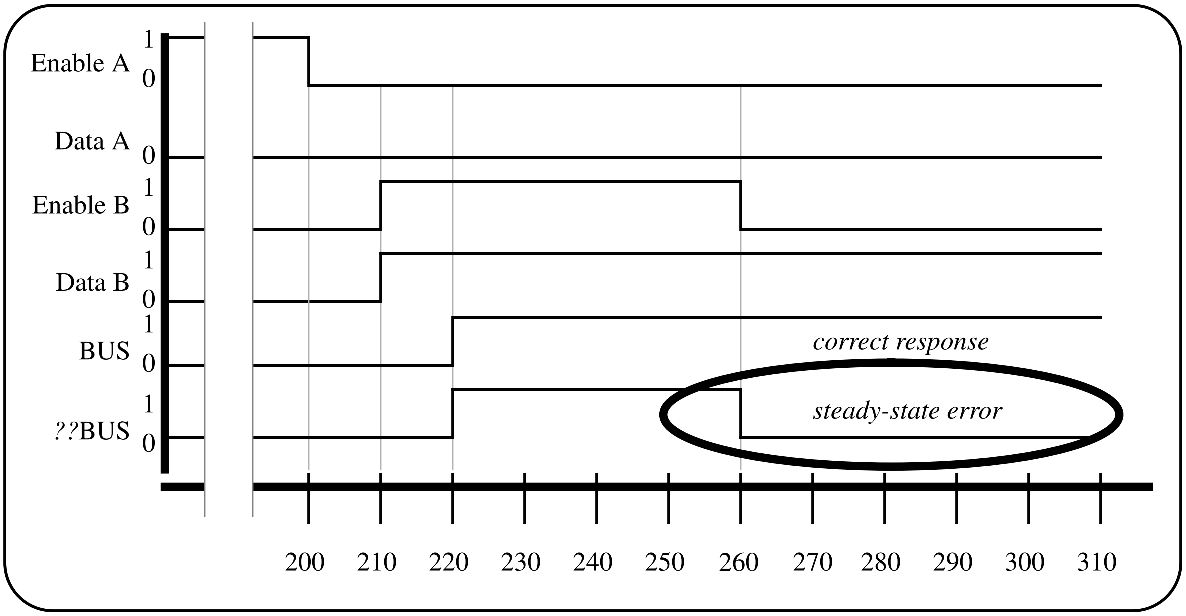
Most modern designs employ busses with multiple drivers that are often dissimilar in drive characteristics and speed. If the simulator cannot account for the combined effects of the drivers, it can predict timing incorrectly, potentially mask transient behavior, or generate erroneous behavior. This example illustrates a bus hold function, common in bus-oriented designs. Since the circuit contains feedback, it demonstrates the possibility of steady-state value discrepancies between the physical design and simulation.




In this example, two tristatable drivers are connected to the common bus, `BUS` (typically, there would be more). Only one receiver, Part `E`, is illustrated, but there are typically more. The bus hold consists of a two-inverter loop, where one inverter, Part `D`, contains small transistors. When any driver is enabled, it overpowers the small devices in Part `D`. When all drivers are disabled, Part `D` provides enough drive to maintain the last driven value. Its large delay arises from its high ON-resistance and the significant capacitance on `BUS`, and is really associated with the time required for Part `D` to charge the bus to the complementary state, assuming all other drivers are disabled. But what if another driver has *already* charged the bus?

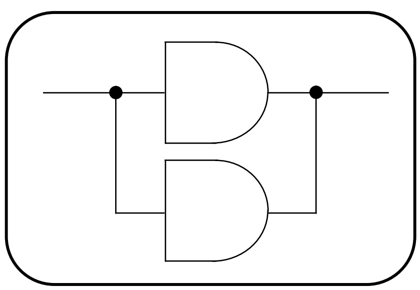
This circuit can be modeled by assigning the tristating drivers a higher drive strength than Part `D` (e.g., DRIVING vs. RESISTIVE strength). The non-inverting drivers are ON when their ENABLE inputs are logic 1, and instantly tristate when their enable inputs are logic 0. The input waveforms shown below initially drive `BUS` to logic 0 via Driver `A`. At time=200, Driver `A` tristates, and `BUS` is maintained at 0 by Part `D`. At time=210, Driver `B` is enabled and Data `B` rises, forcing `BUS` to logic 1 at time=220. Driver `B` is then disabled at time=260, well after all signals should have stabilized.

Does the simulator predict that the bus will lose the last driven logic 1 value? If so, it has introduced a steady-state error! SIMIC properly holds the bus at logic 1.




**Testcase #2 – Paralleled Devices**

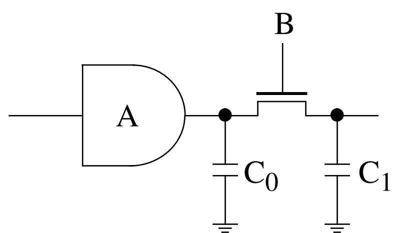
In order to boost drive capabilities beyond that of available cells, a common method is to connect functionally identical cells in parallel. The combination is equivalent to a single gate with the electrical characteristics merged. resulting in a reduction in load-dependent delay.



If the simulator does not merge these properties, then a special model must be manually created to replace the parallel gates with a single device. This is not only prone to error, but it also invalidates the network description if used for other programs, such as place and route. In contrast with other simulators that require the manual workaround to preserve simulation accuracy, SIMIC will group any paralleled combinatorial gates, ideal switches and resistive switches, convert each group to a representative single device, and merge electrical characteristics.

**Testcase #3 Dynamic Loading**

Many simulators provide switch level primitives, either ideal (resistanceless) or resistive. SIMIC has both. In the case of the resistanceless switch, nodes connected by ports of the switch can be dynamically connected (tied) and disconnected during simulation. This causes a change in the loading as seen by the node drivers. The simulator should be able to dynamically adjust the delays of these drivers to reflect the changing load conditions.

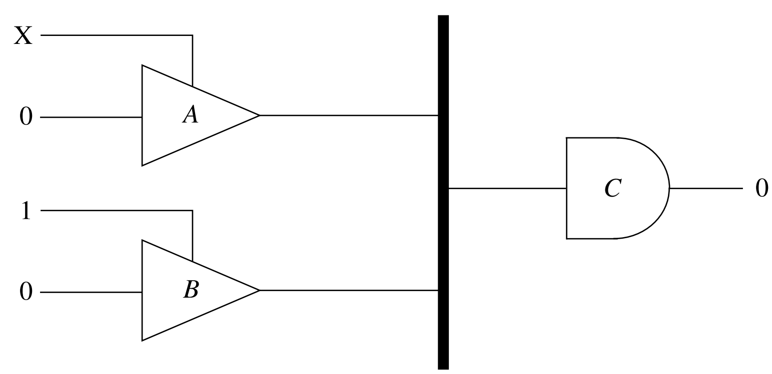



When B=0:
The delay at element A should reflect a loading of $C_0$.

When B=1:
The delay at element A should reflect a loading of $C_0 + C_1$.

**Testcase #4 Resolving Unknowns For Nonconflicting Wire-ties**

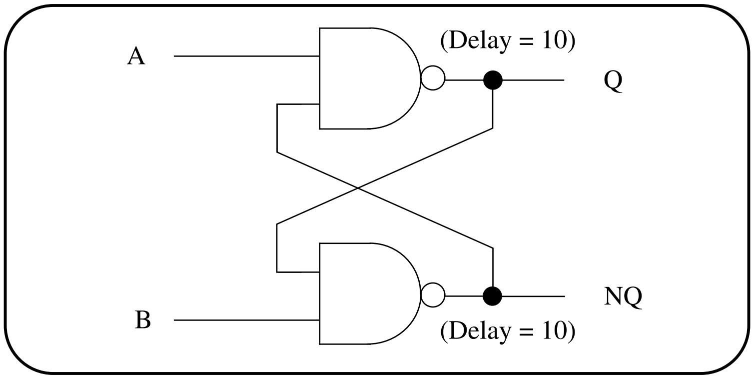
If there is no possibility of conflict at a wire-tie, the simulator should be able to properly determine the value when uncertain strengths and values exist. The following circuit illustrates the problem:




The unknown enable (X) at the Enable input of Part `A` makes the output ambiguous; either it could be tristating (Z) or driving logical-0. Because Part `B` is driving logical-0, there is no possibility of a conflict, and the resultant wire-tied value should be logical-0. SIMIC utilizes an interval representation for ambiguous driven values to properly resolve this type of situation. This representation covers the 2.14 billion possible intervals associated with SIMIC gate and switch primitives.

### Testcase #5 Oscillations

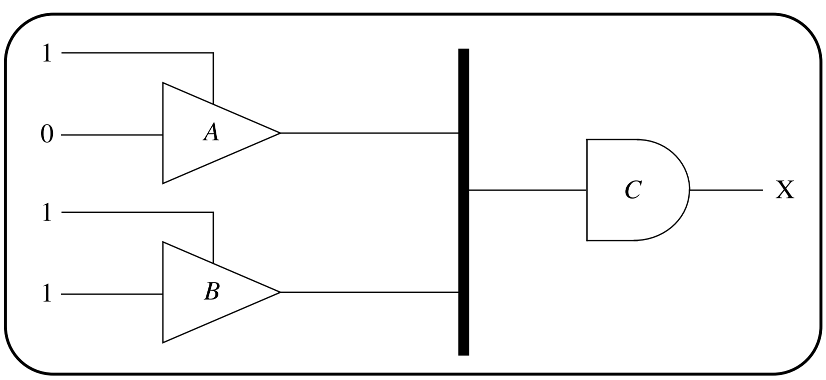
Oscillation can occur if an odd number of inversions are inadvertently placed in a feedback loop, or if transient pulses arise within the loop. Since this situation could cause the simulation to continue indefinitely, or introduce events into the simulation output that might require considerable time to track down, oscillation should be trapped and squelched (force the oscillating signals to an unknown (X) state).




For example, a cross-coupled NAND latch is one of the most common culprits causing inadvertent oscillation. If, in the latch shown above, we let the circuit stabilize with both `A` and `B` logic 0, and then simultaneously change `A` and `B` to logic 1, the circuit becomes unstable and oscillates. If this condition is buried in a large design, can the simulator report the problem rather than ignore it? SIMIC will trap the oscillation, set the oscillating signals to X to quelch activity, and issue a warning message (unless suppressed by the user). In addition, the user may set a breakpoint when oscillation is detected to freeze the circuit state and determine the problem's cause.

### Testcase #6 Wire-tie Conflicts

Wire-ties driven to unknown values because of contending multiple drivers should be trapped and reported. Some simulators provide no warning that a wire-tie conflict has occurred, or will only report conflicts at busses specified by the user. If you miss some wire-tie conflicts during simulation, the implementation in silicon could be power consumptive.




By default, SIMIC reports all wire-tie conflicts. In addition, the user may set a breakpoint when any conflict occurs, in order to investigate and correct the problem.

# Section 2 Timing Verification

After you have verified that the design functions correctly with propagation delays at nominal values, the next step is to check that sufficient tolerances exist between "close events". This is typically done in the post-layout simulation (or additionally, prior to layout using statistical delays), where the effects of wiring delays are included. Stating the requirement for this simulation phase, <u>the simulator should provide reasonable confidence that if the design were committed to silicon, the operation of the physical circuit would remain as predicted</u>.

## Simulation vs. Reality

The fundamental problem associated with using simulation to verify timing tolerances is that the silicon will never really operate <u>exactly</u> as modeled, either in its intended environment or on the tester. That is, even if the cell delay parameters and wiring capacitances are obtained from a large database of production testing and characterization, <u>the simulation results may never be reproduced in the "real-world"</u> because:

(a) Process variations will cause device operation to differ from simulation at the MIN, TYPICAL, and MAX delay points in an unpredictable manner. For example, since p- and n- transistors are deposited at different (therefore independent) steps, some CMOS devices may have fast n-transistors and slow p-transistors, or vice versa, or wiring delays may be larger in some production batches than in others, etc.

(b) Primary input events may not arrive exactly at the times modeled during simulation.

(c) Small propagational delay modeling errors in the cells and/or in the wiring delay estimates can accumulate when events propagate through many levels of logic. Since simulation usually is performed in integral time increments, this effect could be magnified by truncation or rounding of computed delays.

Thus, when performing simulation for timing verification, measures must be incorporated to account for the uncertainties of simulation—the possible differences between computed and actual operation.

## Warning Messages

The simplest method of handling "close events" is to issue warning messages whenever they occur (some simulators don't even provide this basic service). Warning messages are helpful, but they can be overwhelming. <u>Thousands</u> of messages could be generated, of which only a few may be critical. Since manual investigation of each message will be very tedious, large files containing warning messages are generally either be ignored or superficially examined.

The simulator should therefore provide some method of limiting the number of warning messages to a manageable amount. Most don't! In contrast, SIMIC not only supports selection of which signals to monitor, it also allows specification of the maximum number of messages that should be issued for each signal. (If, say, 10 wire-tie conflicts, or 10 close events, are reported at a particular signal, you will suspect that "something is happening" at that part of the circuit; hundreds of messages aren't necessary to attract your attention.)

Another basis for reducing the number of warning message is the observation that some timing problems may be more important than others. For example, timing problems at flip-flops are critical, but transients in combinational logic may not be relevant if this logic does not fan out to sequential elements, or if the transients cannot be clocked into flip-flops. Therefore, SIMIC provides a unique additional option for reducing the number of warning messages—that only problems occurring at flip-flops be reported (either that a timing problem has been detected, or that an unknown value has been clocked into a flip-flop). This option can be *very* effective for quickly isolating and correcting critical timing problems, when used in conjunction with X-pulse propagation (see below); once the sites of steady-state errors have been identified, the simulation can be rerun, this time with warning messages selectively enabled for those sections of the logic that feed the flip-flops in question, to determine where and how the timing problems originated.

## X-Pulse Creation And Propagation

The fundamental problem with relying *solely* on warning messages is that the consequences of "close events" in combinational logic generally cannot be traced, to determine whether their effects are critical, because of inertial filtering.

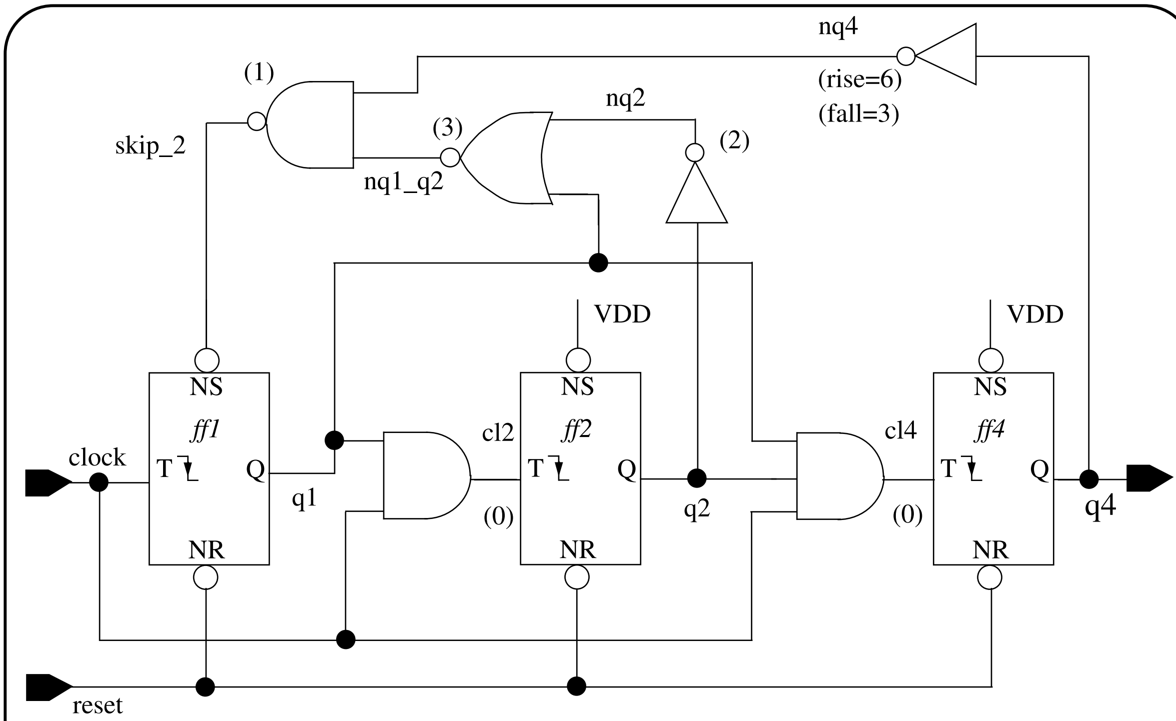
Consider, for example, a spike, which is a transient condition at an element output where, in the course of simulation, (1) an event at the element's inputs causes an event to be scheduled for the output, but then (2) a second input event occurs prior to the scheduled event time that causes the new output value to differ from the scheduled value. (This condition is also called a "glitch"). Since the primary source of delay is capacitive net loading, simulators use an inertial delay model, where transient pulses narrower than an output's response time, i.e., smaller than the time required to charge or discharge the output's net capacitance, are suppressed (filtered out).

Thus, after reporting the occurrence of a spike condition, the simulator filters the resulting output transient, and there is no way to determine which element of the output's fanout cone might have been affected.

A better approach is to introduce an unknown (X) value whenever this situation occurs. If it is blocked from propagation farther downstream in the combinational logic, or propagates to an unloaded element, the "close event" is harmless. Alternatively, if the X value is sensitized to a flip-flop, it would be latched in. This technique, called "X-pulse" propagation (or "spike propagation"), therefore "automatically" distinguishes between irrelevant and relevant close events. Testcase #7 illustrates use of X-pulse propagation to uncover sensitivity to element delays.

Some simulators don't support X-pulse propagation. In those that do, there is no *single* method of implementing it; different simulators incorporate different criteria for determining (a) whether an X-pulse should be generated, and (b) how wide the X-pulse should be. Of all commercial simulators, SIMIC supports the most flexible user-control of X-pulse creation; you can specify these criteria on a per-signal basis.

## Testcase #7 Using X-Pulse Propagation To Uncovering Event Timing Sensitivity




Notes:

(1) Numbers in parentheses are nominal rise and fall delays—if *rise* and *fall* are not specified, then these delays are equal.

(2) Flip-flop *ff1* can respond to a 1 time-unit wide LOW pulse at its NS input.

(3) Flip-flop propagation delay from the fall at T to the Q transition (rise or fall) is any reasonable value, say $\tau$.

<u>Description:</u>

This circuit is a DIVIDE-BY-7 counter containing negative-edge-triggered T flip-flops with active-low asynchronous `SET` and `RESET` inputs. (To exercise this circuit, simply apply a reset pulse and a clock sequence.) Without the feedback logic generating signal `SKIP_2`, the counter would simply divide by 8. However, when the counter state reaches 2 (`Q1`,`Q2`,`Q4`=010), `SKIP_2` goes to logic-0, setting `Q1` to logic-1 and thereby skipping to state 3. The width of the LOW pulse at `SKIP_2` is self-timed, since it disappears as the new state of `Q1` propagates through `NQ1_Q2` back to `SKIP_2`. Assuming that `CLOCK`'s LOW pulse-width is sufficient, this reset pulse should cause no internal flip-flop timing problems (CLOCK's LOW pulse should be at least $\tau$+11 time-units wide, where $\tau$ is the propagation delay from each flip-flop's T input to its Q output).

If you have access to a simulator, you can perform an interesting experiment. First, try simulating this circuit without spike propagation to verify that it indeed divides by 7. Then, enable spike propagation and resimulate. Does the simulator now set the counter state to an unknown value, as SIMIC would? If it doesn't (most simulators will not), you have just missed a real timing problem, since this circuit is very sensitive to timing variations! To see this, simply change the delay of signal `NQ1_Q2` from 3 time-units to 2 and resimulate. The counter will now divide by 6! Testcase #8 illustrates how SIMIC can be used to debug this circuit.

## Combinational Hazards

From the above, you have probably concluded that in order to be capable of verifying an IC design, the simulator must be able to detect potential hazard situations such as spikes, and optionally generate a warning message and/or X-pulse when these situations occur. This is absolutely correct, but there is even more to consider.

SIMIC has the most robust set of combinational timing checks in the industry, including <u>spike</u>, narrow <u>pulse</u>, and <u>near</u> (what if) hazard checks. To illustrate these cases, consider a two-input AND gate with the first input rising and the second falling, as illustrated on Page 15.

(a) If the first input changes much earlier than the second, a large output pulse is created. But if the two input transitions are fairly close, such that the output pulse width, W, is comparable to the signal's average propagation delay, it becomes questionable whether the pulse will actually occur in the real circuit, where input arrival times may differ from simulated event times. In SIMIC, this short pulse is a <u>pulse hazard</u>.

(b) If the input transitions were even closer, the point will be reached where the pulse no longer occurs in simulation due to the inertial delay of the gate (D < output delay). This situation is the classic <u>spike hazard</u> described above.

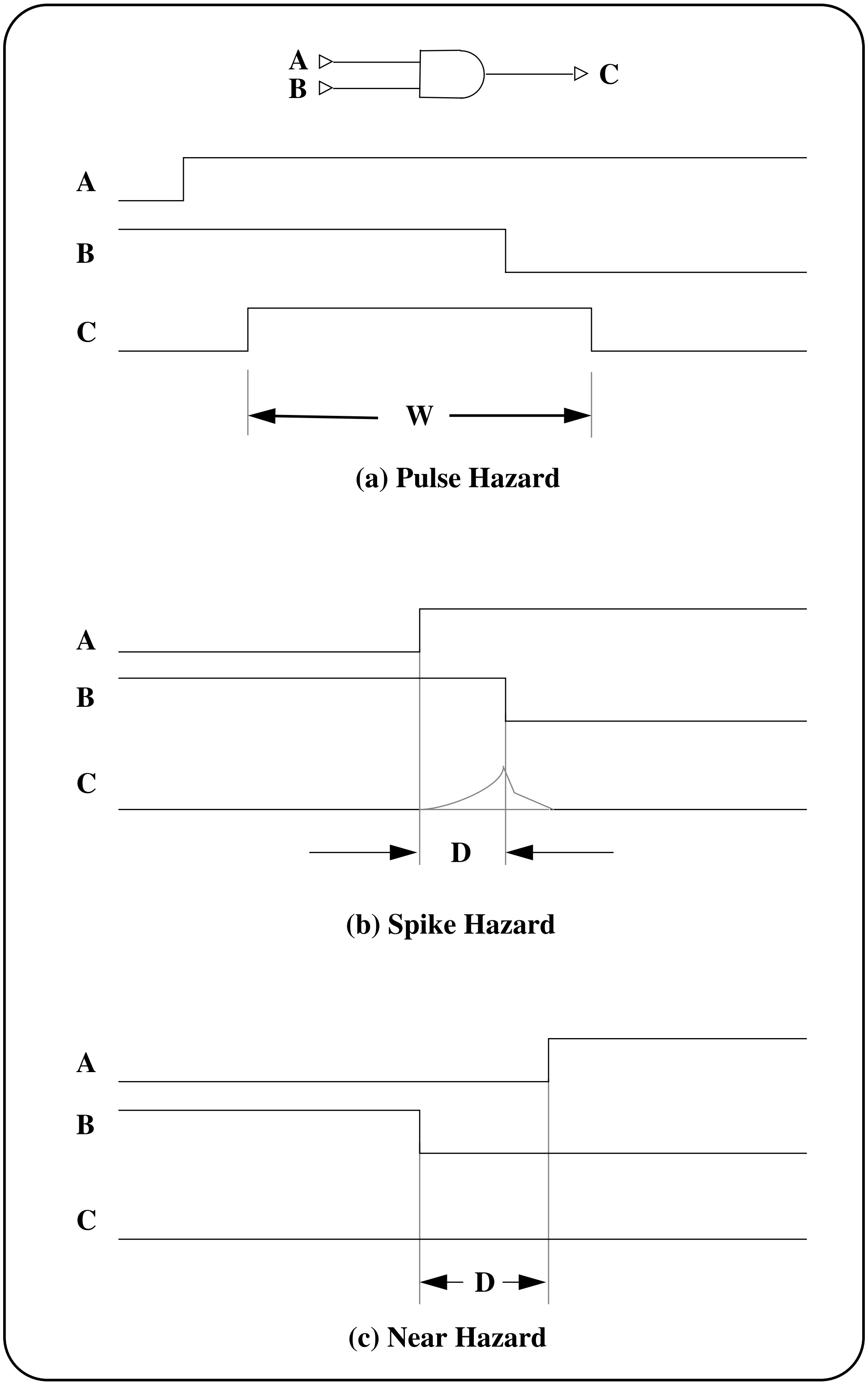
(c) If we continue to move the edges closer, they will eventually occur at the same time and then, the second input transition will occur before the first. Even when simulation produces a "clean" output, SIMIC supports a time analysis window of close input events (D comparable to the output's average propagation delay). If the output's response is sensitive to the order of the input events, SIMIC flags the condition as a <u>near hazard</u>; in the real circuit, the first input transition may actually arrive before the second, causing a pulse that was never predicted during simulation. This is one example of "what-if" simulation. Near hazards can also be propagated as a X-pulses.

Since pulse and near hazards are also situations where differences in timing and input arrival times can cause discrepancies between simulation and reality, the simulator's ability to find timing problems is incomplete without these checks.

## Functional Timing Checks

Combinational hazards are not the only source of timing problems. For example, events can propagate through logic very cleanly, but arrive at flip-flops too close to active clock edges. Thus, the simulator must also be capable of performing timing violations checks for sequential elements. These include setup time, hold time, and pulse-width checks for each applicable element. In addition to issuing warning messages, the simulator should help the designer locate the problem by optionally setting the element output (and possibly the internal state) to unknown (X) when a true hazard occurs.

In order to eliminate improper warnings, or false alarms, these checks should be based on the element's functionality, that is, should verify that the input event in question actually has an effect on the element's state.




**SIMIC Combinational Hazard Checks**

As an example, consider the functional timing checks incorporated into SIMIC. In contrast with other simulators that perform “blind” or unconditional checks based strictly on event times at flip-flop inputs, SIMIC handles timing violations in a functionally consistent manner – warning messages are issued, and/or flip-flops are set to the unknown state, only when the flip-flop’s state is *uncertain*.

SIMIC allows the user or library developer to completely and independently control both warning messages and propagation of unknowns (X) due to violations of the following flip-flop timing checks, which are specifiable as functions of flip-flop loading:

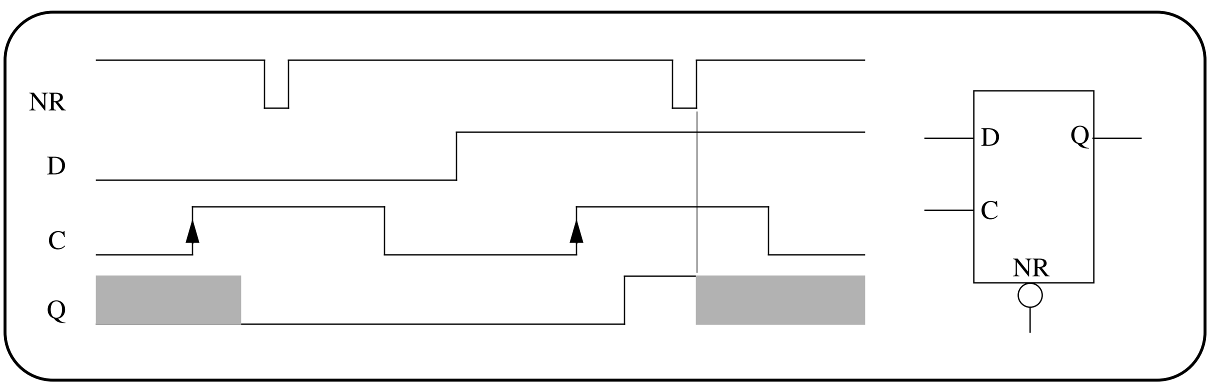
(a) Pulse-width checks – SET, RESET, CLOCK (0 and 1 pulse-widths of clock)

(b) Setup-time checks – minimum time an input must be stable *prior* to the active clock edge:
DATA D- input of D flip-flop or latch, J or K inputs of JK flip-flop
SET When the value of DATA is 0
RESET When the value of DATA is 1.

(c) Hold-time checks – minimum time an input must remain stable *after* the active clock edge: DATA, SET, RESET.

The sequence of inputs shown below illustrate SIMIC’s functionally consistent timing checks to eliminate false alarms. The first active (rising) clock edge reliably clocks in a 0 from the D input. Thus, even though a narrow RESET pulse occurs, the flip-flop’s state is never uncertain, so SIMIC will not issue a warning message nor set its state to X. (However, it can issue an informatory message, since the user may want to know that a violation occurred.)

When the second narrow reset pulse occurs, the final state of the flip-flop *is* uncertain, and SIMIC will issue a warning message, and/or set the flip-flop state unknown, according to the user-specified options.




## What-if Simulation

In order to test tolerances within the design, the simulator should facilitate experimentation. For example, the question “what if setup times were 10% larger?” is easily answered in SIMIC, since a single command can change all setup time limits (or hold time limits, or pulse-width limits, or rise times, etc.) by any given amount, without having to recompile the circuit.

## Debugging Functionality and Timing

When a problem has been discovered, the simulator should provide capabilities to facilitate tracing the problem to its source. In the process, it should allow simulation parameters to be modified so the user can test hypotheses and implement possible corrections. To maximize debugging productivity, the simulator should allow the user to work interactively, starting from the point in the simulation where the problem occurs. Important debugging operations are:

(a) Set Breakpoints. The simulator should provide a robust way of specifying the condition to stop the simulation, so that interactive mode can be entered. For instance, “when signal `XYZ` enters an X state after simulation time 10000”, or “whenever a setup time violation occurs”, etc.

(b) Save and Restore State. The simulator should provide the capabilities to save the state of the network, and then restore it. This operation should be capable of performing a simulation checkpoint (where the same sequence of patterns will be repeated after the restored state), or network initialization function (where new sequences of patterns can be applied to the restored state)

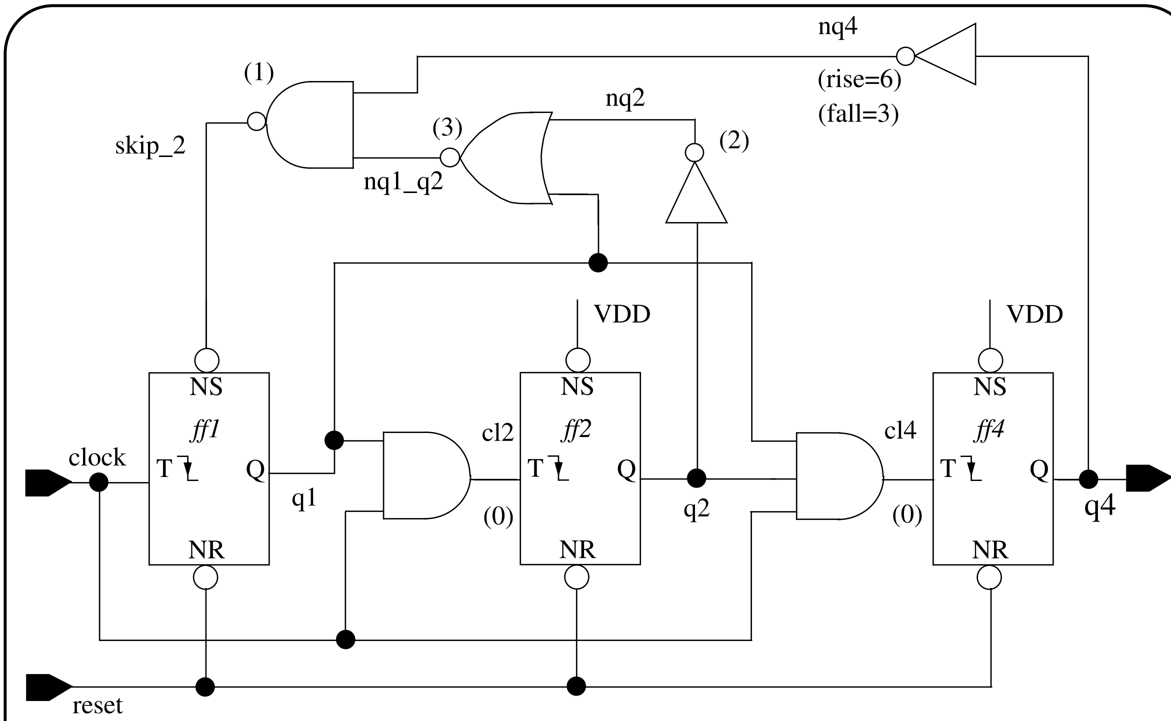
(c) Modify network state. Signal values, plus values internal to memory elements (flip-flops, RAMS, ROMS) should be interactively modifiable.

(d) Modify simulation parameters. Delays, decays, etc. should be interactively modifiable.

(e) Interrogate circuit state and trace circuit activity. The simulator should allow individual signals to be interactively probed, memory locations to be interactively examined, and activity to be traced during simulation for selected sections of the circuit, or for the entire circuit.

Testcase #8 demonstrates a small portion of SIMIC’s debugging capabilities; it illustrates a debugging session for the divide-by-7 circuit of Testcase #7. It is interesting to note that most other simulators will not detect the timing problem inherent in this design.

**Testcase #8 Interactive Debugging**




Notes:

(1) Numbers in parentheses are nominal rise and fall delays—if *rise* and *fall* are not specified, then these delays are equal.

(2) Flip-flop *ff1* can respond to a 1 time-unit wide LOW pulse at its NS input.

(3) Flip-flop propagation delay from the fall at T to the Q transition (rise or fall) is any reasonable value, say $\tau$.

This testcase illustrates a SIMIC debugging session for the DIVIDE-BY-7 circuit (See Testcase #7 for a description of its operation.)

(a) Simulate the circuit with spike propagation disabled (functional verification). The simulation produces the intended divide-by-7 function, but SIMIC issues an informatory message that a spike has occurred (if you do not request spike warning messages, SIMIC issues a summary message after simulation indicating the number of spikes that were detected, but not reported).

(b) Simulate again with spike propagation enabled (timing verification). This time, simulation reveals a problem when the circuit goes from count 3 to count 4; the state of the counter becomes unknown!

(c) Resimulate and generate trace (including causality information) around the time of the problem. The trace information obtained from SIMIC is shown on page 19. To understand this output, consider the lines:

```
2 E     8> TRACE(0->X) NQ1_Q2
  C      >      (0->1) NQ2
  C      >      (1->0) Q1
```

Lines that contain the word "`TRACE`" report a traced event; these lines are followed by one or more lines that start with "`C`", indicating possible causes of this event. The pre-

```
0 E     8> TRACE(1->0) CLOCK
0 E     8> TRACE(1->0) CL4
  C      >      (1->0) CLOCK
0 E     8> TRACE(1->0) CL2
  C      >      (1->0) CLOCK
0 E     8> TRACE(1->0) Q1
  C      >      (1->0) CLOCK
0 E     8> TRACE(1->0) Q2
  C      >      (1->0) CL2
0 E     8> TRACE(0->1) Q4
  C      >      (1->0) CL4
2 E     8> TRACE(0->1) NQ2
  C      >      (1->0) Q2
2 E     8> TRACE(0->X) NQ1_Q2
  C      >      (0->1) NQ2
  C      >      (1->0) Q1
3 E     8> TRACE(1->X) SKIP_2
  C      >      (0->X) NQ1_Q2
3 E     8> TRACE(0->X) Q1
  C      >      (1->X) SKIP_2
5 E     8> TRACE(X->0) NQ1_Q2
  C      >      (0->1) NQ2
  C      >      (0->X) Q1
6 E     8> TRACE(1->0) NQ4
  C      >      (0->1) Q4
6 E     8> TRACE(X->1) SKIP_2
  C      >      (1->0) NQ4
  C      >      (X->0) NQ1_Q2
```

fix "`2 E 8`" indicates that the event, `NQ1_Q2` changing from logic 0 to unknown (X), occurred 2 time-units into the 8-th input state (4-th active clock edge). Two events caused this transition: `NQ2`'s 0→1 transition (also at time `2 E 8`), and `Q1`'s 1→0 transition (two time-units earlier, at `0 E 8`).

We can see that `Q1`, which is the output of *ff1*, went unknown at time 3. The causality information indicates that Q1 went to unknown because `SKIP_2` went unknown. `SKIP_2` went unknown at time 3 because `NQ1_Q2` went unknown. As described above, `NQ1_Q2` went unknown at time 2 because of the fall at `Q1` and the rise at `NQ2`. This clearly indicates that a race condition between these two signals caused a spike to be created at `NQ1_Q2`, which propagated to `SKIP_2` and then to `Q1`.

(d) <u>Determine corrective action</u>. If this action requires only a modification in delay parameters or circuit state to test feasibility, SIMIC allows these changes to be made interactively (here, a delay greater than 2 time-units must be inserted between `Q1` and `NQ1_Q2`). The circuit should then re-simulated with these changes, for verification.

# Section 3 Tester Program Generation

Debugging a test program on a production tester is expensive; it ties up the tester and the test engineer's time, and reduces the amount of time the tester is available to test product. With decreasing prices of computers and workstations, there is strong pressure to use the simulator for test program debugging—at the minimum, to determine timing tolerances for the device on the tester, and at the maximum, to actually simulate the device operating in the tester environment. In all approaches, the stimuli applied during simulation are retimed to emulate tester timing generators, and the device outputs are strobed when the tester would strobe them. Bidirectional busses require special handling; they are sometimes driven and sometimes strobed, depending on who's driving the bus.

After the design has passed functional verification and timing analysis, a set of test vectors must be generated that describe the input stimuli as timing generators and associated patterns applicable to the tester. They may included a slow-speed functional section and a high-speed functional section ("at-speed" tests). Regardless of the simulator's stimulus description language, there should be a compatible mapping to the tester's driver control statements, so that whatever will be applied at the tester drivers will also be applied as stimuli during simulation. The simulator should support a strobing mechanism that accurately emulates strobing on the tester and warns of changing signal values within active strobe windows.

The simulation should provide adequate information to correctly determine the time that the bidirectional pads change from being driven internally to requiring external drive, or vice versa, since the tester must then switch from sensing the pad to driving it, or vice versa. This information is necessary to assure non-destructive testing—preventing the tester from driving conflicting values when the chip drives the pad, and ensuring that these pads are always driven (to avoid damaging chip input buffers because of floating inputs).

Finally, if a problem is found on the tester, there should be a direct method for mapping the test number to simulation time. Changes made to the test program should be easy to backannotate to the stimulus/strobe descriptions for verification.

Can current commercial simulators accept stimulus descriptions corresponding to tester drive specifications and determine whether outputs change within strobe windows? Can information on bidirectional bus states be readily obtained from them? Except for SIMIC, external programs are required to format simulator stimulus descriptions from timing generator constructs and analyze voluminous simulation output data to resolve output strobing and bidirectional pad issues.

In contrast, SIMIC supports a tester emulation mode, where stimuli can be described in terms of timing generators (non-return-to-zero, return-to-zero, return-to-one, and return-to-complement) that allow flexible drive/float control (no-envelop, return-to-drive, and return-to-float). Full time-set switching is supported. Output strobes can be placed exactly where they will occur on the tester, and SIMIC automatically issues warning messages when an output changes within a strobe window. SIMIC generates a tester interface file containing stimulus/response test vectors for each tester period In this file, bidirectional pads properly contain the correct driving or driven values.

# Conclusion

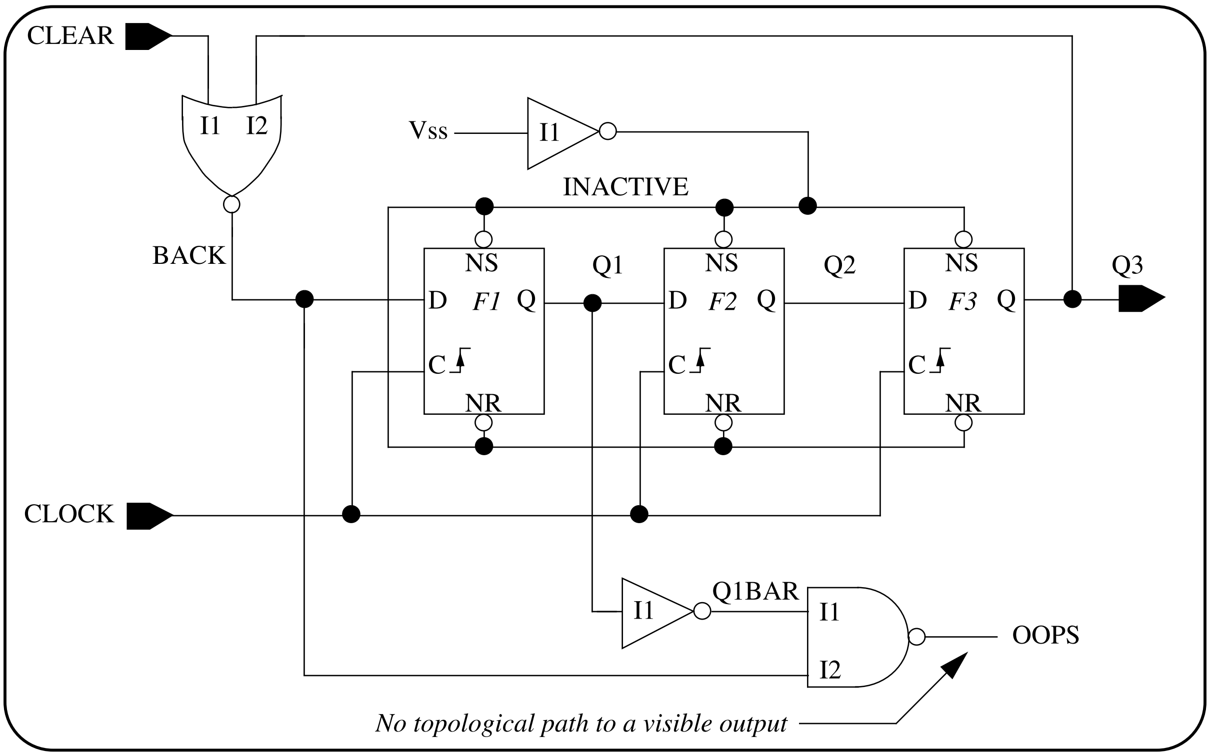
This Guide has described simulation capabilities that we believe are essential for IC design verification. They are not the result of an intellectual exercise, but rather were obtained from years of experience developing and supporting simulators. One such simulator was used by an ASIC design group that turned out an average of one design per day with a 99% track record of first-time working silicon. While supporting this group, we quickly learned that a design verification tool had to be robust, accurate, easy to use, and had to contain an extremely comprehensive suite of functional and timing checks.

We hope this material has been informative and helpful. Many simulation issues were covered; circuit response time, oscillation and wire-tie conflicts, simulation accuracy and component interaction, combinational hazards, X-pulse propagation, functional timing checks, interactive debugging, and tester program debugging. We hope it has given you a better perspective on these issues, on what to look for when evaluating a simulator's capabilities, and on the capabilities you want at your disposal when you debug your next design.

And, with this gained perspective, we hope that you will want SIMIC in your tool suite for that extra insurance against committing designs to silicon that might not be fully scrutinized. After reading this Guide, you now know that SIMIC supports every feature described (and, in fact, a large number of features that were not described), while other commercial simulators lack many of them. After reading this Guide, we believe that you will arrive at the one conclusion that we know to be a fact; that SIMIC is, by far, the best tool available for finding and fixing problems in IC designs.

## Genashor Product Preview—FAULT SIMULATION

Ever have to meet a high fault coverage specification, and wind up spending most of the time weeding out undetectable faults and explaining why faults that were only potential detects should really have been hard detects? Try this circuit, the supplied stimuli should detect 100% of the detectable faults.




<u>Description:</u>

This circuit is a Johnson counter (shift register with complementary feedback) containing positive-edge-triggered D flip-flops with active-low asynchronous `SET` and `RESET` inputs. (If other types of flip-flops are used, the supplied stimuli should be appropriately modified.) The latter inputs, tied to the normally-high signal, `INACTIVE`, are disabled, and topologically invisible elements are introduced, to illustrate two unique features of SIMIC fault simulation:

<u>(1) Resolution of POTENTIALLY-DETECTED faults</u>

Faults on clock and reset lines typically cause the circuit to be uninitializable. Thus, at best, most fault simulators flag these faults as POTENTIAL DETECTS and discard them. SIMIC, however, continues to simulate them and, with its unique algorithm, can resolve those faults that become HARD DETECTS later on.

<u>(2) Removal of undetectable faults</u>

Certain faults are inherently undetectable. This category includes faults that have no topological path to visible outputs (e.g., `OOPS` and `Q1BAR`) as well as certain faults associated with power-rail connections (e.g., all stuck-at-1 faults at part pins connected to `INACTIVE`, the `INACTIVE` signal stuck-at-1, and the stuck-at-0 fault at the input of the inverter generating `INACTIVE`). SIMIC automatically filters out these faults to generate more realistic fault grading. Of course, it will also generate a listing of all filtered undetectable faults.

## FAULT SIMULATION (continued)

```
          CC QQQ
          LL 123
          EO
          AC
          RK

TEST  1: 10 XXX
TEST  2: 11 0XX
TEST  3: 10 0XX
TEST  4: 11 00X
TEST  5: 10 00X
TEST  6: 11 000
TEST  7: 00 000
TEST  8: 01 100
TEST  9: 00 100
TEST 10: 01 110
TEST 11: 00 110
TEST 12: 01 111  ⇐ Clock-related faults and CLEAR-@-1 properly detected when output
TEST 13: 00 111     Q3 executes a 0 -> 1 transition.
TEST 14: 01 011
TEST 15: 00 011
TEST 16: 01 001
TEST 17: 00 001
TEST 18: 01 000
TEST 19: 10 000  ⇐ Begin test for CLEAR-@-0. This fault was not positively detected
TEST 20: 11 000     above since it is possible that the circuit powered up in the '111' state.
TEST 21: 10 000
TEST 22: 11 000
TEST 23: 10 000
TEST 24: 11 000  ⇐ CLEAR-@-0 properly detected after 3 shift cycles transfer CLEAR
                    to output Q3.
```

**Johnson Counter Stimulus/Response Tests For 100% Fault Detection**

The test vectors shown above detects every detectable fault in the Johnson counter. For example, consider the fault `CLEAR`-stuck-at-0; this fault makes the circuit uninitializable. If the counter powers-up in state 111, output `Q3` would be correct through Test 12; however, `Q3` would incorrectly go to 1 at Test 24 if this fault were present. If the counter powers up in any other state, `Q3` would go to 1 earlier than Test 12, and the fault would be detected then. Thus, this fault is detected no later than Test 24, exactly the test that SIMIC flags this fault as having been detected.

The next page illustrates SIMIC fault simulation reports for the Johnson counter, when the above test vectors are used. Here, net faults are designated as *signal_name-@-value* (e.g., `Q3-@-1` means "net `Q3` stuck-at-1"), and pin faults are designated as *part_name;pin-@-value* (e.g., `F1;D-@-0` means "pin `D` of flip-flop `F1` stuck-at-0"). The first report lists all faults removed from consideration because they are inherently undetectable. The second report alphabetically lists all faults simulated and reports the first test that each fault was detected, while the third report provides the same information organized by test number. The 36 faults listed in the latter two reports constitute all detectable faults in the Johnson counter.

# FAULT SIMULATION (continued)

```
TOPOLOGICALLY INVISIBLE FAULTS
OOPS-@-0        OOPS-@-1        OOPS;I1-@-0     OOPS;I1-@-1     OOPS;I2-@-0
OOPS;I2-@-1     Q1BAR-@-0       Q1BAR-@-1       Q1BAR;I1-@-0    Q1BAR;I1-@-1


UNDETECTABLE FAULTS DUE TO POWER RAILS
F1;NR-@-1       F1;NS-@-1       F2;NR-@-1       F2;NS-@-1       F3;NR-@-1
F3;NS-@-1       INACTIVE-@-1    INACTIVE;I1-@-0
```

**SIMIC Report of Undetectable Faults**

```
BACK-@-0 (12)       BACK-@-1 (6)         BACK;I1-@-0 (24)    BACK;I1-@-1 (12)
BACK;I2-@-0 (18)    BACK;I2-@-1 (12)     CLEAR-@-0 (24)      CLEAR-@-1 (12)
CLOCK-@-0 (12)      CLOCK-@-1 (12)       F1;C-@-0 (12)       F1;C-@-1 (12)
F1;D-@-0 (12)       F1;D-@-1 (6)         F1;NR-@-0 (12)      F1;NS-@-0 (6)
F2;C-@-0 (12)       F2;C-@-1 (12)        F2;D-@-0 (12)       F2;D-@-1 (6)
F2;NR-@-0 (12)      F2;NS-@-0 (6)        F3;C-@-0 (12)       F3;C-@-1 (12)
F3;D-@-0 (12)       F3;D-@-1 (6)         F3;NR-@-0 (12)      F3;NS-@-0 (6)
INACTIVE-@-0 (6)    INACTIVE;I1-@-1 (6)  Q1-@-0(12)          Q1-@-1 (6)
Q2-@-0 (12)         Q2-@-1 (6)           Q3-@-0 (12)         Q3-@-1 (6)
```

**SIMIC Report of Detected Faults Sorted By Name**

```
FAULTS DETECTED BY TEST 6
BACK-@-1        F1;D-@-1        F1;NS-@-0       F2;D-@-1         F2;NS-@-0
F3;D-@-1        F3;NS-@-0       INACTIVE-@-0    INACTIVE;I1-@-1  Q1-@-1
Q2-@-1          Q3-@-1

FAULTS DETECTED BY TEST 12
BACK-@-0        BACK;I1-@-1     BACK;I2-@-1     CLEAR-@-1       CLOCK-@-0
CLOCK-@-1       F1;C-@-0        F1;C-@-1        F1;D-@-0        F1;NR-@-0
F2;C-@-0        F2;C-@-1        F2;D-@-0        F2;NR-@-0       F3;C-@-0
F3;C-@-1        F3;D-@-0        F3;NR-@-0       Q1-@-0          Q2-@-0
Q3-@-0

FAULTS DETECTED BY TEST 18
BACK;I2-@-0

FAULTS DETECTED BY TEST 24
BACK;I1-@-0       CLEAR-@-0
```

**SIMIC Report of Detected Faults Organized By Test**