Sliding Window Logic Simulation

Sarah Thompson and Alan Mycroft

Computer Laboratory, University of Cambridge, J J Thomson Avenue, Cambridge CB3 0FD

Abstract

Existing digital logic simulators typically depend on a discrete-time model of circuit behaviour. Whilst this approach is sufficient in many cases for the validation of the behaviour of synchronous circuits, it is not good at identifying glitches that are narrower than the available time resolution. Moreover, it is not generally feasible to accommodate uncertainty in delay time in such a way as to detect at an early design stage glitches that only occur under worst-case layout-specific or environment dependent timing conditions.

This paper presents a technique, based upon abstract interpretation, that, given any synchronous or asynchronous circuit, allows *possible* glitches to be detected within a particular time window given known starting conditions. Since the underlying model is based upon dense (continuous) time, all possible glitches are detected regardless of how narrow they may be. Adopting a window length equivalent to the worst-case uncertainty in delay, then 'sliding' the window in time such that each successive window overlaps the previous window allows all possible glitches to be identified, without a need for exact timing information.

Using this algorithm, it is possible to construct a logic simulator that is capable of automatically detecting possible glitches early in the design life cycle, before layout-specific timing parameters can be determined.

1 Introduction

Abstract interpretation [1, 2] is a long-established technique, most commonly applied to software, that allows abstract properties of systems to be determined.

As a simple example, consider the 'law of signs' in integer arithmetic. It is possible, knowing only the signs of a and b to know with certainty the sign of the result of the integer expression $a \times b$. The sign of the result of the addition $a + b$ may be determined in some cases, but not all. This can be thought of as a very simple kind of abstract interpretation. We might define an abstract multiplication operator $\otimes$, and an abstract addition operator $\oplus$ as follows:

$\otimes$	$-$	0	$+$	$?$
$-$	$+$	0	$-$	$?$
0	0	0	0	0
$+$	$-$	0	$+$	$?$
$?$	$?$	0	$?$	$?$

$\oplus$	$-$	0	$+$	$?$
$-$	$-$	$-$	$?$	$?$
0	$-$	0	$+$	$?$
$+$	$?$	$+$	$+$	$?$
$?$	$?$	$?$	$?$	$?$

where $-$ represents any negative integer, 0 represents zero, $+$ represents any positive integer and $?$ represents any integer whatsoever.

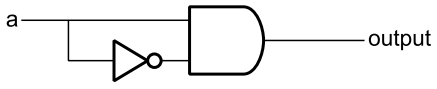
The multiplication $1543 \times -783 = -1208619$ in the concrete world maps to the abstract multiplication $+ \otimes - = -$. Using this technique, it is possible to determine with certainty the sign of the result of a multiplication without actually needing to carry out the multiplication itself. However, the addition $-344 + 762 = 418$ maps to the abstract addition $- \oplus + = ?$, since the abstract values $-$, 0, $+$ and $?$ do not carry enough information for a more accurate result to be determined. Nevertheless, in many cases this approach is still sufficient to fully predict the sign of an integer expression involving addition and multiplication, without any requirement to perform the actual arithmetic.

This paper presents a similar technique that allows the possibility of glitches in digital logic circuits with uncertain delays to be identified, without requiring all possible combinations of delays to be laboriously enumerated.

This work is presented at a relatively early stage – at the time of writing, a simulator based on this approach has not yet been implemented.

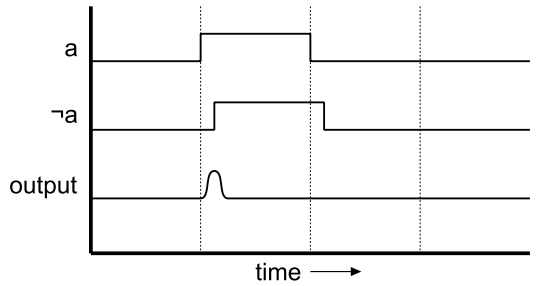
1.1 Motivating Example

The following circuit is probably the simplest possible that exhibits the kind of behaviour that we wish to detect:

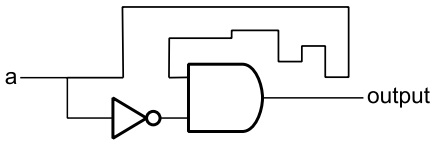


The Boolean equivalence $a \wedge \neg a = 0$ is misleading in this case, since the delays inherent in the inverter and the wires will in many cases cause a glitch to be triggered either by the leading or trailing edge of any pulses applied to the input. If we assume that wire delays are negligible compared with gate delays, it is tempting

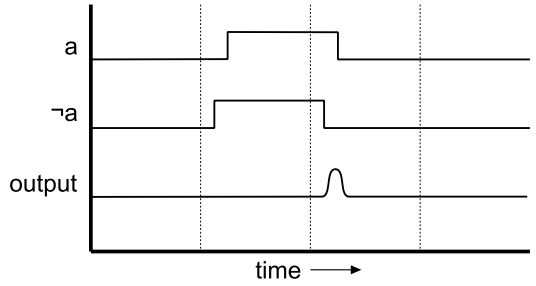
to assume that glitches will appear only on the leading edge:



However, in contemporary full-custom VLSI, it is often the case that wire delays are more significant than gate delays [5]. Due to routing limitations, it could be that the 'direct' path from the input to the and gate might actually take longer to arrive than the path via the inverter:



in which case, glitches will appear on the trailing edge:



When simulating such a circuit, it is typically necessary to make some kind of assumptions about timing. However, since simulation is generally used most heavily in the early stages of the design process, detailed timing information is not yet available. It would be desirable, in such cases, for *possible* glitches to be highlighted, thereby allowing problems to be caught early rather than post-layout (or worse, post-manufacture).

1.2 Identifying Glitches by Abstract Interpretation

Various abstract interpretation based approaches to the detection of glitches, varying in sophistication, are possible [7]. In this paper, we will concentrate on a relatively straightforward approach that can nevertheless still be used to detect possible glitches.

[1] i.e. a signal that is nominally F, that *may* glitch or may not

[2] $a \vee b = \neg(\neg a \wedge \neg b)$

Our technique resembles a Boolean 'logic' to which extra values have been added to encompass transitions as well as steady-state values. A 'time window' during which a transition will occur is assumed in all cases, although the exact time at which the transition will occur is deliberately left unspecified. The values are defined as follows:

F_0 The signal that is 0 for the duration of the window.

$F_?$ A signal that is 0 at the beginning and end of the window, that may contain zero or more glitches[1].

T_0 The signal that is 1 for the duration of the window.

$T_?$ A signal that is 1 at the beginning and end of the window, that may contain zero or more glitches.

$\uparrow_0$ A signal that transitions cleanly from 0 to 1.

$\uparrow_?$ A signal that transitions from 0 to 1 with zero or more intervening glitches.

$\downarrow_0$ A signal that transitions cleanly from 1 to 0.

$\downarrow_?$ A signal that transitions from 1 to 0 with zero or more intervening glitches.

$\perp$ An unknown signal that can not be characterised.

The Boolean operators $\wedge$ and $\neg$ are defined as follows:

$\neg$	
F_0	T_0
$F_?$	$T_?$
T_0	F_0
$T_?$	$F_?$
$\uparrow_0$	$\downarrow_0$
$\uparrow_?$	$\downarrow_?$
$\downarrow_0$	$\uparrow_0$
$\downarrow_?$	$\uparrow_?$
$\perp$	$\perp$

$\wedge$	F_0	$F_?$	T_0	$T_?$	$\uparrow_0$	$\uparrow_?$	$\downarrow_0$	$\downarrow_?$	$\perp$
F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0
$F_?$	F_0	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$\perp$
T_0	F_0	$F_?$	T_0	$T_?$	$\uparrow_0$	$\uparrow_?$	$\downarrow_0$	$\downarrow_?$	$\perp$
$T_?$	F_0	$F_?$	$T_?$	$T_?$	$\uparrow_?$	$\uparrow_?$	$\downarrow_?$	$\downarrow_?$	$\perp$
$\uparrow_0$	F_0	$F_?$	$\uparrow_0$	$\uparrow_?$	$\uparrow_0$	$\uparrow_?$	$F_?$	$F_?$	$\perp$
$\uparrow_?$	F_0	$F_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$F_?$	$F_?$	$\perp$
$\downarrow_0$	F_0	$F_?$	$\downarrow_0$	$\downarrow_?$	$F_?$	$F_?$	$\downarrow_0$	$\downarrow_?$	$\perp$
$\downarrow_?$	F_0	$F_?$	$\downarrow_?$	$\downarrow_?$	$F_?$	$F_?$	$\downarrow_?$	$\downarrow_?$	$\perp$
$\perp$	F_0	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

DeMorgan's law[2] turns out to continue to hold in this extended logic, so the truth table for the $\vee$ operator may be omitted.

1.2.1 Worked Examples

Returning to the example circuit of section 1.1, we will use this technique to determine the behaviour of the circuit. In particular, we will identify its behaviour with respect to steady states (F_0 and T_0) and clean transitions ($\uparrow_0$ and $\downarrow_0$):

a	$\neg a$	$a \wedge \neg a$
F_0	T_0	F_0
T_0	F_0	F_0
$\uparrow_0$	$\downarrow_0$	$F_?$
$\downarrow_0$	$\uparrow_0$	$F_?$

Clearly, the circuit is well-behaved if its inputs are steady; however, either a positive or negative going transition ($\uparrow_0$ or $\downarrow_0$) results in $F_?$, which represents a signal that is 'false', but that *may* glitch. Although timing information is completely omitted in this domain, the abstract interpretation framework guarantees that all possible timing relationships are covered – results obtained in this way are therefore 'failsafe' in that possible problems are always detected.

As a more complex example, the circuits represented by the expressions

$$(a \wedge c) \vee (\neg a \wedge b) \vee (b \wedge c) \tag{1}$$

and

$$(a \wedge c) \vee (\neg a \wedge b) \tag{2}$$

will be compared. With respect to steady-state values for a, b and c, both circuits would appear to be identical, with (2) representing a circuit that may result from naïve optimisation of (1). Our technique can straightforwardly illustrate differences in their dynamic behaviour, however. Consider the critical case $a = \uparrow_0$ and $b = c = T_0$:

$$\begin{aligned} & (a \wedge c) \vee (\neg a \wedge b) \vee (b \wedge c) && (3)\\ = \; & (\uparrow_0 \wedge T_0) \vee (\neg \uparrow_0 \wedge T_0) \vee (T_0 \wedge T_0) && (4)\\ = \; & \uparrow_0 \vee \downarrow_0 \vee T_0 && (5)\\ = \; & T_0 && (6) \end{aligned}$$

However,

$$\begin{aligned} & (a \wedge c) \vee (\neg a \wedge b) && (7)\\ = \; & (\uparrow_0 \wedge T_0) \vee (\neg \uparrow_0 \wedge T_0) && (8)\\ = \; & \uparrow_0 \vee \downarrow_0 && (9)\\ = \; & T_? && (10) \end{aligned}$$

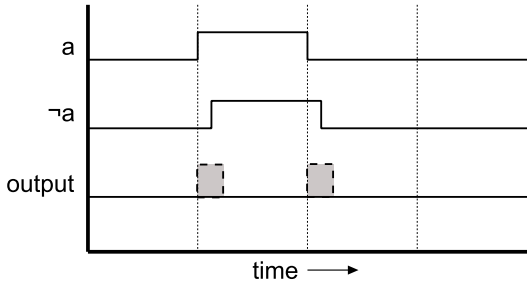
clearly demonstrating the poorer dynamic behaviour of (2).

2 Sliding Windows

The technique described in section 1.2 abstracts away the details of timing. Whilst this is clearly a desirable property in many respects, it is necessary to reintroduce a concept of absolute time in order to use this technique to build a logic simulator.

In practice, this can be achieved by slicing up the simulation in time into overlapping windows[3], where the duration of the window is equivalent to the amount of uncertainty in delay.

Within any particular window, the starting and ending state of inputs are known, allowing an abstract value F, T, $\uparrow$ or $\downarrow$ to be chosen. The circuit is then evaluated as an expression, with gates represented by their abstract counterparts. Results are shown graphically, with possible glitches highlighted appropriately:



3 Memory Elements

Circuits that contain one or more memory elements (D-type flip flops, Müller C-elements, SR latches, etc.) may be simulated using one of the two following approaches:

White Box Delay elements are simulated as their component gates, with feedback handled similarly to the way that it is typically implemented in a conventional logic simulator.

Black Box Delay elements are modeled directly, without an underlying gate-level model.

Whilst both alternatives are feasible, the black box approach will in many cases be more accurate[4]. As a further benefit, this approach makes it straightforward to automatically check design rules during simulation, allowing warnings to be generated if, for example, a clock signal can potentially glitch.

[3] In practice, windows typically need to be overlapped, since under some circumstances a transition occurring at or near the edge of a window may not be properly examined due to quantisation effects

[4] This follows from the well-known feature of abstract interpretation $(f \circ g)^{\#} \sqsubseteq f^{\#} \circ g^{\#}$, which can informally be read as stating that the abstract result of the composition of the functions f and g may be more accurately modeled by abstracting the result of a concrete composition, rather than abstracting f and g separately then performing an abstract composition.

4 Related Work

Don Gaubatz [4] proposes a 4-value 'quaternary' logic that bears some resemblance to the extended logic described in section 1.2. Quaternary logic is similar to the 5-value logic described in [7] – in this paper we choose a more accurate 9-value logic and interpret its values somewhat differently.

Paul Cunningham [3] extends Gaubatz's work in many respects, though his formalism is based on a conventional 2-value logic with transitions handled explicitly as events rather than as values in an extended logic.

Both Gaubatz and Cunningham primarily consider model checking, not simulation, and simulation tools are generally more important for the detection of design errors early in a project's life cycle.

5 Conclusions

The technique presented here offers a clear advantage over existing approaches to logic simulation, in that it can detect problems at a far earlier stage within the design life cycle, thereby offering the possibility of reduced time, cost and risk.

5.1 Future Work

At the time of writing, this technique is essentially a spin-off from broader theoretical work, and is being presented at an early stage. Implementing the technique in a new logic simulator, or alternatively adding a post-processing pass to an existing simulator is clearly desirable as a next step.

The specific abstract interpretation technique defined in section 1.2 is actually one of a number of alternative representations. Empirically comparing the representation used here with the other approaches also listed in [7] in order to determine which alternative is the most effective would be appropriate.

6 Acknowledgments

The authors would like to thank Simon Moore for his comments on an incomplete draft of this paper.

The first author wishes to thank Big Hand Ltd. for financially supporting this work.

References

[1] P. Cousot and R. Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Los Angeles, California), ACM Press, New York, NY, 1977, pp. 238–252.

[2] ———, *Systematic design of program analysis frameworks*, Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (San Antonio, Texas), ACM Press, New York, NY, 1979, pp. 269–282.

[3] P. A. Cunningham, *Verification of asynchronous circuits*, Ph.D. thesis, University of Cambridge, 2002.

[4] D. A. Gaubatz, *Logic programming analysis of asynchronous digital circuits*, Ph.D. thesis, University of Cambridge, 1991.

[5] G. Morelli, *Coralled: Get hold of wire delays*, Electronic Design News, September 25, 2003, pp. 37–46.

[6] A. Mycroft and N. D. Jones, *A relational framework for abstract interpretation*, Lecture Notes in Computer Science: Proc. Copenhagen workshop on programs as data objects, vol. 215, Springer-Verlag, 1984.

[7] S. Thompson and A. Mycroft, *Abstract interpretation of asynchronous circuits*, Manuscript, In preparation.