Technical Report

UCAM-CL-TR-126
ISSN 1476-2986

Number 126

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

Reasoning about the function and timing of integrated circuits with Prolog and temporal logic

M.E. Leeser

February 1988

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Reasoning about the Function and Timing of Integrated Circuits with Prolog and Temporal Logic

M. E. Leeser

February 12, 1988

Abstract

This article describes the application of formal methods to transistor level descriptions of circuits. Formal hardware verification uses techniques based on mathematical logic to formally prove that a circuit correctly implements its behavioral specification. In the approach described here, the structure of circuits and their functional behavior are described with Interval Temporal Logic. These specifications are expressed in Prolog, and the logical manipulations of the proof process are achieved with a Prolog system. To demonstrate the approach the behavior of several example circuits is derived from the behavior of their components down to the transistor level These examples include a dynamic latch which uses a 2-phase clocking scheme and exploits charge storage. Timing as well as functional aspects of behavior are derived, and constraints on the way a circuit interacts with its environment are reasoned about formally.

1 Introduction

The increasing complexity of integrated circuits has generated interest in new methods for the computer aided design of these circuits. One such method is formal hardware verification — using techniques based on mathematical logic to formally prove that a circuit correctly implements its behavioral specification. This approach was inspired by work by Gordon [15] and Barrow [1]. In general, the formal verification of a circuit proceeds as follows. A circuit implementation is described as the composition of circuit elements. Each occurrence of a circuit element is replaced by its behavioral description. The resulting description is then manipulated using rules of mathematical logic to show that the implementation logically implies the specification of the circuit behavior.

1.1 A Simple Example: a CMOS Inverter

* * * Put Figure 1 near here. * * *

As a simple example, consider the formal verification of the CMOS inverter shown in Figure 1. This circuit's behavior will be specified and verified using first-order logic. The first-order logic operators *not* ($\neg$), *and* ($\wedge$) and *or* ($\vee$) should be familiar from boolean algebra. In addition we use *implies* ($\supset$), *for all* ($\forall$) and *there exists* ($\exists$), which will be explained as we go along. Examples discussed in later sections will use Interval Temporal Logic, an extension of first-order logic with special logical operators for reasoning about time.

The behavior of the inverter is defined by a predicate which states that the inverter has two ports and the port labeled *Out* is equal to the logical inverse ($\neg$) of the port labeled *In*. Note that $\equiv_{\text{def}}$ is used for 'is defined by':

$$\mathsf{invert}(In, Out) \quad \equiv_{\mathsf{def}} \quad Out = \neg In$$

The structure of this circuit is described by the first-order logic predicate:

$$\mathsf{invert\text{-}struct}(In, Out) \quad \equiv_{\mathsf{def}}$$
$$\mathsf{ptrans}(In, 1, Out) \wedge \mathsf{ntrans}(In, 0, Out)$$

Here the predicates $\mathsf{ptrans}(G, C_1, C_2)$ and $\mathsf{ntrans}(G, C_1, C_2)$ represent p-type and n-type transistors respectively, where G is the gate node, and C_1 and C_2 are the channel nodes. The constants 1 and 0 specify connections to power and ground.

Formal verification consists of showing that invert-struct is equivalent to invert. As we shall see, this is done by transforming the structural description into a behavioral specification. In this case we show the two are equivalent, but this is not true in general. Frequently the implementation is more detailed than the specification, but we still wish to show that the behavior is correctly implemented. The more general correctness statement is:

$$\text{circuit structure} \supset \text{circuit behavior}$$

Logical implication is defined by:

$$A \supset B \quad \equiv_{\mathsf{def}} \quad \neg A \vee B$$

Note that $A = B$ is more restrictive since if A and B are equal then $A \supset B$, but the converse is not necessarily true.

The behavior of an n-type transistor is specified as an ideal switch whose channel nodes are connected when the gate has value 1. Similarly, the behavior of a p-type transistor is specified as an ideal switch whose channel nodes are connected when the gate has value 0.

$$\begin{array}{lll} \mathsf{ntrans}(G, C_1, C_2) & \equiv_{\mathsf{def}} & (G = 1 \supset (C_1 = C_2)) \\ \mathsf{ptrans}(G, C_1, C_2) & \equiv_{\mathsf{def}} & (G = 0 \supset (C_1 = C_2)) \end{array}$$

There are a few things to note about these behavioral specifications. First, these formulas do not specify the behavior of the transistors when the gates are inactive. In addition, no distinction is made between the two channel nodes.

Formal verification involves proving that the behavior of a circuit can be derived from its structure and the behavior of its components. The first step is to expand the structure of the circuit with the behavioral descriptions of the components in that structure. For complex circuits this may involve several levels of verification. For this simple example, there is only one level of design hierarchy. We proceed by expanding the transistor predicates by their behaviors in the specification of inverter structure:

$$\begin{array}{l} \mathsf{invert\text{-}struct}(In, Out) \quad \equiv_{\mathsf{def}} \\ \quad \mathsf{ptrans}(In, 1, Out) \wedge \mathsf{ntrans}(In, 0, Out) \equiv \\ \quad (In = 0 \supset (1 = Out)) \ \wedge \ (In = 1 \supset (0 = Out)) \end{array}$$

We can prove this by considering two cases: $In = 0$ and $In = 1$. This is called *case analysis*, and is analogous to simulating the circuit for all possible combinations of inputs. It is more efficient than simulation since all possible combinations of inputs are considered for simple components which have only a few inputs. Using case analysis we derive:

$$\begin{array}{l} \mathsf{invert\text{-}struct}(1, Out) = (0 = Out) \quad \text{and} \\ \mathsf{invert\text{-}struct}(0, Out) = (1 = Out) \end{array}$$

It then follows that:

$$\mathsf{invert\text{-}struct}(In, Out) = (Out = \neg In)$$

This is equivalent to the behavioral specification for the inverter stated at the beginning of this section, so we have verified the implementation of the circuit.

Even though this is a very simple example, it gives a flavor of what is involved in hardware verification. In addition, it can be used to illustrate the benefits of modularity in verification. Once an inverter has been proved correct, the derived behavior can be used wherever an inverter appears in the circuit specification. The inverter need not be formally verified again. For an inverter this is a small gain, but for more complex modules which are reused often, such as adders and counters, the gains are significant.

1.2 Formal Hardware Verification of More Complex Circuits

In general, verification is done hierarchically. The behavior of a circuit component is verified by checking that it is logically implied by the composition of the component behaviors. The verified behavior is then used as the description of the circuit component when that component is used in the design process. This is called *behavioral abstraction.* An advantage of hierarchical verification is that a circuit component needs to be verified only once, no matter how many times it is used in the circuit. A component is reused simply by *renaming* its ports to their connections. When components are composed hierarchically, internal lines are hidden. These variables are effectively *existentially quantified.* In addition, *constraints* on the way a circuit interacts with its environment may be described. These constraints are composed and reasoned about in a similar manner to behavioral specifications. When a circuit element is composed with others to form a larger circuit component, some of its ports may become inaccessible to other parts of the circuit. Constraints on these ports must be met. Other constraints which

are not met become constraints of the larger component.

We use Prolog to specify the implementation of circuits and Interval Temporal Logic (ITL) [26] to reason about the function and timing behavior of circuits. In this article we describe how different circuit elements are modeled, and use examples to show how these models are used in hierarchical verification. Composing circuit elements, renaming, hiding external lines, and abstracting the behavior of components are all used in these examples. For each example, constraints as well as behaviors are manipulated. These examples employ Prolog notation. Upper case letters denote Prolog variables. Lower case letters denote constants. The Prolog notation '_' is used for anonymous variables.

The circuits discussed are VLSI circuits specified down to the transistor level. A restricted class of CMOS circuits are considered. These are synchronous circuits where all feedback loops are broken by clocks and there are no branches in combinational logic which merge together again. As we shall see, many timing constraints are satisfied by this class of circuits. Note that the same class of circuits was considered by Lin and Mead [23].

Timing aspects as well as functional aspects of behavior are verified. This allows timing models to be related to higher level behavioral models using *temporal abstraction.* The same description of a circuit can be used for timing as well as other analysis, and different levels of description can be related formally. The designer has increased confidence that various aspects of the design are correct, since different tools all work on related representations of that design. Thus, we ensure that different levels of description actually describe the same circuit. We derive low level timing details such as delay and set-up and hold times from ITL descriptions of circuit components. The timing analysis is done in conjunction with behavioral verification. Delay, for example, depends on the function as well as the timing

characteristics of its components. In addition, this approach allows us to detect errors due to incorrect relations between signals and different clock phases. Such errors are not detected by other timing analysis techniques ([28], [20]), because they do not consider functionality.

1.3 Organization of this Paper

The remainder of this paper describes the different steps for proving a transistor level description of a circuit. A scenario for verifying a circuit which uses these steps is:

1. Specify the behavior in ITL.
2. Specify the circuit structure in Prolog.
3. Use the signal flow analysis algorithm to automatically derive direction of signal flow through the component.
4. Use the PALM system to prove the structure correctly implements the behavioral specification.

Note that this is only one possible scenario. Steps 1 and 2 may be reversed, or more frequently, iterated. In addition, the proof process may detect errors which affect the structure or cause the designer to reconsider the behavioral requirements.

The next section presents the use of Prolog to specify circuit structures, and describes the signal flow analysis algorithm. Section 3 gives a brief introduction to Interval Temporal Logic (ITL). In Section 4 we discuss how different types of circuit components are modeled with ITL and illustrate these models with examples. Then a more complex example which uses 2-phase clocking is presented. These

examples were derived using the PALM (Prolog Assistant for Logic Manipulation) system [22] which was developed to mechanize these manipulations. The article concludes with a discussion of the work presented, related work and conclusions.

2 Prolog for Circuit Specification and Manipulation

Circuits can be specified, simulated and reasoned about using logic programming, which is based on first-order logic. We use the logic programming language Prolog [7] to represent the structure of circuits and to reason about these structures. These Prolog descriptions directly reflect the structure and hierarchy of a circuit as shown in a circuit schematic. In this section, we show how circuits are specified in Prolog and discuss how these specifications can be manipulated.

A *circuit* is an interconnected set of *components*. Components can be composed hierarchically, where components are specified in terms of constituent components. At the bottom of the hierarchy are *primitive* components. We describe CMOS circuits; the primitive components are *n*-type and *p*-type transistors, and power and ground sources. There are no strict rules about the levels of hierarchy. A component may be made up of primitive as well as non-primitive components. Each component has *ports* for external connections. A port may be an *input*, an *output*, or *bidirectional*. A *node* is a junction of ports. Nodes which are *external* to a component are formed by connecting one of the ports of that component to one or more ports of other components. Nodes which are *internal* to a component are formed by connections of the ports of the constituents of that component.

The structure of the CMOS inverter presented in the previous section was defined using first-order logic:

$$\textsf{invert-struct}(In, Out) \equiv_{\textsf{def}} \textsf{ptrans}(In, 1, Out) \wedge \textsf{ntrans}(In, 0, Out)$$

The equivalent Prolog description is:

```
invert-struct(In,Out) :- ptrans(In,1,Out),ntrans(In,0,Out).
```

This is the definitional method for specifying circuits in Prolog. This and other methods are described and compared in [5]. In this method, a circuit is represented as a set of Horn clauses which are a subset of the formulas of first-order logic. A component with n ports is represented as a predicate with n arguments whose left-hand side represents the component being defined. The body of the predicate is a composition of the constituent components which define the component. Constituents are composed with the comma connective. The order of the components in the body is not important. The ':-' connective of Prolog is reinterpreted to mean 'is defined by'. A node is represented by a unique variable name. A node which is named by a variable not appearing in the left-hand side of the clause is an internal node.

* * * Put Figure 2 near here. * * *

Figure 2 shows a dynamic latch. Its structural description in Prolog is:

```
dlatch-struct(L,D,Q,Φ1,Φ2) :-
    invert-mux(L,D,Q,Φ1,Y),
    shiftstage(Y,Q,Φ2).
```

Where the structures of invert-mux and shiftstage are specified by:

invert-mux-struct(A, B, C, Φ, D) :-
    two-one-mux(A, B, C, X, Φ),
    invert(X, D).

shiftstage-struct(A, B, Φ) :-
    trans-gate(Phi, A, C),
    invert(C, B).

The structure of a clocked two-to-one multiplexer, shown in Figure 9, is specified as:

two-one-mux(G, A, B, X, Φ) :-
    trans-gate$((G \wedge Phi), A, X, M)$,
    trans-gate$((\neg(G) \wedge Phi), B, X, M)$.

There are several things to note about this example. First, the ports of these components do not have direction specified. As we shall see later, port directions can be determined automatically and need not be specified by the user. Second, in the top level specification of dlatch-struct, the variable Y defines the 'hidden' node between the output of the invert-mux, and the input of the shiftstage. This variable does not appear in the left-hand side of the specification for dlatch-struct. In addition, there is no rigid definition of different levels of hierarchy. For example, the definition of shiftstage-struct mixes transmission gates and inverters. This example uses many levels of hierarchy to describe a simple circuit component. This was not necessary; the dynamic latch structure could have been specified with one level of hierarchy where all components were transistors. This was not done for several reasons. The example illustrates the use of hierarchy in circuit description. Using hierarchical descriptions makes it easier to prove behavior correct. In addition, the proofs of several of the components have been used in deriving the

behavior of other components. For example, the shiftstage is also a component of a shift register.

Specifying circuits in this manner has several advantages. The descriptions directly reflect the structure and hierarchy of a circuit as shown in a schematic, and are therefore easy to write. These descriptions also lend themselves to easy modular specification for several reasons. The component name is explicitly part of the specification. Internal connections are named by variables which do not appear in the left-hand side of the clause and are effectively hidden from users of the structure. In addition, the definition of ports of components is inherently non-directional. This is important for specifying components, such as pass transistors and transmission gates, which have bidirectional ports.

Another advantage of this approach is that the specifications can be manipulated or directly executed by Prolog systems. Clocksin [5] describes tools for manipulating these Prolog specifications. These tools include symbolic simulation of circuits by direct execution of their descriptions, gate assignment, circuit rewriting and component specialization. The Advanced Silicon Compiler Project at the University of California at Berkeley also uses Prolog descriptions of circuits at the transistor level and above. Their tools for manipulating these descriptions include the Prolog Timing Analyzer [30], which calculates the delays of all nodes using a lumped RC delay model, and a simulated annealing algorithm to assign sizes to transistors in a VLSI schematic [9].

Another tool we have written for manipulating Prolog descriptions of circuits determines the signal flow through a network of transistors [6]. Signal flow analysis is used to determine the direction of all transistors in a circuit, and the directions of ports of components of a circuit. These directions are used when deriving the behavior of circuits. As we shall see, the Interval Temporal Logic description of

transistor behavior assumes that transistor direction is assigned. The signal flow analyzer has been used with all the examples presented in this paper. The result of signal flow analysis for the dynamic latch is shown in Figure 3.

* * * Put Figure 3 near here. * * *

3 Interval Temporal Logic

Temporal logic is a formalism for reasoning about time which has been used for specifying and proving properties of software and hardware. There are several ways of reasoning about time with logic. Classical temporal logic adds operators for reasoning about time to the logic operators introduced in Section 1.1. Interval Temporal Logic (ITL), due to Moszkowski [26], is a development of classical linear time temporal logic which adds operators for reasoning about intervals of time. In this section, the subset of ITL which we use to specify and reason about the function and timing of hardware is presented. The operators presented in this section are summarized in Appendix A.

An ITL formula describes behavior on an interval of time. ITL operators include $\Box$ (always), $\bigcirc$ (next), and ; (chop). An intuitive description of these operators follows.

* * * Put Figure 4 near here. * * *

An *interval* is a non-empty sequence of states. (The notation $\langle\rangle$ is used to delimit intervals.) A *state* can be viewed as an instantaneous snapshot of a system. The *length* of an interval is one less than the number of states it contains. An interval of length 0 is called an *empty* interval; it has one state and is just an instant

of time. A unit interval has length one. Throughout this article, we assume all unit intervals have the same duration. For example, consider Figure 4, which might be a voltage waveform sampled at discrete instances of time. In Figure 4, $\langle s_0, s_1, s_2, s_3 \rangle$ and $\langle s_2 \rangle$ are intervals. The state $\langle s_2 \rangle$ is an *empty* interval.

Formulas are true or false with respect to an interval. A formula w which contains no temporal operators is true on an interval if it is true in the first state of that interval. In Figure 4, the formula $X = 0$ is true on the interval $\langle s_0, s_1, s_2, s_3 \rangle$.

$\Box w$ (always w) is true on $\langle s_0, s_1, \ldots s_n \rangle$ if w is true in every subinterval which finishes in the final state. In Figure 4, $\Box X = 1$ is true on the interval $\langle s_1, s_2, s_3 \rangle$. It is also true on several other intervals including $\langle s_1, s_2 \rangle$ and $\langle s_2, s_3 \rangle$.

Similarly, $\bigcirc w$ (next w) where w is a formula, is true on an interval $\langle s_0, s_1, \ldots, s_n \rangle$, if w is true on the sub-interval $\langle s_1, \ldots, s_n \rangle$. In Figure 4, the formula $\bigcirc X = 1$ is true on the interval $\langle s_0, s_1, s_2, s_3 \rangle$.

The chop operator (;) allows an interval to be broken in two. The formula $w_1 \,;\, w_2$ can be read as w_1 *followed by* w_2. $w_1 \,;\, w_2$ is true on an interval $\langle s_0, s_1, \ldots, s_n \rangle$ if there exists an intermediate state s_i where both w_1 and w_2 are true, w_1 is true on the interval $\langle s_0, s_1, \ldots, s_i \rangle$, and w_2 is true on the subinterval $\langle s_i, \ldots, s_n \rangle$.

The formula *skip* is true of any interval of length one. It is often necessary to use skip in formulas employing the chop operator. For example, the signal X in Figure 4 rises from 0 to 1. The temporal logic formula

$$(X = 0); skip; (X = 1)$$

is true on the interval $\langle s_0, s_1, s_2, s_3 \rangle$. The chop operator splits an interval into two sub-intervals which share a common state. It is impossible for the signal X

to be both 0 and 1 at the same time, so it is necessary to use *skip* to represent the interval of length one where X is changing. Here *skip* represents the interval $\langle s_0, s_1 \rangle$ which has length one.

Temporal operators for expressing such concepts as temporal equality and delay can be built out of formulas using the operators described above. These and other operators are defined in Appendix A. Formal syntax and semantics of ITL are presented in [26]. ITL constructs will be explained when they are used in the rest of this article.

Most variables in these logical descriptions are *signals*. Moszkowski [27] describes delayless combinational elements and memory elements; his signals are boolean values. We describe switch-level models of transistor circuits, which depend on capacitive strength as well as logical value. Thus Moszkowski's definition of signals have been extended, as have the logical and temporal operators which operate on them. Signals are $\{value, strength\}$ pairs. Each of the fields *value* and *strength* can have value 0 or 1. The *value* field represents the boolean logic value of a signal. The *strength* field represents the strength of a signal which may be either capacitive (0) or driven (1). The use of the strength field is illustrated in the next section.

All the logical operators described so far examine the value field of a signal only. For example, $A \wedge B$ is true if the value of A is 1 and the value of B is 1. This applies to the operator ' = ' as well. The symbol ' $\Leftrightarrow$ ' is used to denote equivalence of signals. $A \Leftrightarrow B$ if and only if the value fields and the strength fields of A and B are equal. Thus the following statements are true:

$$\{0,1\} = \{0,0\}$$
$$\{0,1\} \Leftrightarrow \{0,1\}$$

The functions weaken and strengthen operate only on the strength field of a signal.

The weaken function takes a signal and returns a signal with the same value field whose strength is 0. The strengthen function is described similarly.

Signals are combined at a node using the join operator ($\sqcup$). The result of joining two signals of different strength is a signal with the same value and strength as the stronger signal. The result of joining two signals with the same value and strength is a signal with equal value and strength. Note that joining two signals with different value but the same strength results in an error.

Temporal assignment ($\rightarrow$) allows one signal at the beginning of an interval to be assigned to another signal at the end of the interval. More than one signal may be assigned to another signal. When this occurs, the join of the two signals is assigned to the resulting signal. This can be described in ITL:

$$((A \rightarrow B) \wedge (C \rightarrow B) \supset (A \sqcup C \rightarrow B))$$

This property of the join operator implies that the value of a node can always be changed by another signal contributing to that node. Eventually a node becomes hidden in a circuit description, so the number of signals which contribute to that node is limited. In essence, we are combining the join function with temporal assignment. Others ([21], [10]) have used explicit join components in their circuit representations.

4 Using ITL to Reason about the Behavior of Circuits

In this section we show how ITL is used to model the behavior of circuits, and show with examples how these models are manipulated. First, the models for the behavior of circuit elements are discussed. These include combinational elements,

transistors, memory elements and clocks. The models describe two types of information: the dependence of outputs on inputs (i. e. function and timing), and constraints on the inputs. Constraints specify the way the circuit model interacts with its environment. Constraints include such timing information as set-up and hold times, and are described and reasoned about with the same logic as that used for the timing and functional behavior of circuits. The implicit form of the correctness statement for the behavior of a circuit is:

$$\begin{array}{l}\text{constraints on inputs} \wedge \text{circuit structure} \supset \\ \qquad \text{circuit behavior} \wedge \text{constraints on outputs}\end{array}$$

Note that the term *gate* is used sometimes to refer to the electrode of a transistor and sometimes to refer to a combinational element. The term's meaning should be clear from context.

4.1 Modeling Combinational Elements

Our main objective is to model timing as well as functional behavior of circuit elements. The timing behavior of all combinational elements is modeled as delay. Breuer and Friedman [2] present many hardware delay models, and Moszkowski [26] shows how several of these can be expressed in ITL. We use Breuer and Friedman's transport delay, which Moszkowski has defined in ITL. The definition of delay states that signal B is signal A delayed by m units of time. *len m* is true on any interval of length exactly m.

$$A \operatorname{del}^m B \quad \equiv_{\text{def}} \quad \Box \mathit{len}\ m \supset (A \rightarrow B)$$

* * * Put Figure 5 near here. * * *

Combinational elements with several inputs $(A, B, \ldots)$ and one output (X) have the general form:

$$\mathrm{f}(A, B, \ldots)\ \mathrm{del}^m\ X$$

Here $\mathrm{f}(A, B, \ldots)$ is a function of the inputs which contains no temporal operators. For example, the 3-input and gate shown in Figure 5 is described by:

$$(A \wedge B \wedge C)\ \mathrm{del}^m\ X$$

The delay is viewed as being lumped at the output. Note that m represents the worst case delay from inputs to output for this gate.

When a cell has more than one output we use one predicate for each output and compose these to produce the description of the cell. For example, a cell with outputs $X, Y, \ldots$ is described by:

$$\mathrm{f}(A, B, \ldots)\ \mathrm{del}^m\ X \wedge$$
$$\mathrm{g}(A, B, \ldots)\ \mathrm{del}^n\ Y \wedge \ldots$$

ITL transport delay has several useful properties. For example, it is cumulative:

$$A\ \mathrm{del}^m\ B \wedge B\ \mathrm{del}^n\ C \supset A\ \mathrm{del}^{m+n}\ C$$

Here the output is the delay of an input signal. The output may also receive a delayed *function* of the inputs. Functional composition applies:

$$\mathrm{f}(A)\ \mathrm{del}^m\ B \wedge \mathrm{g}(B)\ \mathrm{del}^n\ C \supset \mathrm{g}(\mathrm{f}(A))\ \mathrm{del}^{m+n}\ C$$

This delay model is valid subject to the constraint that the inputs are well-behaved. A signal is defined as well-behaved if it does not glitch. A glitch is a pulse of duration less than g, where g is a characteristic of the technology being used. This property is also expressed in ITL, where g is a global constant:

$$\begin{aligned}&\textsf{well-behaved}(A) \quad \equiv_{\textsf{def}} \\ &\qquad \Box(\uparrow\downarrow A \supset \mathit{len} > g) \wedge \\ &\qquad \Box(\downarrow\uparrow A \supset \mathit{len} > g)\end{aligned}$$

$\uparrow\downarrow A$ and $\downarrow\uparrow A$ are ITL predicates which are true of intervals which contain a rising pulse and a falling pulse, respectively:

$$\begin{array}{lll}\uparrow\downarrow A & \equiv_{\textsf{def}} & (A \approx \{0,\_\})\ ;\ \mathit{skip}\ ;(A \approx \{1,\_\})\ ;\ \mathit{skip}\ ;\ (A \approx \{0,\_\}) \\ \downarrow\uparrow A & \equiv_{\textsf{def}} & (A \approx \{1,\_\})\ ;\ \mathit{skip}\ ;(A \approx \{0,\_\})\ ;\ \mathit{skip}\ ;\ (A \approx \{1,\_\})\end{array}$$

Here temporal equality ($\approx$) is used to denote that a signal has a certain value over an interval. For example, $A \approx \{1,\_\}$ is true of an interval if signal A is equal to $\{1,\_\}$ in every state of that interval.

Transport delay transfers this **well-behaved** property from its inputs to its output, with a time shift:

$$\textsf{well-behaved}(A) \ \wedge\ A\ \textsf{del}^m\ B \quad \supset \quad \bigcirc^m(\textsf{well-behaved}(B))$$

Here $\bigcirc^m$ is an abbreviation for m $\bigcirc$ operators applied in sequence.

4.2 Modeling Transistors

Switch-level models [4] are used to describe circuit primitives: n-type and p-type transistors. Previously we defined a bidirectional model without delay for these components. From now on, we will only present models for n-type transistors. The model for the complementary p-type transistors are analogous.

The simple switch-level model can be extended with timing information. In order to do this we must also specify directionality. Directions of transistors are derived using the Prolog program described in [6]. When the gate of the transistor is on,

the signal flows from source to drain. Source and drain nodes are not symmetrical. If the model is constrained such that the source is stable when the gate changes, the delay, m, from source to drain can be expressed in ITL:

$$\textsf{ntrans}(G, S, D, m) \quad \equiv_{\text{def}} \quad \Box(G = \{1, \_\}) \wedge \mathit{len}\ m \supset (S \rightarrow D)$$

The '_' is a Prolog anonymous variable. Since the '=' operator only examines the value field of the gate signal, it does not matter what the strength field is.

This model is subject to the constraints:

$$\Box(G \approx \{1, \_\} \supset \mathit{stb}\ S) \wedge$$
$$\textsf{well-behaved}(G)$$

4.3 The CMOS Inverter Example Revisited

We will now reconsider the CMOS inverter presented at the start of this article. Here delay and capacitive strength will be modeled. The inverter behavior is specified in ITL by:

$$\textsf{invert}(\mathit{In}, \mathit{Out}, m) \quad \equiv_{\text{def}} \quad \Box \mathit{len}\ m \supset ((\textsf{strengthen}\ \neg \mathit{In}) \rightarrow \mathit{Out}) \qquad (1)$$

Here **strengthen** specifies that the output of the inverter is always driven. This is true whether or not the input is driven. Note that Prolog variables are used for the node names. When this behavior is used as the definition of a component of a cell, Prolog makes a copy of this formula and replaces these variables by the names of the wires to which this particular instance of a gate is connected.

The implementation of the inverter is specified in ITL by:

$$\textsf{invert-struct}(In, Out, m) \equiv_{\textsf{def}} \textsf{ntrans}(In, \{0,1\}, Out, m) \wedge \textsf{ptrans}(In, \{1,1\}, Out, m)$$

Here $\{0,1\}$ represents a connection to ground and $\{1,1\}$ represents a connection to power or V_{dd}. For simplicity, we assume equivalent delay on the two transistors.

We proceed by proving invert-struct correctly implements invert. The outline of the derivation of behavior for the inverter is given below. First we derive the behavior of the inverter, then we derive the constraints.

4.3.1 Deriving Behavior

The first step is to replace the parts ntrans and ptrans by their behavioral specifications:

$$\begin{array}{l}\textsf{ntrans}(In, \{0,1\}, Out, m) \wedge \textsf{ptrans}(In, \{1,1\}, Out, m) = \\ (\Box(In = \{1, \_\}) \wedge len\ m \supset (\{0,1\} \rightarrow Out)) \wedge \\ (\Box(In = \{0, \_\}) \wedge len\ m \supset (\{1,1\} \rightarrow Out))\end{array}$$

Next, using the inverse distributive property of $\Box$: $(\Box a) \wedge (\Box b) = \Box a \wedge b$:

$$\begin{array}{l}\Box(In = \{1, \_\}) \wedge len\ m \supset (\{0,1\} \rightarrow Out) \wedge \\ \quad (In = \{0, \_\}) \wedge len\ m \supset (\{1,1\} \rightarrow Out)\end{array}$$

Using the associative property of $\wedge$ to rearrange the order of the $len\ m$ and $In = X$ statements, applying the rule $(a \wedge b \supset c) = (a \supset (b \supset c))$ twice, and applying the rule $(a \supset b) \wedge (a \supset c) = (a \supset (b \wedge c))$ yields:

$$\begin{array}{l}\Box len\ m \supset (In = \{0, \_\} \supset (\{1,1\} \rightarrow Out)) \wedge \\ \qquad\qquad (In = \{1, \_\} \supset (\{0,1\} \rightarrow Out))\end{array}$$

The next step is to use case analysis on In. Since In is a signal, it has four possible values corresponding to the cross product of the boolean value and strength fields. Only the boolean value field is examined because of the definition of '=', so these four values are covered by the cases: $[\{0,\_\},\{1,\_\}]$. Case 1: $In = \{0,\_\}$. We replace $\{0,\_\} = \{0,\_\}$ with *true* and $\{0,\_\} = \{1,\_\}$ with *false*, and apply the rules $(false \supset X) = true$, $(true \supset Y) = Y$ and $Z \wedge true = Z$, where X, Y, and Z stand for arbitrary logical formulas. This results in:

$$\Box\, In = \{0,\_\} \supset (len\ m \supset (\{1,1\} \rightarrow Out))$$

This is the case when $In = \{0,\_\}$, so $\{1,1\}$ can be replaced with strengthen $\neg In$:

$$\Box\, In = \{0,\_\} \supset (len\ m \supset ((\text{strengthen } \neg In) \rightarrow Out))$$

Similarly for the case $In = \{1,\_\}$:

$$\Box\, In = \{1,\_\} \supset (len\ m \supset ((\text{strengthen } \neg In) \rightarrow Out))$$

Combining the cases results in the behavioral description:

$$\Box len\ m \supset ((\text{strengthen } \neg In) \rightarrow Out)$$

This equation is the same as the definition for behavior given in formula 1. We have shown that the specified behavior for the inverter can be deduced from its structure and the behavior of its components. Note that we have verified both functional and timing aspects of behavior.

4.3.2 Deriving Constraints

Constraints are reasoned about in a similar way to behavior. The constraints for the inverter are derived by composing the constraints of the components. In the case of the inverter, the constraints which arise from the *n*-type and the *p*-type transistors are simply *and*ed together (and the duplicate predicate well-behaved is removed):

$$\Box(In \approx \{0, \_\} \supset stb\{1,1\}) \wedge$$
$$\Box(In \approx \{1, \_\} \supset stb\{0,1\}) \wedge$$
$$\text{well-behaved}(In)$$

The formulas $stb\{1,1\}$ and $stb\{0,1\}$ are always true since constants are always stable. This conjunction therefore reduces to the single constraint for the inverter:

$$\text{well-behaved}(In)$$

4.4 Pass Transistors and Transmission Gates

The circuits that we are interested in include steering elements as well as combinational elements. In steering logic, the steering element is used as a switch to conditionally connect two nodes together. Steering logic is appropriate when the logic function can be conceptualized as signals being conditionally steered through a network. Steering elements include nMOS pass transistors and CMOS transmission gates. A pass transistor has neither channel node connected directly to power or ground. In CMOS a *p*-type and an *n*-type transistor are connected with common source and drain connections to form a transmission gate as shown in Figure 6.

* * * Put Figure 6 near here. * * *

Whenever the *n*-type transistor of the transmission gate is on, the *p*-type transistor is on as well. Due to threshold drops, the transmission of a logical '1' is degraded when passed through an *n*-type device, while the transmission of a logical '0' is degraded when passed through a *p*-type device. Both types of devices are used in a transmission gate so that both 0's and 1's can be transmitted without degradation. Our model for a transistor does not represent threshold effects. In this model, a transmission gate behaves equivalently to an *n*-type pass transistor and is treated as such. The *p*-type transistor in this model is redundant. The model used is therefore the same as the *n*-type transistor model presented earlier. It is common in switch-level models ([4], [32]) not to represent threshold drops.

4.5 Modeling Charge Storage

The transistor model used above does not specify the behavior of the output node when the transistor is off. Because of the capacitance in MOS circuits, nodes which become isolated as a result of transistors turning off retain their previous driven value for as long as several milliseconds. This phenomenon is known as charge storage. There are many sources of capacitance in a MOS circuit including capacitance due to interconnect and capacitance on the gate and channel nodes of transistors. Charge storage is frequently exploited in MOS circuit design. Circuits which make use of this phenomenon include the CMOS dynamic latch in Figure 2. The charge storage in this circuit is largely due to the input capacitance of the inverters. This is due in turn to the gate capacitance of the transistors which make up the inverter.

Pass transistors and transmission gates are modeled using the temporal logic equations presented in the previous section. We model charge storage by connecting an explicit 'ideal capacitance' to the drain node of the transistor as shown in Figure 7.

* * * Put Figure 7 near here. * * *

This capacitance is largely due to the gates of the transistors being driven. However, the effects we are interested in modeling only influence the circuit when a node is isolated. This occurs only when a switch is open. Therefore, we model this capacitance on the drain of a transistor. We assume the drain drives a capacitive load. In fact, this load usually arises from the capacitance on the gates of the transistors being driven.

The capacitor is described in ITL by:

$$\mathsf{cap}(D) \quad \equiv_{\mathsf{def}} \quad \Box \mathit{len}\ 1 \supset (\mathsf{weaken}\ D \rightarrow D)$$

This ITL formula states that for any interval of length 1, the signal D at the end of the interval is a weakened version of the signal D at the beginning of the interval. The two versions of D have the same value. In other words, a capacitor connected to node D always retains its last logical value, but weakens the strength of the signal. The complete specification of an *n*-type transistor with capacitance is:

$$\Box((G = \{1, \_\}) \wedge \mathit{len}\ m \supset (S \rightarrow D)) \wedge$$
$$\Box \mathit{len}\ 1 \supset (\mathsf{weaken}\ D \rightarrow D)$$

This states that m time units after the transistor turns on, node D is driven. If the transistor is on and S is a strong signal, then this overrides the memory due to the capacitor. When the transistor is off, the memory due to the capacitor retains the last driven value on the node. Note that the overriding effect is due to the property of temporal assignment discussed in Section 3.

This capacitance is *ideal* for two reasons. First, no capacitive value is assigned to it. Two ideal capacitances on a node have the same effect as one capacitance on the node:

$$\mathsf{cap}(D) \wedge \mathsf{cap}(D) = \mathsf{cap}(D)$$

Second, we assume that charge never leaks away. This assumption is valid provided that the charge gets refreshed at least once every r units of time, where r is a global constant of the technology. This in turn occurs if the transistor connected to the node turns on at least once every r units of time. Expressing that node G turns on at least once every r units of time in ITL:

$$\Box(\downarrow\uparrow G) \supset (\mathit{len} < r)$$

Note that we assume G falls and rises at least once.

A transistor with a capacitor connected to its drain thus has two constraints on the signal at its gate node: the constraint above, and the requirement expressed by the well-behaved predicate. These constraints are combined in a single predicate, called control:

$$\begin{aligned}\mathsf{control}(G) = {} & \mathsf{well\text{-}behaved}(G) \wedge \\ & \Box(\downarrow\uparrow G) \supset (\mathit{len} < r)\end{aligned}$$

In summary, the behavior of an n-type transistor with gate source and drain nodes labeled G, S and D respectively, with delay m from source to drain, and with explicit capacitance on the drain node is specified:

$$\begin{aligned}&\Box((G = \{1, \_\} \wedge \mathit{len}\ m) \supset (S \rightarrow D)) \wedge \\ &\Box \mathit{len}\ 1 \supset (\mathsf{weaken}\ D \rightarrow D)\end{aligned}$$

The constraints on this behavior are:

$$\begin{aligned}&\mathsf{control}(G) \wedge \\ &\Box G \approx \{1, \_\} \supset \mathit{stb}\ S\end{aligned}$$

This is the complete transistor model we will use.

From these constraints, it can be inferred that the drain node is well-behaved. Note that the capacitance is modeled on the drain node of every transistor. The examples presented in this article have been simplified so only capacitance which is significant is considered. For example, the capacitance on the output node of an inverter is not considered because that node is always driven.

4.6 Clocking

In this section we consider more complex clocked circuits. Various clocking strategies for CMOS circuits are discussed in Weste and Eshraghian [33]. The circuits presented here employ a 2-phase clocking strategy with non-overlapping clocks. This is referred to as pseudo 2-phase clocking [33] because in reality four different clock phases are used by the circuit. These phases $(\Phi_1, \bar{\Phi}_1, \Phi_2, \bar{\Phi}_2)$ are shown in Figure 8.

* * * Put Figure 8 near here. * * *

Φ_1 and Φ_2 are non-overlapping; it is never the case that Φ_1 and Φ_2 are both high at the same time. This is expressed in ITL by:

$$\begin{array}{l} \Box(\Phi_1 \approx \{1,1\} \supset (\Phi_2 \approx \{0,1\})) \wedge \\ \Box(\Phi_2 \approx \{1,1\} \supset (\Phi_1 \approx \{0,1\})) \end{array}$$

For simplicity we will assume that $\bar{\Phi}_1$ is always equal to $\neg\Phi_1$, and similarly for Φ_2.

In the class of circuits considered, clocks are used to control transmission gates. Thus Φ_1 and Φ_2 must satisfy the requirements of the control predicate:

$\text{control}(\Phi_1) \wedge \text{control}(\Phi_2)$

These properties of clock signals need to be established only once. Whenever a transistor is encountered which is gated by a clock, it can be assumed that the control constraint on that transistor is automatically met. In the restricted class of circuits under consideration, pass transistors are gated either by clocks, or by signals *and*ed with clocks. In the latter case, the control constraint simplifies to establishing that the signal is stable when the clock is true.

4.7 Example 2: a Clocked 2-to-1 Multiplexer

* * * Put Figure 9 near here. * * *

In this section we derive the behavior and constraints of the CMOS clocked 2-to-1 multiplexer shown in Figure 9. In the next section, the behavior of the multiplexer will be used to derive the behavior of the dynamic latch. The clocked multiplexer behaves like a 2-to-1 multiplexer when the clock (Φ) is high, and retains its previous value when the clock is low. This example makes use of charge storage and clocking models discussed above.

4.7.1 Deriving Behavior

The behavior of the multiplexer is described in ITL as:

$$\begin{aligned}&\text{two-one-mux}(G, A, B, X, \Phi, m) \quad \equiv_{\text{def}} \\ &\quad (\Box(\Phi = \{1,1\}) \wedge \mathit{len}\ m \supset \\ &\qquad (\mathit{if}\ G = \{1, \_\}\ \mathit{then}\ (A \rightarrow X)\ \mathit{else}\ (B \rightarrow X))) \wedge \\ &\quad (\Box \mathit{len}\ 1 \supset (\mathit{weaken}\ X \rightarrow X))\end{aligned}$$

This behavior definition uses the conditional:

$$\textit{if } a \textit{ then } b \textit{ else } c \quad \equiv_{\text{def}} \quad (a \supset b) \wedge (\neg a \supset c)$$

An outline of the derivation of behavior for the multiplexer follows. As usual, it begins with the description of the implementation of the circuit.

$$\begin{aligned}&\textsf{two-one-mux-struct}(G, A, B, X, \Phi, m) \quad \equiv_{\text{def}} \\ &\qquad \textsf{trans-gate}((G \wedge \Phi), A, X, m) \wedge \\ &\qquad \textsf{cap}(X) \wedge \\ &\qquad \textsf{trans-gate}((\neg G \wedge \Phi), A, X, m) \wedge \\ &\qquad \textsf{cap}(X)\end{aligned}$$

Next, the components can be replaced by their definitions. Note that the model for a transmission gate is the same as the model for an *n*-type transistor. One of the capacitances is removed since $\textsf{cap}(X) \wedge \textsf{cap}(X) = \textsf{cap}(X)$:

$$\begin{aligned}&\Box(((G \wedge \Phi) = \{1,1\}) \wedge \textit{len } m \supset (A \rightarrow X)) \wedge \\ &\Box(((\neg G \wedge \Phi) = \{1,1\}) \wedge \textit{len } m \supset (B \rightarrow X)) \wedge \\ &\Box \textit{len } 1 \supset (\textit{weaken } X \rightarrow X)\end{aligned}$$

Using straightforward manipulations of logical formulas, the first two conjuncts are manipulated to the form:

$$\begin{aligned}&\Box(\Phi = \{1,1\}) \wedge \textit{len } m \supset \\ &\qquad (G = \{1, \_\} \supset (A \rightarrow X)) \wedge \\ &\qquad (\neg(G = \{1, \_\}) \supset (B \rightarrow X))\end{aligned}$$

Using the definition of the conditional given above, and *and*ing with the capacitive behavior, results in the behavior for the multiplexer given at the start of this section.

4.7.2 Deriving Constraints

The constraints for the multiplexer are derived by composing the constraints of the parts. The constraints of a transmission gate are the same as those for an *n*-type transistor. The constraints of the parts are:

1. control$(G \wedge \Phi)$
2. $\Box((G \wedge \Phi) \approx \{1, \_\}) \supset stb\ A$
3. control$(\neg G \wedge \Phi)$
4. $\Box((\neg G \wedge \Phi) \approx \{1, \_\}) \supset stb\ B$

These constraints simplify under the assumptions about clocking. It is assumed that any clock, Φ, in the system meets the requirement control(Φ). The first and third constraints can be simplified. In fact, We only need to show that the gate signals are well-behaved, and that the output node X is driven often enough. These requirements are met by the stronger constraint $\Box\,\Phi \supset stb(G)$. This is stronger since it can be shown that:

$$
\begin{aligned}
&(\Box(\Phi \supset stb\ G) \wedge \text{control}(\Phi)) \quad \supset \\
&\qquad \text{well-behaved}(G \wedge \Phi) \wedge \\
&\qquad \text{well-behaved}(\neg G \wedge \Phi)
\end{aligned}
$$

This stronger constraint states that if G is stable whenever Φ is true, then the resulting signal is well-behaved. We still need to show that X is driven often enough. This is also a direct consequence of the stronger constraint since, when Φ is true either G or $\neg G$ is true. Thus, the output is driven once every clock cycle. The remaining two constraints, constraints 2 and 4 above, cannot be resolved until the component is used in a larger circuit. Thus the constraints for this device are:

$$
\begin{aligned}
&\text{two-one-mux-constraints}(G, A, B, X, \Phi) \quad \equiv_{\text{def}} \\
&\qquad (\Box\,\Phi \supset stb\ G) \wedge \\
&\qquad (\Box((G \wedge \Phi) \approx \{1, \_\}) \supset stb\ A) \wedge \\
&\qquad \Box((\neg G \wedge \Phi) \approx \{1, \_\}) \supset stb\ B
\end{aligned}
$$

5 Components which Use 2-phase Clocking

In the previous section, we discussed how clocking behavior can be described in ITL, and presented an example which uses one phase of a 2-phase clocking scheme. In this section we discuss how components which are clocked on different clock phases are composed. Weste and Eshraghian [33] describe the operation of different memory structures which employ pseudo 2-phase clocking. One of these structures is the dynamic latch which we discuss below.

* * * Put Figure 10 near here. * * *

A generalized element using a 2-phase clocking scheme is shown in Figure 10. $func1(A)$ is evaluated when Φ_1 is high, and stored by the capacitor connected to node B when Φ_1 is low. Similarly, $func2(B)$ is evaluated when Φ_2 is high, and the result is available at node C when Φ_2 is low. Remember that the clock cycle is a continuous cycle of Φ_1 high and Φ_2 low, followed by both clocks low, followed by Φ_1 low and Φ_2 high followed by both clocks low. Thus, when $func2(B)$ is evaluated, B is storing the previously evaluated version of $func1(A)$. Also note that, in a correctly operating circuit, Φ_1 and Φ_2 are never both high at the same time. This behavior is captured in ITL by the following rule:

$$\begin{aligned}
&\Box(((\Phi_1 = \{1,1\}) \wedge (\Phi_2 = \{0,1\}) \supset (len\ M \supset (func1(A) \rightarrow B))) \wedge \\
&\quad ((\Phi_1 = \{0,1\}) \wedge (\Phi_2 = \{0,1\}) \supset stb\ B) \wedge \\
&\quad ((\Phi_1 = \{0,1\}) \wedge (\Phi_2 = \{1,1\}) \supset (len\ N \supset (func2(B) \rightarrow C)))) \quad \supset \\
&\Box((\Phi_2 = \{1,1\}) \wedge lenN \supset func2(func1(\ \mathrm{latched}(\Phi_1, A)) \rightarrow C)
\end{aligned}$$

This rule means that if two elements are connected in series, with the first element clocked by Φ_1 and the second element clocked by Φ_2, and the node between them is modeled with an ideal capacitance, then the behavior of the combined circuit

element is the composed functional behavior of the two elements. The value of the input is latched on Φ_1. This is specified by the function latched(Φ_1, A)), which returns the value A had when Φ_1 was $\{1,1\}$. The delay is the delay from Φ_2 characteristic of the second element.

5.1 Example 3: A Dynamic Latch

Next we will consider the behavior of the CMOS dynamic latch shown in Figure 2. This circuit is an instance of the generalized circuit element which employs 2-phase clocking. We have already considered the structure of this circuit, and determined the direction of signal flow through its transistors, as shown in Figure 3. The operation of this common CMOS structure is described in Weste and Eshraghian [33]. When the latch signal L is high during Φ_1, a new value is stored in the latch. Otherwise, the previous value is saved.

The temporal logic description of the behavior of the latch is derived from the behavior of its parts. A sketch of the derivation is given below. A more complete presentation of this example is given in [22].

The structure of the latch is:

$$\begin{aligned}\textsf{dlatch-struct}(L, D, Q, \Phi_1, \Phi_2, R) \quad &\equiv_{\text{def}} \\ &\textsf{invert-mux}(L, D, Q, \Phi_1, y, m) \wedge \\ &\textsf{shiftstage}(y, Q, \Phi_2, p)\end{aligned}$$

The invert-mux is the composition of a 2-to-1 multiplexer described in the previous section and a CMOS inverter. Its behavior, derived from the behaviors of its components, is:

$$\begin{aligned}&\textsf{invert-mux}(A,B,C,\Phi,D,m) \quad \equiv_{\textsf{def}} \\ &\quad (\Box((\Phi = \{1,1\}) \wedge \mathit{len}\ m \quad \supset \\ &\qquad (\mathit{if}\ A = \{1,\_\}\ \mathit{then}\ (\textsf{strengthen}\ \neg B \rightarrow D) \\ &\qquad\qquad \mathit{else}\ (\textsf{strengthen}\ \neg C \rightarrow D))) \wedge \\ &\qquad (\Phi = \{0,1\} \supset \mathit{stb}D))\end{aligned}$$

The behavior of the shiftstage, derived from the behaviors of its components is:

$$\begin{aligned}&\textsf{shiftstage}(A,B,\Phi,M) \quad \equiv_{\textsf{def}} \\ &\quad (\Box\,\Phi = \{1,1\} \quad \supset \quad (\mathit{len}\ M \supset (\textsf{strengthen}\ \neg A \rightarrow B))) \wedge \\ &\quad (\Box\,\neg(\Phi \approx \{1,1\}) \quad \supset \quad (\mathit{stb}B))\end{aligned}$$

There is one constraint on the behavior of the shiftstage:

$$\Box((\Phi \approx \{1,1\}) \quad \supset \quad (\mathit{stb}A))$$

To derive the behavior of the latch, we first expand the behaviors of the parts, and then apply logical rules. One of the logical rules used is the 2-phase clocking rule discussed in this section.

It is straightforward, if tedious, to show that the behaviors of the components imply the behavior of the latch, given in the following definition:

$$\begin{aligned}&\textsf{dlatch}(L,D,Q,\Phi_1,\Phi_2,R) \quad \equiv_{\textsf{def}} \\ &\quad (\Box\ \mathit{if}\ L = \{1,\_\}\ \mathit{then} \\ &\qquad ((\Phi_2 = \{1,1\}) \wedge \mathit{len}\ p \supset \textsf{strengthen latched}(\Phi_1, D) \rightarrow Q) \\ &\qquad \mathit{else}((\Phi_2 = \{1,1\}) \wedge \mathit{len}\ p \supset \textsf{strengthen latched}(\Phi_1, Q) \rightarrow Q)) \wedge \\ &\quad (\Box\,\neg(\Phi_2 = \{1,1\}) \supset \mathit{stb}\ Q) \wedge \\ &\quad (R = p)\end{aligned}$$

The constraints of the parts for the dynamic latch are:

1. $\Box((L \wedge \Phi_1) \approx \{1,\_\}) \supset \mathit{stb}\ D$
2. $\Box((\neg L \wedge \Phi_1) \approx \{1,\_\}) \supset \mathit{stb}\ Q$
3. $\Box(\Phi_1 \approx \{1,1\}) \supset \mathit{stb}\ L$
4. $\Box(\Phi_2 \approx \{1,1\}) \supset \mathit{stb}\ y$

The first three constraints are the constraints from the 2-to-1 multiplexer of the invert-mux component, and the fourth constraint is due to the shift stage. The constraints we wish to have for the dlatch state that the load signal L is stable when Φ_1 is high, and that the input D is stable when it is being loaded into the latch. These are expressed in ITL:

$$\Box(((L \wedge \Phi_1) \approx \{1, \_\}) \supset stb\ D)$$
$$\Box((\Phi_1 \approx \{1,1\}) \supset stb\ L)$$

These two constraints are identical to the first and third constraints above. Constraints 2 and 4 can be eliminated by manipulating the behavioral equations of the parts of the latch. Constraint 4 is true if the input to the shiftstage is stable during Φ_2. This in turn is true if the delay through the invert-mux is less than the length of Φ_1 plus the separation between Φ_1 and Φ_2. Constraint 3 is true if the output Q is stable during Φ_1. This in turn is true if the separation between Φ_2 and Φ_1 is longer than the delay through the inverter which drives Q. These requirements put additional constraints on the relative timing of the different clock phases.

6 Discussion

Formal hardware verification has several advantages over conventional methods such as simulation for verifying circuits. With formal verification, signals are manipulated symbolically. Thus, by proving that an implementation meets its specification, the designer is sure that it has that behavior for all cases. In simulation the designer can only be sure of the behavior for the cases tested. Another advantage of verification is the ability to exploit modularity. A functional block such as a latch need only be proved once no matter how many times it is used in the circuit. In addition, a portion of the design can be proved to meet its

specification before the rest of the design is complete. Thus errors can be caught early in the design process. Modularity also allows small changes to a design to be handled easily since only those portions of the design which have been altered need to be reverified.

These advantages do not imply that formal verification will replace simulation. Formal verification can be viewed as another tool in the designer's tool box which gives added confidence in the correctness of designs. Even though a circuit can be proven to correctly implement its behavioral specification, the specification itself could be wrong. It is therefore useful to simulate the specification to check that it exhibits the required behavior. A subset of ITL can be simulated with the executable programming language Tempura [25]. The Tempura interpreter simulates this subset by finding values for the Tempura variables which result in the temporal logic formulas being valid. Moszkowski shows how this approach can be used to simulate ITL descriptions of circuits. Such simulation is useful for checking that the specification does in fact describe the desired behavior of the device.

A circuit is verified by showing that its behavior can be derived from the mathematical models of the behavior of the components which make it up. If a mathematical model does not capture some physical aspect of the circuit, that physical aspect will not be captured in the proof. For example, in a proof system whose primitives are combinational elements which ignore delay, delay characteristics of a circuit cannot be derived. Similarly, if charge-sharing is not modeled, charge-sharing bugs will not be detected. In these cases other tools such as timing analyzers and design rule checkers are required. It is important to note that there will always be aspects of behavior which are not captured in a model. This applies to models used in simulation as well as those used in formal methods. The user of a

model should be aware of these shortcomings.

The research presented in this article provides a framework for investigating more detailed models of transistor behavior. There are several low level aspects of transistor behavior which our model does not capture. For example, the model cannot express threshold drops. Thus, the need for p-type as well as n-type transistors in CMOS transmission gates is not apparent in this model. In addition, charge sharing bugs, races or hazards cannot be detected. Note that the examples have been restricted to circuits in which races and hazards will not arise. These, as well as charge sharing, are important aspects of incorrect MOS circuit behavior which will arise in a more general class of MOS circuits. Consideration should go into incorporating them into the model. Weise [32] describes a verification system which identifies races and hazards.

We consider delays of circuit elements at the transistor level, but use a simple model for composing these delays. Delays of elements connected in series are added, and the maximum delay of elements connected in parallel is used. This aspect of timing modeling should be improved. Ousterhout [29] presents several switch-level delay models which could be incorporated into our approach.

A simple improvement would result from taking into consideration resistive and capacitive effects on delay. For example, the delay of a component could be modeled by:

$$intrinsic\ delay + R_{Load} \times C_{Load}$$

Here R_{Load} is a function of the output of the component being modeled and C_{Load} is the sum of the capacitances of the devices being driven. Further improvements could be gained by taking into consideration the waveform shape of the input wave-

form which is dependent on the fan-out and drive capability of the preceding stage of the circuit. Currently we assume that all waveform changes are instantaneous.

Since wires add an appreciable amount of delay in MOS technologies these should be modeled, and the capacitance of the material of the wire should be considered. This has not yet been done since this information is highly dependent on the actual layout of a circuit. The research presented here would benefit from being incorporated into a system which relates layout to circuit implementation.

We have described how to formally derive low level timing and functional behavior from transistor level descriptions of hardware components. This only addresses one aspect of the design cycle. This approach should be incorporated into a computer aided design system which encourages designers to apply formal methods from design conception to layout. An important advantage of formal methods is the ability to relate different levels of description of a circuit. A system which incorporates formal methods should allow many different tools to work on different levels of representation of a circuit, and should keep the various different levels consistent. Subrahmanyam [31] describes an expert system for VLSI design which incorporates formal methods as well as more conventional tools. Milne [24] describes a more revolutionary approach, which does not attempt to incorporate existing tools, where a system can be described at various different levels from specification to layout and the different levels can be formally related.

7 Related Work

Other formalisms have been used for formal verification. These include first-order logic, higher-order logic and temporal logic.

Several approaches to formal hardware verification have been based on first-order

logic [1], [19]. These systems automate many aspects of the task of hardware verification and have been used to verify complex hardware designs. Drawbacks in the use of first-order logic include difficulty in representing time directly, and difficulty in expressing temporal and behavioral abstractions between different levels of description. Systems which use higher-order logic [15], [16], provide several advantages over first-order logic, including a more direct way of expressing abstraction and a more elegant way of representing time with signals modeled as functions from time to values. Herbert [18] discusses modeling timing and function of digital circuits using higher-order logic. Several large examples from real designs which have been verified at Cambridge using higher-order logic include the ECL chip of the Fast Ring [17] and the VIPER microprocessor [8].

A third formalism used for hardware verification is temporal logic. A major difference between using temporal logic and higher-order logic is the way time is modeled. In temporal logic, time is implicit in the logical operators. In higher-order logic, time is usually represented by an explicit time variable. For example, the behavior of an inverter with input In and output Out and delay m is expressed in higher-order logic by:

$$\mathsf{invert}(In, Out, m) \quad \equiv_{\mathrm{def}} \quad \forall t.Out(t + m) = \neg In(t)$$

The same inverter, described in ITL, has the behavior:

$$\mathsf{invert}(In, Out, m) \quad \equiv_{\mathrm{def}} \quad \Box len\, m \supset (\neg In \rightarrow Out)$$

Temporal logic has been used for specifying software, hardware and communications protocols. Moszkowski uses Interval Temporal Logic to specify and reason about digital circuits [27]. The circuits he describes are clocked circuits which

have delayless combinational elements and memory elements as primitives. Several other temporal logics have been used for hardware specification and verification. LTTL [13] has been used to specify hardware at the register-transfer level and above. The logic programming language Tokio [12] can be used to execute LTTL specifications. The result is a simulation of a specification similar to that provided by Tempura. Other uses of temporal logic in verification include using Computation Tree Logic to specify and verify the behavior of asynchronous and sequential circuits [3], [11]. Extend Temporal Logic [14] has been used to describe and reason about VLSI circuits, including a dynamic latch similar to the one discussed in this article. Here the behavior of a circuit is validated by verifying that the specification of the input signals *and*ed with the temporal logic description of the circuit implies a specification of the output signals. Note that the user must specify the inputs and outputs, so the result resembles simulation more than formal verification.

8 Conclusions

We have presented an approach to hardware verification which uses temporal logic to reason about MOS VLSI circuits at the transistor level. This approach has been demonstrated by deriving the behavior of several examples from the behavior of their components down to the transistor level. These examples included a dynamic latch which uses a 2-phase clocking scheme and exploits charge sharing. The properties derived include the functional behavior of the latch, constraints on when the inputs must be stable, the time at which the outputs are available, and constraints on the lengths of the different clock phases.

We have described the application of formal tools to one step in the design hierarchy: from the transistor level to the gate level. The advantages of using formal

methods include the advantages gained by hierarchical and incremental analysis, and by manipulating inputs and outputs symbolically. In addition, formal methods allow different levels of description to be related formally. By incorporating many levels of description into a design system based on formal methods, errors which occur when one level is translated into another will be detected. Such errors are not detected by current design tools.

Acknowledgements

Thanks to the members of the Hardware Verification Group who have contributed to this research. Special thanks to John Herbert and Jeff Joyce for many useful discussions on timing and transistor modeling. Thanks also to William Clocksin, Robert Cooper, Mike Gordon, John Herbert and Jeff Joyce for reading and commenting on earlier drafts of this paper.

References

[1] H. G. Barrow. Proving the correctness of digital hardware designs. *VLSI Design*, 64–77, July 1984.

[2] M. A. Breuer and A. D. Friedman. *Diagnosis & Reliable Design of Digital Systems.* Computer Science Press, Inc., 1976.

[3] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, C-35(12):1035–1043, December 1986.

[4] R. E. Bryant. *A Switch-Level Simulation Model For Integrated Logic Circuits.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1981.

[5] W. F. Clocksin. Logic programming and digital circuit analysis. *Journal of Logic Programming*, 4:59–82, 1987.

[6] W. F. Clocksin and M. E. Leeser. Automatic determination of signal flow through MOS transistor networks. *Integration*, 4:53–63, 1986.

[7] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, second edition, 1984.

[8] A. Cohn. A proof of correctness of the Viper microprocessor: the first level. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 27–71, Kluwer Academic Publishers, 1988.

[9] A. Despain, Y. Patt, V. Srini, et al. Aquarius. *Computer Architecture News*, 15(1):22–34, 1987.

[10] I. S. Dhingra. Formal validation of an integrated circuit design style. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 293–321, Kluwer Academic Publishers, 1988.

[11] D. L. Dill and E. M. Clarke. Automatic verification of asynchronous circuits using temporal logic. *IEE Proceedings E: Computers and Digital Techniques*, 133(5):276–282, September 1986.

[12] M. Fujita, S. Kono, H. Tanaka, and T. Moto-oka. Aid to hierarchical and structured logic design using temporal logic and Prolog. *IEE Proceedings E: Computers and Digital Techniques*, 133(5):283–294, September 1986.

[13] M. Fujita, H. Tanaka, and T. Moto-oka. Verification with Prolog and temporal logic. In T. Uehara and M. Barbacci, editors, *Computer Hardware Description Languages and Their Applications*, pages 103–114, North Holland, 1983.

[14] A. Fusaoka, H. Seki, and K. Takahashi. Description and reasoning of VLSI circuit in temporal logic. *New Generation Computing*, 2:79–90, 1984.

[15] M. J. C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, North-Holland, 1986.

[16] F. K. Hanna and N. Daeche. Specification and verification of digital systems using higher-order predicate logic. *IEE Proceedings E: Computers and Digital Techniques*, 133(5):242–254, September 1986.

[17] J. Herbert. The application of formal specification and verification to a hardware design. In C. J. Koomen and T. Moto-oka, editors, *Computer Hardware Description Languages and Their Applications*, pages 434–451, North Holland, 1985.

[18] J. M. J. Herbert. *Application of Formal Methods to Digital System Design.* PhD thesis, Computer Laboratory, University of Cambridge, 1986.

[19] W. A. Hunt, Jr. *FM8501: A Verified Microprocessor.* PhD thesis, Institute for Computing Science, The University of Texas at Austin, 1986.

[20] N. P. Jouppi. TV: an nMOS timing analyzer. In *Proceedings of the 3rd Caltech VLSI Conference*, pages 71–76, 1983.

[21] J. J. Joyce. *Hardware Verification of VLSI Regular Structures.* Technical Report 109, University of Cambridge Computer Laboratory, July 1987.

[22] M. E. Leeser. *Reasoning about the Function and Timing of Integrated Circuits with Prolog and Temporal Logic.* PhD thesis, Computer Laboratory, Cambridge University, 1987.

[23] T-M Lin and C. A. Mead. A hierarchical timing simulation model. *IEEE Transactions on Computer-Aided Design*, CAD-5(1):188–197, January 1986.

[24] R. Milne. Design transformation and chip planning. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 23–43, North-Holland, 1986.

[25] B. C. Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, 1986.

[26] B. C. Moszkowski. *Reasoning about Digital Circuits.* PhD thesis, Department of Computer Science, Stanford University, July 1983.

[27] B. C. Moszkowski. A temporal logic for multilevel reasoning about hardware. *Computer*, 10–19, February 1985.

[28] J. K. Ousterhout. Crystal: a timing analyzer for nMOS VLSI circuits. In *Proceedings of the 3rd Caltech VLSI Conference*, pages 57–69, 1983.

[29] J. K. Ousterhout. Switch-level delay models for digital MOS VLSI. In *21st ACM/IEEE Design Automation Conference*, pages 542–548, 1984.

[30] J. D. Pincus and A. M. Despain. Delay reduction using simulated annealing. In *23rd ACM/IEEE Design Automation Conference*, pages 690–695, 1986.

[31] P. A. Subrahmanyam. Synapse: an expert system for VLSI design. *Computer*, 78–89, July 1986.

[32] D. W. Weise. *Formal Multilevel Hierarchical Verification of Synchronous MOS VLSI Circuits.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, August 1986.

[33] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective*, chapter 5.4. Addison-Wesley Publishing Company, 1985.

A ITL Operators

The ITL operators used in this article are defined in the table below. These definitions are given either with the $\mathcal{M}$ function, as a definition in terms of other ITL operators, or by listing its properties. The function $\mathcal{M}$ maps formulas and *intervals* to truth values. For example, the definition of $\bigcirc w$ is given as:

$$\mathcal{M}_{s_0 \ldots s_n}[\![\bigcirc w]\!] = true \quad \text{iff} \quad n \geq 1 \text{ and } \mathcal{M}_{s_1 \ldots s_n}[\![w]\!] = true$$

This says that $\bigcirc w$ is true of an interval $\langle s_0, s_1, s_2, s_3 \rangle$ if and only if (iff) w is true of the interval $\langle s_1, s_2, s_3 \rangle$ which starts at the next instant of time.

Operator Name	Formula	Interpretation/Definition
always	$\Box w$	$\mathcal{M}_{s_0 \ldots s_n}[\![\Box w]\!] = true$ iff $\mathcal{M}_{s_i \ldots s_n}[\![w]\!] = true$ for all $i \leq n$
next	$\bigcirc w$	$\mathcal{M}_{s_0 \ldots s_n}[\![\bigcirc w]\!] = true$ iff $n \geq 1$ and $\mathcal{M}_{s_1 \ldots s_n}[\![w]\!] = true$
chop	$w_1; w_2$	$\mathcal{M}_{s_0 \ldots s_n}[\![w_1; w_2]\!] = true$ iff $\mathcal{M}_{s_0 \ldots s_i}[\![w_1]\!] = true$ and $\mathcal{M}_{s_i \ldots s_n}[\![w_2]\!]$ for some i $0 \leq i \leq n$
fin	$fin\ w$	$\mathcal{M}_{s_0 \ldots s_n}[\![fin\ w]\!] = true$ iff $\mathcal{M}_{s_n}[\![w]\!] = true$
length	$len\ n$	$\mathcal{M}_{s_0 \ldots s_m}[\![len\ n]\!] = true$ iff $m = n$
skip	$skip$	$skip \equiv_{\text{def}} len\ 1$
temporal equality	$A \approx B$	$A \approx B \equiv_{\text{def}} \Box(A = B)$
temporal assignment	$A \rightarrow B$	$A \rightarrow B \equiv_{\text{def}} \forall c.((A \Leftrightarrow c) \supset fin(B \Leftrightarrow c))$ for c static
temporal stability	$stb\ A$	$stb\ A \equiv_{\text{def}} \exists c.(A \approx c)$ for c static

Operator Name	Formula	Interpretation/Definition
delay	$A \text{ del}^m B$	$A \text{ del}^m B \quad \equiv_{\text{def}} \quad \Box \, len \, m \supset (A \rightarrow B)$
signal equivalence	$\{V_1, S_1\} \Leftrightarrow \{V_2, S_2\}$	$\{V_1, S_1\} \Leftrightarrow \{V_2, S_2\} \quad \equiv_{\text{def}}$ $V_1 = V_2 \ \wedge \ S_1 = S_2$
weaken	weaken$\{V, S\}$	weaken $\{V, S\} \Leftrightarrow \{V, 0\}$
strengthen	strengthen$\{V, S\}$	strengthen $\{V, S\} \Leftrightarrow \{V, 1\}$
join	$V_1 \sqcup V_2$	$\{V_1, 1\} \sqcup \{V_2, 0\} \Leftrightarrow \{V_1, 1\}$ $\{V_2, 0\} \sqcup \{V_1, 1\} \Leftrightarrow \{V_1, 1\}$ $A \sqcup A \Leftrightarrow A$

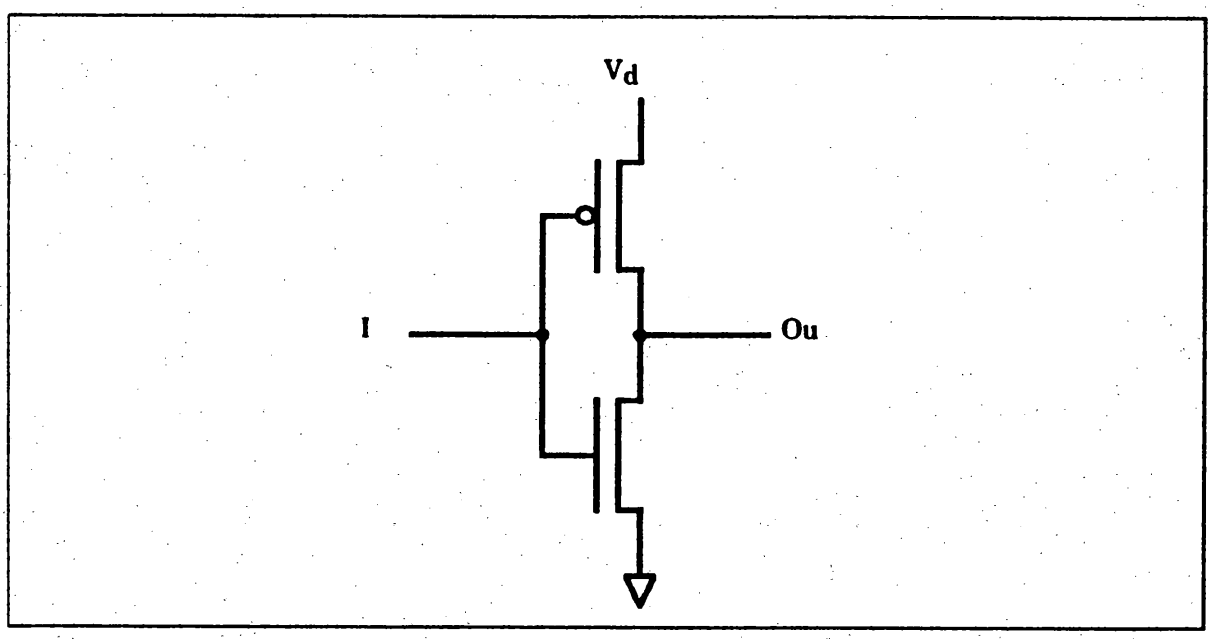

Figure 1: A CMOS Inverter

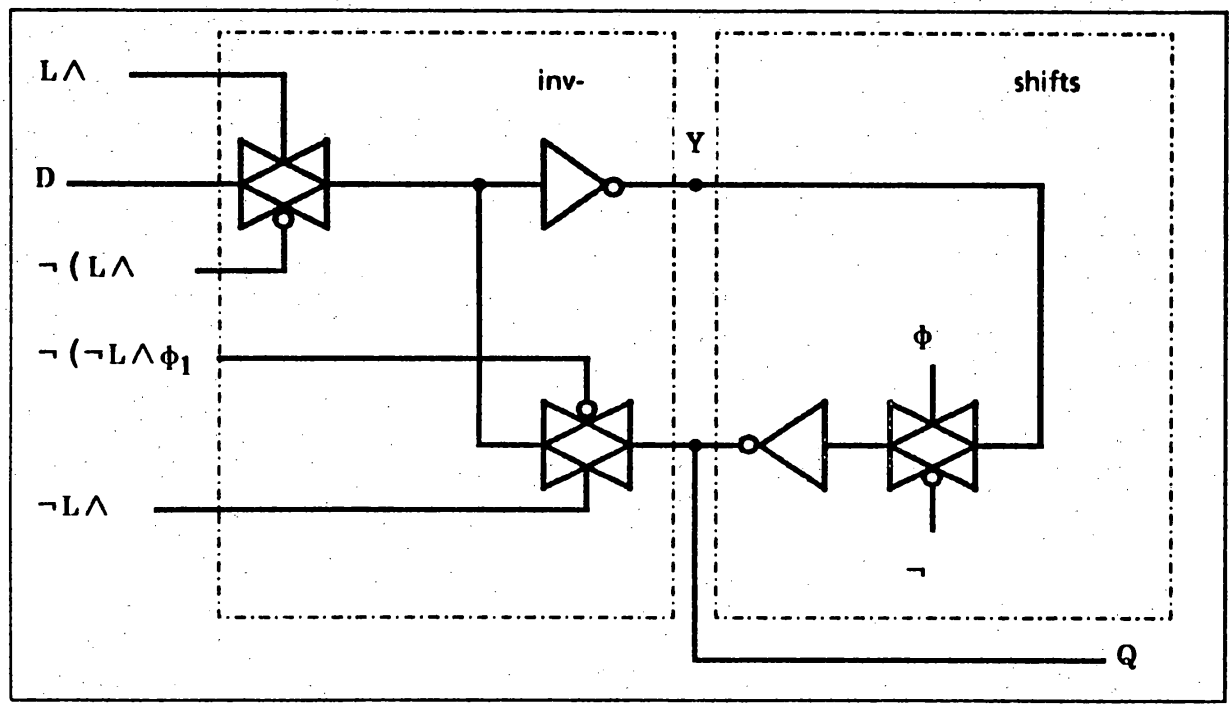

Figure 2: A CMOS Dynamic Latch

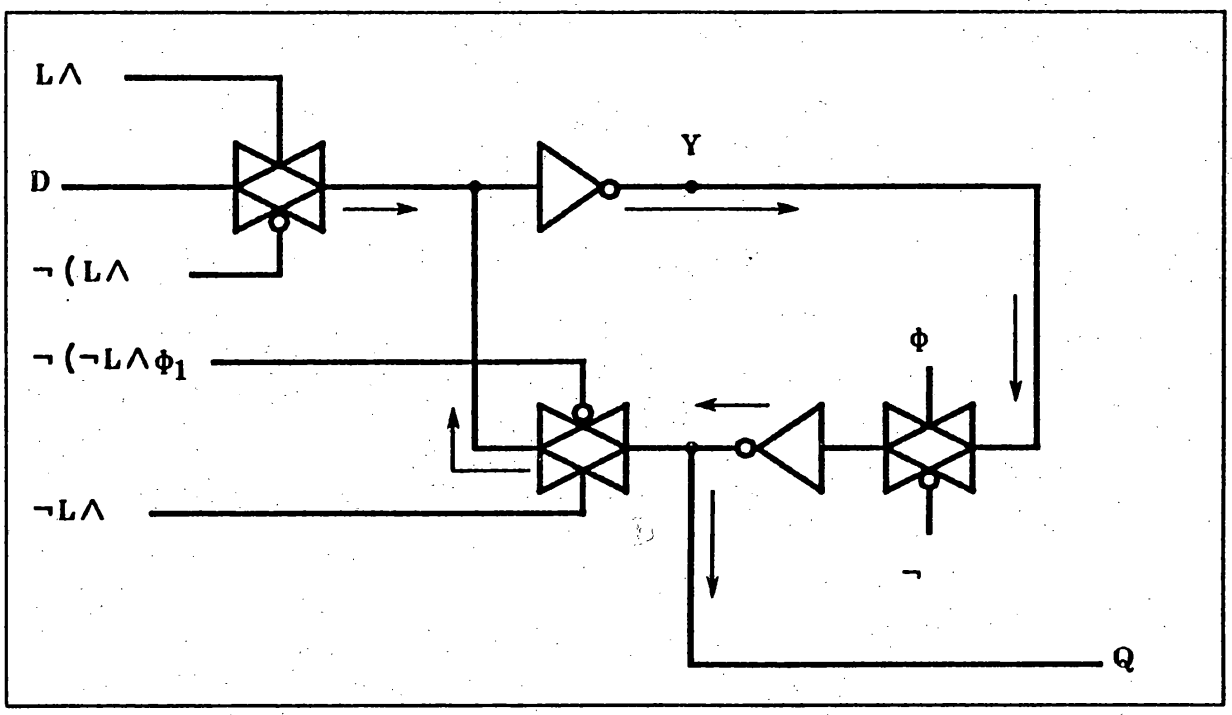

Figure 3: A CMOS Dynamic Latch with Signal Flow

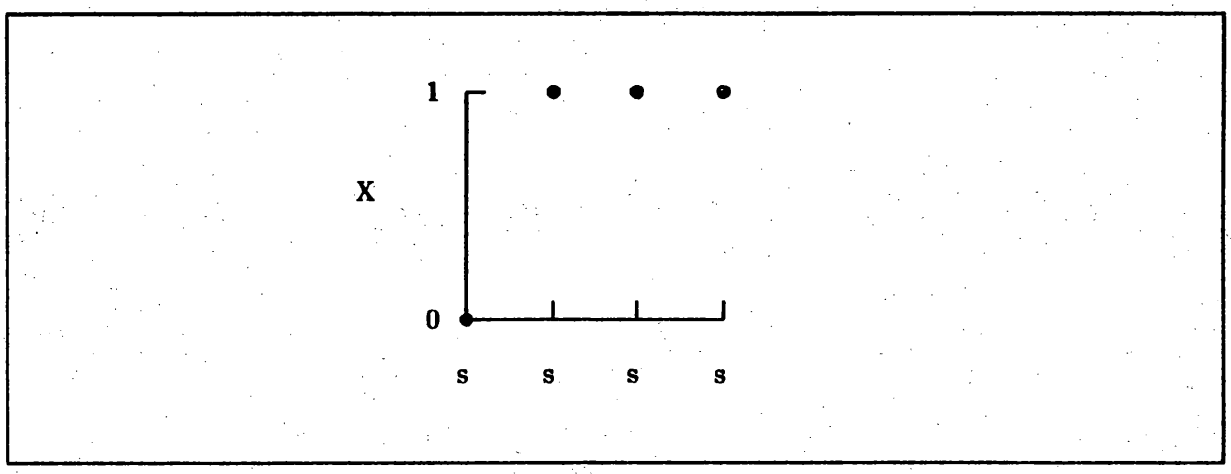

Figure 4: A Rising Bit Signal

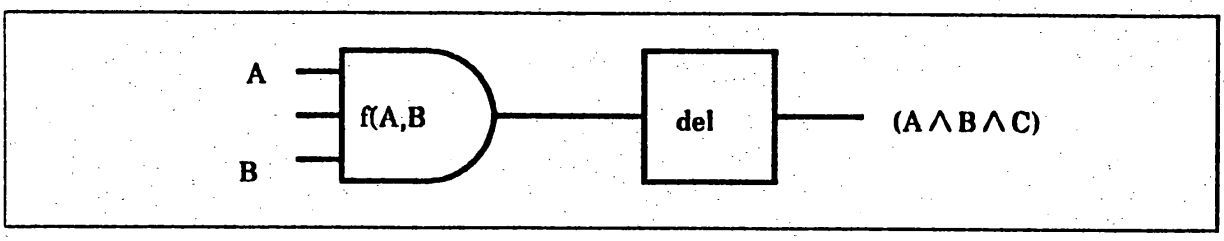

Figure 5: Model of a 3-Input **and** Gate with Delay

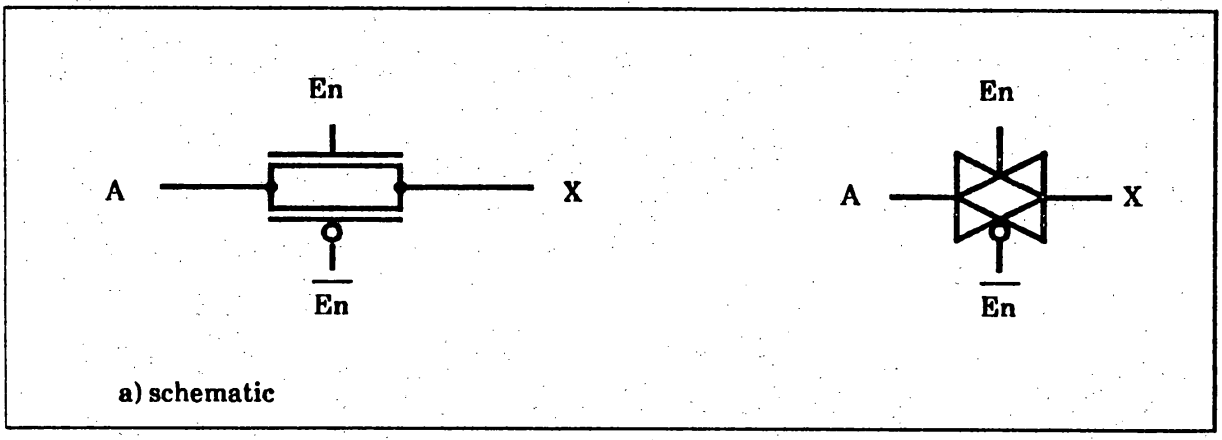

Figure 6: A CMOS Transmission Gate

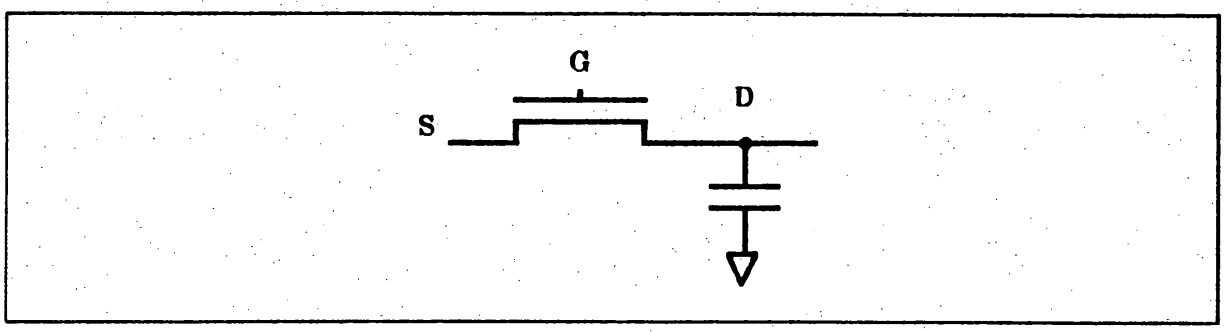

Figure 7: A Pass Transistor with Explicit Capacitance

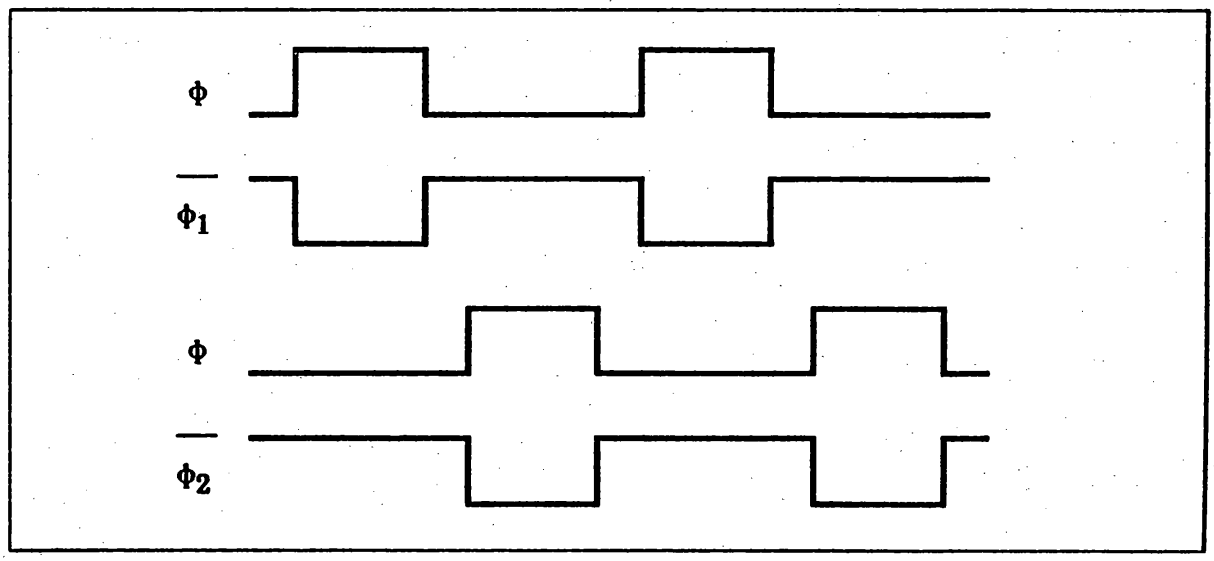

Figure 8: 2-Phase Non-overlapping Clocks

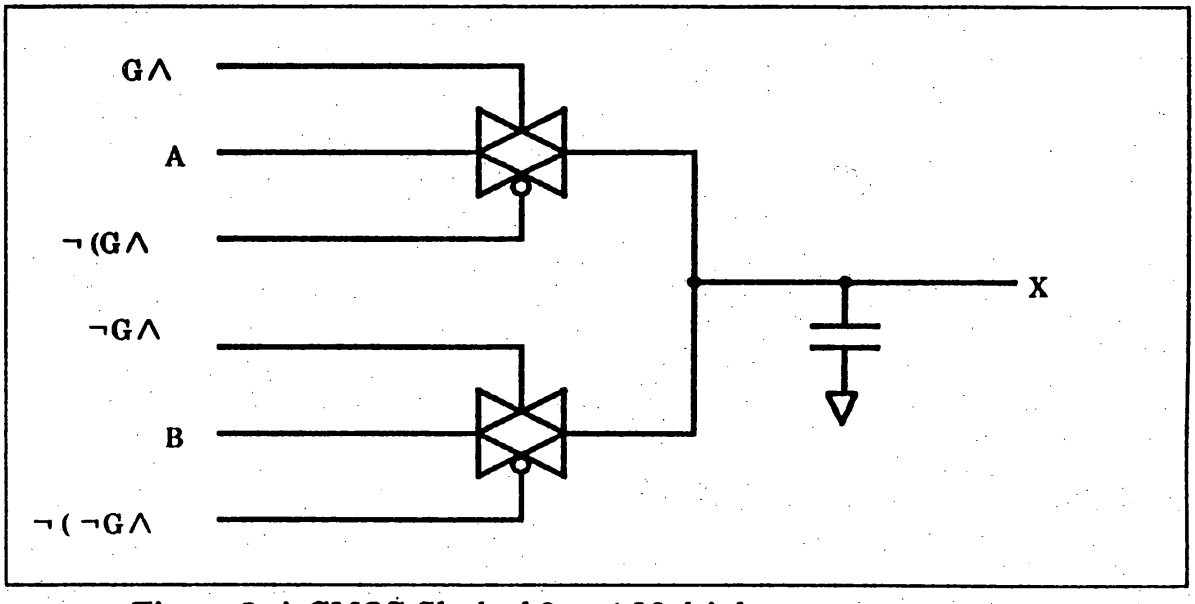

Figure 9: A CMOS Clocked 2-to-1 Multiplexer

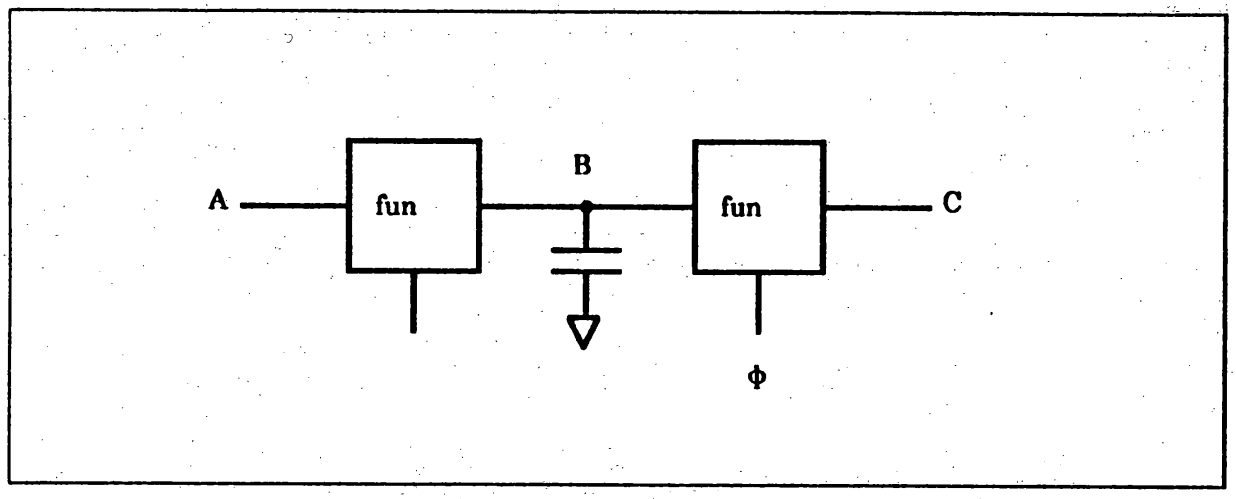

Figure 10: An Element which Employs 2-phase Clocking