A Glitch in the Theory of Delay-Insensitive Circuits[1]

Bradley C. Kuszmaul

NE43-228
M.I.T. Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

bradley@abp.lcs.mit.edu
http://theory.lcs.mit.edu/~bradley/

A Puzzle ('Another Glitch')

There is a questionable assumption made in the theory and practice of delay-insensitive circuits, as presented by Dill [Dil88b] and Burns and Martin [BM88]. The problem is illustrated by the following puzzle.

> **Puzzle:** Consider an OR gate (see Figure 1) with inputs labeled a and b, and an output labeled c. Suppose that the circuit initially has $a = 1$, $b = 0$ and $c = 1$; The b input then rises, and then some finite positive time later, a drops. Does output c glitch? [2]

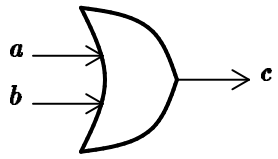


Figure 1: An OR gate. The inputs are labeled a and b. The output is labeled c.

The answer to this puzzle depends on your assumptions about the OR gate. Dill's verifier and Martin's translator both assume that c does not glitch in this case. Our position is that c does glitch. We will refer to this class of glitch as an *internal glitch*. (By *glitch*, we mean that the circuit produces an unpredictable output, and possibly enters an unpredictable state. We will discuss glitches in more detail and with more care later in the paper.)

This paper investigates the puzzle of this new glitch, in the following linear, self-timed sequence of events.

- The introduction is presented (you are almost done reading it).
- We review the work of Martin and of Dill, with an eye to understanding what kinds of glitches are avoided by Martin's designs, and what kind certification Dill's verifier makes of circuits. We also demonstrate that Martin's circuits and Dill's verifier both make the assumption that the internal-glitch is not a problem.
- We show that the internal-glitch really can be a problem, by demonstrating two implementations of OR gates (implemented at different levels of abstraction), both of which suffer from internal-glitches.
- We explore three different fixes to the internal-glitch problem. The fixes include ignoring the problem, fixing the problem, and exploiting the problem to build even better circuits. We demonstrate our fixes on a few selected circuits.
- We conclude with a summary and some directions for future work.

[1] Published in the *1990 ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (Tau '90)*, held August 14–17 at the The University of British Columbia, Vancouver, British Columbia.

[2] A remark on notation: This paper uses the value 0 to indicate the logical value FALSE, and the value 1 to indicate the logical value TRUE.

Review

In order to show how Martin's circuits and Dill's verifier may fail under certain assumptions, we will review Martin's and Dill's work 'by example'. We start with Dill's work and then use it to verify a small piece of Martin's translator.

Trace Theory 'by example'

In Dill's work, a circuit's *state* is represented by a mapping of signal names to binary values. Thus, the initial state of our puzzle is $a = 1$, $b = 0$, and $c = 1$. Those three equations give the state of the circuit. Dill adopts the following shorthand for the mapping: The state is represented by a vector of digits (where the only constraint on the ordering of the vector is that it is used consistently). Thus in our example, we would use a vector of length three, where the first digit gives the value of a, the second digit gives the value of b, and the third digit gives the value of c. Thus the initial state of our puzzle is 101.

Dill builds a finite state machine (called the *state graph* of the circuit) to represent the legal transitions for a circuit. Each arc of the finite state machine is labeled with a signal name — An arc labeled with a signal name x connects state q to state q', then the finite state machine moves from state q to q' if the value of signal x changes (either by rising from 0 to 1 or by falling from 1 to 0.) Since the value of x is determined by the state name, it is enough to know that x changes in order to compute the new value of x. In particular, the state name of q is different from q' only in the bit corresponding to signal x. Transitions which can not occur are not shown. Transitions which *should not* occur go to a special state called state F, the 'failure' state. The only successor of state F is state F.

A circuit which enters the failure state is said to *glitch*. (Dill uses the terminology that the circuit *fails*.) A circuit which glitches produces unpredictable outputs and enters an unpredictable state. It is worse than that the circuit enters, nondeterministically, some valid state and produces, nondeterministically, some valid output. The resulting output and state may be invalid. A glitch may result in no change on the output, or it may result in an arbitrarily brief change on the output (a spike), or the output may become invalid. The circuit could even fail by entering the *smoking* state, from which the circuit and the surrounding power supply may never emerge.

Certain states are called 'unstable'. A stable state is one in which the value of the outputs agrees with the 'DC' specification of the circuit, i.e., the boolean function of the circuit, as determined by the inputs of the circuit (and any hidden state of the circuit). The other states, except for F, are called unstable states. If changing an input would move the circuit from an unstable state to a stable state, then the circuit glitches (this rule really only works for boolean gates — Dill presents a formalism for computing the state graphs of more general circuits with hidden state, but the simpler rules will suffice for this part of the discussion.)

Figure 2 shows Dill's state graph for an OR gate. For example, from state 000, no arc labeled c is shown, because if the circuit is in state 000 the circuit will not raise c. From state 000 it is legal for a to change, in which case the circuit enters state 100. From state 100, the circuit may enter state 101 (if c changes) or 110 (if b changes), but is is illegal for a to change because the circuit would be moving from an unstable state to a stable state via an input transition. Thus from state 100, a transition on a leads to state F (shown in the center of the cube).

Note that Dill's answer to our puzzle is 'the output does not glitch', because the circuit does not fail for the sequence of states and transitions given by

$$101 \stackrel{b\uparrow}{\rightarrow} 111 \stackrel{a\downarrow}{\rightarrow} 011.$$

Circuit Generation 'By Example'

The circuts generated by Burns and Martin assume that the answer to our puzzle is 'the output does not glitch'. This subsection shows how our puzzle creeps into those circuits. Burns and Martin transform a program written in a variant of Communicating Sequential Processes (CSP) [Hoa78] into delay-insensitive circuits using syntax directed translation. We will focus on one particular rule, the 'translation rule for sequencing', which is illustrated pictorially by Figure 3.

The general method of transformation of a CSP program into a circuit is that a statement is transformed into a circuit which has a four-phase handshake interface for control flow (and possibly some other wires

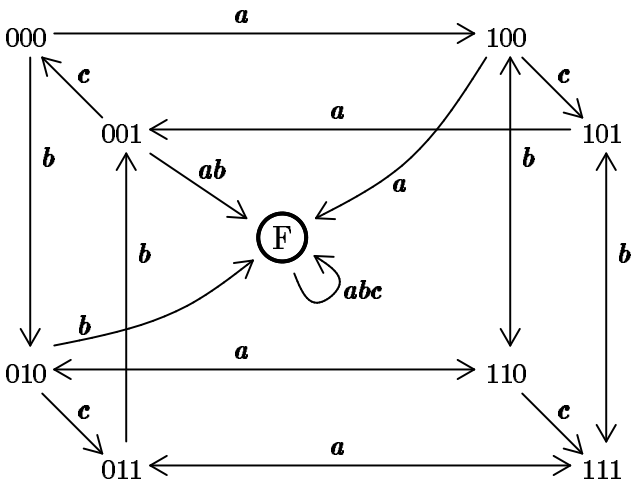


Figure 2: The state graph of an OR gate. (This figure is borrowed, with minor corrections, from [Dil88b], Page 55.)

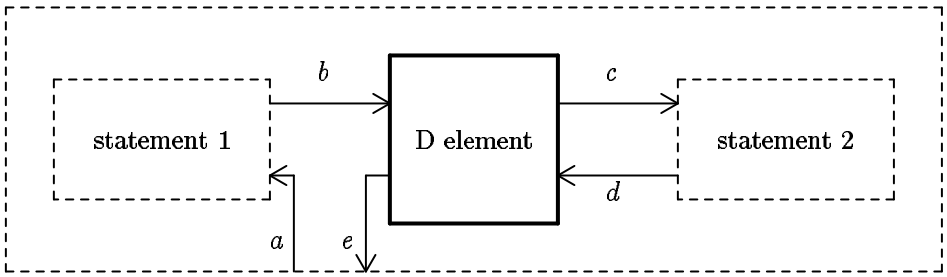


Figure 3: The translation rule for sequencing (borrowed from Page 42 of [BM88].)

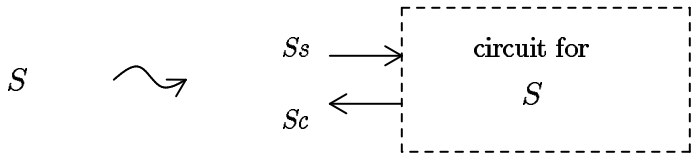


Figure 4: The translation of a statement, S, into a circuit with input S_s and output S_c.

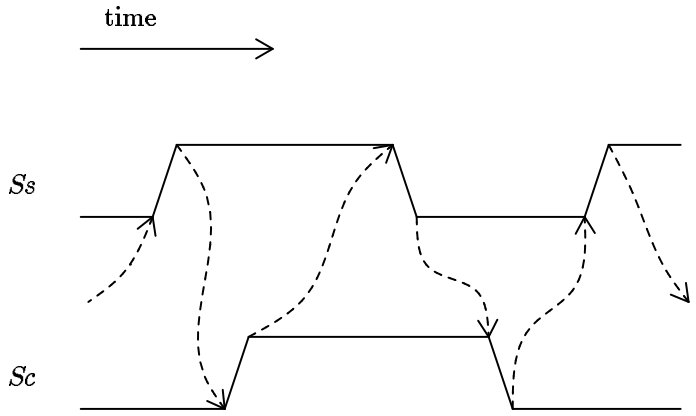


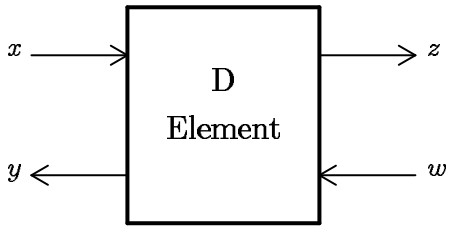
Figure 5: How four-phase handshaking works: S_s rises, and then S_c may rise, and then S_s may fall, and then S_c may fall, returning the circuit to the initial state, and then S_s may rise again and so forth.

to account for data-flow — here we focus on the control flow.) Statements are composed using sequencing, iteration, and other control flow constructs, and the four-phase handshake interfaces are hooked up to implement the operations in the order implied by the control flow of the original CSP program.

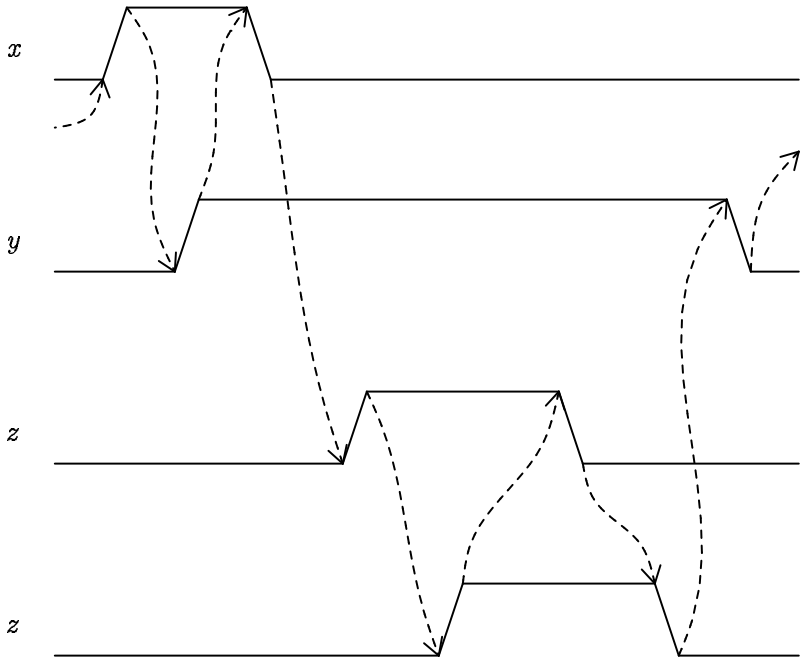
Thus given a statement S, one generates a circuit with an input signal (named S_s) and an output signal (named S_c.) (See Figure 4.) The input signal S_s is used to start the execution of statement S, and the output signal S_c is used to signal the termination of statement S. The $\langle S_s, S_c \rangle$ pair operate using a four-phase handshaking protocol, shown as a timing diagram in Figure 5. The operational semantics of the signals are that when S_s rises, the statement may start execution. Some time later, S_c rises to indicate that the statement has started to execute, at which point S_s is allowed to drop, and then sometime after S_s has dropped and the statement has finished execution S_c drops, returning the circuit to its initial state, so that the statement may be executed again sometime in the future. In this paper, refer to the $\langle S_s, S_c \rangle$ pair of signals as the *controlling interface* for the circuit implementing statement S.

The case we will focus on, the strict sequencing of two statements, is implemented by the transformation shown in Figure 3. Given the translations of two statements S_1 and S_2, we want to build a circuit in which S_1 is guaranteed to finish executing before S_2 starts execution. The **D**-element is used to sequence the two statements, and works as follows. The **D**-element and the specification for its timing is shown in Figure 6. When refering to a **D**-element in this paper, we call the pair of wires $\langle x, y \rangle$ the master interface, and the pair of wires $\langle z, w \rangle$ the slave interface. The **D**-element guarantees that any statement controlled by the slave interface will not start executing until very late in the master interface's 4-phase cycle (i.e., z will not rise until after x has fallen. Thus if two statements are hooked together as shown in Figure 3, the first statement will have finished executing (indicated by b falling) before the second statement starts execution (controlled by c rising). The circuit as a whole implements the strict sequentialization of S_1 and S_2, and has a new controlling interface (implemented by the pair of signals $\langle a, e \rangle$.)

The **D**-element is one of the places where our puzzle becomes applicable to circuits generated by Burns and Martin. The implementation of the **D**-element used by Martin and Burns is derived using the techniques discussed by Martin in [Mar86]. In particular, the assumption that forks operate isochronously within a small



(a) The external interface of the **D**-element.



(b) A timing diagram to specify the behavior of the **D**-element.

Figure 6: Specification of the **D**-element.

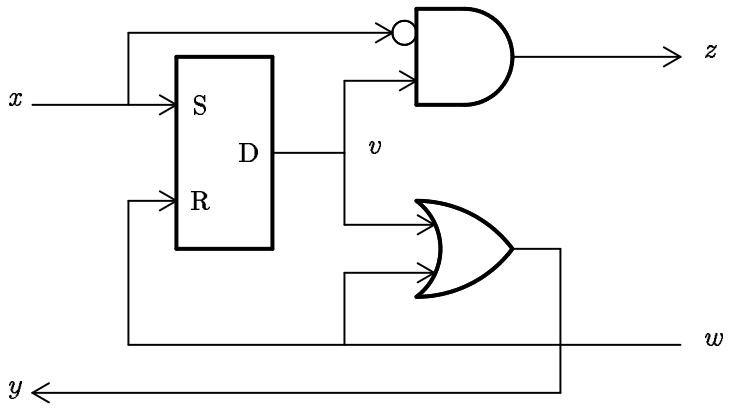


Figure 7: The implementation of the **D**-element, as suggested by Martin and Burns.

circuit is exploited to distribute a single value to different gates *at the same time* (or more properly, the difference in the times of arrival at the different gates is so small that the arrivals might as well be at the same time.) This is precisely the point at which the puzzle of the internal-glitch creeps in. (Actually, the internal-glitch creeps in at an even lower level, in the production rule for an OR gate. The details of how the OR gate is produced can be found on Page 24 of Burns's S.M. thesis [Bur88]. The production rule is not quite strong enough to prevent the internal-glitch.)

The implementation of the **D**-element suggested by Martin and Burns is shown in Figure 7. The signals x, y, z, and w are the external signals specified in Figure 6. The signal v is internal. An OR gate is used, along with an AND gate (with an inverted input), and an SR-latch is used. (The behavior of the SR-latch is that if $S = 1$ then the output becomes 1, and if $R = 0$ then the output becomes 0. It is a failure for $S = R = 1$.)

The verification of Martin's **D**-element can be performed using Dill's technique as shown in Figure 8. To represent the state, we use a 5-element bit vector, containing the values, in order, of x, y, z, w, and v. All of the internal states of the verification are shown. Note that except for edges leading to the failure state, the state graph is just a loop. I.e., From any given state, there is exactly one signal which may legally make a transition.

Note that the sequence of transitions

$$01101 \stackrel{w\uparrow}{\rightarrow} 01111 \stackrel{v\downarrow}{\rightarrow} 01110$$

is exactly the situation originally mentioned in our puzzle. Also, there is a corresponding situation in the sequence of transitions

$$00000 \stackrel{x\uparrow}{\rightarrow} 10000 \stackrel{v\uparrow}{\rightarrow} 10001$$

where the AND gate with inverted input glitches according to our reasoning, but not according to the (implicit) assumptions made by Martin and Dill.

Is the Internal-Glitch Real?

This section demonstrates that the internal-glitch can occur, given weak enough assumptions about our primitives. We have reviewed the kinds of hazards avoided by Martin's design strategy and detected by Dill's verifier, and we noted that both Martin and Dill would answer 'no glitch' to our puzzle. In order to show that their circuits can suffer from the internal-glitch we make weaker assumptions about the circuits than are (apparently) made by Martin and Dill. One can expect a certain degree of controversy about which assumptions are correct. Rather than first state the assumptions explicitly, we will demonstrate a few implementations of boolean gates in which the internal glitch appears, and then we will explore the

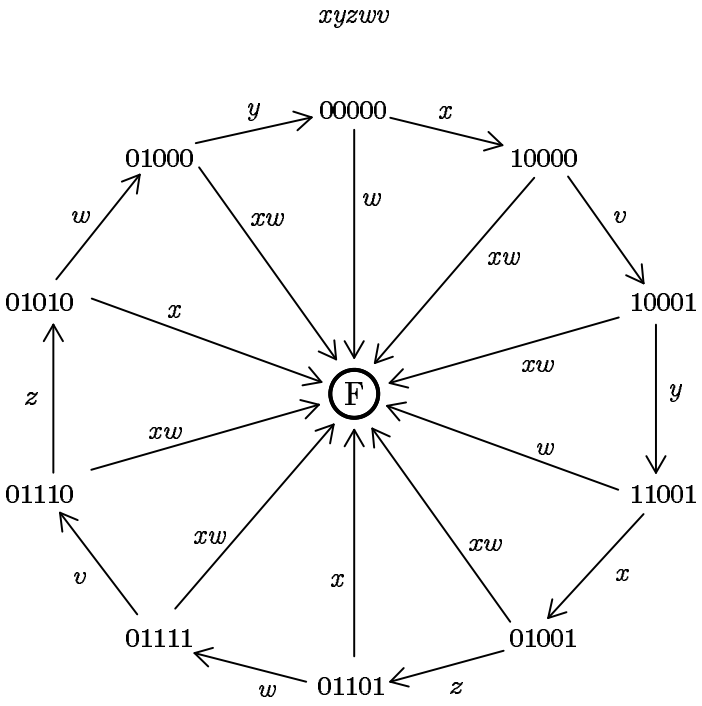


Figure 8: The verification of Martin's **D**-element using Dill's techniques. The state vector contains the values, in order, of x, y, z, w, and v. All of the internal states of the verification are shown.

assumptions. We can demonstrate the internal glitch by using a boolean (discrete) circuit model (where values are always either valid 1 or valid 0); We can also demonstrate the internal glitch by using a continous model if we look inside the transistor-level implementation of a typical logic device. The idea in both cases is to cause the events triggered by the rise of b to occur very slowly compared to the speed of the events triggered by the fall of a.

Internal Glitch in the Discrete Model of Logic

A logic designer might choose to implement an OR gate as shown in Figure 9 as two inverters feeding a NAND gate. (An alternative design, in CMOS, might be to implement the OR gate as a NOR gate feeding a single inverter. The NOR implementation gate has fewer transistors, and might be of smaller area, but in CMOS the NAND implementation would probably be faster. We concentrate on the NAND implementation, since it meets our needs.)

Now if we try to reason about this circuit, we will discover that it does not meet Dill's specification for

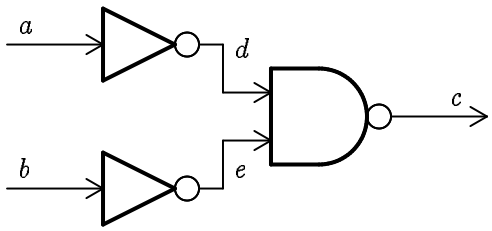


Figure 9: The implementation of an OR gate as two inverters feeding a NAND gate.

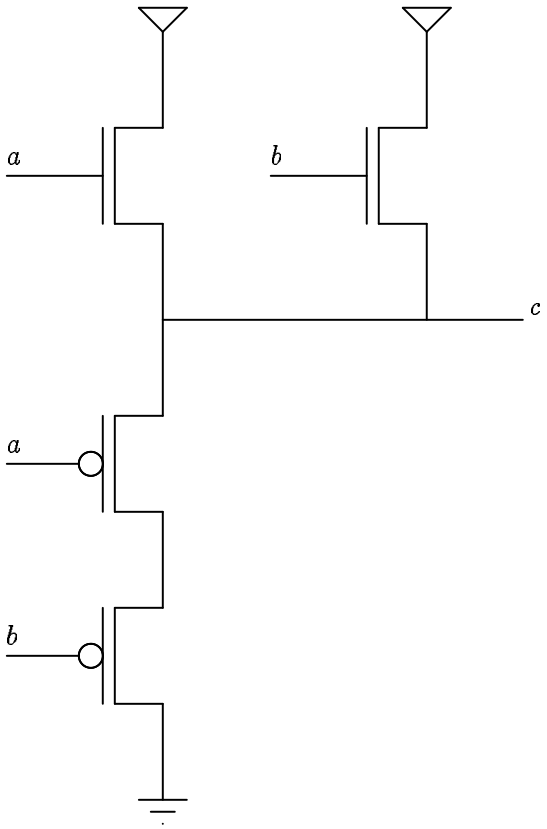


Figure 10: An OR gate implemented using a nice underlying technology in which $V_t = 0$.

an OR gate. Specifically, if we denote the state of this circuit by a 5 element bit-vector, denoting the values a, b, c, d, and e in order, and start in the stable state 10101, consider what happens when we raise b then drop a. When b rises and then a falls, there is a race condition between a falling and e falling. If e falls first, then the 'correct' thing happens: a then falls, and then d rises, with c remaining a stable 1. However if the fall of e is delayed long enough by the inverter generating e, then a could fall, then d could rise, then c could fall, then e could rise and finaly c could rise. The net result is that this circuit may allow c to briefly fall, which does not meet the specification of an OR gate. (The specification of the OR gate is given by the state graph in Figure 2, where there is no arc allowing c to change when the circuit is in state 011.)

Internal Glitch in the Continuous Model of Logic

In this subsection, we will demonstrate an internal glitch using circuits built from transistors, resistors, and capacitors. We do this in order to show that the internal glitch demonstrated in the previous subsection has some basis in the lower level abstractions used to build logic devices. Figure 10 shows an OR gate implemented using transistors from a synthetic underlying technology in which V_t, the threshold voltage, is zero.

The assumption that $V_t = 0$ means that the transistors are just as good at passing 1's as they are at passing 0's. We make this assumption in order to be able to easily use an OR gate as our example, but if we wanted to use the kind of transistors found in MOS technologies [WE85], we could rephrase the example in terms of a NAND gate with the corresponding puzzle state transition given by

$$011 \stackrel{b\downarrow}{\rightarrow} 001 \stackrel{a\uparrow}{\rightarrow} 101.$$

Instead, we choose to use the OR gate, and to make some simplifying assumptions about the underlying

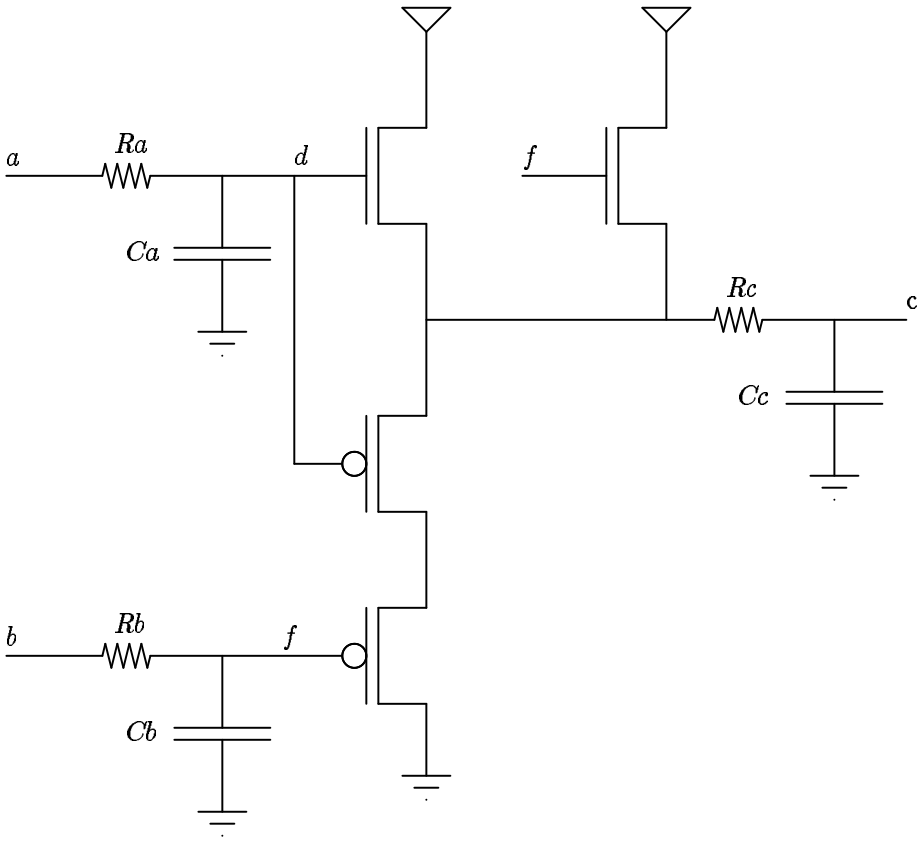


Figure 11: An OR gate implementation using ideal transistors with explicit resistance and capacitance.

transistor level abstraction.

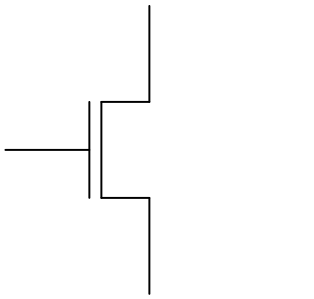
The DC characteristics of the circuit shown in Figure 10 certainly agree with the DC characteristics of an OR gate, however we intend to show that the AC characteristics can be manipulated by the designer so that the circuit does not meet Dill's specification for an OR gate. We do this by modeling the circuit as shown in Figure 11 using an idealized model in which all of the resistance and capacitance is explicit. We have lumped together certain resistors and capicitors in order to keep the model simple. We also assume that this circuit operates in an equipotential region (i.e., the circuit acts the same from any point of reference, or in other words, there is no problem with reasoning about time). Our idealized transistors have DC characteristics as shown in Figure 12, in which horizontal axis is the gate voltage (ranging from 0 to 1 volts) and the vertical axis is the conductivity of the transistor across the gate (ranging from 0 to 1 mhos). Figure 12 Part (a) shows what we will call a non-inverting transistor (or n-transistor); Part (b) shows the n-transistor's DC transfer characteristic; Part (c) shows an inverting transistor (or p-transistor); Part (d) shows the p-transistors's DC transfer characteristic.

If we assume that the circuit starts out with $b = f = 0$ (volts) and $a = f = c = 1$ (volts), and that at time $t = 0$, the signal b rises (say, as a perfect step function), then the value of f is given (in volts) by

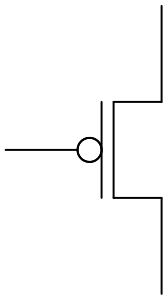
$$f = \begin{cases} 0 & \text{if } t \leq 0, \\ 1 - e^{-\frac{t}{R_b C_b}} & \text{if } t \geq 0. \end{cases}$$

Transistor T2 switches on and transistor T4 switches off at the time when $f = 0.5V$, which is

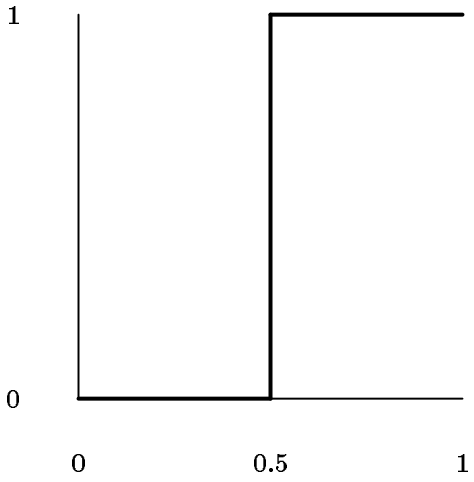
$$t_\beta = -R_b C_b \ln \frac{1}{2}.$$



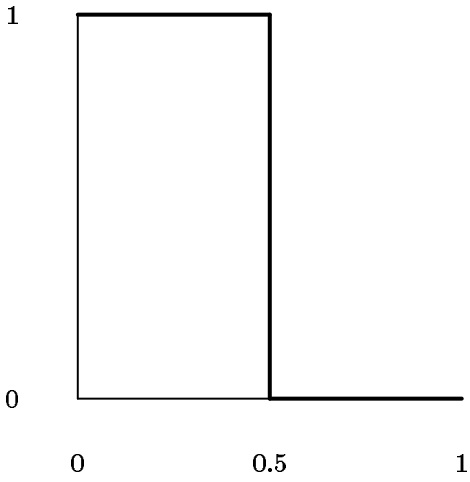
(a) Non-inverting n-transistor.



(c) Inverting p-transistor.



(b) N-transistor DC characteristic.



(d) P-transistor DC characteristic.

Figure 12: Ideal transistors with their DC characterics. The horizontal axis of the DC characteristic graph is the gate voltage (ranging from 0 to 1 volts). The vertical axis is conductivity of the transistor across the gate (ranging from 0 to 1 mhos).

If at time $t = \epsilon$, the signal a falls (as a perfect step function), then the value of d is given (in volts) by

$$d = \begin{cases} 1 & \text{if } t \leq \epsilon, \\ e^{-\frac{t-\epsilon}{R_b C_b}} & \text{if } t \geq \epsilon. \end{cases}$$

Transistor T1 switches off and transistor T3 switches on at the time when $d = 0.5V$, which is

$$t_\alpha = \epsilon - R_a C_a \ln \frac{1}{2}.$$

If $t_\alpha < t_\beta$ then charge will be pulled off of capacitor C_c, and by choosing a sufficiently small value for $R_c C_c$, we can cause the voltage at c to briefly drop to 0. This occurs if

$$\epsilon < (R_a C_a - R_b C_b) \ln \frac{1}{2},$$

which can be guaranteed by making the $R_b C_b$ product large compared to the $R_a C_a$ product. Viewing the RC product as the time constant for an exponential delay, we can see that this is exactly the condition that we stated earlier — the events triggered by the rise of b occur slowly compared to the events triggered the fall of a.

Figure 13 shows a simulation of the circuit, using numerical methods, to show what happens in our circuit when $R_a C_a = 0.1F$, $R_b C_b = 0.5F$, $R_c C_c = 0.1F$. The voltage at c, V_c shows a clear glitch. A range in speed of a factor of five is a little extreme, even for CMOS (and especially for transistors on a single chip), but the point is that without making *some* assumption about the relative RC constants of a circuit, it is easy to end up with an OR-gate that suffers from internal-glitch.

Bad Gates

We have demonstrated using two very different models of logic circuits, that given weak enough assumptions about the implementation of an OR gate, the 'internal glitch' of an OR gate can indeed occur. I.e., it is possible to design OR gates that have correct DC characteristics, but incorrect AC characteristics. It is clear that our OR gates do not meet the specification given Dill, and therefore our OR gates can not be used in Martin's circuits.

Dill uses 'stable' states to generate the state-graph for a boolean gate. One must do something more detailed than that to avoid internal-glitches; One must assume that any time an input changes, the state becomes becomes 'unstable' in some deeper sense. Thus we know something is fishy if, when an input changes, the circuit moves from one stable-state to another stable-state. The circuit has moved into an unstable state without us realizing it, and there is no way to determine when the circuit will become stable again. The circuit needs to be modified so that we can always tell that the circuit has become stable after an input has changed. Primitives that meet the weak-conditions described by Seitz ([MC80], Chapter 7) meet the informal conditions described here.

Fixing the Problem

There are at least three ways to fix-up the internal-glitch:

1. Assume it is not a problem,
2. use better primitive components, or
3. exploit the underlying assumptions at a higher level.

Assume the Problem Away

One could argue that since our OR gates simply do not meet the specification of an OR gate as given by Dill and used by Martin, the problem is not really there; One should just use an OR gate that does meet the specification. Since Martin and Dill both explicitly state that some of the primitives must be

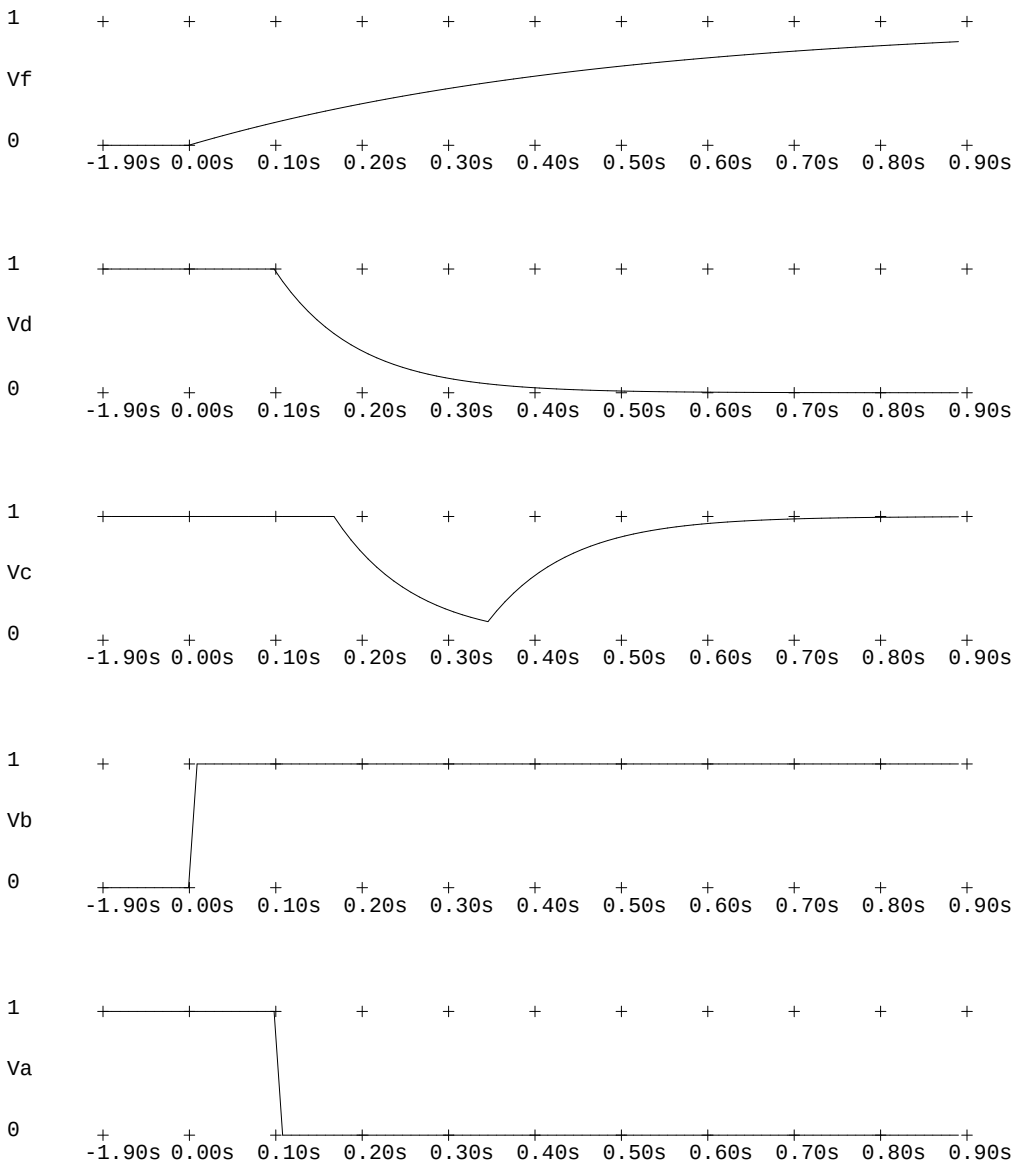


Figure 13: A simulation of the circuit of Figure 11, with $R_aC_a = 0.1F$, $R_bC_b = 0.5F$, $R_cC_c = 0.1F$.

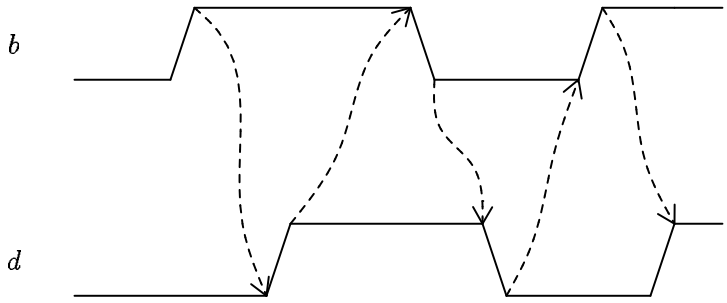


Figure 14: Timing of a two-phase, non-return-to-zero (NRZ) protocol, where b is the input and d is the acknowledgement. Note that this looks the same as Figure 5, but there is a difference: A complete cycle consists of only two transistions (one b and one on d instead of four transistions. A single cycle of the four-phase protocol is just the composition of two cycles of the two-phase protcol.

designed using analogue circuit techniques (for example the mutual-exclusion element can not be designed using only discrete logic values [AG89]), it seems reasonable to assume that the OR gate must be designed, using analogue circuit techniques, to meet their specification. This approach has its merits, since in fact, the circuits designed by Martin seem to actually work. However, the assumption about OR gates is only stated indirectly through the state-graph. We would like an assumption about the primitives which is more general, and easier to state.

We propose the following, additional assumption:

> **new assumption** (the *atomic-gate* assumption): The primitive gates operate instantaneously, but the outputs may be delayed arbitrarily.

This new assumption puts all of the delays into the wires, which is a nice place for them.

In fact, this assumption is really part of the isochronous-fork assumption made by Martin and Dill. The isochronous-fork assumption does not make sense unless one also assumes that gates are atomic. Correspondingly, the atomic-gate assumption does not make sense unless one also assumes that there are isochronous forks.

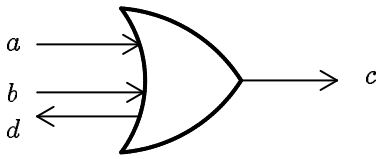
The question is whether this assumption is a good one. It seems reasonable to be able to decompose an OR gate into smaller primitives, but this assumption does not allow such a decomposition, which is a mark against the assumption. On the other hand, if the assumption greatly simplifies the circuitry, and if the assumption actually does hold for OR gates, then why not assume it?

Use Better Primitives

Another approach is to use better primitive components. The problem with the way the OR gate is used in Martin's circuit is that there is no way to verify that the OR gate has correctly registered the rise of b if a has already risen. There is no way to safely exit the state where $b = a = 1$. A solution is to require that the gate acknowledge the risen of b. This is something like the weak-conditions described by Seitz ([MC80], Chapter 7).

One way to acknowledge the rise of b would be to use a two-phase, non-return-to-zero (NRZ) protocol. The NRZ protocol consisits of two signals, one carrying a bit of data, and one returning to the logical source, acknowledging the receipt of the bit. The timing of the NRZ protocol is shown in Figure 14. This approach is discussed by Seitz (Chapter 7 of [MC80]). Note that with a NRZ protocol, we can avoid the use of isochronous forks in the construction of our circuits.

In the extreme, one could replace every logical signal with a NRZ pair, but that would be overkill. We can get away with putting an NRZ protocol in just a few places, substantially reducing the cost of the circuitry, and making the circuitry easier to reason about. In particular, we can use a primitive OR gate that has an NRZ protocol on one of the inputs and a plain 'single-ended' protocol on the other input.



(a) The lopsided-OR gate.

inputs		outputs	
a	b	$c := a \vee b$	$d := b$
0	0	0	0
0	1	1	1
1	0	1	0
1	1	1	1

(b) The boolean function computed by the gate. (DC Characteristics).

Figure 15: Lopsided-OR gate with DC characteristics.

Figure 15 shows an asymetrical OR gate, which we call a lopsided-OR gate, with its DC characteristics. The domain constrains of a lopsided-OR gate are specified by a state-graph in Figure 16. Each state is labeled with the value of a, b, c, and d, in order. The failure state is not shown explicitly; If an input signal changes and there is no explicit arc for that input signal, then the next state is the failure state. Note that two states can not be reached by valid transitions from the initial state.. Among the constraints are

- if $b = 1$ and $d = 0$ then a may not fall (because we must wait for b's rise to be regsitered by the circuit before allowing a to fall), and
- input a may not rise if $b = 1$ (because there would be no way to verify that a's rise is registered by the circuit), and
- if $a = 1$ and $c = 0$ then b may not rise (because there would be no way to verify that a's rise had been registered by the circuit).

The constraint on the internal design of the gate is that if $a = 1$, and b rises, then d must not rise until it can be guaranteed that c will not glitch if a falls.

The b input to a lopsided-OR gate follows a two-phase, non-return-to-zero (NRZ) protocol to provide handshaking between the lopsided-OR gate and the supplier of the b input.

One can build other lopsided gates. E.g., a lopsided-AND gate can be implemented using a lopsided-OR gate and some inverters as shown in Figure 17. Note that a real gate designer would actually build the lopsided-AND gate directly, but we use the lopsided-OR gate in order to simplify the presentation and to show how lopsided gates meet can be composed where the ordinary gates could not be composed. In general one can invert an NRZ signal using a pair of inverters, one on the data signal, and one on the acknowledge signal. The lopsided-AND in Figure 17 is build out of a lopsided-OR with inverted inputs and outputs. The single-ended a input and single-ended c output are just inverted with single inverters, while the NRZ b input is inverted with a pair of inverters.

A note for implementors of lopsided-OR gates: If we can build an OR gate that satisfies the atomic-gate assumption, then we can use that OR gate to build a lopsided-OR gate as shown in Figure 18. We assume that the region within the dotted box is a isochronous region. The d signal is simply derived from b by a (nonisochronous) fork.

We can use the lopsided gates to implement a **D**-element as shown in Figure 19. Note that there is only one fork, and we do not depend on that fork operating isochronously. It is straightforward, using trace analysis, to verify that this circuit will operate correctly. The problem in the original **D**-element circuit was that the two boolean gates experienced internal-glitches. The NRZ protocol is used to guarantee that the gates will not experience an internal-glitch, since, e.g., in the case of the OR gate, the falling of v is strictly preceeded by the rise of the w_{ack} signal.

We can use lopsided gates to implement Martin's most complex published circuit, the Distributed Mutual Exclusion circuit. This circuit is shown in Figure 20. An explanation of this circuits function can be found

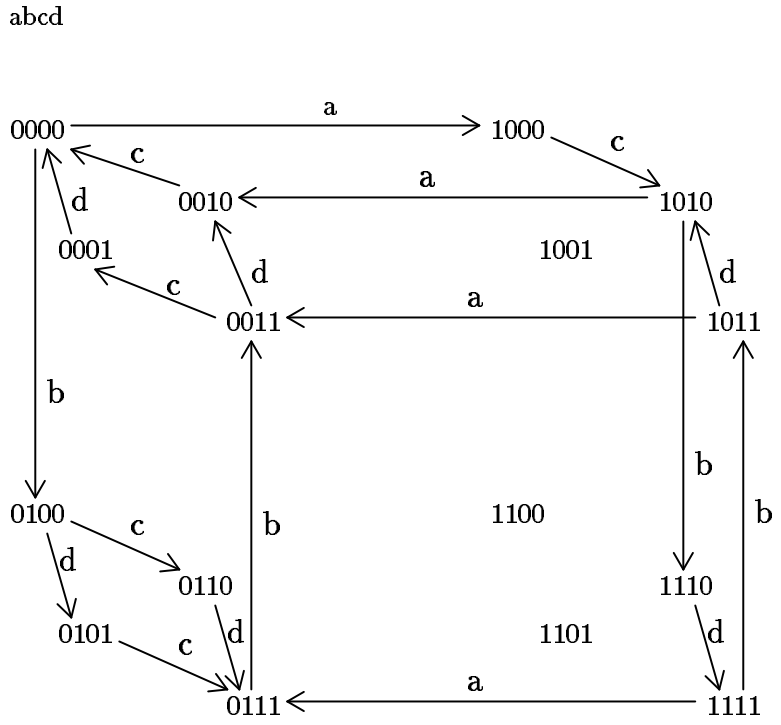


Figure 16: State graph for lopsided-OR gate. The states are labeled with the values of a, b, c, and d, in order. The failure state is not explicitly shown.

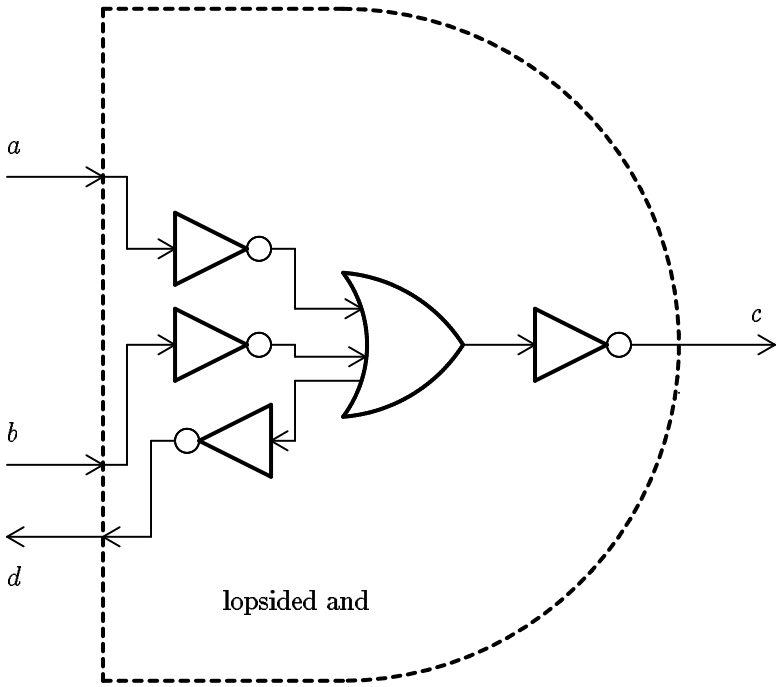


Figure 17: A lopsided-AND gate can be built using a lopsided-OR gate and some inverters.

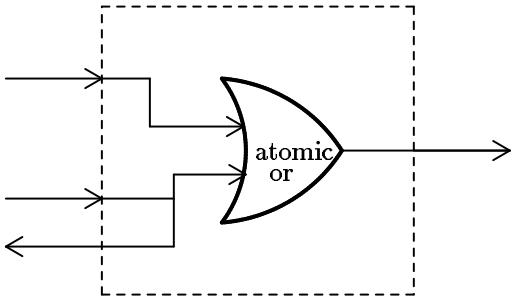


Figure 18: How to build a lopsided-OR gate out of an atomic OR gate. The region inside the dotted box is an isochronous region.

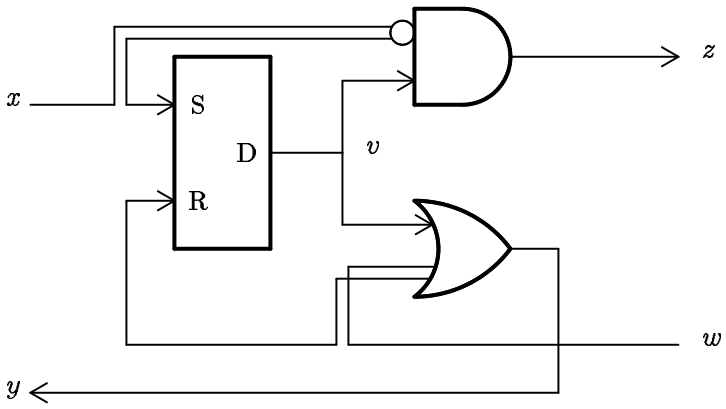


Figure 19: A **D**-element implemented with lopsided gates.

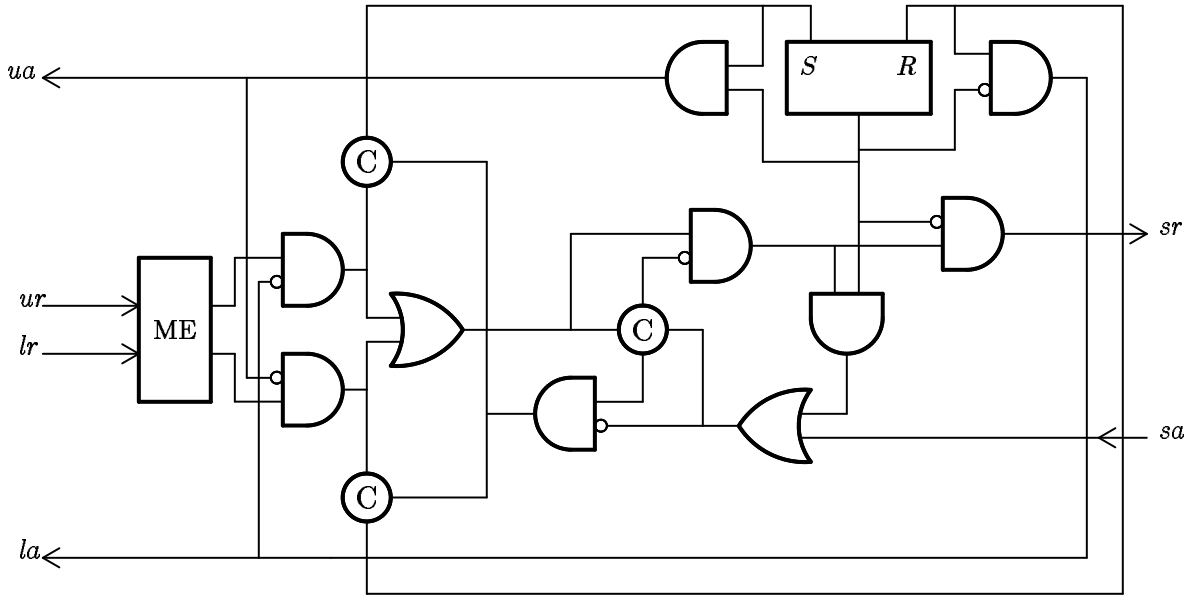


Figure 20: Martin's DME circuit, as verified by Dill. (Borrwed from [Dil88b], Page 63.)

in [Dil88b], complete with an important correction. The construction of the circuit, in its earlier, uncorrected, form can be found in [Mar85].

In order to implement the DME we need a few extra primitives that use an NRZ interface (see Figure 21.) Part (a) shows the an NRZ SR-latch, Part (b) shows an NRZ 'double-C' element. The NRZ SR-latch has NRZ inputs for s and r and NRZ outputs for dt and df (the stored-data and inverted stored-data signals respectively). Informally, the new function constraints on the latch are:

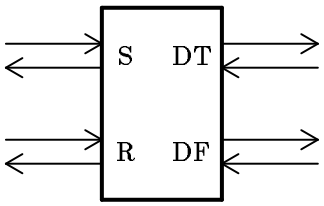
- If the latch is being set, then it must pull down df (and recieve an acknowledge) before pulling up dt.
- If the latch is being reset, then it must pull down dt (and recieve an acknowledge) before pulling up df.

The NRZ double-C element implements a pair of mueller C elements, sharing one of the inputs. Informally, the new function constraints on a double-C element are that both C elements must have 'registered' the shared input before either C element is allowed to change. We will skip the formal statement of the function and domain constraints on these new NRZ elements.

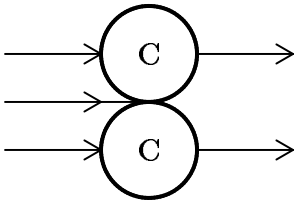
One might ask "why does one need a double-C element?" The answer is that we want to take a single-ended signal and use it to control two C-elements. If we had a NRZ signal pair, we could split the signal using an NRZ fork (which essentially would include a C-element inside it to produce the acknowledge signal) and feed the two C-elements with NRZ signal pairs. The approach found here seems more convenenient, in this case, than the use of NRZ signal pairs everywhere, because we can get away without using NRZ pairs on the inputs to the double-C element.

The problem with the way the C-elements are used in Martin's circuits is that if one of the inputs does not change, the other is allowed to change briefly without the C-element changing state or otherwise acknowledging the brief change. This is another case of an internal glitch — there is no way to prevent the the internal delays of the C-element from conspiring against us to cause the circuit to fail.

The new delay-insensitive DME circuit which avoids internal-glitches is shown in Figure 22. The author has verified, by hand, that this circuit does not fail and it avoids internal glitches. It would be a good idea to verify the circuit with an automatic verifier.



(a) A NRZ SR-latch.



(b) A NRZ double-C element.

Figure 21: New state-holding primitives to avoid internal glitches.

See the DME BUILT WITH NRZ primitives, attached.

Figure 22: Martin's DME circuit, re-engineered to avoid internal glitches.

Exploit the Assumption

A third way to address the puzzle of the internal-glitch is to really exploit the deeper assumptions being made by Martin and Dill — assumptions that allow the use of OR gates without worrying about the internal-glitch.

One such assumption is that, locally, there is a strong positive correlation between the delays in one gate (or transistor) and the delays in another. Lakshmikumar et al [KHC86] present the results of a very scientific study characterizing mismatch in MOS transistors. The study includes some theory to explain how mismatch is introduced into the system, and some good measurements with error bars to follow up on the theory.

The use of matched delays to generate timing signals is apparently a common practice in the design of circuits which must not glitch.

An example of putting a matched-transistor assumption to practice can be found in an SRAM design by Schuster et al [SCD*84]. Schuster uses matched transistors to perform self-timing of the ENABLE signal on a sense-amplifier in an SRAM design. The problem faced in SRAM design is that the sense-amplifiers on the bit-lines should not be enabled too early (or the sense-amplifiers may sense a false answer), and it should not be enabled too late (or the circuit will run slowly). The solution used by Schuster, shown in Figure 23, is to use the word-line (which triggers the output of a bit of data onto each bit-line) to also pull down a 'dummy-line'. The dummy-line controls the sense-amplifier. The features of the dummy-line are carefully designed so that the delay on the dummy-line will track the delays on the bit-lines (even in the face of process variations, and temperature and supply voltage fluctuations).

The use of matched delays even shows up in undergraduate circuit design courses. A common trick taught to undergraduates, to help them avoid glitches in finite state machines generating a complex clock signal, is to include all the min-terms in any sum-of-products structure. The students avoid the external-glitch by assuming that the internal-glitch is not a problem (because the AND gates all run at about the same speed). Of course, this strategy is risky, and we often tell undergraduates never to generate a clock signal through a logic gate.

Several possible assumptions have been proposed in the literature. For example, Seger [Seg89] proves that it is possible to verify a a circuit against race conditions in polynomial time if the gates satisfy the following property

> **Constant-Factor Assumption:** There is some constant c such that for any two gates, the speed, s_1, of the first gate, is related to the speed, s_2 of the second gate by
>
> $$\frac{1}{c}s_1 \leq s_2 \leq cs_1.$$

As an example to show the advantages of strengthening ones assumptions about circuits, we present a reimplementation of Martin's distributed mutual exclusion element (DME). Figure 24 shows a finite-state-machine (FSM) to implement the DME element. Figure 25 shows a DME circuit, based on the FSM, reengineered to take advantage of a stronger assumption. The circuit was designed using finite-state-machine design techniques, without worrying "too much" about the order in which gates evaluate their inputs and assert their outputs.

We have not worked out a precise description of all of the conditions needed for correct operation of the circuit of Figure 25. One example is that when sr goes true, we require that sa does not go true until la has had a chance to reset, i.e., about two gate delays.

Our new DME implementation is substantially smaller and faster than the previous DME implementations. Our estimate of the gate count is that there are about 13 gates in our implementation of the DME cell (in addition to the ME cell), as compared with 24 gates in Martin's implementation. Our estimate of the speed of our circuit is that the critical paths are all shorter. I.e., the longest delay from an input changing to an output changing is about 4 gate-delays in the new circuit, where in the old circuit it was about 8 gate-delays.

Conclusions and Directions for Future Work

Circuit designers often depend on unstated assumptions to design 'correct' circuits. The circuits designed by Martin and verified by Dill depend on a fairly strong assumption about primitive gates. That assumption

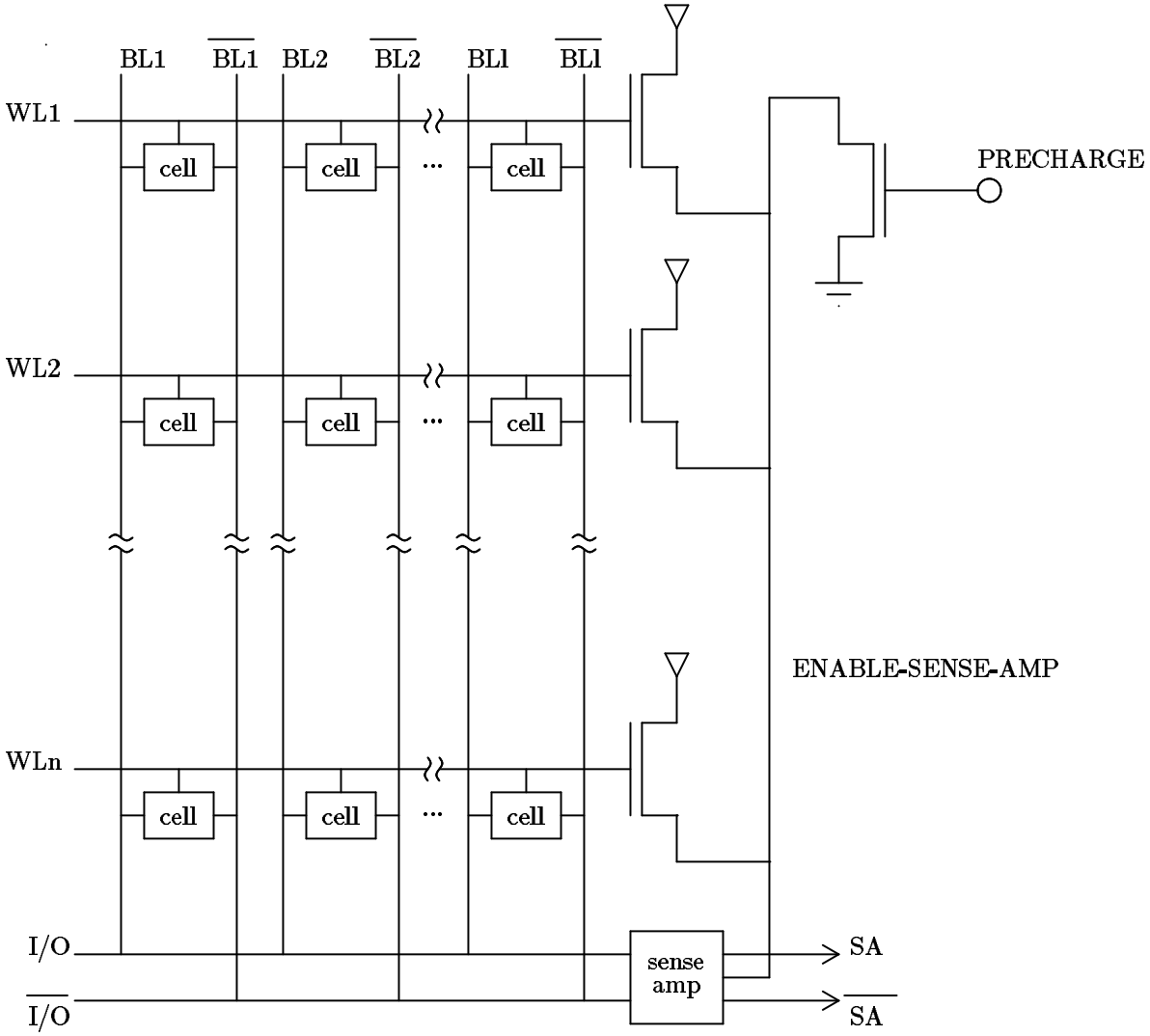


Figure 23: A self-timed DRAM circuit. The word-line enables output of bits onto the bit-lines and also pulls down the sense-amplifier's ENABLE signal.

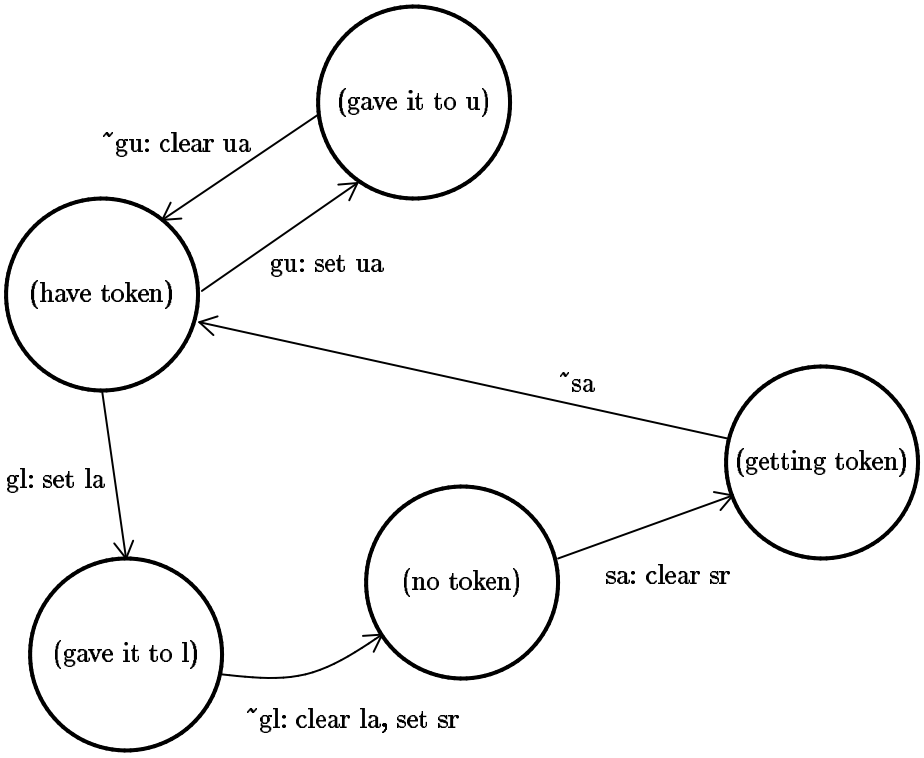


Figure 24: A finite-state-machine to implement the DME element.

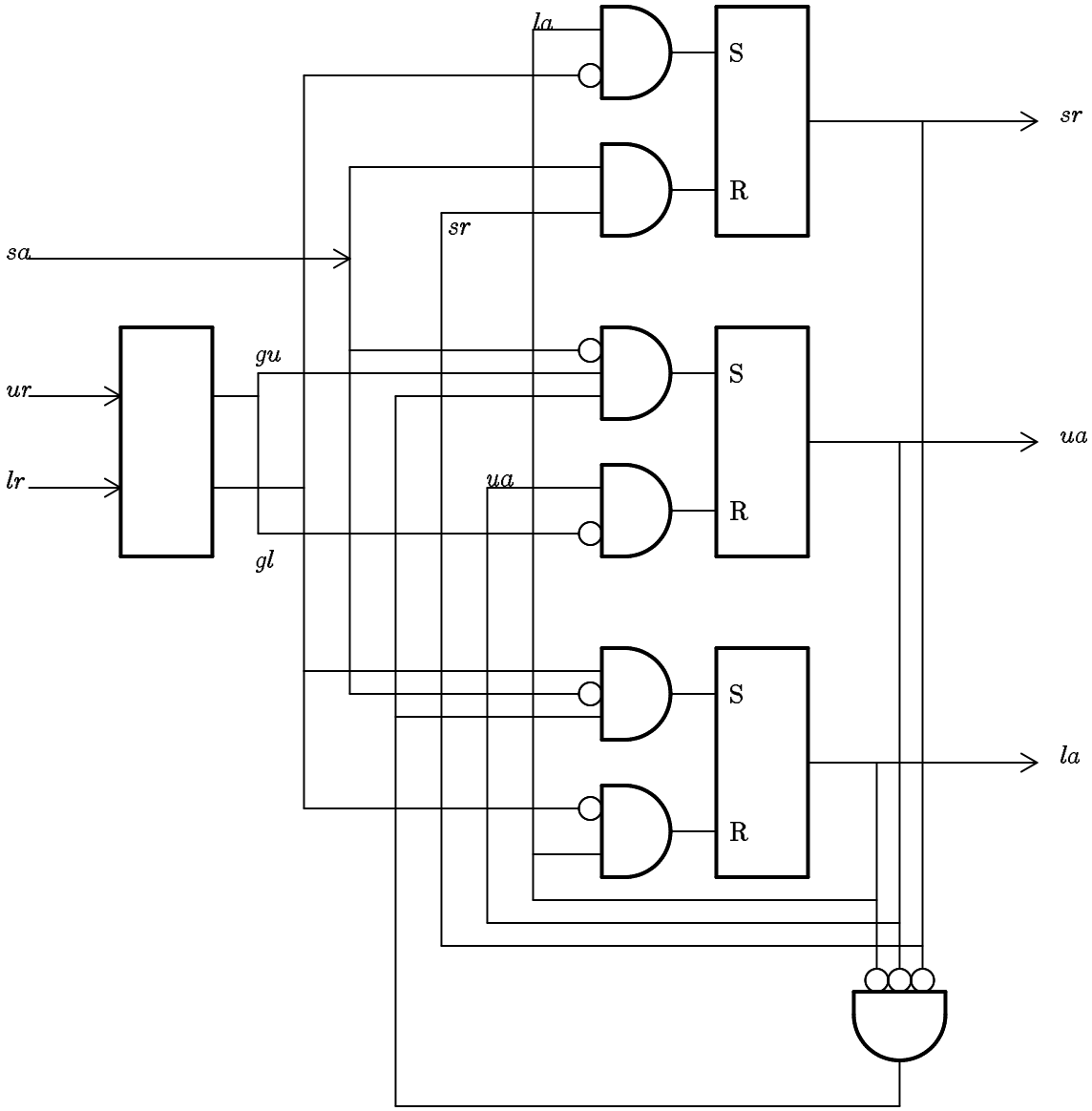


Figure 25: A DME circuit which is delay-sensitive.

is probably actually satisfied by the gates used in those circuits, but it is important to at least state the assumptions clearly. It does not suffice to assume 'isochronous forks', since the gates may still fail. One can work around the strong assumptions made by Martin and Dill by using NRZ signals in certain situations, and by rephrasing the primitives slightly. Engineers concerned with building real VLSI circuits often make other, even stronger assumptions. We showed, using the DME circuit as an example, that circuits designed with these stronger assumptions can be made smaller and faster than the corresponding circuits designed with the weaker assumptions.

There are several ways to improve or extend the results of this paper. Some suggestions for future work include the following:

- Articulate the internal-glitch phenomena more clearly. This paper argues, mostly by examples, that the internal-glitch is a real problem. Our intution is that there is something even more fundamental about the internal-glitch phenomenon, something that the arguments in this paper have not quite articulated. One approach might be to find a clearer, or more useful, restatement of the atomic-gate assumption, and explain why the assumption is too strong (or alternatively, why the assumption is reasonable.)

- Show how to methodically generate circuits that avoid the internal-glitch phenomenon.

  The modified circuits shown in this paper (i.e, the **D**-element and the DME circuit) were generated using informal techniques. The informal approach used to design the circuits is to start with Martin's circuit and then use Dill's trace algorithm to look for internal-glitches. When an internal glitch is discovered, one figures out which order things needed to be done, and wire up an NRZ signal to enforce the correct order. Sometimes (e.g., for the double-C element), neither order is correct, and one needs to synthesize a new primitive. An open area of research is to formalize this informal process.

  Another approach to generating circuits free of internal-glitches is to transform CSP directly into correct circuits. There are at least two subproblems in such an approach:

  - Provide syntax directed rules for transforming CSP into circuits using NRZ signals.
  - Characterize the legal transformations that simplify circuits using NRZ signals into circuits using single-ended signals.

  The third, delay-sensitive, DME circuit was designed using finite-state-machine design techniques rather than by transforming a CSP program. Again, the techniques were informal, and could be formalized.

  The author found it very difficult to transform a finite-state-machine into a delay-insensitive circuit without using CSP as an intermediate representation. Is there a methodology to directly transform a finite-state-machine into an efficient delay-insensitive circuit?

- The new DME circuits should be verified by appropriate automatic verifiers. The delay-insensitive circuits can be modified using a straightforward adaptation of Dill's algorithms (where the primitives are specified to fail on internal-glitches). The delay-sensitive circuits may need new techniques. Seger [Seg89] showed some theoretical bounds on finding race conditions in the worst-case circuits. Is there a way to generate circuits which never hit the worst case? One of Seger's results was that as the assumptions about the circuits become stronger, it becomes harder to prove the worst-case correct circuits correct. This must be because the family of correct circuits becomes larger, because the circuits which could be verified under the weaker assumptions must automatically be correct under the stronger assumptions.

- Work out the details of the advantages of designing with stronger assumptions. I.e., from a mathematical perspective, how many gates can we save and what speed improvements can we achieve if we make stronger assumptions about the primitives.

- Provide a good abstraction and design methodology for exploiting relatively strong assumptions about delay-matching.

- A methodology could be developed to use a heirarchy of clocking strategies, each suitable for some particular level of abstraction or level in the physical packaging heirarchyof machines. Such a methodology should be able to exploit locality, and yet be insensitive to any unknown properties of the various delays in the system. One expects that gates on the same VLSI chip will have delays which are very well matched, while gates on different chips will not be well matched. This suggests that one should use a heirarchy of design methodologies roughly corresponding to the packaging heirarchy of the circuit; As the circuit becomes larger, the assumptions about the relative speed of various components becomes weaker. Thus one might assume very close transistor matches in the design of a single gate or standard-cell, e.g., allowing one to build atomic gates. At a slightly larger scale, one can weaken the assumptions by, for example, allowing for the gates to be slightly less well matched, and doing race-detection. For circuits spanning several VLSI chips, one can use delay-insensitive techniques which is based on even weaker assumptions.
- How should systems be clocked? This raises many questions, such as: What should be the role of self-timing in system design? What should be the role of synchronous clocking in system design?

References

[AG89] James H. Anderson and Mohamed G. Gouda. *A New Explanation of the Glitch Phenomenon.* Technical Report TR-88-23 (revised July, 1989), Department of Computer Sciences, The University of Texas at Austin, July 1989.

[BM88] Steven M. Burns and Alain J. Martin. Syntax-directed translation of concurrent programs into self-timed circuits. In *Advanced Research in VLSI. Proceedings of the Fifth MIT Conference,* 1988, pages 35–50. (A more detailed presentation can be found in Burns's master's thesis [Bur88].)

[Bur88] Steven M. Burns. *Automated Compilation of Concurrent Programs into Self-timed Circuits.* Technical Report Caltech-CS-TR-88-2, CALTECH, 1988. (Master's thesis..)

[Dil88a] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* MIT Press, 1988. (An ACM Distinguished Dissertation 1988.)

[Dil88b] David L. Dill. Trace theory for automatic hierarchical verification of speed-independent circuits. In *Advanced Research in VLSI. Proceedings of the Fifth MIT Conference,* 1988, pages 51–65. (This is a conference paper version of Dill's Ph.D. Thesis [Dil88a].)

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM,* 21(8):666–677, August 1978.

[KHC86] Kadaba R. Kakshmikumar, Robert A. Hadaway, and Miles A. Copeland. Characterization and modeling of mismatch in MOS transistors for precision analog design. *IEE Journal of Solid-State Circuits,* SC-21(6):1057–1066, December 1986.

[Mar85] Alain J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In *1985 Chapel Hill Conference on Very Large Scale Integration,* 1985, pages 245–260. (The resulting circuit has a bug, a corrected version appears in [Dil88b].)

[Mar86] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing,* 1(4):226–234, October 1986.

[MC80] Carver A. Mead and Lynn A. Conway. *Introduction to VLSI Systems.* Addison-Wesley, 1980.

[SCD*84] Stanley E. Schuster, Barbara Chappell, Victor Di Lonardo, and Peter E. Britton. A 20 ns 64K (4K $\times$ 16) NMOS RAM. *IEEE Journal of Solid-State Circuits,* SC-19(5):564–571, October 1984.

[Seg89] C.-J. Seger. The complexity of race detection in VLSI circuits. In *Advanced Research in VLSI. Proceedings of the 1989 Decennial Caltech Conference,* March 1989, pages 335–350.

[WE85] Neil Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective.* Addison-Wesley, 1985.