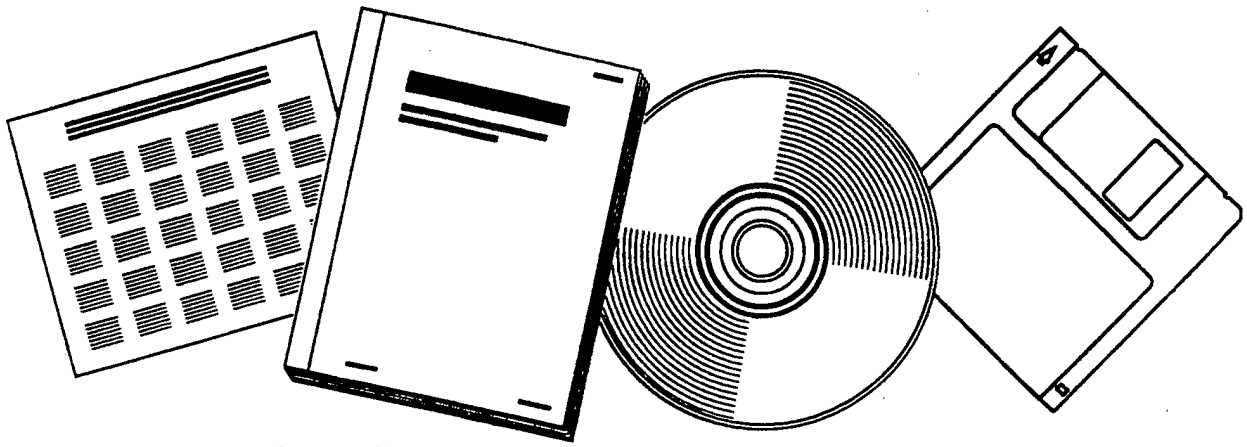
A TEMPORAL LOGIC FOR MULTI-LEVEL REASONING ABOUT HARDWARE

DEPARTMENT OF COMPUTER SCIENCE
STANFORD, CA

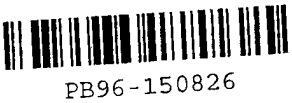
DEC 82



U.S. DEPARTMENT OF COMMERCE
National Technical Information Service

December 1982

Report No. STAN-CS-82-952



A Temporal Logic for Multi-Level Reasoning about Hardware

by

Ben Moszkowski

Department of Computer Science

Stanford University
Stanford, CA 94305

BIBLIOGRAPHIC INFORMATION

PB96-150826

Report Nos: STAN-CS-82-952

Title: Temporal Logic for Multi-Level Reasoning about Hardware.

Date: Dec 82

Authors: B. Moszkowski.

Performing Organization: Stanford Univ., CA. Dept. of Computer Science.

Sponsoring Organization: *National Science Foundation, Washington, DC.*Defense Advanced Research Projects Agency, Arlington, VA.*Air Force Office of Scientific Research, Bolling AFB, DC.

Contract Nos: DARPA-N00039-82-C-0250, NSF-MCS79-09495, NSF-MCS81-11586, AFOSR-81-0014

Supplemental Notes: Presented at the IFIP International Conference on Computer Hardware Description Languages and Their Applications (6th), Pittsburgh, PA., May 1983.

NTIS Field/Group Codes: 62E (Information Theory), 62A (Computer Hardware), 62B (Computer Software)

Price: PC A03/MF A01

Availability: Available from the National Technical Information Service, Springfield, VA. 22161

Number of Pages: 26p

Keywords: *Mathematical logic, *Logic design, Learning machines, Artificial intelligence, Computer systems programs, Computer systems hardware, *Temporal logic, *Multi-level reasoning.

Abstract: The paper describes a logical notation for reasoning about digital circuits. The formalism provides a rigorous and natural basis for device specification as well as for proving properties such as correctness of implementation. Conceptual levels of circuit operation ranging from detailed quantitative timing and signal propagation up to functional behavior are integrated in a unified way. A temporal predicate calculus serves as the formal core of the notation, resulting in a versatile tool that has more descriptive power than any conventional hardware specification language. The logic has been applied to specifying and proving numerous properties of circuits ranging from delay elements up to the Am2901 ALU bit slice. Presentations of a delay model and a multiplication circuit illustrate various features of the notation.

A Temporal Logic for Multi-Level Reasoning about Hardware

Ben Moszkowski
Department of Computer Science
Stanford University
Stanford, California 94305

Abstract

This paper describes a logical notation for reasoning about digital circuits. The formalism provides a rigorous and natural basis for device specification as well as for proving properties such as correctness of implementation. Conceptual levels of circuit operation ranging from detailed quantitative timing and signal propagation up to functional behavior are integrated in a unified way.

A temporal predicate calculus serves as the formal core of the notation, resulting in a versatile tool that has more descriptive power than any conventional hardware specification language. The logic has been applied to specifying and proving numerous properties of circuits ranging from delay elements up to the Am2901 ALU bit slice. Presentations of a delay model and a multiplication circuit illustrate various features of the notation.

The work presented here was supported in part by the National Science Foundation under a Graduate Fellowship, Grants MCS79-09495 and MCS81-11586, by DARPA under Contract N00039-82-C-0250, and by the United States Air Force Office of Scientific Research under Grant AFOSR-81-0014.

This paper is part of the author's Ph.D. dissertation under the supervision of Professor Zohar Manna and will appear in the IFIP Sixth International Conference on Computer Hardware Description Languages and Their Applications, Pittsburgh, Pennsylvania, May, 1983.

§1 Introduction

Computer systems continue to grow in complexity and the distinctions between hardware and software keep on blurring. Out of this has come an increasing awareness of the need for behavioral models suited for specifying and reasoning about both digital devices and programs. Contemporary hardware description languages (for example [1,15,19]) are not sufficient because of various conceptual limitations:

- Most such tools are intended much more for simulation than for mathematically sound reasoning about digital systems. Many compromises are made so that the descriptions can be executed.
- Difficulties arise in developing circuit specifications that out of necessity must refer to different levels of behavioral abstraction.
- What formal tools there are for such languages cannot in general deal with the inherent parallelism and nondeterminism of circuits.

The formalism presented in this paper overcomes these problems and unifies in a single notation digital circuit behavior that is generally described by means of the following techniques:

- Register transfer operations
- Flowgraphs and transition tables
- Tables of functions
- Timing diagrams
- Schematics and block diagrams

The notation is based on discrete time intervals and combines aspects of standard temporal logics [12,17] with features of dynamic logic [7]. Halpern et al. [6] shows that useful subsets of the logic are decidable and of relatively reasonable computational complexity. This indicates that partial automation of reasoning may be practical. The formalism's applicability is by no means limited to the goals of computer-assisted verification and synthesis of circuits. This type of notation, with appropriate "syntactic sugar," could provide a fundamental and rigorous basis for communicating, reasoning or teaching about digital concepts and devices. Simulation-based languages could for example use such a logic as a vehicle for describing the intended semantics of delays and other features. Thus, semi-automated correctness checking is really only one part of a much bigger picture.

Before outlining the formalism, the paper discusses related work. The temporal logic is then informally introduced by way of sample properties. Following this, the formalism serves as a basis for specifying and reasoning about various aspects of a simple delay element as well as of a hardware multiplication circuit. Quantitative timing as well as algorithm development are discussed.

§2 Related Work

Gordon's work [4] on register-transfer systems uses a denotational semantics to provide a concise means for reasoning about clocking, feedback, instruction-set implementation and bus communication. No quantitative timing properties are considered and the notation has some difficulties in describing operations occurring over multiple cycles. Wagner [20] presents a semi-automated proof development system for reasoning about signal transitions and register transfer behavior. Unfortunately the notation suffers from a lack of formality that is difficult to remedy. Malachi and Owicki [11] utilize a temporal logic to model self-timed digital systems by giving a set of axioms. No indication is included on how to generalize the work to the entire domain of digital circuits. The work of Bochmann [2] describes and verifies properties of an arbiter, a device for regulating access to shared resources. The presentation, by means of a temporal logic, reveals some tricky aspects in reasoning about such components although the concepts used are not as rigorously developed as they may appear to be and do not easily generalize. As in the previous works, no quantitative timing issues are examined.

Leinwand and Lamdan [9] use a type of Boolean algebra to model signal transitions. Applications include systems with feedback and critical timing constraints. The use of the notation for non-trivial examples is very unintuitive. Patterson [16] explores the verification of firmware. This work views the problem from the sequential programming standpoint without describing the underlying digital circuitry and related issues of concurrency and timing. There is also work by Meinen [13] on register transfer behavior and McWilliams [10] on worst-case time constraints.

Eveking [3] uses predicate calculus with an explicit time variable to explore verification in the Conlan language. Although such an approach can in principle describe circuits, the proliferation of variables representing explicit time points becomes a major hindrance from a practical as well as theoretical standpoint. Many high-level temporal concepts become easily obscured amid all the notation.

A number of people have used temporal logics to describe computer communication protocols [5,8,18]. However, the precise connections between protocols and

the underlying hardware and software are still rather unclear as are the relative advantages of the different techniques employed.

§3 Notational Preliminaries

Before the logic is introduced, it is necessary to say a little about the kinds of mathematical entities used here for modelling digital signals.

Data Values

Values are limited to natural numbers, $\perp$ (read "bottom"), and finite-length vectors constructed using these elements. Both 0 and 1 as well as $\perp$ serve as bits, with 0 standing for low voltage, 1 for high voltage and $\perp$ representing voltages that are out of range. Finite-length vectors can be formed containing natural numbers and $\perp$. The following are sample values:

$$0, \quad 3, \quad \perp, \quad \langle 0 \rangle, \quad \langle 1, 2 \rangle, \quad \langle \rangle, \quad \langle \perp, 1, 1 \rangle$$

Bit Operations

Four basic operations defined on bits are *complement* ($\ominus$), *and* ($\otimes$), *or* ($\oslash$) and *exclusive-or* ($\oplus$). The symbols $\ominus$, $\otimes$ and $\oslash$ are used instead of $\neg$, $\wedge$ and $\vee$ in order to distinguish notationally between bit expressions and formulas in the underlying predicate calculus. Here are corresponding truth tables extended to include $\perp$:

$\ominus$	
0	1
1	0
$\perp$	$\perp$

$\otimes$	0	1	$\perp$
0	0	0	0
1	0	1	$\perp$
$\perp$	0	$\perp$	$\perp$

$\oslash$	0	1	$\perp$
0	0	1	$\perp$
1	1	1	1
$\perp$	$\perp$	1	$\perp$

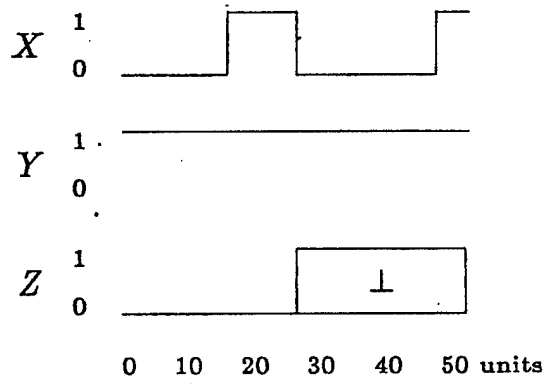
$\oplus$	0	1	$\perp$
0	0	1	$\perp$
1	1	0	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$

§4 Informal Overview of Temporal Operators

The temporal logic provides a basis for describing periods of time such as in timing diagrams. Concepts such as signal response and oscillation are readily

expressible. Examples serve to introduce the various operators used later in this paper. This presentation has been kept rather informal although the entire logic is explored in detail in Halpern et al. [6] and Moszkowski [14].

Time is modeled as being discrete and finite. The following figure is a typical timing diagram:



This represents the behavior of the signals X, Y and Z over a period of 50 units of time. The signal X goes up and down twice, while Y is stable with the value 1. Initially Z equals 0 for over 20 units, after which it equals $\perp$. Notice that all times are relative. This approach is used because the properties to be examined depend solely on distances between points, independent of any absolute times.

The group of signals can be modeled as a finite *temporal interval* σ mapping variables and times to values. The behavior of intervals is concisely expressible by temporal formulas presented below. Given such a formula p, the construct $\sigma \vDash p$ means p is true for the interval σ. The notation $\vDash p$ signifies that the formula p is true of all intervals. Please keep in mind that all operators discussed can be expressed in terms of a small collection of fundamental notions. The properties shown are deducible from a basic set of logical rules.

4.1 Initial and Terminal Equality

The formula $beg(X = Y)$ is true for an interval σ if within σ the two signals X and Y have equal starting values. Similarly, the construct $fin(X = Y)$ is true for an interval σ if X and Y end up equal in σ.

Examples for a given interval σ:

Concept	*Formula*
X and Y start equal and end complements	$\sigma \vDash [beg(X = Y) \wedge fin(X = \ominus Y)]$
X ends equal to 1 and Y ends equal to 0	$\sigma \vDash fin(X = 1 \wedge Y = 0)$

Properties that are true for all intervals:

$$\models \neg \mathit{fin}(X = Y) \equiv \mathit{fin}(\neg(X = Y))$$

The signals X and Y do not end equal if and only if they end up not equal.

$$\models \mathit{fin}((X \circledcirc Y) = 1) \supset \mathit{fin}(X = 1 \wedge Y = 1)$$

If the bit-and of X and Y ends up equaling 1, both X and Y end up equal to 1.

4.2 Temporal Equality

Two signals X and Y are *temporally equal* in an interval σ if they have the same values at all times. This is written $X \approx Y$ and differs from the constructs for initial and terminal equality, which only examine signals' values at the extremes of the interval.

Examples:

Concept	*Formula*
The signal X is 0 throughout the interval	$\sigma \models X \approx 0$
The bit-and of X and Y everywhere equals 0	$\sigma \models (X \circledcirc Y) \approx 0$
X agrees everywhere with the complement of Y	$\sigma \models X \approx \ominus Y$

Properties:

$$\models X \approx Y \supset f(X) \approx f(Y)$$

If two signals are temporally equal, then any function applied to one of them temporally equals the same function applied to the other.

$$\models X \approx 0 \supset X \circledcirc Y \approx 0$$

If X temporally equals 0, then the bit-and of it with another signal also equals 0 everywhere.

$$\models \langle X, Y \rangle \approx \langle 0, 1 \rangle \equiv [X \approx 0 \wedge Y \approx 1]$$

The pair $\langle X, Y \rangle$ temporally equals $\langle 0, 1 \rangle$ exactly if the signal X temporally equals 0 and Y temporally equals 1.

4.3 Temporal Stability

A signal X is *stable* if it has a constant, defined value. The notation used is $stb\,X$. In the case of a bit signal, this means that the signal is always 0 or always 1, that is

$$stb\,X \equiv [X \approx 0 \vee X \approx 1]$$

Example: (this and further examples will omit the symbols "$\sigma \models$")

Concept	*Formula*
The complement of X is stable	$stb \ominus X$

Properties:

$$\models \quad [X \approx 1] \equiv [stb\,X \wedge beg(X = 1)]$$

The signal X always equals 1 if and only if X is stable and initially equals 1.

$$\models \quad stb\,X \equiv stb \ominus X$$

A bit signal is stable if and only if its complement is.

$$\models \quad [stb\,X \wedge stb\,Y] \supset stb(X \oslash Y)$$

If two bit signals are stable, then so is their bit-or. The converse is not always true.

$$\models \quad stb\langle X, Y\rangle \equiv [stb\,X \wedge stb\,Y]$$

A pair is stable exactly if the two individual signals are.

4.4 Temporal Length

Quantitative timing properties are handled by a special object *len* whose value for any interval σ equals the length of σ.

Examples:

Concept	*Formula*
The interval is at least m units in length	$len \geq m$
The signal X is stable and σ measures at least m units	$stb\,X \wedge len \geq m$

The predicate *empty* is true exactly if the interval has length 0. The predicate *skip* is true if the interval has length exactly 1. Since time is discrete, this is the minimum nonzero width.

4.5 Examining Subintervals

For a formula p and interval σ, the construct $\boxdot p$ is true if p is true in *all* subintervals of time contained within σ including σ itself. Note that the "a" in $\boxdot$ simply stands for "all" and is not a variable. The formula $\diamondsuit p$ is true if the formula p itself is true in at least one subinterval of σ.

Examples:

Concept	*Formula*
In some subinterval of length $\geq m+n$, X is stable	$\diamondsuit([len \geq m+n] \wedge stb\, X)$
In all subintervals $< m$ units, X is stable	$\boxdot([len < m] \supset stb\, X)$

Properties:

$$\models \quad \boxdot p \supset p$$

If a formula p is true in all subintervals then it is true in the primary interval.

$$\models \quad \diamondsuit p \equiv \neg \boxdot \neg p$$

A formula is true in some subinterval if and only if the formula is not everywhere false.

$$\models \quad \boxdot(p \wedge q) \equiv [\boxdot p \wedge \boxdot q]$$

The logical-and of two formulas p and q is true in every subinterval if and only if both formulas are true everywhere.

$$\models \quad \diamondsuit p \equiv \diamondsuit \diamondsuit p$$

A formula is somewhere true exactly if there is some subinterval in which the formula is somewhere true.

$$\models \quad [\boxdot p \wedge \diamondsuit q] \supset \diamondsuit(p \wedge q)$$

If p is true in all subintervals and q is true in some subinterval then both are simultaneously true in at least one.

$$\models \quad [X \approx Y] \equiv \boxdot(X = Y)$$

Two signals are temporally equal in an interval exactly if they are equal in every subinterval.

$$\models \quad stb\, X \supset \boxdot stb\, X$$

If X is stable in the overall interval, X is also stable in every subinterval.

4.6 Initial Subintervals

The operators $\boxdot$ and $\diamondsuit$ are similar to $\boxplus$ and $\diamondsuit$ but only look at *initial* subintervals starting at time 0.

Example:

Concept	*Formula*
X is initially stable for at least the first m units	$\diamondsuit(stb\,X \wedge len \geq m)$

4.7 Temporal Dependence

It is useful to specify that a signal X remains stable as long as another signal Y does. X is said to *depend* on Y, written $X\ dep\ Y$. This can be expressed using the temporal formula

$$X\ dep\ Y \equiv \boxdot(stb\,Y \supset stb\,X)$$

Examples:

Concept	*Formula*
X and Y remain stable while Z does	$\langle X, Y\rangle\ dep\ Z$
X remains stable as long as the pair $\langle Y, Z\rangle$ does	$X\ dep\ \langle Y, Z\rangle$

Properties:

$$\models\ [X\ dep\ Y \wedge stb\,Y] \supset stb\,X$$

If X depends on Y and Y is stable, then so is X.

$$\models\ [X\ dep\ Y \wedge Y\ dep\ Z] \supset X\ dep\ Z$$

Dependence is transitive.

$$\models\ beg(X = 0) \supset (X \circledcirc Y)\ dep\ X$$

If X initially equals 0, then the bit-and of X and Y depends on X.

$$\models\ [X\ dep\ Z \wedge Y\ dep\ Z] \equiv \langle X, Y\rangle\ dep\ Z$$

The variables X and Y depend on Z exactly if the pair $\langle X, Y\rangle$ does.

4.8 Adjacent Subintervals

Given a time interval, the formula $p;q$ is true if there is at least one way to divide the interval into two adjacent subintervals σ and σ' such that the formula p is true in the first one, σ, and the formula q is true in the second, σ'. In particular, a rising signal can be described by the predicate $\uparrow X$:

$$\uparrow X \equiv [(X \approx 0);\ skip;\ (X \approx 1)]$$

This says that X is 0 for a while and then jumps to 1. The gap of quantum length represented by the test *skip* is necessary here since a signal cannot be 0 and 1 at exactly the same instant. Falling signals are analogously described by the construct $\downarrow X$:

$$\downarrow X \equiv [(X \approx 1);\ skip;\ (X \approx 0)]$$

Examples:

Concept	*Formula*
X is stable and Y goes up	$stb\, X \wedge \uparrow Y$
The bit-or of X and Y falls	$\downarrow(X \oslash Y)$
In every subinterval where X rises, Y falls	$\boxdot(\uparrow X \supset \downarrow Y)$
X goes up and then back down	$\uparrow X; \downarrow X$

Properties:

$$\models\ (\uparrow X \wedge \uparrow Y) \supset [\uparrow(X \otimes Y) \wedge \uparrow(X \oslash Y)]$$

If two bit signals rise, so do their bit-and and bit-or.

$$\models\ \downarrow X \equiv \uparrow \ominus X$$

A bit signal falls exactly if its complement rises.

$$\models\ [\uparrow X \wedge beg(Y = 0) \wedge (Y\ dep\ X)] \supset \uparrow(X \oslash Y)$$

If X rises and in addition Y initially equals 0 and depends on X, then the bit-or of X and Y also rises.

These operators can be extended to include quantitative information specifying minimum periods of stability before and after the transitions. For example, timing details can be added to the operator $\uparrow$:

$$\uparrow^{m,n} X \equiv [(X \approx 0 \wedge len \geq m);\ skip;\ (X \approx 1 \wedge len \geq n)]$$

A negative pulse with quantitative information can be described as shown below:

$$\downarrow\uparrow^{l,m,n} X \equiv$$
$$[(X \approx 1 \wedge len \geq l);\ skip;$$
$$(X \approx 0 \wedge len \geq m);\ skip;\ (X \approx 1 \wedge len \geq n)]$$

4.9 Temporal Assignment

The formula $X \rightarrow Y$ is true for an interval if X's initial value equals Y's final value.

Example:

Concept	*Formula*
Z ends up with the complement of Y's initial value	$\ominus Y \rightarrow Z$

Properties:

$$\models \quad stb\, X \supset (X \rightarrow X)$$

A stable signal's initial and final values agree.

$$\models \quad [(X \rightarrow Y);(Y \rightarrow Z)] \supset (X \rightarrow Z)$$

If Y gets X's value and then Z gets Y's, the net result is that Z gets X's initial value.

$$\models \quad (\ominus X \rightarrow Y) \equiv (X \rightarrow \ominus Y)$$

The bit signal Y gets the complement of X's value exactly if Y's complement gets the value of X itself.

$$\models \quad [(\ominus Z \rightarrow Z);(\ominus Z \rightarrow Z)] \supset (Z \rightarrow Z)$$

If a signal is twice complemented, it ends up with its original value.

4.10 Repetition

An interval can be broken up into an arbitrary number of successive subintervals, each satisfying some formula p. The construct p^n has the same meaning

as

$$\underbrace{p; \quad . \quad . \quad . \quad ;p}_{n \text{ times}}$$

For the case of $n = 0$, an interval σ satisfies the operator exactly if σ's length is 0.

Examples:

Concept	*Formula*
The signal Y twice goes up and down	$(\uparrow Y; \downarrow Y)^2$
Z is complemented n times	$(\ominus Z \rightarrow Z)^n$

Properties:

$$\models \quad (\ominus X \rightarrow X)^n \supset [X \oplus (n \bmod 2) \rightarrow X]$$

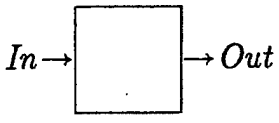
After a series of n complements, X ends up with the initial value of the exclusive-or of X and $(n \bmod 2)$. For instance, if n is even, X ends up unchanged.

$$\models \quad (p^m)^n \equiv p^{mn}$$

If a formula p is repeated m times within a further repetition of n cycles, the net result is the same as iterating p a total of mn times.

§5 Simple Delay Element

Delay is of fundamental importance in digital systems. One of the simplest types of delay elements has the following structure:



n-unit delay

Here *In* is the input bit signal and *Out* is the associated output. The variable n is a fixed natural number indicating the time delay between a value appearing on the input and later on the output. The following statement uses intervals to characterize this behavior:

> *In every subinterval of length exactly n units, the initial input value agrees with the final output one.*

The next predicate *Delay* captures the required interaction:

$$Delay(In, Out, n) \quad \equiv_{\text{def}} \quad \boxdot[(len = n) \supset (In \rightarrow Out)]$$

Properties:

- *A delay element is also a delay element in every subinterval:*

$$\models \quad Delay(In, Out, n) \supset \boxdot Delay(In, Out, n)$$

- *Zero delay is the same as temporal equality:*

$$\models \quad Delay(In, Out, 0) \equiv (In \approx Out)$$

- *Two connected delays result in a combined delay:*

$$\models \quad [Delay(In1, Out1, n1) \wedge Delay(In2, Out2, n2) \wedge Out1 \approx In2] \supset Delay(In1, Out2, n1 + n2)$$

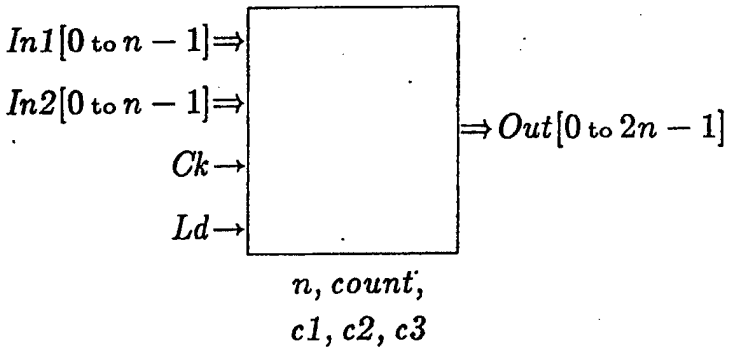
Note that the total delay $n1 + n2$ is the sum of the delays $n1$ and $n2$.

An alternative delay model can be given containing an internal state of $n + 1$ bits that are shifted as in a queue. The two distinct models are formally equivalent as can be expressed and demonstrated with the temporal logic.

The object *len* is used in the definition of *Delay* to measure time. Actually, other metrics seem possible. For example, some variable might represent the number of clock cycles or machine instructions executed in each interval. The properties of delay remain basically the same.

§6 Multiplication Circuit

The hardware multiplier considered here is motivated by one discussed in Wagner's work on hardware verification [20]. The desired device behavior is first described followed by a look at implementation techniques. The multiplier has the following general structure:



The circuit accepts two numbers and after a given number of clock cycles yields the product. The numbers are represented as unsigned n-bit vectors *In1* and *In2* while the output *Out* is a $2n$-bit one. In addition to the vector inputs and output, there are two input bits *Ck* and *Ld* which control operation. The signal *Ck* serves as the clock input and *Ld* initiates the loading of the vectors to be multiplied. The field *count* tells how many clock cycles are required. The values *c1*, *c2* and *c3* are timing coefficients used in the behavioral description.

6.1 Additional Notation

Because the multiplier deals with numbers and their representation as bit vectors, it is convenient to introduce some extra notation before giving the device's formal description:

- Subscripts on a vector $V = \langle v_0, \ldots, v_n \rangle$ normally range from 0 on the left to n on the right. The construct $V[i]$ follows this style. However, to simplify reasoning about the correspondence between a bit vector and its numerical equivalent, a slightly different convention is adapted. The alternative notation $V\{i\}$ indexes V from the *right* with the right-most element having subscript 0. For example:

$$\underset{\uparrow}{\langle 1, 0, \bot \rangle\{0\} = \bot}, \quad \underset{\uparrow}{\langle 1, 0, \bot \rangle\{1\} = 0}, \quad \underset{\uparrow}{\langle 1, 0, \bot \rangle\{2\} = 1}$$

  For a vector V and $i \geq j$, the expression $V\{i \text{ to } j\}$ forms a new vector out of the elements indexed from i down to j. If $i < j$, the empty vector is returned. For example,

$$\langle 0, 9, \bot, 2 \rangle\{3 \text{ to } 1\} = \langle 0, 9, \bot \rangle, \quad \langle 0, 1 \rangle\{0 \text{ to } 0\} = \langle 1 \rangle, \quad \langle \bot, 1, 0, 1 \rangle\{1 \text{ to } 2\} = \langle\rangle$$

- The predicate *def* X is true for a scalar value X if X does not equal $\bot$. In this case, X is *defined*. A vector is defined exactly if all its components are. For

example, the following values are defined:

$$0, \quad 3, \quad \langle 1, 0 \rangle, \quad \langle\rangle$$

The values given below are not defined:

$$\perp, \quad \langle \perp, \perp \rangle, \quad \langle \perp, 0 \rangle$$

- The function *nval* converts a bit vector to its unsigned numeric value. For example,

$$nval(\langle 0, 1, 1 \rangle) = 3, \qquad nval(\langle 1, 1, 0, 0 \rangle) = 12$$

If any element of the vector is undefined, *nval* yields $\perp$ as the result. Thus,

$$nval(\langle 1, \perp, 0, 0, 1 \rangle) = \perp$$

6.2 Overview of Description Techniques

In what follows, the predicate $Multiplier(M)$ specifies that desired behavior of a multiplication circuit. The device's various inputs, outputs and timing coefficients are represented as fields of the single parameter M. An iterative, timing-independent multiplication algorithm is then presented which computes a product by a series of successive additions. Later, the predicate $Implementation(H)$ characterizes a device which computes sums and in fact has the algorithm's steps embedded within it. A logical implication is then given, showing how $Implementation(H)$ realizes $Multiplier(M)$.

6.3 Formal Specification of Multiplication Circuit

The predicate *Multiplier* formally characterizes the circuit's desired structure and behavior. The single parameter M is a tuple representing the multiplier. For example, the expression $M.Ck$ equals the clock input. The predicate's definition makes reference to other predicates given later:

$$\begin{aligned} &Multiplier(M) \quad \equiv_{\text{def}} \\ &\quad MultStructure(M) \\ &\quad \wedge \ \boxdot_a \, Calculate(M) \end{aligned}$$

The predicate *MultStructure* presents M's fields. The predicate *Calculate* gives the control sequencing required to perform a multiplication. The operator $\boxdot_a$ indicates that *Calculate* must be true in all subintervals.

Definition of *MultStructure*:

The definition below of *MultStructure* contains information on the physical structure of the multiplier. Variables starting in upper case represent signals while lower-case ones are constant. Labels such as "Inputs:" are comments included to classify the various circuit fields. For example, $M.In1$ is an input bit vector.

$$\begin{array}{l} MultStructure(M) \quad \equiv_{\text{def}} \\ \quad \text{Inputs:} \\ \qquad (Ck, Ld)\text{: } Bit, \\ \qquad In1\{n-1 \text{ to } 0\}\text{: } Bit, \\ \qquad In2\{n-1 \text{ to } 0\}\text{: } Bit \\ \quad \text{Outputs:} \\ \qquad Out\{2n-1 \text{ to } 0\}\text{: } Bit \\ \quad \text{Parameters:} \\ \qquad n\text{: } nat, \\ \qquad count\text{: } nat, \\ \qquad c1, c2, c3\text{: } time \end{array}$$

For brevity, the prefix "$M.$" is omitted when a field is referenced below.

Definition of *Calculate*:

If the inputs behave as specified by the predicate *Control*, the output *Out* ends up with the product of the initial values of *In1* and *In2*. Recall that the function *nval* converts a bit sequence to the corresponding numerical value.

$$\begin{array}{l} Calculate(M) \quad \equiv_{\text{def}} \\ \quad Control(M) \supset \\ \qquad \big[nval(In1) \cdot nval(In2)\big] \rightarrow nval(Out) \end{array}$$

Definition of *Control*:

The predicate *Control* describes the required sequencing of the inputs so that a multiplication takes place. The computation first loads the circuit and then keeps the load line inactive while the clock is cycled.

$$Control(M) \quad \equiv_{\text{def}} \quad Load(M);\ ([Ld \approx 0] \ \wedge \ Cycling(M))$$

Definition of *Load*:

Loading is done as indicated by the predicate *Load*. The clock is cycled as given by the predicate *SingleCycle*. The control signal *Ld* starts with the value 1 and together with the other inputs *In1* and *In2* remains initially stable as long as the clock input *Ck* does.

$$Load(M) \quad \equiv_{\text{def}} \quad SingleCycle(M) \wedge beg(Ld = 1) \wedge \langle Ld, In1, In2 \rangle \; dep \; Ck$$

Definition of *SingleCycle*:

An individual clock cycle consists of a negative pulse:

$$SingleCycle(M) \quad \equiv_{\text{def}} \quad \downarrow\uparrow^{c1,c2,c3} Ck$$

The clock signal falls from 1 to 0 and then rises back to 1. The three times given indicate the minimum widths of the levels during which the clock is stable.

Definition of *Cycling*:

The overall cycling of the clock is as follows:

$$Cycling(M) \quad \equiv_{\text{def}} \quad (SingleCycle(M))^{count}$$

A total of *count* individual cycles must be performed one after the other, where each is a negative pulse satisfying the predicate *SingleCycle*.

Variants of the Specification

The predicate *Multiplier* does not represent the only way to describe the multiplier circuit. Alternative approaches based on an internal state can be shown to be formally equivalent to the one given here. A useful extension to this description specifies that once the output is computed, it remains stable as long as the control inputs do. If desired, additional quantitative timing details can readily be included.

6.4 Development of Multiplication Algorithm

The specification predicate *Multiplier* intentionally makes no reference to any particular technique for multiplying. Since the process of multiplication does not

generally depend on any specific circuit timing, it is natural to separate algorithmic issues from other implementation details. The temporal logic now serves as a basis for deriving a suitable circuit-independent algorithm for determining the product and in the next section as a means for describing hardware that realizes this method. The synthesis process can be viewed as a proof in reverse, starting with the goal and ending with the necessary assumptions to achieve it.

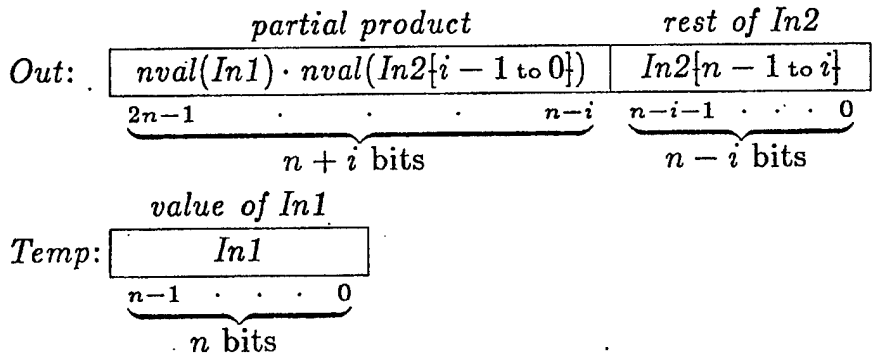
The aim here is to obtain an algorithm describing some way for doing the multiplication. The variables n, *In1*, *In2* and *Out* are represented as fields of a variable A. The predicate *Goal* below specifies the desired result:

$$\begin{aligned} Goal(A) \quad &\equiv_{\mathbf{def}} \\ &beg(def\ In1 \ \wedge \ def\ In2) \supset \\ &\qquad [nval(In1) \cdot nval(In2)] \rightarrow nval(Out) \end{aligned}$$

If the data inputs *In1* and *In2* are initially defined, the output *Out* should end up with their product. The presentation given here reduces the problem of multiplying the two n-bit vectors to that of using repeated additions to determine successively larger partial products. The algorithm consists of initialization followed by n successive iterations. After i iterations of the loop, for $i \leq n$, the initial product of *In1* and the least significant i bits of *In2*, that is,

$$nval(In1) \cdot nval(In2\{i-1 \text{ to } 0\})$$

is computed and available in the upper $n+i$ bits of *Out*. Neither *In1* nor *In2* is guaranteed to remain stable once initialization is complete. However, their initial values must be used throughout the calculation. The lower $n-i$ bits of *Out* hold the unexamined bits of *In2* (i.e., $In2\{n-1 \text{ to } i\}$). In addition, an extra n-bit variable *Temp* is introduced in order to remember the original value of *In1*. The following figure informally depicts the situation after i steps:



After n steps, *Out* equals the desired $2n$-bit multiplication result.

The predicate *Assert* below precisely specifies this behavior over i iterations for $i \leq n$. Note that both inputs *In1* and *In2* must be initially defined for the operations to properly take place.

$$
\begin{aligned}
&Assert(A, i) \quad \equiv_{\textbf{def}} \\
&\quad beg(def\ In1 \ \wedge \ def\ In2) \supset \\
&\qquad \left[nval(In1) \cdot nval(In2\{i-1 \text{ to } 0\})\right] \rightarrow nval(Out\{2n-1 \text{ to } n-i\}) \\
&\qquad\quad \wedge \ In2\{n-1 \text{ to } i\} \rightarrow Out\{n-i-1 \text{ to } 0\} \\
&\qquad\quad \wedge \ In1 \rightarrow Temp
\end{aligned}
$$

After n steps, the product must be computed. For $i = n$, *Assert* indeed observes this requirement:

$$Assert(A, n) \supset Goal(A) \qquad (*)$$

Expressed in the logic, the algorithm takes the following form:

$$Init(A); (Step(A))^n$$

In the next two subsections, the predicates *Init* and *Step* are given in detail. Both *Init* and *Step* are derived so as to maintain *Assert* after looping i times for any $i \leq n$:

$$\left[i \leq n \ \wedge \ Init(A); (Step(A))^i\right] \supset Assert(A, i) \qquad (**)$$

The properties $(*)$ and $(**)$ together ensure that n iterations of the loop calculate the product:

$$Init(A); \ (Step(A))^n \supset Goal(A)$$

Deriving the Predicate *Init*

The initialization requirement can be obtained by making sure *Init* satisfies *Assert* for $i = 0$:

$$Init(A) \supset Assert(A, 0)$$

Simplification of *Assert* yields the constraint

$$\begin{array}{l} \mathit{Init}(A) \supset \\ \quad \mathit{beg}(\mathit{def}\ \mathit{In1} \wedge \mathit{def}\ \mathit{In2}) \supset \\ \qquad 0 \rightarrow \mathit{nval}(\mathit{Out}\{2n-1 \text{ to } n\}) \\ \qquad\quad \wedge\ \mathit{In2} \rightarrow \mathit{Out}\{n-1 \text{ to } 0\} \\ \qquad\quad \wedge\ \mathit{In1} \rightarrow \mathit{Temp} \end{array}$$

This can be achieved by the definition

$$\begin{array}{l} \mathit{Init}(A) \quad \equiv_{\mathbf{def}} \\ \quad \mathit{beg}(\mathit{def}\ \mathit{In1} \wedge \mathit{def}\ \mathit{In2}) \supset \\ \qquad \langle 0, \ldots, 0 \rangle \rightarrow \mathit{Out}\{2n-1 \text{ to } n\} \\ \qquad\quad \wedge\ \mathit{In2} \rightarrow \mathit{Out}\{n-1 \text{ to } 0\} \\ \qquad\quad \wedge\ \mathit{In1} \rightarrow \mathit{Temp} \end{array}$$

Deriving the Predicate *Step*

The iteration step should be constructed so that after i iterations for any $i < n$, *Step* can inductively widen the scope of the assertion to $i+1$ increments:

$$[i < n \wedge \mathit{Assert}(A, i); \mathit{Step}(A)] \supset \mathit{Assert}(A, i+1)$$

Each step achieves this by selectively adding *Temp*'s n bits to *Out*, depending on *Out*'s least bit, $\mathit{Out}\{0\}$. Only the top n bits of *Out* are actual inputs for the sum. The top $n+1$ bits store the result. The remaining $n-1$ bits of *Out* are simply shifted right. For *Temp* the requirement reduces to the formula

$$\begin{array}{l} \mathit{Step}(A) \supset \\ \quad \mathit{beg}(\mathit{def}\ \mathit{Temp}) \supset (\mathit{Temp} \rightarrow \mathit{Temp}) \end{array}$$

This guarantees that *Temp* continues to remember the initial value of *In1*.

The constraint for *Out* is

$$\begin{array}{l} \mathit{Step}(A) \supset \\ \quad \mathit{beg}(\mathit{def}\ \mathit{Out} \wedge \mathit{def}\ \mathit{Temp}) \supset \\ \qquad [\mathit{nval}(\mathit{Out}\{2n-1 \text{ to } n\}) + \mathit{Out}\{0\} \cdot \mathit{nval}(\mathit{Temp})] \\ \qquad\quad \rightarrow \mathit{nval}(\mathit{Out}\{2n-1 \text{ to } n-1\}) \\ \qquad \wedge\ \mathit{Out}\{n-1 \text{ to } 1\} \rightarrow \mathit{Out}\{n-2 \text{ to } 0\} \end{array}$$

Thus the overall incremental step can be realized by the definition

$$
\begin{aligned}
&\mathit{Step}(A) \quad \equiv_{\text{def}} \\
&\quad \mathit{beg}(\mathit{def}\ \mathit{Out} \ \wedge\ \mathit{def}\ \mathit{Temp}) \supset \\
&\qquad \big[\mathit{nval}(\mathit{Out}\{2n-1 \text{ to } n\}) + \mathit{Out}\{0\} \cdot \mathit{nval}(\mathit{Temp})\big] \\
&\qquad\quad \rightarrow \mathit{nval}(\mathit{Out}\{2n-1 \text{ to } n-1\}) \\
&\qquad\ \wedge\ \mathit{Out}\{n-1 \text{ to } 1\} \rightarrow \mathit{Out}\{n-2 \text{ to } 0\} \\
&\qquad\ \wedge\ \mathit{Temp} \rightarrow \mathit{Temp}
\end{aligned}
$$

6.5 Description of Implementation

The circuit specified below performs the iterative algorithm just given. The definition includes relevant timing information and is broken down into parts describing the implementation's physical structure and behavior. The primary predicate *Implementation* overviews operation. The device's fields are shown by *ImpStructure*. The predicate *LoadPhase* specifies device operation for initially loading the inputs. Once this is achieved, the predicate *MultPhase* indicates how to perform the individual multiplication steps.

$$
\begin{aligned}
&\mathit{Implementation}(H) \quad \equiv_{\text{def}} \\
&\quad \mathit{ImpStructure}(H) \\
&\qquad \wedge\ \boxdot(\mathit{LoadPhase}(H) \ \wedge\ \mathit{MultPhase}(H))
\end{aligned}
$$

Definition of *ImpStructure*:

The structure of the implementation differs from that of the original specification by the addition of the internal state *Temp* for maintaining the value of *In1* and by the omission of a *count* field giving the required number of clock cycles for computing a product.

$ImpStructure(H) \quad \equiv_{\text{def}}$

Inputs:

(Ck, Ld): Bit,
$In1\{n-1 \text{ to } 0\}$: Bit,
$In2\{n-1 \text{ to } 0\}$: Bit

Outputs:

$Out\{2n-1 \text{ to } 0\}$: Bit

Internal:

$Temp\{n-1 \text{ to } 0\}$: Bit

Parameters:

n: nat,
$c1, c2, c3$: $time$

Definition of *LoadPhase*:

The body of *LoadPhase* specifies how to load the inputs as described in the algorithm:

$$LoadPhase(H) \quad \equiv_{\text{def}}$$
$$Load(H) \supset Init(H)$$

The predicate *Load* gives the required loading sequence for the circuit inputs. The predicate *Init* refers to algorithm's initialization predicate. The definition of *Load* is identical to that of its namesake in *Multiplier*:

$$Load(H) \quad \equiv_{\text{def}}$$
$$SingleCycle(H) \wedge beg(Ld = 1) \wedge \langle Ld, In1, In2 \rangle \; dep \; Ck$$

Individual clock cycles are also defined as in *Multiplier*:

$$SingleCycle(H) \quad \equiv_{\text{def}} \quad \downarrow\uparrow^{c1,c2,c3} Ck$$

Definition of *MultPhase*:

When the load signal is inactive at 0, the circuit can be clocked to perform a single iteration. The algorithm's predicate *Step* takes place over two clock cycles.

$$MultPhase(H) \quad \equiv_{\text{def}}$$
$$[Ld \approx 0 \wedge (SingleCycle(H))^2] \supset Step(H)$$

Implementation Theorem

The correspondence between the implementation *Implementation* and the original multiplier device specification *Multiplier* is now given by the theorem

$$\models \quad \mathit{Implementation}(H) \supset \mathit{Multiplier}(M)$$

where the mapping from H's fields to M's is

$$\begin{array}{lcll} M.field & \approx & H.field, & \text{for the fields } In1,\ In2 \text{ and } Out \\ M.n & = & H.n & \\ M.count & = & 2H.n & \\ M.field & = & H.field, & \text{for the fields } c1,\ c2 \text{ and } c3 \end{array}$$

The value of $M.count$ corresponds to the $2n$ clock cycles needed for doing the iterative computation.

The behavioral description *Implementation* can itself be realized by some even lower-level specification containing further details about the timing and using a still more concrete algorithm. For example, the iterative steps are decomposible into separate adds and shifts. If desired, the development ultimately examines such things as propagation through gates.

§7 Conclusion and Future Plans

Compared with conventional hardware description languages, the approach used here permits direct reasoning about signal, device and algorithm behavior at various levels of detail. In addition, the concepts relating specifications with implementations and hardware with register-transfer operations can be rigorously expressed within a single mathematical framework. A disadvantage arises from the inability to directly execute arbitrary descriptions.

Standard temporal logics and other such notations have not been designed to concisely handle the kinds of quantitative timing properties and signal transitions found in the examples considered. The intervals of time provide a unifying means for presenting various features.

The material presented only scratches the formalism's surface. Halpern et al. [6] and Moszkowski [14] cover many details of the logic, describing and comparing devices ranging from delay elements up to the Am2901 ALU bit slice developed by Advanced Micro Devices, Inc. Future work will examine microprocessors, buses and protocols, DMA, firmware and instruction sets, as well as the combined semantics of hardware and software.

§8 Acknowledgements

Many thanks go to the following people for discussions and suggestions concerning the notation's readability (or lack thereof): Patrick Barkhordarian, Russell Greiner, Kevin Karplus, Amy Lansky, Yoni Malachi, Fumihiro Maruyama, Gudrun Polak, Alex Strong, Carolyn Talcott and Pierre Wolper. Professors John McCarthy and Zohar Manna gave much support and guidance as this research developed. Joseph Halpern provided valuable insights with regard to the logic's theoretical complexity. If it had not been for my friends at Siemens AG and the Polish Academy of Sciences, it is unlikely I would have undertaken this investigation. Late-night trans-Atlantic discussions with Mike Gordon helped provide a sense of intrigue. Highest-quality chocolate and enthusiasm were always available from the Trischlers.

References

1. M. R. Barbacci. Instruction Set Processor Specifications (ISPS): The notation and its applications. *IEEE Transactions on Computers C–30*, 1 (January 1981), pages 24–40.
2. G. V. Bochmann. Hardware specification with temporal logic: An example. *IEEE Transactions on Computers C–31*, 3 (March 1982), pages 223–231.
3. H. Eveking. The application of Conlan assertions to the correct description of hardware. Proceedings of the IFIP TC–10 Fifth International Conference on Computer Hardware Description Languages and their Applications, Kaiserslautern, West Germany, September, 1981, pages 37–50.
4. M. Gordon. Register transfer systems and their behavior. Proceedings of the IFIP TC–10 Fifth International Conference on Computer Hardware Description Languages and their Applications, Kaiserslautern, West Germany, September, 1981, pages 23–36.
5. B. T. Hailpern and S. Owicki. Verifying network protocols using temporal logic. Tech. Rept. 192, Computer Systems Laboratory, Stanford University, June, 1980.
6. J. Halpern, Z. Manna, and B. Moszkowski. A hardware semantics based on temporal intervals, forthcoming.
7. D. Harel. *First-Order Dynamic Logic*. Springer-Verlag, Berlin, 1979. Number 68 in the series *Lecture Notes in Computer Science*.
8. L. Lamport. Specifying concurrent program modules. Opus 60, Computer Science Laboratory, SRI International, June, 1981.
9. S. Leinwand and T. Lamdan. Algebraic analysis of nondeterministic behavior. Proceedings of the 17-th Design Automation Conference, Minneapolis, June, 1980, pages 483–493.

10. T. M. McWilliams. Verification of timing constraints on large digital systems. Proceedings of the 17-th Design Automation Conference, Minneapolis, June, 1980, pages 139–147.
11. Y. Malachi and S. S. Owicki. Temporal specifications of self-timed systems. VLSI Systems and Computations, Rockville, Maryland, 1981, pages 203–212. This book is the proceedings of a conference held at the Carnegie-Mellon University in October, 1981.
12. Z. Manna and A. Pnueli. The modal logic of programs. Tech. Rept. STAN–CS–79–751, Department of Computer Science, Stanford University, September, 1979.
13. P. Meinen. Formal semantic description of register transfer language elements and mechanized simulator construction. Proceedings of the 4-th International Symposium on Computer Hardware Description Languages, Palo Alto, California, October, 1979, pages 69–74.
14. B. Moszkowski. *Reasoning about Digital Circuits.* Ph.D. Thesis, Dept. of Computer Science, Stanford University, forthcoming.
15. A. C. Parker and J. J. Wallace. SLIDE: An I/O hardware description language. *IEEE Transactions on Computers C–30,* 6 (June 1981), pages 423–439.
16. D. A. Patterson. Strum: Structured microprogram development system for correct firmware. *IEEE Transactions on Computers C–25,* 10 (October 1976), pages 974–985.
17. N. Rescher and A. Urquart. *Temporal Logic.* Springer-Verlag, New York, 1971.
18. R. L. Schwartz and P. M. Melliar-Smith. Temporal logic specification of distributed systems. Proceedings of the Second International Conference on Distributed Computing Systems, Paris, France, April, 1981, pages 446–454.
19. S. Y. H. Su, C. Huang and P. Y. K. Fu. A new multi-level hardware design language (LALSD II) and translator. Proceedings of the IFIP TC–10 Fifth International Conference on Computer Hardware Description Languages and their Applications, Kaiserslautern, West Germany, September, 1981, pages 155–169.
20. T. Wagner. *Hardware Verification.* Ph.D. Thesis, Dept. of Computer Science, Stanford University, September 1977.

Reproduced by NTIS

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Technical Information Service
Springfield, VA 22161 (703) 487-4650

PB96150826

BA

BIN: M72 04-25-97
INVOICE: 424259
SHIPTO: 1*89997
PAYMENT: CSH*CPDAG