Specification and Verification of Gate-Level VHDL Models of Synchronous and Asynchronous Circuits

David M. Russinoff
Computational Logic, Inc., Austin, Texas

Contract NAS1-18878
January 1995
Abstract

We present a mathematical definition of a hardware description language (HDL) that admits a semantics-preserving translation to a subset of VHDL. Our HDL includes the basic VHDL propagation delay mechanisms and gate-level circuit descriptions. We also develop formal procedures for deriving and verifying concise behavioral specifications of combinational and sequential devices. The HDL and the specification procedures have been formally encoded in the computational logic of Boyer and Moore, which provides a LISP implementation as well as a facility for mechanical proof-checking. As an application, we design, specify, and verify a circuit that achieves asynchronous communication by means of the biphase mark protocol.
## Contents

1 Introduction
   1.1 Hardware Modeling .................................. 1
   1.2 Behavioral Specifications ............................ 2
   1.3 Asynchronous Communication ......................... 3
   1.4 Nqthm Formalization ................................ 3

2 Definition of the Language
   2.1 S-expressions ....................................... 4
   2.2 Waveforms ........................................... 4
   2.3 Behavioral Modules .................................. 5
   2.4 Structural Modules .................................. 8
   2.5 Simulation .......................................... 10

3 Specification of Synchronous Circuits
   3.1 Combinational Modules ............................... 17
   3.2 Sequential Modules .................................. 17
   3.3 Sequential Values ................................... 20
   3.4 Behavior of dff ...................................... 21
   3.5 Parameters .......................................... 24
   3.6 The Main Theorem ................................... 25

4 Asynchronous Communication
   4.1 Smooth and Quasi-Smooth Waveforms ................. 30
   4.2 Describing Output as Input .......................... 30
   4.3 Eliminating Metastability ............................ 31
   4.4 The Main Theorem .................................... 32

5 Biphasic Mark
   5.1 Sending ............................................. 37
   5.2 Receiving .......................................... 37
   5.3 Moore’s Theorem ..................................... 38
   5.4 Basic Components ................................... 39
   5.5 The Sender .......................................... 39
   5.6 The Receiver ....................................... 43
   5.7 The Main Theorem .................................... 46

6 NASA’s Reliable Computing Platform ................... 51

Appendix: Nqthm Formalization ............................ 54

A Language Definition .................................... 54

B Properties of the Simulator ............................. 68

C Synchronous Sequential Circuits ........................ 81
1 Introduction

NASA Langley Research Center has conducted a research program in formal methods, focusing on the development of a practical verification methodology for fault-tolerant digital flight-control systems. Computational Logic, Inc. (CLI) is one of several organizations that have participated in this program. The first phase of the program addressed the application of formal methods to various key design problems. During this phase, CLI produced results in three areas:

(1) The formal design and verification of a circuit that achieves Byzantine agreement among four synchronous processors [1];

(2) The mechanical verification of the Interactive Convergence clock synchronization algorithm [22];

(3) The formalization of the Biphase Mark protocol for asynchronous communication [15].

The second phase of the program is concerned with exploring the integration of these results in the design of a verified reliable computing platform (RCP) [9, 10] for real-time control. This paper is a report on CLI's effort during this phase.

1.1 Hardware Modeling

A prerequisite for the realization of NASA's goals is a hardware description language (HDL) that is both (a) amenable to formal verification and (b) suitable for representing asynchronous systems of communicating processors. Much of our effort has been devoted to the development of a language that meets these requirements.

Our previous research in hardware modeling and verification has been based on an HDL developed at CLI by Brock and Hunt [5]. The utility of the Brock-Hunt HDL as a verification tool, as demonstrated in the verification of the FM3001 microprocessor [4], stems from the simplicity of its semantics. All circuits designed in this language are assumed to be driven by an implicit global clock. Simulation of a circuit amounts to a computation of a sequence of states corresponding to clock cycles. Thus, no explicit representation of time or propagation delays is provided, so that the class of circuits that can be satisfactorily modeled is limited. In particular, the language is unsuitable for any application involving asynchrony.

Commercial event-driven simulation languages provide for a broader range of hardware behaviors. VHDL [11], in particular, has gained wide acceptance in the hardware design community as a validation tool. Since the limitations of simulation as a method of validation are well known, a formal verification system based on VHDL would have clear practical value. Unfortunately, like most programming languages in common use, the semantics of VHDL are complicated and obscure. There have been various attempts to formalize these semantics [2, 8, 19, 21], but none of these have provided an effective verification methodology.

We have undertaken, therefore, to identify a core subset of VHDL that is small enough to admit a clear and simple semantic definition, providing for correctness proofs of comprehensive behavioral specifications, but extensive enough to provide realistic
gate-level descriptions of the circuits involved in our intended application. Thus, we have avoided complicated language constructs and focused on the VHDL models of time, signal behavior, propagation delay, and event-driven simulation.

The definition of our language is presented in Section 2. Its syntax, based on the S-expressions of LISP (subsection 2.1), is more abstract and amenable to direct formal analysis than the standard VHDL syntax [11]. The correspondence between the two is straightforward—a simple translator from our language to VHDL is described elsewhere [12]. Here, we concentrate on a mathematical treatment of the abstract language. This begins in Subsection 2.2, where we present the notions of time and waveform, on which the semantics of the language are based. We also define two waveform transformations that embody the main propagation delay modes of VHDL, transport and inertial, and derive their fundamental properties.

In Subsection 2.3, we describe the form and execution of behavioral modules, which are used to model gates and also to specify abstractly the behavior of circuits. Subsection 2.4 discusses structural modules, which provide hierarchical descriptions of circuits in terms of connections among their components. For the purpose of illustration, we exhibit the actual VHDL code generated by the translator for modules of both types.

The semantics of the language are given by an interpreter function, sim, which produces a list of waveforms that represent the output generated by a module in response to a given list of input waveforms. The definition of sim is presented in Subsection 2.5, along with a number of basic results pertaining to its behavior.

1.2 Behavioral Specifications

During the course of the design process, a typical hardware device is modeled at various levels of abstraction. An initial abstract model, derived from a given behavioral specification, is gradually refined to produce a concrete model, such as a network of gates, which is more amenable to implementation. A design is validated by demonstrating the equivalence of these representations.

This is most commonly effected through simulation. In VHDL, a circuit component may be associated with various alternative architectures, which describe the component at different levels of abstraction. The equivalence of architectures may be confirmed through comparative simulations. Once a sufficiently low-level VHDL architecture has been derived and validated in this manner, it may be implemented directly.

We propose to replace simulation with formal verification. In our VHDL subset, circuit components are represented concretely at the gate level. In Section 3, we shall describe a methodology for deriving abstract behavioral specifications and proving that they are satisfied by these gate-level models.

In Subsection 3.1, we consider the relatively simple class of combinational circuits, i.e., circuits that are free of cyclic paths. Each output of such a circuit is naturally associated with a certain Boolean function of the inputs. This association is commonly stated as follows: the value of an output at any time may be computed by applying the associated function to the current input values. Obviously, this description is valid only with respect to hardware models that ignore propagation delay. We shall derive a more accurate specification of combinational circuits and verify its validity in the context of our model.
The analysis of sequential circuits is considerably more complicated. While the abstract sequential machine model is well understood, its precise relationship with the actual behavior of the hardware that it is intended to describe is not. The sequential machine characterization is traditionally based on the extravagant assumption that signal values may change only at discrete points occurring at regular time intervals. This allows the behavior of a signal to be represented abstractly as a sequence of values. The value of an output over a given interval is then expressed as a function of the sequence of past input values. Of course, the underlying model again must disregard propagation delay. This approximation seems questionable, since the functionality of the basic state-holding elements generally depends critically on the presence of delays.

In Subsection 3.2, we define a class of sequential circuits that may be characterized as synchronous resettable rising-edge-triggered devices. The basic memory element employed in their construction is a resettable clocked d-flip-flop, composed of nand gates, described in Subsection 3.3. In Subsections 3.4–3.5, we establish a procedure for deriving high-level sequential machine descriptions for the class of circuits. In Subsection 3.6, we prove a theorem that gives a precise statement of the relationship between the sequential machine description of a circuit and its behavior as defined by our gate-level semantics.

1.3 Asynchronous Communication

The utility of our approach with respect to the NASA RCP depends on our ability to model asynchronous communication between individually synchronous processors. This problem is addressed in Section 4. We present a solution based on Moore's model of asynchrony [15]. After reviewing this model, we prove a theorem that demonstrates its applicability to a class of circuits defined in our language. Each of these circuits consists of a pair of sequential circuits that are driven by independent clocks of approximately equal periods. They communicate with the aid of a latch that serves to smooth the sender's output, allowing it to be read by the receiver.

In Section 5, we present a concrete definition of such a circuit that achieves asynchronous communication by means of the well known biphase mark protocol [18]. The circuit design and the proof of its correctness are both based on [15].

1.4 Nqthm Formalization

The decision to base our language on S-expressions was motivated by our desire to support its analysis with the use of the Nqthm system of Boyer and Moore [3]. Nqthm is based on a constructive formal logic for which the intended model is the domain of S-expressions. Thus, there is a correspondence between the formulas of this logic and informal propositions about S-expressions. A user of the system may extend the logic by adding axioms that correspond to definitions of computable functions over this domain.

Mechanical support for the Nqthm logic is provided by a LISP implementation that includes (1) an evaluator that computes values of functions defined in the logic, and (2) a theorem prover that may be used to derive logical consequences of the axioms. Since these theorems may be interpreted as propositions about functions of S-expressions, the prover may be used to verify (formally and mechanically) the correctness of properties of these functions that have been derived by traditional (informal) mathematical methods.
All of the functions involved in the construction of our language, which we describe informally, meet the computability requirement for encoding as Nqthm definitions [3]. In fact, we have developed an Nqthm theory, presented in Appendix A, that formalizes these functions, including the module recognizers that form the syntax of the language and the interpreter that constitutes its semantics. Thus, we have a complete LISP implementation of our language, provided by the Nqthm evaluator.

Moreover, all of our results, which are justified by informal (but mathematically rigorous) proofs, correspond in a natural way to Nqthm formulas. Thus, these proofs could, in principle, be checked mechanically by the Nqthm prover, thereby increasing our confidence in their validity at the expense of some effort. At the time of this writing, mechanical proofs have been generated for most of the results of Section 2 (see Appendix B), as well as most of the results pertaining to specific circuits, including the components of the biphase mark implementation (Appendix C).

Another benefit of the Nqthm formalization is that it provides a basis for a LISP implementation of the translator from our syntax to that of VHDL [12]. This potentially allows commercial VHDL synthesis tools to be used to implement our programs in silicon. As another application of more immediate interest, we have actually executed (the translations of) many of our programs using the Vantage VHDL simulator. For the simulations that we have tested, which include all of those described herein, the Vantage results were identical to those produced by our LISP-based interpreter. Since the official description of VHDL [11] is often ambiguous, this offers useful evidence that we have achieved our goal of semantically capturing the VHDL subset in which we are interested.

2 Definition of the Language

2.1 S-expressions

Along with the set \( N \) of natural numbers, we posit a set \( B = \{ T, F \} \) and an infinite set \( L \), the elements of which are called Boolean and literal atoms, respectively. These three sets are assumed to be pairwise disjoint, and any element of their union is called an atom. We further assume that no atom is an ordered pair of atoms, and we recursively define an S-expression to be an atom or an ordered pair of S-expressions. \( S \) denotes the set of all S-expressions. Three basic operations on \( S \) are defined: If \( z = (x, y) \in S \times S \), then \( \text{car}(z) = x \), \( \text{cdr}(z) = y \), and \( \text{cons}(x, y) = z \).

We also assume the existence of various distinct literal atoms, which we shall mention as we proceed. Among these is the atom INFINITY. We define a generalized number to be an atom that is either INFINITY or an element of \( N \). Both the order relation and the addition operation on \( N \) are extended to the set of generalized numbers in the natural manner: for any \( n \in N \), \( n < \text{INFINITY} \) and \( n + \text{INFINITY} = \text{INFINITY} + n = \text{INFINITY} \).

A list is an S-expression that is either the literal atom \( \text{NIL} \) or an ordered pair \( z \in S \times S \) such that \( \text{cdr}(z) \) is a list. The list \( \text{NIL} \) is denoted alternatively as \( () \), and a non-NIL list \( z \) is denoted as \( (a_1 \ldots a_n) \), where \( a_1 = \text{car}(z) \) and \( (a_2 \ldots a_n) \) denotes \( \text{cdr}(z) \). In this case, \( n \) is the length of \( z \), and \( a_1, \ldots, a_n \) are its members. For \( 1 \leq i \leq n \), \( \text{nth}(i, z) \) is defined to be \( a_i \). A list is a bit vector if each of its members is a Boolean atom.

A function \( f : B^n \rightarrow B \) is an \( n \)-ary Boolean function. The following Boolean func-
tions are called elementary: the 0-ary functions to and fo, with values T and \( \mathcal{F} \), respectively; the unary function not1; the binary functions and2, or2, nand2, nor2, xor2; the ternary functions and3, or3, nand3, nor3, xor3; the quaternary functions and4, or4, nand4, nor4, and xor4; and the quinary functions and5, or5, nand5, nor5, and xor5. The definitions of these functions are assumed to be understood.

For the purpose of encoding Boolean function calls, we also assume that each elementary Boolean function \( f \) is associated with a unique literal atom \( f \) that is denoted with the same name as \( f \). Thus, the function not1 is associated with the literal atom \( \not 1 = \overline{1} \). We define a Boolean term over a list \( L \) of distinct literal atoms to be an S-expression that is either (a) a member of \( L \), or (b) a list \((f \tau_1 \ldots \tau_n)\), where \( f \) is an \( n \)-ary elementary Boolean function and each \( \tau_i \) is a Boolean term over \( L \).

Let \( L = (s_1 \ldots s_k) \) be a list of distinct literal atoms and let \( V = (v_1 \ldots v_k) \) be a bit vector. Then \( \text{pairlist}(L, V) \) is the list \( \mathcal{A} = ((s_1, v_1) \ldots (s_k, v_k)) \), which is called an association list. If \( r \) is a Boolean term over \( L \), then we define \( \text{eval}(r, \mathcal{A}) \) to be (a) \( v_i \), if \( r = s_i \), or (b) \( f(\text{eval}(r_1, \mathcal{A}), \ldots, \text{eval}(r_n, \mathcal{A})) \), if \( r = (f \tau_1 \ldots \tau_n) \).

### 2.2 Waveforms

Let \( T \) be the quotient set determined by the equivalence relation on \( N \times N \times N \) that identifies each \( n \in N \) with the pair \( (n, 0) \in N \times N \). An element of \( T \) is called a time object. Thus, any element of \( N \) or \( N \times N \) denotes a unique time object, with the understanding that for \( n \in N \), \( n \) and \((n, 0)\) denote the same object.

The motivation for this ordered-pair model of time is the need to provide records of the behavior of zero-delay devices. The components of a time object \((n, k)\) may be interpreted as follows: \( n \) represents the number of time units, which we arbitrarily take to be picoseconds, that have elapsed since the start of a simulation; \( k \) represents the number of successive delta cycles that have occurred during the current time unit.

Thus, \( T \) is ordered according to the lexicographic order on \( N \times N \), which is consistent with the natural ordering of \( N \): for time objects \( t_1 = (n_1, k_1) \) and \( t_2 = (n_2, k_2) \), \( t_1 \leq t_2 \) iff \( n_1 \leq n_2 \) and either \( n_1 < n_2 \) or \( k_1 \leq k_2 \). Thus the minimum element of \( T \) is the time object that is denoted alternatively as \( 0 \) or \((0, 0)\). For \( t_1, t_2 \in T \), the interval \( \{t \in T : t_1 \leq t < t_2\} \) will be denoted as \([t_1, t_2)\).

An event is an ordered pair \( e = (v, t) \), where \( v = \text{value}(e) \in \mathcal{B} \) and \( t = \text{time}(e) \in T \). Let \( w = ((v_1, t_1) \ldots (v_0, t_0)) \) be a list of events. If \( t_i > t_{i-1} \) and \( v_i \neq v_{i-1} \) for \( 0 < i \leq n \), and \( t_0 = 0 \), then \( w \) is a waveform. Note that according to this definition, successive events of a waveform must have different values; in VHDL terminology, all transactions are events. This restriction is consistent with the absence of implicit signals from our subset: since there is no way to detect transactions other than events (e.g., by means of the ACTIVE and TRANSACTION attributes), they may be ignored.

We define \( \hat{w} : T \to \mathcal{B} \) by \( \hat{w}(t) = v_j \), where \( j \) is the greatest value of \( i \) satisfying \( t_i \leq t \); \( \hat{w}(t) \) is called the value of \( w \) at \( t \). Note that \( \hat{w}_1 = \hat{w}_2 \) iff \( w_1 = w_2 \). If \( t = t_2 \), then we shall say that \( w \) has a new value at \( t \). We also define the history of \( w \) relative to \( t \) to be the waveform \( \text{hist}(w, t) = ((v_j, t_j) \ldots (v_0, t_0)) \).

A packet is a list of waveforms, \( p = (w_1 \ldots w_n), n \geq 0 \). For any \( t \in T \), the value of \( p \) at \( t \) is the bit vector \( \hat{p}(t) = (\hat{w}_1(t) \ldots \hat{w}_n(t)) \); \( p \) has a new value at \( t \) if any member of \( p \) does. The history of \( p \) relative to \( t \) is the packet \( \text{hist}(p, t) = (\text{hist}(w_1, t) \ldots \text{hist}(w_n, t)) \).
The behavior of each signal occurring in a circuit will be modeled as a waveform. During the course of a simulation, these waveforms are updated at various times. When a waveform is considered in the context of a current time \( t_0 \), each of its members \( e \) is viewed as a past, current, or future event, according to the relationship between \( \text{time}(e) \) and \( t_0 \). Past and present events are immutable, but future events are subject to deletion as they are superseded by newly scheduled events, as described below.

Whenever a new event \( e \) is to be scheduled for a signal, \( \text{time}(e) \) is computed from the current time \( t_0 = (n, k) \) and a delay \( d \in \mathbb{N} \) that is associated with the signal, by means of an addition operation from \( \mathbb{T} \times \mathbb{N} \) to \( \mathbb{T} \), defined as follows:

\[
(n, k) \oplus d = \begin{cases} 
(n + d, 0) & \text{if } d \neq 0 \\
(n, k + 1) & \text{if } d = 0.
\end{cases}
\]

Thus, regardless of delay, when a new event \( e = (v, t_v) \) is scheduled on a waveform \( w \) at time \( t_0 \), we have \( t_0 < t_v \). The scheduling may be performed by either of two procedures, corresponding to the transport and inertial delay modes of VHDL. Note that the definitions of these procedures are somewhat different from the processes described in [11], due to our restricted notion of waveform.

Transport delay is the simpler of the two: each event \( (v', t') \) with \( t' > t_v \) is deleted from \( w \), and \( e \) is then consed to the result, unless that result already has value \( v \) at \( t_v \). The updated waveform \( w' \) is computed as the value of \( \text{transport}(w, v, t_v) \), which is defined recursively as follows:

1. Let \( \text{car}(w) = (v_f, t_f) \). If \( t_f \geq t_v \), then \( w' = \text{transport}(	ext{cdr}(w), v, t_v) \); otherwise:
2. If \( v_f = v \), then \( w' = w \); otherwise:
3. \( w' = \text{cons}((v, t_v), w) \).

Alternatively, \( w' \) may be described in terms of the function \( \tilde{w}' \):

\[
\tilde{w}'(t) = \begin{cases} 
v & \text{if } t \geq t_v \\
\tilde{w}(t) & \text{if } t < t_v.
\end{cases}
\]

Inertial delay is somewhat more complicated: every event \( (v', t') \) with \( t' > t_0 \) is deleted from \( w \), and if \( \tilde{w}(t_0) \neq v \), then a single event with value \( v \) is consed to the result. If \( \tilde{w}(t_v) = v \), then the time of this event is the time of the last event of \( w \) that precedes \( t_v \); otherwise, it is \( t_v \). Note that this procedure takes the current time \( t_0 \) as an additional argument, and requires that \( t_0 < t_v \). The recursive definition of \( w' = \text{inertial}(w, v, t_0, t_v) \) is given as follows:

1. Let \( \tilde{w} = \text{hist}(w, t_0) \). If \( \tilde{w}(t_0) = v \), then \( w' = \tilde{w} \); otherwise:
2. Let \( \text{car}(w) = (v_f, t_f) \). If \( t_f \geq t_v \), then \( w' = \text{inertial}(	ext{cdr}(w), v, t_0, t_v) \); otherwise:
3. If \( v_f = v \), then \( w' = \text{cons}((v, t_f), \tilde{w}) \); otherwise:
4. \( w' = \text{cons}((v, t_v), \tilde{w}) \).
Transport mode is often used to model wires (along which pulses of arbitrarily small duration are propagated to the delayed signal), while gate outputs are generally modeled by inertial delay. The difference between the two modes is illustrated in Fig. 1. The diagram labelled (a) represents the waveform
\[ w = ((\mathcal{T}, 9) (\mathcal{F}, 8) (\mathcal{T}, 6) (\mathcal{F}, 5) (\mathcal{T}, 3) (\mathcal{F}, 1) (\mathcal{T}, 0)) \].

The results of updating \( w \) at time 1 by scheduling an event with time 7 and value \( \mathcal{T} \), in both transport and inertial modes, are
\[ \text{transport}(w, \mathcal{T}, 7) = ((\mathcal{T}, 6) (\mathcal{F}, 5) (\mathcal{T}, 3) (\mathcal{F}, 1) (\mathcal{T}, 0)) \]
and
\[ \text{inertial}(w, \mathcal{T}, 1, 7) = ((\mathcal{T}, 6) (\mathcal{F}, 1) (\mathcal{T}, 0)) \],
as shown in (b) and (c), respectively.

The following is a useful summary of both propagation functions. Each result may be proved by a straightforward induction. Note that (b) is consistent with our earlier informal observation that past and present events are immutable:

**Lemma 2.1** Let \( w \) be a waveform, let \( t_0, t_1, \) and \( t_2 \) be natural numbers with \( t_0 < t_2 \), and let \( w' \) be either \( \text{transport}(w, v, t) \) or \( \text{inertial}(w, v, t) \). Then
(a) \( \dot{w}'(t) = v \) for \( t \geq t_2 \);
(b) \( \dot{w}'(t) = \dot{w}(t) \) for \( t < t_0 \);
(c) if \( t_1 \leq t_0 \leq t_2 \leq t_v \) and \( \dot{w}(t) = u \) for \( t \in [t_1, t_2) \), then \( \dot{w}'(t) = u \) for \( t \in [t_1, t_2) \).

A similar induction shows that both procedures are "idempotent" in the following sense:

**Lemma 2.2** If \( w \) is a waveform and \( t_0, t_v, t'_0, t'_v \) are natural numbers with \( t_0 < t_v \), \( t'_0 < t'_v \), \( t_0 < t'_0 \), and \( t_v < t'_v \), then
(a) \( \text{transport}(\text{transport}(w, v, t_v), v, t'_v) = \text{transport}(w, v, t_v) \);
(b) \( \text{inertial}(\text{inertial}(w, v, t_v), v, t'_0, t'_v) = \text{inertial}(w, v, t_v, t_v) \).
2.3 Behavioral Modules

The simplest programs of our language are the behavioral modules, which contain explicit information concerning propagation delay and the functional dependence of outputs on inputs.

A behavioral module is a list \( M = (\text{BEHAV} \ I \ O \ T \ P \ D) \), where

(1) \( \text{BEHAV} \) is the identifying literal atom for modules of this type;
(2) \( I = I(M) = (r_1 \ldots r_m) \) is a list of literal atoms called the inputs of \( M \);
(3) \( O = O(M) = (s_1 \ldots s_n) \) is a list of literal atoms called the outputs of \( M \);
(4) \( T = T(M) = (\tau_1 \ldots \tau_n) \) is a list of elementary Boolean terms over \( I(M) \), called the output terms of \( M \);
(5) \( D = D(M) = (d_1 \ldots d_n) \) is a list of natural numbers, the delays of \( M \);
(6) \( P = P(M) = (p_1 \ldots p_n) \) is a list of literal atoms called the propagation modes of \( M \), each of which is either \text{TRANSPORT} or \text{INERTIAL}.

The members of the list \( (r_1 \ldots r_m, s_1 \ldots s_n) \) are required to be distinct and are called the signals of \( M \).

Note that each output is associated with a term, a mode, and a delay. If every term is either an atom or a list of atoms, (i.e., contains no nested function calls), then \( M \) is primitive.

Gates are generally modeled as primitive modules with inertial delays. For example, we represent a simple 2-input nand gate as the primitive module \text{nand2}:

\[
(\text{BEHAV} \ (A \ B) \ (C) \ ((\text{nand2} \ A \ B)) \ (2000) \ (\text{INERTIAL}))
\]

We may define a similar behavioral module, with \( n \) inputs and 1 output, corresponding to each elementary \( n \)-ary Boolean function, arbitrarily taking the delay to be 2000 in each case. In the sequel, we shall refer to these primitive modules without explicitly listing their definitions.

For the purpose of illustration, the following primitive module \( m \) is defined to have one output of each propagation mode:

\[
(\text{BEHAV} \ (A \ B) \ (C \ D) \ ((\text{nand2} \ A \ B) \ (\text{not} \ A)) \ (2000 \ 5000) \ (\text{INERTIAL} \ \text{TRANSPORT}))
\]

The VHDL code corresponding to a behavioral module consists of

(a) an entity declaration, consisting of a port clause listing the input signals as ports of mode \text{IN} and the output signals as ports of mode \text{OUT}, all of type \text{BIT};
(b) an architecture body, consisting of a concurrent signal assignment statement corresponding to each output signal.

The code (generated by our translator) for the module \( m \) defined above is displayed in Figure 2(a). Note that our time units are interpreted by the translator as picoseconds, and hence the delays are expressed as 2 and 5 nanoseconds. Note also that there is no mention of inertial delay in the translation, since this is the VHDL default mode.

Another example of a behavioral module is the 1-bit adder \text{adder1}:
ENTITY m IS
  PORT(a, b: IN BIT; c, d: OUT BIT)
END m;

ARCHITECTURE m OF m IS
BEGIN
  c <- a NAND b AFTER 2 NS;
  d <- TRANSPORT NOT a AFTER 5 NS;
END m;

ENTITY adder2 IS
  PORT(a, b, c: IN BIT; l, h: OUT BIT)
END adder2;

ARCHITECTURE adder2 OF adder2 IS
COMPONENT hand
  PORT(a, b: IN BIT; l, h: OUT BIT);
END COMPONENT;
SIGNAL _l, t2, t3, _4, t5, t6, t7: BIT;
BEGIN
  II: hand PORT MAP (a, b, _l);
  I2: hand PORT MAP (a, _l, t2);
  I3: hand PORT MAP (b, _l, t3);
  I4: hand PORT MAP (t2, t3, _4);
  I5: hand PORT MAP (_4, t4, t5);
  I6: hand PORT MAP (_4, t5, t7);
  I7: hand PORT MAP (t5, t4, _7);
  I8: hand PORT MAP (t5, _7, l);
  I9: hand PORT MAP (t7, _7, l);
END adder2;

Figure 2: VHDL Code

((BEHAV (A B C) (L H)
  ((XOR3 A B C) (OR2 (AND2 A (OR2 B C)) (AND2 B C))))
  (12000 10000)
  (INERTIAL INERTIAL))

The two outputs of this module represent the 2-bit sum of the three input bits. Since the higher-order “carry” output bit is not expressed as an elementary function of the inputs, this is not a primitive module.

Let \( s = nth(j, O(M)) \) be an output of a behavioral module \( M \). Let \( \tau = nth(j, T(M)) \) be the corresponding term. For any bit vector \( V \) of the same length as \( I(M) \), we define the combinational value of \( s \) w.r.t. \( V \) as \( cv(s, V, M) = eval(\tau, pairlist(I(M), V)) \).

We shall say that a list of waveforms is an input (resp., output) packet for a module \( M \) if it has the same length as \( I(M) \) (resp., \( O(M) \)). The semantics of behavioral modules are defined by a function \( exec \) of four arguments: (1) a module \( M \), (2) an input packet \( p_{in} \) for \( M \), (3) an output packet \( p_{out} = (w_1 \ldots w_n) \) for \( M \), and (4) a time object \( t_0 \). The value of \( exec(M, p_{in}, p_{out}, t_0) \) is the updated output packet \( p_{out}' = (w'_1 \ldots w'_n) \) that results from “executing” \( M \) at \( t_0 \). It is defined as follows: For \( i = 1, \ldots, n \), let \( w_i \) be the combinational value of \( nth(i, O(M)) \) w.r.t. \( p_{in}(t_0) \), and let \( t_i = t_0 + nth(i, D(M)) \). Then \( w'_i \) is either \( transport(w_i, v_i, t_i) \) or \( inertial(w_i, v_i, t_0, t_i) \), according to \( nth(i, P(M)) \).

Our first observation concerning the behavior of \( exec \) is that its value depends only on the current values of the input:

Lemma 2.3 Let \( p_1 \) and \( p_2 \) be input packets and let \( p_{out} \) be an output packet for a
behavioral module $M$. For any $t_0 \in T$, if $\hat{p}_1(t_0) = \hat{p}_2(t_0)$, then $\text{exec}(M, p_1, p_{\text{out}}, t_0) = \text{exec}(M, p_2, p_{\text{out}}, t_0)$.

Two other basic properties may be derived as consequences of Lemmas 2.1(b) and 2.2:

**Lemma 2.4** Let $p_{\text{in}}$ and $p_{\text{out}}$ be an input packet and an output packet for a behavioral module $M$. For any $t_0 \in T$, $\text{hist}(\text{exec}(M, p_{\text{in}}, p_{\text{out}}, t_0), t_0) = \text{hist}(p_{\text{out}}, t_0)$.

**Lemma 2.5** Let $p_{\text{in}}$ and $p_{\text{out}}$ be an input packet and an output packet for a behavioral module $M$ and let $t_0$ and $t_1$ be time objects. If $t_0 < t_1$ and $\hat{p}_{\text{in}}(t_0) = \hat{p}_{\text{in}}(t_1)$, then $\text{exec}(M, p_{\text{in}}, \text{exec}(M, p_{\text{in}}, p_{\text{out}}, t_0), t_1) = \text{exec}(M, p_{\text{in}}, p_{\text{out}}, t_0)$.

### 2.4 Structural Modules

Our language also includes modules that represent hierarchically constructed circuits. These structures contain information concerning interconnections among the modules of which they are composed.

A *structural module* is a list $M = (\text{STRUCT} I O S LI LO)$, where

1. $\text{STRUCT}$ is the identifying literal atom for modules of this type;
2. $I = I(M) = (r_1 \ldots r_m)$ is a list of literal atoms called the (global) inputs of $M$;
3. $O = O(M) = (s_1 \ldots s_n)$ is a list of literal atoms called the (global) outputs of $M$;
4. $S = S(M) = (\mu_1 \ldots \mu_k)$ is a list of (structural or behavioral) modules, called the submodules of $M$;
5. $LI = LI(M) = (A_1 \ldots A_k)$, where for $j = 1, \ldots, k$, $A_j = (a_{j1} \ldots a_{jm_j})$ is a list of literal atoms called the $j^\text{th}$ local inputs of $M$, and $m_j$ is the length of $I(\mu_j)$;
6. $LO = (B_1 \ldots B_k)$, where for $j = 1, \ldots, k$, $B_j = (b_{j1} \ldots b_{jn_j})$ is a list of literal atoms called the $j^\text{th}$ local outputs of $M$, and $n_j$ is the length of $O(\mu_j)$.

The members of the list $(r_1 \ldots r_m b_{11} \ldots b_{1n_1} \ldots b_{k1} \ldots b_{kn_k})$, consisting of the global inputs and all local outputs, are required to be distinct and are called the signals of $M$. There is no such constraint on the global outputs or local inputs, but each local input must be a signal of $M$, and each global output must be a local output.

Note that the local inputs and outputs of $M$ correspond to its submodules. Thus, intuitively, the submodules of a structure generate signals that are distinct from each other and from the structure's inputs. Each signal may be connected to arbitrarily many submodule inputs. A signal other than a global input may serve as any number of global outputs, but global inputs and outputs are distinct.

One additional constraint must be imposed on structural modules: in order to ensure that any simulation (as defined in the next section) of a module terminates, our structures are required to be free of zero-delay cyclic paths. Several preliminary definitions will be needed in order to make this notion precise.

We shall define a computable function that measures the (possibly infinite) maximum length of any path of signals within a structure along which the total delay is 0. The definition will be based on an auxiliary function, $\delta(M, s, E, L)$, the arguments of which are to be understood as follows:
(1) $M$ may be either the top-level structure or one of its components at any level of
the hierarchy;

(2) $s$ is a signal of $M$;

(3) $E = (e_1 \ldots e_n)$ is a list of generalized numbers corresponding to $O(M)$. For each
$i$, $e_i$ is intended to represent the maximum length of any path that starts at the
$i$th output and leads out of $M$. Such a list is called an environment for $M$;

(4) $L$ is a list of signals of $M$, each of which is known to lie on some infinite path.

Under these assumptions, we may think of $\delta = \delta(M, s, E, L)$ as the maximum length of
a path starting at $s$. It is computed recursively as follows:

(1) If $s$ is a member of $L$, then $\delta = \INFINITY$. Otherwise:

(2) Let $\Delta_1 = \max\{e_i : s = s_i\}$, where $O(M) = (s_1 \ldots s_n)$. (The maximum of the
null set is taken to be 0.)

(3) Suppose $M$ is behavioral. Let $D(M) = (d_1 \ldots d_n)$. If $s$ is an input of $M$ and some
d_i > 0, then let $\Delta_2 = 1 + \max\{e_i : d_i = 0\}$; otherwise, $\Delta_2 = 0$.

(4) Suppose $M$ is structural with $S(M) = (\mu_1 \ldots \mu_k)$. For $1 \leq i \leq k$, let
$nth(i, LI(M)) = (a_{i1} \ldots a_{im_i})$, $nth(i, LO(M)) = (b_{i1} \ldots b_{im_i})$, $I(\mu_i) = (a_{i1} \ldots a_{im_i})$, and
let $E_i$ be the environment $(e_{i1} \ldots e_{im_i})$ for $\mu_i$, where for $1 \leq k \leq m_i$, $e_{ik} = \delta(M, b_{ik}, E, \text{cons}(s, L))$. Let $\delta_{ij} = \delta(\mu_i, a_{ij}, E_i, \text{NIL})$ for $i = 1, \ldots, k$ and $j = 1, \ldots, m_i$. Let $\Delta_2 = \max\{\delta_{ij} : s = a_{ij}\}$.

(5) $\delta = \max(\Delta_1, \Delta_2)$.

The function $\Delta$ is defined by $\Delta(M, s, E) = \delta(M, s, E, \text{NIL})$. Next, we define the
relative $\delta$-depth of a module $M$ with respect to an environment $E$ to be the number $\rho$
computed as follows:

(1) Let $D_0$ be the maximum value of $\Delta(M, s, E)$ over all signals $s$ of $M$. If $M$ is
behavioral, then $\rho = D_0$. Otherwise:

(2) Let $M$ be structural with $S(M) = (\mu_1 \ldots \mu_k)$. For $1 \leq i \leq k$, let
$nth(i, LO(M)) = (b_{i1} \ldots b_{im_i})$ and let $D_i$ be the relative $\delta$-depth of $\mu_i$ with respect to the environment
$(\Delta(M, b_{i1}, E) \ldots \Delta(M, b_{im_i}, E))$. Then $\rho = \max(D_0, D_1, \ldots, D_k)$.

Finally, we define the $\delta$-depth of $M$ to be its relative $\delta$-depth with respect to the
environment $(0 \ldots 0)$. This represents the length of the longest 0-delay path through
$M$. If it is not $\INFINITY$, we shall say that $M$ is $\delta$-acyclic. All structural modules in
our language are required to have this property.

Although we have gone to considerable effort to formalize the VHDL "delta delay"
mechanism, the examples in which we are interested exhibit only positive delays. Our
first example is the structural module adder2, composed of nine nand gates and intended
as a gate-level "implementation" of the behavioral module adder1:
The VHDL code corresponding to a structural module consists of

(a) an entity declaration, consisting of a port clause listing the inputs as ports of mode IN and each output as a port, either of mode BUFFER, if it occurs as a local input, or of mode OUT, if it does not;

(b) an architecture body, consisting of a component declaration corresponding to each module that occurs as a submodule, a signal declaration corresponding to each local output that it not a global output (and hence does not already occur as a port), and a component instantiation statement corresponding to each submodule.

The code for adder2 is shown in Figure 2(b), and a circuit diagram appears in Figure 3(b). Later, we shall compare the behaviors of adder1 and adder2.

Of course, a signal path may be cyclic, provided that some signal in the path is associated with a positive delay. This is an important feature of our language, as it allows the modeling of state-holding devices. Figure 3(a) shows a clocked resetable d-flip-flop, which is modeled by the structural module diff:

(STRUCT (CLK RST D) (Q QN)
    (not1 and2 nand2 nand2 nand3 nand2 nand2 nand2)
    ((RN) (DD) (A1) (B1) (A2) (B2) (Q) (QN)))

In addition to five 2-input nand gates, the submodules of diff include an inverter not1, an a 2-input and gate and2, and a 3-input and gate nand3, the definitions of which are assumed to be understood.

We shall define the semantics of structural modules by means of a function step, based on the exec function of Section 4. Note that the notions of input and output packets may be naturally applied to any module. For a structural module $M$, however, instead of a simple output packet, the third argument of step must be an object that consists of a waveform corresponding to each signal generated by each component of $M$. Thus, for any module $M$, we define a bundle for $M$ to be a list $B$ such that (a) if $M$ is behavioral, then $B$ is an output packet for $M$; (b) if $M$ is a structure with $O(M) = (s_1 ... s_n)$, then $B = (\beta_1 ... \beta_k)$, where $\beta_i$ is a bundle for $\mu_i$, $i = 1, ..., k$.

Let $B$ be a bundle for a module $M$ and let $s$ be a signal of $M$ that is not an input of $M$. The waveform for $s$ determined by $B$ is the waveform $w$ that is computed as follows: (a) if $M$ is behavioral and $s = nth(j, O(M))$, then $w = nth(j, B)$; (b) if $M$ is structural and $s = nth(j, nth(i, LO(M)))$, then $w$ is the waveform for $nth(j, nth(i, S(M)))$ determined by $nth(i, B)$.

The output packet for $M$ determined by $B$, denoted as outp$(M,B)$, is defined as follows: (a) if $M$ is behavioral, then outp$(M,B) = B$; (b) if $M$ is structural with $O(M) = (s_1 ... s_n)$, then outp$(M,B) = (w_1 ... w_n)$, where for $1 \leq j \leq n$, $w_j$ is the waveform for $s_j$ determined by $B$. 

12
Let $M$ be a structural module with $nth(i, LI(M)) = (a_1 \ldots a_m)$. Let $p$ be an input packet and let $B$ be a bundle for $M$. The $i$th input packet determined by $p$ and $B$, denoted as $inp(i, M, p, B)$, is the input packet $(w_1 \ldots w_m)$ for $nth(i, S(M))$, where for $1 \leq j \leq m$, $w_j$ is computed as follows: (a) if $s_j$ is a global input $nth(k, I(M))$, then $w_j = nth(k, p)$; (b) if $s_j$ is a local output, then $w_j$ is the waveform for $s_j$ determined by $B$.

We may now define $step$. Let $p$ and $B$ be an input packet and a bundle, respectively, for an arbitrary module $M$, and let $t \in T$. Then $step(M, p, B, t)$ is the bundle $B'$, defined as follows: (a) if $M$ is behavioral, then $B' = exec(M, p, B, t)$ if $p$ has a new value at $t$, and $B' = B$ if not; (b) if $M$ is structural with $S(M) = (\beta_1 \ldots \beta_k)$ and $B = (\beta_1 \ldots \beta_k)$, then $B' = (\beta'_1 \ldots \beta'_k)$, where $\beta'_i = step(\mu_i, inp(i, M, p, B), \beta_i, t)$.

Thus, the execution of a structure at time $t$ amounts to the execution of each behavioral component for which the value of some input signal changes at $t$.

We have the following generalization of Lemma 2.3:

**Lemma 2.6** Let $p_1$ and $p_2$ be input packets and let $B$ be a bundle for a module $M$. Let $t_0 \in T$. If $hist(p_1, t_0) = hist(p_2, t_0)$, then $step(M, p_1, B, t_0) = step(M, p_2, B, t_0)$.

The history of a structural bundle $(\beta_1 \ldots \beta_k)$ relative to a time $t$ is recursively defined as $hist(B, t) = (hist(\beta_1, t) \ldots hist(\beta_k, t))$. Lemma 2.4 may be generalized as follows:

**Lemma 2.7** Let $p$ and $B$ be an input packet and a bundle for a module $M$. For any $t_0 \in T$, $hist(step(M, p, B, t_0), t_0) = hist(B, t_0)$.

### 2.5 Simulation

Let $p$ and $B$ be an input packet and a bundle for a module $M$. For any $t \in T$, we define $t_{next}(t, p, B, M)$ to be the minimum element of the set of all $t' \in T$ that occur as times of events in the waveforms of $p$ and $B$ and that satisfy $t' > t$, if this set is nonempty; otherwise, $t_{next}(t, p, B, M)$ is undefined.
A simulation of $M$ consists of repeated applications of $\text{step}$, which are performed by the function $\text{run}$. For $t_0, t_f \in T$, we define $\text{run}(M, p, B, t_0, t_f)$ to be the bundle $B'$ that is computed recursively as follows: Let $t_{\text{next}} = t_{\text{next}}(t_0, p, B, M)$. If $t_{\text{next}}$ is defined and $t_{\text{next}} \leq t_f$, then $B' = \text{run}(M, p, \text{step}(M, p, B, t_{\text{next}}), t_{\text{next}}, t_f)$; otherwise, $B' = B$.

It is not obvious that this is a valid recursive definition, i.e., that it is satisfied by a unique function. This may be established by exhibiting some measure of the arguments that decreases with each recursive call. More precisely, it suffices to define a function $\text{meas}$ such that under the assumptions imposed on the arguments of $\text{run}$,

$$\text{meas}(M, p, \text{step}(M, p, B, t_{\text{next}}), t_{\text{next}}, t_f) \prec \text{meas}(M, p, B, t_0, t_f)$$

with respect to some well-founded order "\prec". (In fact, this is the requirement for admissibility of Nqthm function definitions.)

We may construct an appropriate measure based on a function $\phi(M, p, B)$ that computes an upper bound on the delta component of any time object that occurs in any waveform during the course of a simulation. For each signal $s$ of $M$ or any module occurring in $M$, this function computes the sum of (a) the length of the longest 0-delay path through $M$ starting at $s$ and (b) the largest delta component that occurs in the waveform of $p$ or $B$ that corresponds to $s$. $\phi(M, p, B)$ is the maximum of these sums. (We omit the actual recursive definition of $\phi$, which parallels that of $\delta$-depth.)

Now, if $t_0 = (m_i, k_i)$ and $t_f = (m_f, k_f)$, then we define

$$\text{meas}(M, p, B, t_0, t_f) = (m_f - m_i, \phi(M, p, B) - k_i).$$

It may be shown that with respect to the lexicographic order "\prec" on $\mathbb{N} \times \mathbb{N}$, this function satisfies the property stated above. Note that its definition, and hence that of $\text{run}$, ultimately depends on the assumption that $M$ is $\delta$-acyclic.

The function $\text{meas}$ provides an induction scheme for deriving properties of $\text{run}$. The following, for example, is proved by induction as an immediate consequence of Lemma 2.7:

**Lemma 2.8** Let $p$ and $B$ be an input packet and a bundle for a module $M$. For any $t_0, t_f \in T$, $\text{hist}(\text{run}(M, p, B, t_0, t_f), t_0) = \text{hist}(B, t_0)$.

The next lemma, similarly proved by induction, provides for the decomposition of a simulation interval:

**Lemma 2.9** If $p$ and $B$ are an input packet and a bundle for a module $M$, and $t_0 \leq t' \leq t_f$, then $\text{run}(M, p, B, t_0, t_f) = \text{run}(M, p, \text{run}(M, p, B, t_0, t'), t' , t_f)$.

Another property of $\text{run}$ that is important in the analysis of circuit behavior is the following basic result, which describes the behavior of a structural module in terms of that of its components. It is interesting that its proof requires the two properties of $\text{step}$ that are stated in Lemmas 2.6 and 2.7, namely that module execution is neither predictive (with respect to input) nor retroactive (with respect to output).

**Lemma 2.10** Let $p$ and $A = (\alpha_1 \ldots \alpha_k)$ be an input packet and a bundle for a structural module $M$ with $S(M) = (\mu_1 \ldots \mu_k)$. Let $t_0, t_1 \in T$ and $B = (\beta_1 \ldots \beta_k) = \text{run}(M, p, A, t_0, t_1)$. Then $\beta_i = \text{run}(\mu_i, b_i, \alpha_i, t_0, t_1)$, where $b_i = \text{inp}(i, M, p, B)$, $i = 1, \ldots, k$. 

14
Proof: Let $A' = (\alpha'_1 \ldots \alpha'_k) = \text{step}(M, p, A, t')$, where $t' = t_{\text{next}}(t_0, p, A, M)$. Then by definition of \text{step}, $\alpha'_i = \text{step}(\mu_i, \alpha_i, t')$, where $\alpha_i = \text{inp}(i, M, p, A)$, and by definition of \text{run}, $B = \text{run}(M, p, A', t', t_1)$. By induction, we may assume that $\beta_i = \text{run}(\mu_i, b_i, \alpha'_i, t', t_1)$.

It follows from Lemmas 2.7 and 2.8 that $\text{hist}(A, t') = \text{hist}(B, t')$. Consequently, $\text{hist}(\alpha_i, t') = \text{hist}(b_i, t')$. By Lemma 2.6, $\alpha'_i = \text{step}(\mu_i, b_i, \alpha_i, t')$. Thus, we have $\beta_i = \text{run}(\mu_i, b_i, \text{step}(\mu_i, b_i, \alpha_i, t'), t', t_1)$.

Let $t'' = t_{\text{next}}(t_0, b_i, \alpha_i, \mu_i)$. Clearly, if $t''$ is defined, then $t'' \geq t'$. If $t'' = t'$, then

$$
\text{run}(\mu_i, b_i, \alpha_i, t_0, t_1) = \text{run}(\mu_i, b_i, \text{step}(\mu_i, b_i, \alpha_i, t''), t'', t_1) = \text{run}(\mu_i, b_i, \text{step}(\mu_i, b_i, \alpha_i, t'), t', t_1) = \beta_i.
$$

In the remaining case,

$$
\text{run}(\mu_i, b_i, \alpha_i, t_0, t_1) = \text{run}(\mu_i, b_i, \alpha_i, t', t_1) = \text{run}(\mu_i, b_i, \text{step}(\mu_i, b_i, \alpha_i, t'), t', t_1) = \beta_i. \Box
$$

The definition of our top-level simulation function $\text{sim}$ depends on $\text{run}$ as well as a function $\text{init}$, which generates an initial bundle from a module and an input packet. First, for a given module $M$, we define the bundle $B_0(M)$:

1. If $M$ is behavioral, then $B_0(M)$ is the output packet $(w_0 \ldots w_0)$ for $M$, where $w_0 = (\mathcal{F}, 0)$.
2. If $M$ is structural and $S(M) = (\mu_1 \ldots \mu_k)$, then $B_0(M) = (B_0(\mu_1) \ldots B_0(\mu_k))$.

Thus, every waveform of $B_0(M)$ is the trivial $w_0$, which has the constant value $\bar{w}_0(t) = \mathcal{F}$. Prior to simulation, each of these waveforms is updated by executing every behavioral component of $M$. The result is the bundle $\text{init}(M, p)$, defined as follows:

1. If $M$ is behavioral, then $\text{init}(M, p) = \text{exec}(M, p, B_0(M), 0)$;
2. If $M$ is structural with $S(M) = (\mu_1 \ldots \mu_k)$, then $\text{init}(M, p) = (\text{init}(\mu_1, \text{init}(1, M, p, B_0(M))) \ldots \text{init}(\mu_k, \text{init}(k, M, p, B_0(M))))$.

Now, given an input packet $p$ for $M$ and a time object $t$, we define

$$
\text{sim}(M, p, t) = \text{run}(M, p, \text{init}(M, p), 0, t).
$$

We note the following restatements of Lemmas 2.9 and 2.10:

**Lemma 2.11** If $p$ is an input packet for a module $M$, and $t_1 \leq t_2$, then $\text{sim}(M, p, t_2) = \text{run}(M, p, \text{sim}(M, p, t_1), t_1, t_2)$.

**Lemma 2.12** Let $p$ be an input packet for a structural module $M$ with $S(M) = (\mu_1 \ldots \mu_k)$. Let $t \in T$ and $B = (\beta_1 \ldots \beta_k) = \text{sim}(M, p, t)$. Then $\beta_i = \text{sim}(\mu_i, b_i, t)$, where $b_i = \text{inp}(i, M, p, B)$, $i = 1, \ldots, k$.
Figure 4: Simulation of $m$

As a simple example, a simulation of the primitive module $m$ is illustrated in Figure 4. The waveforms corresponding to the inputs $A$ and $B$ are

$$w_A = ((\mathcal{T}, 60000) (\mathcal{F}, 21000) (\mathcal{T}, 20000) (\mathcal{F}, 10000) (\mathcal{T}, 0))$$

and

$$w_B = ((\mathcal{T}, 70000) (\mathcal{F}, 30000) (\mathcal{T}, 0)),$$

respectively. These are shown along with the waveforms

$$w_C = (((\mathcal{F}, 72000) (\mathcal{T}, 12000) (\mathcal{F}, 0)))$$

and

$$w_D = ((\mathcal{F}, 65000) (\mathcal{T}, 26000) (\mathcal{F}, 25000) (\mathcal{T}, 15000) (\mathcal{F}, 0))$$

of the output $\text{sim}(m, (w_A w_B), 80000) = (w_C w_D)$.

This example exhibits a fundamental difference between transport and inertial delay: an input pulse of duration less than the delay, as occurs in $w_A$, is not reflected in an inertial output.

All of the simulation results that we report herein were produced by the Nqthm implementation of $\text{sim}$ and have been matched with the output of the corresponding Vantage simulations of the VHDL translations of these modules. One further observation is warranted, however, in support of the claim that our language definition adheres to the VHDL standard [11]. There is an apparent discrepancy between the definition of $\text{sim}$ and the standard: in our language, each output waveform of a behavioral module is updated whenever there is a change in any input value. In VHDL, on the other hand, in the absence of any instruction to the contrary (i.e., an explicit “sensitivity list”), a signal's waveform is updated only in response to changes in those inputs on which the signal is functionally dependent.

Consider, for example, the output $D$ of the module $m$. The VHDL code corresponding to this signal (Figure 2) is executed only in response to events of the input waveform $w_A$. However, according to our definitions of $\text{exec}$ and $\text{step}$, its waveform is also updated whenever the value of $B$ changes, e.g., at time $30000$ in our example.

Nonetheless, as illustrated in Figure 4, the behavior of this output signal is completely independent from that of $B$, in accordance with the VHDL standard. In order to understand this, consider the waveform $w$ that represents this signal before the execution of $m$ at time $21000$. The updated waveform after this execution is $w' = \text{transport}(w, \mathcal{T}, 26000)$. Although $w'$ is further updated when the value of $B$ changes at $30000$, the value of (NOT1 $A$) remains $\mathcal{T}$, and hence, by Lemma 2.2, the resulting waveform is $\text{transport}(w', \mathcal{T}, 35000) = w'$.

The above argument is based on the simple observation that at the time of any change in input during a simulation of a behavioral module, the output packet is the
result of executing the module at that time. In fact, an interesting property of our simulator is that this holds true even when there is no input change, i.e., regardless of whether the execution actually occurs:

**Lemma 2.13** Let \( p \) be an input packet for a behavioral module \( M \), let \( t \in T \), and let \( B = \text{sim}(M, p, t) \). Then \( B = \text{exec}(M, p, B, t) \).

Proof: It is easily shown by induction and Lemma 2.5, that if \( B_0 = \text{exec}(M, p, B_0, t_0) \) and \( B_1 = \text{run}(M, p, B_0, t_0, t_1) \), then \( B_1 = \text{exec}(M, p, B_1, t_1) \). The lemma is an instance of this result, with \( t_0 = t, B_0 = \text{init}(M, p), t_1 = t, \) and \( B_1 = B \). □

3 Specification of Synchronous Circuits

In order to simplify our analysis of circuit behavior, we shall assume in the sequel that delays associated with outputs of behavior modules are positive. (All of the examples in which we are interested conform to this assumption.) It follows that every time object occurring in a waveform produced by the simulator may be represented as a simple natural number. Thus, we may replace \( T \) by \( N \) and “\( \oplus \)” by “\( + \)”.

3.1 Combinational Modules

Before undertaking a characterization of synchronous sequential circuits, we shall consider the relatively simple class of combinational circuits. Let \( p = (s_1 \ldots s_p) \) be a list of signals of a structural module \( M \) such that for each \( i, 1 < i \leq p \), there exists \( j \) such that \( s_i \) is a member of \( \text{nth}(j, \text{LI}(M)) \) and \( s_i \) is a member of \( \text{nth}(j, \text{LO}(M)) \). Then \( p \) is a path in \( M \) from \( s_1 \) to \( s_p \). If \( s_1 = s_p \), then \( p \) is a loop in \( M \). An arbitrary module \( M \) is combinational if either (a) \( M \) is behavioral or (b) \( M \) is structural with no loops and all of its submodules are combinational.

The notion of combinational value, which previously applied only to outputs of behavioral modules, may be extended to combinational modules. Let \( s \) be any signal of a combinational module \( M \) and let \( V \) be a bit vector of the same length as \( I(M) \).

1. If \( s = \text{nth}(j, I(M)) \), then \( \text{cv}(s, V, M) = \text{nth}(j, V) \);

2. If \( M \) is structural and \( s = \text{nth}(j, \text{nth}(i, \text{LO}(M))) \), where \( \mu = \text{nth}(i, S(M)) \) and \( (a_1 \ldots a_m) = \text{nth}(i, \text{LI}(M)) \), then

\[
\text{cv}(s, V, M) = \text{cv}(\text{nth}(j, O(\mu)), (\text{cv}(a_1, V, M) \ldots \text{cv}(a_m, V, M)), \mu).
\]

We shall describe the behavior of combinational modules in terms of the function \( \text{cv} \). Our analysis begins with the following characterization of behavioral modules:

**Lemma 3.1** Let \( s = \text{nth}(j, O(M)) \) be the \( j \)-th output of a behavioral module \( M \), let \( d = \text{nth}(j, D(M)) \) be the corresponding delay, and let \( w = \text{nth}(j, \text{sim}(M, p, t_f)) \).

Assume that for all \( t \in [t_1, t_2] \), the combinational value of \( s \) w.r.t. \( \hat{\psi}(t) \) is \( v \), where \( t_1 + d \leq t_2 \) and \( t_1 \leq t_f \). Then for all \( t \in [t_1 + d, t_2 + d] \), \( \hat{\psi}(t) = v \).
Proof: Let \( p_1 = \text{sim}(M, p, t_1) \). Then according to Lemma 2.13, \( p_1 = \text{exec}(M, p, p_1, t_1) \). It follows from Lemma 2.1(a) that the value of \( \text{nth}(j, p_1) \) is \( v \) for all \( t \geq t_1 + d \).

We claim that if \( p' \) is any output packet for \( M \) such that \( \text{nth}(j, p') \) has value \( v \) throughout \([t_1 + d, t_2 + d]\), then so does \( \text{nth}(j, \text{run}(M, p, p', t', t_j)) \), for any \( t' \geq t_1 \). Once this claim is proved, the lemma will follow from Lemma 2.11 upon substituting \( p_1 \) and \( t_1 \) for \( p \) and \( t \).

The claim is proved by induction. It suffices to show that if \( p \) has a new value at \( t'' = t_{\text{next}}(t', p, p', M) \), and \( p'' = \text{exec}(M, p, p', t'') \), then \( \text{nth}(j, p'') \) has value \( v \) throughout \([t_1 + d, t_2 + d]\).

If \( t'' \geq t_2 \), then the desired result follows from Lemma 2.1(c). Thus, we may assume \( t'' < t_2 \) and hence, the combinational value of \( s \) w.r.t. \( p(t'') \) is \( v \). In this case, \( \text{nth}(j, p'') \) has value \( v \) on \([t_1 + d, t'' + d]\) by Lemma 2.1(c), and on \([t'' + d, t_2 + d]\) by Lemma 2.1(a). □

Lemma 3.1 is illustrated by the simulation of adder1 shown in Fig. 5, where we compare its behavior with that of the combinational module adder2. Note, for example, that the output \( L \) of adder1, with corresponding term \( (\text{XOR} 3 \ A \ B \ C) \), has the combinational value \( F \) throughout the interval from 40000 to 80000, and thus, since its delay is 12000, the actual value of the signal is \( F \) from 52000 to 92000. Note also that this simple behavior is not shared by the combinational module adder2.

However, we shall derive a generalization of Lemma 3.1 that provides similar (although somewhat weaker) behavioral specifications of arbitrary combinational modules. First, we associate each signal \( s \) of a combinational module \( M \) with two parameters, called the minimum and maximum delays of \( s \), which represent the range of total delays.
along all paths connecting the inputs of $M$ to $s$. These are defined as follows:

1. If $s$ is a member of $I(M)$, then $d_{\text{min}}(s,M) = d_{\text{max}}(s,M) = 0$;

2. If $M$ is behavioral and $s = \text{nth}(j,O(M))$, then
   
   $$d_{\text{min}}(s,M) = d_{\text{max}}(s,M) = \text{nth}(j,D(M))$$

3. If $M$ is structural and $s = \text{nth}(j,\text{nth}(i,LO(M)))$, where
   
   $\mu = \text{nth}(i,S(M))$ and $(a_1 \ldots a_m) = \text{nth}(i,LI(M))$, then

   $$d_{\text{min}}(s,M) = d_{\text{min}}(\text{nth}(j,O(\mu)), \mu)
   + \text{min}(d_{\text{min}}(a_1,M),\ldots,d_{\text{min}}(a_m,M)),$$

   $$d_{\text{max}}(s,M) = d_{\text{max}}(\text{nth}(j,O(\mu)), \mu)
   + \text{max}(d_{\text{max}}(a_1,M),\ldots,d_{\text{max}}(a_m,M))$$

**Lemma 3.2** Let $s = \text{nth}(j,O(M))$ be the $j^{th}$ output of a combinational module $M$, $d = d_{\text{min}}(s,M)$, $d' = d_{\text{max}}(s,M)$, and

$$w = \text{nth}(j,\text{outp}(M,\text{sim}(M,p,t_f))).$$

Assume that $\hat{p}$ is constant on the interval $[t_1,t_2)$, where $t_1 + d' \leq t_2$ and $t_1 \leq t_f$. Let $v = cv(s,\hat{p}(t_1),M)$. Then for all $t \in [t_1 + d',t_2 + d)$, $\hat{w}(t) = v$.

Proof: For behavioral $M$, the conclusion follows from Lemma 3.1. For structural $M$, we shall show that it holds more generally for any local output $s$ of $M$ and the waveform $w$ for $s$ determined by $B = \text{sim}(M,p,t_f)$. The proof is by induction on the length of the longest path in $M$ terminating at $s$.

Suppose $s$ is a local output, say $s = \text{nth}(j,\text{nth}(i,LO(M)))$. Let $\mu = \text{nth}(i,S(M))$, $\beta = \text{nth}(i,B)$, $(a_1 \ldots a_m) = \text{nth}(i,LI(M))$, and

$$b = \text{inp}(i,M,p,B) = (w_1 \ldots w_m).$$

Then $w = \text{nth}(j,\text{outp}(\mu,\beta))$, and by Lemma 2.12, $\beta = \text{sim}(\mu,b,t_f)$.

For $1 \leq \ell \leq m$, let $d_{\ell} = d_{\text{min}}(a_\ell,M)$, $d'_{\ell} = d_{\text{max}}(a_\ell,M)$, and $v_{\ell} = cv(a_\ell,\hat{p}(t_1),M)$.

If $a_\ell$ is a local output of $M$, then by inductive hypothesis, $\hat{w}_{\ell}(t) = v_{\ell}$ for all $t \in [t_1 + d'_{\ell},t_2 + d_{\ell})$; otherwise, $a_\ell$ is an input, and the same is true trivially. Thus, $\hat{b}(t) = (v_1 \ldots v_m)$ for all $t \in [t_1 + \Delta,t_2 + \delta)$, where $\Delta = \text{max}(d'_{1},\ldots,d'_{m})$ and $\delta = \text{min}(d_{1},\ldots,d_{m})$.

By the definition of $cv$,

$$v = cv(\text{nth}(j,O(\mu)),(v_1 \ldots v_m),\mu) = cv(\text{nth}(j,O(\mu)),\hat{b}(t_1 + \Delta),\mu).$$

Since $\mu$ is combinational, $\hat{w}(t) = v$ for all

$$t \in \left[t_1 + \Delta + d_{\text{max}}(\text{nth}(j,O(\mu)),\mu), t_2 + \delta + d_{\text{min}}(\text{nth}(j,O(\mu)),\mu)\right]$$

$$= [t_1 + d',t_2 + d). \square$$
As an example, consider the output signal L of the combinational module adder2. By tracing all paths from the inputs to L, we may compute \( cv(L, (a b c), \text{adder2}) \) as a nested nand2 expression that may be shown to be tautologically equivalent to \( \text{xor3}(a, b, c) \). By a similar calculation, we have

\[
dmin(L, \text{adder2}) = 4000 \quad \text{and} \quad dmax(L, \text{adder2}) = 12000.
\]

Thus, according to Lemma 3.2, if \( t_1 + 12000 \leq t_2, t_1 \leq t_f \), and the input packet \( p \) for adder2 has the constant value \( \hat{p}(t) = (a b c) \) for \( t \in [t_1, t_2) \), then

\[
w = nth(1, outp(\text{adder2}, sim(\text{adder2}, p, t_2)))
\]

has the value \( \hat{w}(t) = \text{xor3}(a, b, c) \) for \( t \in [t_1 + 12000, t_2 + 4000) \). This result is illustrated in Fig. 5: since the input packet has the constant value \( (T T T) \) on the interval \([20000, 40000)\), the value of the first output is \( \text{xor3}(T, T, T) = T \) on the interval \([32000, 44000)\).

### 3.2 Sequential Modules

We shall describe a class of sequential circuits that may be characterized as synchronous resettable rising-edge-triggered devices. The flip-flop \( \text{dff} \) of Subsection 2.4 will be used as a primitive in the construction of these circuits.

Let \( M \) be a structural module with \( I(M) = (r_1 \ldots r_m) \), where \( m \geq 2 \), \( S(M) = (\mu_1 \ldots \mu_k) \), and for \( i = 1, \ldots, k \), \( nth(i, LI(M)) = (a_{i1} \ldots a_{im}) \) and \( nth(i, LO(M)) = (b_{i1} \ldots b_{in}) \). Let \( q \in \mathbb{N} \). Then \( M \) is a sequential module with multiplicity \( q = \text{mult}(M) \) if either (a) \( q = 0 \) and \( M = \text{dff} \), or (b) \( 0 < q \leq k \) and the following conditions hold:

1. For \( 1 \leq i \leq q \), \( \mu_i \) is a sequential module;
2. For \( q < i \leq k \), \( \mu_i \) is a combinational module;
3. For \( 1 \leq i \leq k \) and \( 1 \leq j \leq m \), \( a_{ij} = r_1 \) iff \( i \leq q \) and \( j = 1 \);
4. For \( 1 \leq i \leq k \) and \( 1 \leq j \leq m \), \( a_{ij} = r_2 \) iff \( i \leq q \) and \( j = 2 \);
5. If \((s_1 \ldots s_p)\) is a path in \( M \) with \( s_1 = s_p \), then for some \( i \) and \( j \), where \( 1 \leq i \leq p \) and \( 1 \leq j \leq q \), \( s_i \) is a member of \( nth(j, LO(M)) \);
6. If \((s_1 \ldots s_p)\) is a path in \( M \) with \( s_1 \) a global input and \( s_p \) a global output of \( M \), then for some \( i \) and \( j \), where \( 1 \leq i \leq p \) and \( 1 \leq j \leq q \), \( s_i \) is a member of \( nth(j, LO(M)) \).

Throughout the remainder of this section, we shall assume that \( M \) is a sequential module with \( I(M), S(M), LI(M), \) and \( LO(M) \) as denoted above. Note that \( M \) must have at least two inputs, \( r_1 \) and \( r_2 \), which we call the clock and reset, respectively; the other inputs are called data. According to (3) and (4), if \( M \neq \text{dff} \), then the clock and reset of \( M \) are connected to the clock and reset, respectively, of each sequential submodule of \( M \), and to no other submodule inputs.

We define a path in \( M \) to be combinational if it contains no signal that is a local output of a sequential submodule. According to (5) of the definition, \( M \) contains no
combinational loop; according to (6), no combinational path connects an input to an output.

We define a signal \( s \) of \( M \) to be native if there is no combinational path from any global input to \( s \); the signals \( Q \) and \( \overline{Q} \) of \( \text{dff} \) are also defined to be native. Thus, all outputs of \( M \) are native signals.

A native signal \( s \) of \( M \) is registered if either (a) \( M = \text{dff} \) and \( s \) is an output of \( M \), or (b) \( M \neq \text{dff} \) and \( s \) is a local output \( b_{ij} \) where \( i \leq q \) and \( \text{nth}(j, O(\mu_i)) \) is a registered signal of \( \mu_i \). This property will have special significance in connection with asynchronous communication.

Two examples of sequential modules are diagrammed in Fig. 6. The enabled d-flip-flop, \( \text{dff} \), is defined to be the following structure:

\[
\text{(STRUCT} \\
\text{(CLK RST EN D)} \\
\text{(Q QN)} \\
\text{(\text{dff not1 nand2 nand2})} \\
\text{(CLK RST S4) (EN) (S1 Q) (D EN) (S2 S3))} \\
\text{(Q QN) (S1) (S2) (S3) (S4)))}
\]

Clearly, this module satisfies the definition, with \( \text{mult} (\text{dff}) = 1 \).

The 3-bit counter \( \text{count3} \) is a sequential module of multiplicity 3, defined as follows:

\[
\text{(STRUCT} \\
\text{(CLK RST EN)} \\
\text{(QO Q1 Q2)} \\
\text{(\text{dff edff edff and2 xor2})} \\
\text{(CLK RST EN QNO) (CLK RST EN S3) (CLK RST EN S2)} \\
\text{(QO Q1) (S1 Q2) (QO Q1))} \\
\text{(QO QNO) (Q1 QM1) (Q2 QM2) (S1) (S2) (S3)))}
\]

Note that all outputs of both of these modules are registered.

### 3.3 Sequential Values

Our description of the behavior of sequential modules will be based on a function that computes a sequence of values for each output corresponding to a given sequence of input values. The definition of this function involves the notion of state. An object \( \Sigma \) is a state of \( M \) if

1. \( M = \text{dff} \) and \( \Sigma \in \mathbb{B} \),
2. \( \text{mult}(M) = 1 \) and \( \Sigma \) is a state of \( \mu_1 \), or
3. \( \text{mult}(M) = q > 1 \) and \( \Sigma = (\sigma_1 \ldots \sigma_q) \), where for \( i = 1, \ldots, q \), \( \sigma_i \) is a state of \( \mu_i \).

Thus, a state associates a Boolean value with each flip-flop. The reset state \( \Sigma_0(M) \) is the state for which each of these values is \( \mathcal{F} \):

1. \( \Sigma_0(\text{dff}) = \mathcal{F} \);
2. If \( \text{mult}(M) = 1 \), then \( \Sigma_0(M) = \Sigma_0(\mu_1) \);
If $\text{mult}(M) = q > 1$, then $\Sigma_0(M) = (\Sigma_0(\mu_1) \ldots \Sigma_0(\mu_q))$.

A data vector for $M$ is a bit vector of length $m - 2$, the components of which correspond to the data inputs of $M$. We shall define a function $\text{next}(V, \Sigma, M)$ that computes a state of $M$ from a data vector $V$ and a state $\Sigma$. This definition requires two auxiliary functions.

First, for a native signal $s$ and a state $\Sigma$ of $M$, we define the native value of $s$ determined by $\Sigma$, denoted as $\text{nv}(s, \Sigma, M)$, as follows:

1. $\text{nv}(Q, \Sigma, \text{diff}) = \Sigma$ and $\text{nv}(\text{QN}, \Sigma, \text{diff}) = \text{not1}(\Sigma)$;
2. If $\text{mult}(M) = 1$ and $s = \text{btj}$, then
   \[ \text{nv}(s, \Sigma, M) = \text{nt'(nth(j, O(#l))), } \Sigma, \mu_1); \]
3. If $\text{mult}(M) = q > 1$ and $s = \text{bij}$, where $i < q$, then
   \[ \text{nv}(s, \Sigma, M) = \text{nv}(\text{nth}(j, O(\mu_1))), \Sigma, \mu_1); \]
4. If $\text{mult}(M) = q > 1$ and $s = \text{bij}$, where $i > q$, then
   \[ \text{nv}(s, \Sigma, M) = \text{cv}(\text{nth}(j, O(\mu_1)), (\text{nv}(\text{a}_{i1}, \Sigma, M) \ldots \text{nv}(\text{a}_{im}, \Sigma, M)), \mu_1). \]

Now, let $V = (v_3 \ldots v_m)$ and $\Sigma$ be a data vector and a state of $M$, respectively. We define the resultant value of a signal $s$ determined by $V$ and $\Sigma$, denoted as $\text{rv}(s, V, \Sigma, M)$, as follows:

1. If $s = r_i$ is a data input of $M$, then $\text{rv}(s, V, \Sigma, M) = v_i$;
2. If $s$ is native to $M$, then $\text{rv}(s, V, \Sigma, M) = \text{nv}(s, \Sigma, M)$;
3. If $\text{mult}(M) = q > 0$ and $s = b_{ij}$, where $i > q$, then
   \[ \text{rv}(s, V, \Sigma, M) = \text{cv}(\text{nth}(j, O(\mu_1)), (\text{rv}(\text{a}_{i1}, V, \Sigma, M) \ldots \text{rv}(\text{a}_{im}, V, \Sigma, M)), \mu_1). \]

We may now define the function $\text{next}$. Let $\text{mult}(M) = q$ and for $i = 1, \ldots, q$, let
\[ L_i = (\text{rv}(\text{a}_{i1}, V, \Sigma, M) \ldots \text{rv}(\text{a}_{im}, V, \Sigma, M)). \]
Then $\text{next}(V, \Sigma, M) = \Sigma'$, where

1. If $q = 0$ (i.e., $M = \text{diff}$), then $\Sigma' = v_3$;
2. If $q = 1$, then $\Sigma' = \text{next}(L_1, \Sigma, \mu_1)$;
3. If $q > 1$ and $\Sigma = (\sigma_1 \ldots \sigma_q)$, then
   \[ \Sigma' = (\text{next}(L_1, \sigma_1, \mu_1) \ldots \text{next}(L_q, \sigma_q, \mu_q)). \]
Figure 6: (a) edff (b) count3

Now, let $V = (V_3 \ldots V_m)$, where for $i = 3 \ldots m$, $V_i = (v_{i1} \ldots v_{im})$ is a bit vector of length $n$. $V$ may be viewed as a Boolean matrix, the rows of which correspond to the data inputs of $M$. Each column of this matrix, $V_j = (v_{3j} \ldots v_{mj})$, where $j = 1, \ldots, n$, is a data vector for $M$. A sequence of $n + 1$ states is determined by $V$ as follows:

$$\text{state}(j, V, M) = \begin{cases} \Sigma_0(M) & \text{if } j = 0 \\ \text{next}(V_j, \text{state}(j - 1, V, M), M) & \text{if } 0 < j \leq n. \end{cases}$$

For any native signal $s$ of $M$, the $j^{th}$ sequential value of $s$ determined by $V$ is defined as

$$sv(j, s, V, M) = nv(s, \text{state}(j, V, M), M).$$

Thus, the sequential values corresponding to a given matrix of input values are determined by the functions $nv$ and $\text{next}$. As an illustration, we shall analyze the behavior of these functions for the modules $\text{edff}$ and $\text{count3}$. Clearly, a state of $\text{edff}$ is a state of $\text{dff}$, i.e., a Boolean value. If $\Sigma$ is such a state and $V = (v_3 v_4)$ is a data vector, then

$$rv(Q, V, \Sigma, \text{edff}) = nv(Q, V, \Sigma, \text{edff}) = nv(Q, \Sigma, \text{dff}) = \Sigma$$

and

$$rv(QN, V, \Sigma, \text{edff}) = nv(QN, V, \Sigma, \text{edff}) = nv(QN, \Sigma, \text{dff}) = \text{not}1(\Sigma).$$

Expanding the definition of $rv$, we have

$$rv(S4, V, \Sigma, \text{edff}) = \text{nand}2(\text{nand}2(\text{not}1(v_3), \Sigma), \text{nand}2(v_3, v_4)),$$

which is also the value of $\text{next}(V, \Sigma, \text{edff})$. A trivial calculation yields the following:

**Proposition 3.1** Let $\Sigma$ and $V = (v_3 v_4)$ be a state and a data vector for $\text{edff}$. Then

$$nv(Q, V, \Sigma, \text{edff}) = \Sigma \quad \text{and} \quad nv(QN, V, \Sigma, \text{edff}) = \text{not}1(\Sigma);$$

$$\text{next}(V, \Sigma, \text{edff}) = \begin{cases} v_4 & \text{if } v_3 = \text{T} \\ \Sigma & \text{if } v_3 = \text{F}. \end{cases}$$
A state of count3 is a vector of 3 Boolean values, corresponding to the mult(count3) = 3 occurrences of edff. If \( \Sigma = (\sigma_0, \sigma_1, \sigma_2) \) and \( V = (v_3) \) are a state and a data vector, then
\[
\begin{align*}
rv(S1, V, \Sigma, \text{count3}) &= \text{and2}(\sigma_0, \sigma_1), \\
rv(S2, V, \Sigma, \text{count3}) &= \text{xor2}(\text{and2}(\sigma_0, \sigma_1), \sigma_2), \\
rv(S3, V, \Sigma, \text{count3}) &= \text{xor2}(\sigma_0, \sigma_1),
\end{align*}
\]
and it follows from Proposition 3.1 that
\[
\text{next}(V, \Sigma, \text{count3}) =
\begin{cases}
\text{not1}(\sigma_0) \text{xor2}(\sigma_0, \sigma_1) \text{xor2}(\text{and2}(\sigma_0, \sigma_1), \sigma_2) & \text{if } v_3 = T \\
\Sigma & \text{if } v_3 = F.
\end{cases}
\]
This result is conveniently expressed in terms of the function \( \text{inc}(W) \), defined as follows for an arbitrary bit vector \( W \):
\[
\begin{align*}
(1) & \text{ If } W = \text{NIL}, \text{ then } \text{inc}(W) = \text{NIL}; \text{ otherwise:} \\
(2) & \text{ If car}(W) = T, \text{ then } \text{inc}(W) = \text{cons}(F, \text{inc}(\text{cdr}(W)))); \text{ otherwise:} \\
(3) & \text{inc}(W) = \text{cons}(T, \text{cdr}(W)).
\end{align*}
\]

**Proposition 3.2** Let \( \Sigma = (\sigma_0, \sigma_1, \sigma_2) \) and \( V = (v_3) \) be a state and a data vector for count3. Then
\[
\begin{align*}
nv(Q0, V, \Sigma, \text{count3}) &= \sigma_0, \\
nv(Q1, V, \Sigma, \text{count3}) &= \sigma_1, \\
nv(Q2, V, \Sigma, \text{count3}) &= \sigma_2;
\end{align*}
\]
\[
\text{next}(V, \Sigma, \text{count3}) = \begin{cases}
\text{inc}(\Sigma) & \text{if } v_3 = T \\
\Sigma & \text{if } v_3 = F.
\end{cases}
\]

### 3.4 Behavior of dff

Naturally, the behavior of sequential modules depends on that of the primitive dff. A precise behavioral specification of dff is given by the following lemma, the proof of which is an elaboration of the informal argument found in [20]:

**Lemma 3.3** Let \( t_1 + 4000 \leq t_-, t_- + 6000 \leq t_2 \), and \( t_1 \leq t_f \). Let \( p = (w_{\text{CLK}} \ w_{\text{NST}} \ w_{\text{D}}) \) be an input packet for dff, and suppose that
\[
\begin{align*}
w_{\text{CLK}}(t) &= \begin{cases} F & \text{for all } t \in [t_1 - 6000, t_1) \cup [t_-, t_2) \\ T & \text{for all } t \in [t_1, t_-) \end{cases}, \\
w_{\text{NST}}(t) &= r \text{ for all } t \in [t_1 - 8000, t_1), \\
w_{\text{D}}(t) &= d \text{ for all } t \in [t_1 - 6000, t_1).
\end{align*}
\]
and
\[
\bar{w}_{\text{CLK}}(t) = d \text{ for all } t \in [t_1 - 6000, t_1).
\]

Let \( \text{sim}(\text{dff}, p, t_f) = ((w_{\text{HH}}) (w_{\text{DD}}) (w_{\text{A}}) (w_{\text{B}}) (w_{\text{A}}) (w_{\text{B}}) (w_{\text{H}})) \) and let \( v = \text{and2}(\text{not1}(r), d) \). Then \( \bar{w}_{\text{A}}(t) = v \) and \( \bar{w}_{\text{B}}(t) = \text{not1}(v) \text{ for all } t \in [t_1 + 6000, t_2 + 4000) \). Moreover, if these same values hold for all \( t \in [t_1, t_1 + 4000) \), then they also hold for all \( t \in [t_1 + 4000, t_1 + 6000) \).
Proof: By Lemmas 3.1 and 2.12, we have \( \dot{w}_{\text{in}}(t) = \text{not} 1(r) \) for all \( t \in [t_1 - 6000, t_1 + 2000] \). Applying the same two lemmas again, we have \( \dot{w}_{\text{in}}(t) = v \) for all \( t \in [t_1 - 4000, t_1 + 2000] \). Similarly, \( \dot{w}_{\text{in}}(t) = \text{not} 1(v) \) for \( t \in [t_1 - 2000, t_1 + 4000] \), and hence \( \dot{w}_{\text{in}}(t) = v \) for \( t \in [t_1, t_1 + 4000] \).

We shall consider the case \( v = \text{F} \); the case \( v = \text{T} \) is similar. In this case, \( \dot{w}_{\text{in}}(t) = \text{F} \) for \( t \in [t_1, t_1 + 2000] \), and hence \( \dot{w}_{\text{in}}(t) = \text{F} \) for \( t \in [t_1, t_1 + 6000] \).

Let \( t' \) be the least time such that \( t' > t_1 + 2000 \) and some waveform in the set \( \{ w_{\text{A}_1}, w_{\text{B}_1}, w_{\text{A}_2}, w_{\text{B}_2} \} \) assumes a new value at \( t' \). Then \( \dot{w}_{\text{A}_1}(t) = \dot{w}_{\text{A}_2}(t) = \text{F} \) for \( t \in [t_1 + 2000, t'] \). Since \( t' \geq t_1 + 4000 \), it follows that \( \dot{w}_{\text{A}_1}(t) = \dot{w}_{\text{A}_2}(t) = \text{F} \) for \( t \in [t_1 + 4000, t' + 2000] \). Similarly, \( \dot{w}_{\text{A}_2}(t) = \text{F} \) for \( t \in [t_1 + 2000, \min(t' + 4000, t_1 + 2000)] \). Thus, only \( w_{\text{A}_2} \) can possibly assume a new value at \( t' \), and this requires that \( t' > t_1 + 2000 \).

Hence, \( \dot{w}_{\text{B}_1}(t) = T \) and \( \dot{w}_{\text{B}_2}(t) = F \) for \( t \in [t_1 + 2000, t_1 + 2000] \). It follows that \( \dot{w}_{\text{in}}(t) = T \) for \( t \in [t_1 + 4000, t_1 + 4000] \), and hence \( \dot{w}_{\text{in}}(t) = T \) for \( t \in [t_1 + 2000, t_1 + 2000] \).

Let \( t'' \) be the least time such that \( t'' > t_1 + 6000 \) and either \( w_0 \) or \( w_{\text{in}} \) assumes a new value at \( t'' \). By an argument similar to the above, it is easily shown that \( t'' \geq t_2 + 4000 \). Thus, \( \dot{w}_0(t) = \text{F} \) for \( t \in [t_1 + 6000, t_2 + 4000] \), and \( \dot{w}_{\text{in}}(t) = \text{F} \) for \( t \in [t_1 + 4000, t_1 + 4000] \).

Now suppose that \( \dot{w}_0(t) = \text{F} \) for \( t \in [t_1 + 4000, t_1 + 4000] \). It follows that \( \dot{w}_0(t) = \text{F} \) for \( t \in [t_1 + 4000, t_2 + 4000] \).

3.5 Parameters

Our objective is to impose constraints on the input to a sequential module that will allow its outputs to be described in terms of sequential values. In particular, the clock input will be required to exhibit periodic behavior. We shall call each event of its associated waveform a rising or falling edge, according to whether its value is \( \text{T} \) or \( \text{F} \). An interval between two successive rising edges is called a cycle. Each of the remaining inputs will be required to maintain a stable value over a prescribed interval preceding each rising edge. For the reset input \( r_2 \), this value is \( \text{F} \) for an initial cycle, and \( \text{T} \) for every cycle thereafter.

Under these constraints, we shall show that the behavior of \( M \) admits a fairly simple description. A state of \( M \) will be associated with each rising edge. This state may be computed from the data values prior to the edge and the previous state by the function \( \text{next} \). The values of the outputs, which may change only during a short interval following a rising edge, are the corresponding sequential values.

We shall describe the behavior of the signals of \( M \) in terms of several parameters. First, we associate with each input other than the clock a setup time, which represents the duration over which the signal is required to hold constant prior to a rising edge. For the case \( M = \text{dff} \), as suggested by Lemma 3.3, we define

\[
\text{setup}(\text{rst}, \text{dff}) = 8000 \quad \text{and} \quad \text{setup}(\text{d}, \text{dff}) = 6000.
\]

Now suppose \( \text{mult}(M) = q > 0 \) and let \( s \) be any signal of \( M \) other than \( r_1 \). Assume \( \text{setup}(s', M) \) has been defined for each \( s' \neq s \) that lies on a combinational path starting at \( s \). For \( i = 1, \ldots, k \), let \( \zeta_i \) be defined as follows:
(1) If \( s \neq a_{ij} \) for all \( j, 1 \leq j \leq m_i \), then \( \zeta_i = 0 \); otherwise:

(2) If \( i \leq q \), then \( \zeta_i \) is the maximum setup\( (n_l(j, \mu_i)), \mu_i \) such that \( s = a_{ij}, j = 2, \ldots, m_i \); otherwise:

(3) \( i > q \), and \( \zeta_i \) is the maximum sum

\[
d_{max}(n_l(j, O(\mu_i)), \mu_i) + \text{setup}(b_{ij}, M)
\]

such that \( \text{setup}(b_{ij}, M) > 0, j = 1, \ldots, n_i \).

Then \( \text{setup}(s, M) = \max(\zeta_1, \ldots, \zeta_k) \).

Each native signal of \( M \) is associated with a minimum and a maximum delay, which determine an interval during which the signal's value may change following a rising edge. For the case \( M = \text{dff} \), we define

\[
d_{min}(Q, \text{dff}) = d_{min}(QN, \text{dff}) = 4000,
\]

\[
d_{max}(Q, \text{dff}) = d_{max}(QN, \text{dff}) = 6000.
\]

Now suppose \( \text{mult}(M) = q > 0 \) and let \( s = b_{ij} \) be any native signal of \( M \).

(1) If \( i \leq q \), then

\[
d_{min}(s, M) = d_{min}(n_l(j, O(\mu_i)), \mu_i),
\]

\[
d_{max}(s, M) = d_{max}(n_l(j, O(\mu_i)), \mu_i);
\]

(2) If \( i > q \), then

\[
d_{min}(s, M) = d_{min}(n_l(j, O(\mu_i)), \mu_i) + \min(d_{min}(a_{i1}, M), \ldots, d_{min}(a_{im_i}, M)),
\]

\[
d_{max}(s, M) = d_{max}(n_l(j, O(\mu_i)), \mu_i) + \max(d_{max}(a_{i1}, M), \ldots, d_{max}(a_{im_i}, M)).
\]

We also define three parameters pertaining to the behavior of the clock input of \( M \), called the clock high, the clock low, and the minimum period of \( M \). These represent the minimum durations between a rising edge and the next falling edge, a falling edge and the next rising edge, and successive rising edges, respectively. First, we define \( h_{igh}(\text{dff}) = 4000, l_{ow}(\text{dff}) = 6000 \), and \( p_{er}(\text{dff}) = 10000 \). For \( \text{mult}(M) = q > 0 \), we define

\[
h_{igh}(M) = \max(h_{igh}(\mu_1), \ldots, h_{igh}(\mu_q));
\]

\[
l_{ow}(M) = \max(l_{ow}(\mu_1), \ldots, l_{ow}(\mu_q));
\]
\[
\text{per}(M) = \max(P_1, P_2, P_3),
\]

where

\[
P_1 = \max\{\text{per}(\mu_i) : 1 \leq i \leq q\};
\]

\[
P_2 = \max\{\text{setup}(\tau_i, M) : 2 \leq i \leq m\};
\]

\[
P_3 = \max\{\text{setup}(b_{ij}, M) + \text{dmax}(\text{nth}(j, O(\mu_i)), \mu_i) : 1 \leq i \leq q, 1 \leq j \leq n_i\}.
\]

Consider, for example, the circuits edff and count3. First, the setup times for the signals of edff may be computed directly from the definitions, by tracing along all combinational paths. For example,

\[
\begin{align*}
\text{setup}(\text{RST, edff}) &= 8000, \\
\text{setup}(\text{EN, edff}) &= 12000, \\
\text{setup}(D, \text{edff}) &= 10000;
\end{align*}
\]

The setups for count3 follow trivially:

\[
\begin{align*}
\text{setup}(\text{RST, count3}) &= 8000, \\
\text{setup}(\text{EN, count3}) &= 12000.
\end{align*}
\]

In fact, it follows from our definitions that the reset input of every sequential module is 8000.

All outputs of both of these devices are registered. It follows that the minimum and maximum delay of each output are 4000 and 6000, respectively.

Similarly, the clock high and low of each device (in fact, of any sequential device) are 4000 and 6000, respectively, as determined by dff. Calculation of the minimum period, on the other hand, involves a comparison of various setups and delays. In the case of edff, the minimum period is found to be

\[
\text{setup}(Q, \text{edff}) + \text{dmax}(Q, \text{edff}) = 10000 + 6000 = 16000;
\]

for count3, it is

\[
\text{setup}(Q0, \text{count3}) + \text{dmax}(Q, \text{edff}) = 14000 + 6000 = 20000.
\]

3.6 The Main Theorem

The input constraints for sequential modules will be expressed in terms of the functions setup, high, low, and per. First, we define a waveform \( w \) to be an \( n \)-cycle pulse based at \( t_0 \) with high \( h \), low \( \ell \), and period \( \pi = h + \ell \) if for \( k = 0, \ldots, n - 1 \),

\[
\hat{w}(t) = \begin{cases} 
T & \text{for all } t \in [t_0 + k\pi, t_0 + k\pi + h) \\
F & \text{for all } t \in [t_0 + k\pi + h, t_0 + (k+1)\pi).
\end{cases}
\]

If \( h \geq \text{high}(M), \ell \geq \text{low}(M), \) and \( \pi \geq \text{per}(M) \), then \( w \) is an admissible pulse for \( M \).
Let $V = (v_1 \ldots v_n)$ be a bit vector and let $\pi > u > 0$. Let $w$ be a waveform such that for $k = 1, \ldots, n$, $\dot{w}(t) = v_k$ for all $t \in (t_0 + k\pi - u, t_0 + k\pi)$. Then $w$ is a stable $n$-cycle waveform based at $t_0$ with setup $u$, value list $V$, and period $\pi$. If $u = \text{setup}(r_2, M)$, $v_1 = T$, and $v_2 = \ldots = v_r = F$, then $w$ is an admissible reset waveform for $M$.

For $i = 1, \ldots, k$, let $w_i$ be a stable $n$-cycle waveform based at $t_0$ with value list $V_i$, setup list $U$, and period $\pi$. Let $V = (V_1 \ldots V_k)$, $U = (u_1 \ldots u_k)$, and $W = (w_1 \ldots w_k)$. Then $W$ is a stable $n$-cycle packet based at $t_0$ with value matrix $V$, setup list $U$, and period $\pi$. If $k = m - 2$ and $u_i = \text{setup}(r_{i+2}, M)$ for $i = 1, \ldots, k$, then $W$ is an admissible data packet for $M$.

Let $w_1$ be an admissible $(n + 2)$-cycle pulse for $M$ based at $t_0$ with period $\pi$. Let $w_2$ be an admissible $(n + 1)$-cycle reset waveform for $M$ based at $t_0$ with period $\pi$. Let $w_3 \ldots w_m$ be an admissible $n$-cycle data packet for $M$ based at $t_0 + \pi$ with value matrix $V$ and period $\pi$. Then $(w_1 \ldots w_m)$ is an admissible $n$-cycle input packet for $M$ based at $t_0$ with value matrix $V$ and period $\pi$.

We may now state a behavioral specification for sequential modules:

**Theorem 3.1** Let $s = \text{nth}(j, \text{out}(M))$ be the $j$th output of a sequential module $M$, $d' = \text{dmax}(s, M)$, and $w = \text{nth}(j, \text{outp}(M, \text{sim}(M, p, t_f)))$.

Assume that $p$ is an admissible $n$-cycle input packet for $M$ based at $t_0$ with value matrix $V$ and period $\pi$, where $t_f \geq t_0 + (n + 1)\pi$. For $i = 0, \ldots, n$, let $v_i = sv(i, s, V, M)$. Then $w$ is a stable $(n + 1)$-cycle waveform based at $t_0 + \pi$ with setup $\pi - d'$, value list $(v_0 \ldots v_n)$, and period $\pi$.

Assume further that $s$ is a registered signal of $M$ and $v_{i-1} = v_1$ for some $i, 1 \leq i \leq n$. Then $\dot{w}(t) = v_i$ for all $t \in [t_0 + (i + 1)\pi, t_0 + (i + 2)\pi]$.

**Theorem 3.1** is an immediate consequence of the following:

**Lemma 3.4** Let $s = \text{nth}(j, \text{out}(M))$ be the $j$th output of a sequential module $M$, $d = \text{dmin}(s, M)$, $d' = \text{dmax}(s, M)$, and $w = \text{nth}(j, \text{outp}(M, \text{sim}(M, p, t_f)))$.

Assume that $p$ is an admissible $n$-cycle input packet for $M$ based at $t_0$ with value matrix $V$ and period $\pi$. Let $t_0 + (n + 1)\pi = t_1$, $t_1 + \pi = t_2$, and assume $t_1 \leq t_f$. Let $v = sv(n, s, V, M)$. Then $\dot{w}(t) = v$ for all $t \in [t_1 + d', t_2 + d]$.

Suppose further that $s$ is a registered signal of $M$. If $n > 0$ and $sv(n-1, s, V, M) = v$, then $\dot{w}(t) = v$ for all $t \in [t_1 + d, t_2 + d]$.

**Proof:** For the case $M = \text{dff}$, the lemma is simply a restatement of Lemma 3.3. Thus, we may assume that $M \neq \text{dff}$ and proceed by induction on the structure of $M$. Let $V = (V_3 \ldots V_m)$, where for $i = 3, \ldots, m, V_i = (v_{i1} \ldots v_{in})$. For $j = 0, \ldots, n$, let $\Sigma_j = \text{state}(j, V, M)$.

Let $B = \text{sim}(M, p, t_f)$, and for each signal $s$ of $M$, let

$$w_s = \begin{cases} \text{nth}(i, p) & \text{if } s \text{ is a global input } r_i, \\ \text{the waveform for } s \text{ determined by } B & \text{if } s \text{ is a local output } b_{ij}, \end{cases}$$

If $s$ is not $r_1$ or $r_2$, then for $0 \leq \ell < n$, let
val(s, \ell) = rv(s, (v_{m(\ell+1)} \ldots v_{m(\ell+1)}), \Sigma_\ell, M).

If s is native, then by definition we have

val(s, \ell) = sv(s, \Sigma_\ell, M) = s\nu(\ell, s, V, M).

Thus, for native s, we extend the definition to \ell = n by

val(s, n) = sv(n, s, V, M).

For any \ell \in \mathbb{N}, let \ell^t = t_0 + (\ell + 1)\pi, so that \ell_1 = \ell^n and \ell_2 = \ell^{n+1} = \ell^n + \pi. We shall prove, by induction on \ell, that the following three statements hold for each \ell \leq n:

(a) For each \ell, 1 \leq i \leq q, inp(i, M, p, B) is an admissible \ell-cycle input packet for \mu_i based at \ell_0 with value matrix

\((\text{val}(a_{i_3}, 0) \ldots \text{val}(a_{i_3}, \ell - 1)) \ldots (\text{val}(a_{i_m}, 0) \ldots \text{val}(a_{i_m}, \ell - 1)))

and period \pi.

(b) For each native signal s = b_{ij} of M,

\dot{w}_s(t) = val(s, \ell) for all t \in [\ell + d\text{max}(s, M), \ell + d\text{min}(s, M)];

if s is a registered signal of M, then the same is true for the interval

\[[\ell + d\text{min}(s, M), \ell + d\text{min}(s, M)];

(c) If \ell < n, then for each signal s of M other than \tau_1 and \tau_2,

\dot{w}_s(t) = val(s, \ell) for all t \in [\ell + d\text{max}(s, M), \ell^t+1).

The lemma will then follow from (b), taking \ell = n.

Proof of (a): For \ell = 0, this follows from (3) and (4) in the definition of sequential module. For \ell > 0, we must also invoke the inductive hypothesis that (c) holds with \ell replaced by \ell - 1.

Proof of (b): We induct on the length of the longest combinational path terminating at s. Let s = b_{ij}. In the base case, where \ell \leq q, the result follows from the inductive assumption that the lemma holds for the sequential submodule \mu_i, Lemma 2.12, and (a). In the inductive case, where \ell > q, it follows from Lemmas 2.12 and 3.2.

Proof of (c): This is similarly proved by induction on the length of the longest combinational path terminating at s. In the base case, s is either a global input \tau_i, \ell \geq 3, or a local output b_{ij}, \ell \leq q. If s = \tau_i, then the claim follows directly from the admissibility of the input packet p. Suppose s = b_{ij}, \ell \leq q. It follows from (b) that

\dot{w}_s(t) = val(s, \ell) for all t \in [\ell + d\text{max}(s, M), \ell^t+1).

According to the definition of per(M),

\pi \geq setup(b_{ij}, M) + d\text{max}(nth(j, O(\mu_i)), \mu_i).

Hence,

\ell^t + d\text{max}(s, M) = \ell^{t+1} - \pi + d\text{max}(nth(j, O(\mu_i)), \mu_i) \leq \ell^{t+1} - setup(b_{ij}, M).

The induction is completed as in the proof of (b). □
4 Asynchronous Communication

Suppose we have a circuit in which an output of one sequential module, called the sender, is connected to a data input of another, called the receiver. Under suitable conditions on the sender's input, its output waveform is guaranteed by Theorem 3.1 to be stable with respect to the period of the sender's clock. On the other hand, in order to apply the results of Section 3 to the behavior of the receiver, we must be able to assume that its input is stable with respect to the period of its own clock. In general, this is true only for a synchronous circuit, in which the two modules are driven by the same clock. In this section, we shall examine the asynchronous case, in which the two clock inputs have different periods.

Our treatment of this problem is based on Moore's model of asynchrony [15]. In this model, the behavior of a signal is characterized abstractly by three quantities: a base time, a period, and a bit vector (representing the values assumed on successive cycles). Moore postulates that the receiver's input vector is determined by a function \( asynch \), the arguments of which include the sender's output vector, the two periods, and the two base times. In this section, we shall present Moore's function \( asynch \) and establish the applicability of his model to certain circuits represented in our language. In Section 5, we shall employ a theorem of Moore to show that if the sender's and receiver's periods are known to be approximately equal, then communication may be achieved by means of a well known protocol.

4.1 Smooth and Quasi-Smooth Waveforms

The communication protocol is motivated by the observation that if the time at which the receiver samples its input may be approximated by the sender, then the sender may successfully communicate a value by redundantly writing the value on sufficiently many successive cycles to guarantee that it is the value read by the receiver. For this purpose, the assumption that the sender's output waveform is stable is too weak; the waveform must be known to be constant on each cycle during some critical interval. With this requirement in mind, we define a stable waveform to be smooth if its setup time coincides with its period. Thus, \( w \) is a smooth \( n \)-cycle waveform based at \( t_0 \) with value list \( V = (v_1 \ldots v_n) \) and period \( \pi \) if for \( i = 1, \ldots, n \), \( \hat{w}(t) = v_i \) for all \( t \in [t_0 + (k-1)\pi, t_0 + k\pi) \).

A somewhat weaker notion of smoothness is needed to describe waveforms that are constant over some but not all cycles. First, we define a list \( V = (v_1 \ldots v_n) \) to be a generalized bit vector if each \( v_i \) is either Boolean or the literal atom \( Q \). In this case, we shall call \( w \) a quasi-smooth \( n \)-cycle waveform based at \( t_0 \) with value list \( V \) and period \( \pi \) if for \( i = 1, \ldots, n \), either \( v_i = Q \) or \( \hat{w}(t) = v_i \) for all \( t \in [t_0 + (k-1)\pi, t_0 + k\pi) \). (Thus, the value \( Q \) corresponds to cycles of unknown behavior.)

Our first objective is to derive a nontrivial representation of an output waveform of a sequential device as a quasi-smooth waveform. For this purpose, we make the following definition: If \( v \) is a Boolean atom and \( V \) is a bit vector, then \( smooth(v, V) \) is the generalized bit vector \( V' \), where

1. If \( V = \text{NIL} \), then \( V' = \text{NIL} \); otherwise:
2. If \( \text{car}(V) = v \), then \( V' = \text{cons}(v, smooth(v, cdr(V))) \); otherwise:

30
(3) \( V' = \text{cons}(Q, \text{smooth}(\text{car}(V), \text{cdr}(V))) \).

Thus, if \( v = v_0 \) and \( V = (v_1 \ldots v_n) \), then \( V' = (v'_1 \ldots v'_n) \), where for \( i = 1, \ldots, n \),

\[
v'_i = \begin{cases} v_i & \text{if } v_i = v_{i-1} \\ Q & \text{if } v_i \neq v_{i-1}. \end{cases}
\]

**Lemma 4.1** Let \( s = \text{nth}(j, O(M)) \) be a registered output of a sequential module \( M \). Let 
\( w = \text{nth}(j, \text{outp}(M, \text{sim}(M, p, t_f))) \), where \( p \) is an admissible \( n \)-cycle input packet for \( S \) based at \( t_0 \) with value matrix \( V \) and period \( \pi \), and \( t_f \geq t_0 + (n + 1)\pi \).

Let \( U = (sv(0, s, V, M) \ldots sv(n, s, V, M)) \). Then \( w \) is an \( n \)-cycle quasi-smooth waveform based at \( t_0 + 2\pi \) with value list \( \text{smooth}(\text{car}(U), \text{cdr}(U)) \) and period \( \pi \).

Proof: For \( 0 \leq k \leq n \), let \( U_k = (sv(n - k, s, V, M) \ldots sv(n, s, V, M)) \) and \( V_k = \text{smooth}(\text{car}(U_k), \text{cdr}(U_k)) \). We shall prove, by induction on \( k \), that \( w \) is a \( k \)-cycle quasi-smooth waveform based at \( t_0 + (n - k + 2)\pi \) with value list \( V_k \) and period \( \pi \).

The base case \( k = 0 \) holds vacuously. For \( k > 0 \), since \( \text{cdr}(V_k) = V_{k-1} \), we need only consider \( \text{car}(V_k) \) and the behavior of \( w \) on \( [t_0 + (n - k + 2)\pi, t_0 + (n - k + 3)\pi) \).

If \( \text{car}(V_k) = Q \), there is nothing to prove. In the remaining case, \( \text{car}(V_k) = \text{car}(U_k) = \text{car}(U_{k-1}) \), i.e., \( sv(n - k, s, V, M) = sv(n - k + 1, s, V, M) \), and the result follows from Theorem 3.1. \( \square \)

### 4.2 Describing Output as Input

Next, for a given quasi-smooth waveform with period \( \pi_r \) (representing that of the sender's clock), we would like to derive an alternative representation as a quasi-smooth waveform with a given period \( \pi_s \) (that of the receiver's clock). Let \( w \) be an \( n \)-cycle quasi-smooth waveform based at \( t_s \) (a rising edge of the sender's clock) with value list \( V = (v_1 \ldots v_n) \) and period \( \pi_s \). Assume \( t_s \leq t_r < t_s + \pi_r \) (where \( t_r \) represents a rising edge of the receiver's clock). We shall construct a list of values \( V' = \text{warp}(V, t_s, t_r, \pi_s, \pi_r) \) such that \( w \) is a quasi-smooth waveform based at \( t_r \) with value list \( V' \) and period \( \pi_r \).

The definition of \( \text{warp} \) requires several auxiliary functions.

Let \( t \) satisfy \( t_s < t \leq t_s + n\pi_s \). Choose \( k \) so that \( t_s + (k - 1)\pi_s < t \leq t_s + k\pi_s \). Then \( 1 \leq k \leq n \). (\( k \) represents the number of cycles of the sender that intersect the interval \( [t_r, t_s) \).) We define

\[
\text{sig}(V, t_s, t, \pi_s) = \begin{cases} v_k & \text{if } v_1 = v_2 = \ldots = v_k \\ Q & \text{if not.} \end{cases}
\]

Under the same constraints on \( t \), choose \( \ell \) so that \( t_s + \ell\pi_s < t < t_s + (\ell + 1)\pi_s \). Then \( 0 \leq \ell \leq n \). (\( t_s + \ell\pi_s \) represents the maximum sender's rising edge that is not exceeded by \( t \).) We define

\[
t^+_{t_s}(V, t_s, t, \pi_s) = t_s + \ell\pi_s
\]

and

\[
\text{lst}^+(V, t_s, t, \pi_s) = (v_{t+1} \ldots v_n).
\]
Now we may define $V' = \text{warp}(V, t_s, t_r, \pi_s, \pi_r)$: If $t_r + \pi_r > t_s + n\pi_s$, then $V' = \text{NIL}$; otherwise,

$$V' = \text{cons}(\text{sig}, \text{warp(lst}^+, t_r^+, t_r + \pi_r, \pi_r)),$$

where $\text{sig} = \text{sig}(V, t_s, t_r + \pi_r, \pi_r)$, $\text{lst}^+ = \text{lst}^+(V, t_s, t_r + \pi_r, \pi_r)$, and $t_r^+ = t_r^+(V, t_s, t_r + \pi_r, \pi_r)$.

**Lemma 4.2** Let $w$ be a quasi-smooth $n$-cycle waveform based at $t_s$ with value list $V$ and period $\pi_r$. Let $\pi_r > 0$ and $t_s \leq t_r < t_s + \pi_s$. Let $V' = \text{warp}(V, t_s, t_r, \pi_s, \pi_r)$ and let $n'$ be the length of $V'$. Then $w$ is a quasi-smooth $n'$-cycle waveform based at $t_r$ with value list $V'$ and period $\pi_r$.

Proof: We may assume $t_r + \pi_r \leq t_s + n\pi_s$, for otherwise, $n' = 0$. Let $V = (v_1 \ldots v_n)$ and let $\text{sig}, \text{lst}^+$, and $t_r^+$ be defined as in the definition of $\text{warp}$. By induction, we may further assume that $w$ is a quasi-smooth $(n' - 1)$-cycle waveform based at $t_r + \pi_r$ with value list $\text{cdr}(V') = \text{warp(lst}^+, t_r^+, t_r + \pi_r, \pi_r)$ and period $\pi_r$. We need only show that either $\text{car}(V') = \text{sig} = \text{Q}$, or $w$ has the constant value $\text{sig}$ on the cycle $[t_r, t_r + \pi_r)$.

Suppose $\text{sig} \neq \text{Q}$. Choose $k$ so that $t_s + (k - 1)\pi_s < t_r + \pi_r \leq t_s + k\pi_s$. According to the definition of $\text{sig}$, $\text{sig} = v_1 = v_2 = \ldots = v_k$, and hence, $w(t) = \text{sig}$ for all $t \in [t_s, t_s + k\pi_s) \supseteq [t_r, t_r + \pi_r)$.

### 4.3 Eliminating Metastability

Lemmas 4.1 and 4.2 together provide a representation of a registered output waveform from the sender as a quasi-smooth waveform with respect to the receiver's clock. In order to achieve communication, we shall design a clocked state-holding device, called a $d$-latch, that converts a quasi-smooth input to a stable output. In our asynchronous circuit, this device will share the receiver's clock, and its output will be connected to the receiver's input.

The $d$-latch will consist of an inverter and three nand gates. Its functionality will depend on the relative delays of these components. Thus, along with our standard gates not1 and nand2, both of which have delay 2000, we shall require the following faster nand gate, fnand2:

$$\text{(BEHAV} (A B) (\text{NAND2} A B)) (1000) (\text{INERTIAL})$$

We define $d$-latch to be the following module, which is diagrammed in Fig. 7:

$$\text{(STRUCT} (\text{CLK} D) (S2)}$$

$(\text{not1 nand2 nand2 faand2})$  
$((\text{CLK}) (\text{CLK} D) (S1 S3) (S0 S2))$  
$((\text{S0}) (S1) (S2) (S3))$)

Unlike all other circuits that we have encountered, the specified behavior of $d$-latch will also depend on the unique character of inertial delay. In particular, we shall need the following result:
Lemma 4.3 Let $n_{th}(j, O(M)) = s$ be the $j^{th}$ output of a behavioral module $M$. Let $n_{th}(j, D(M)) = d$ and $n_{th}(j, P(M)) = \text{INERTIAL}$. Let $p$ be an input packet for $M$, let $v$ be the combinational value of $s$ w.r.t. $p(t_o)$, and let $w = n_{th}(j, \text{sim}(M, p, t_o))$.

(a) If $\hat{w}(t_0) = v$, then $w = \text{hist}(w, t_0)$;
(b) If $\hat{w}(t_0) \neq v$, then $w = \text{cons}((v, t_1), \text{hist}(w, t_0))$, where $t_0 < t_1 \leq t_0 + d$.

Proof: By Lemma 2.13 and the definition of $\text{exec}$,

$$w = \text{inertial}(w, v, t_0, t_0 + d).$$

The lemma follows from the definition of $\text{inertial}$.

The behavioral specification of $d\text{latch}$ is an instance of the following, with $d_0 = d_1 = 2000$ and $d_3 = 1000$.

Lemma 4.4 Let $G_0$ be the inverter

\[(\text{BEHAV} \ (A) \ (\text{NOT1} \ A) \ (d_0) \ (\text{INERTIAL}))\]

and for $i = 1, 2, 3$, let $G_i$ be the nand gate

\[(\text{BEHAV} \ (A \ B) \ (\text{AND2} \ A \ B) \ (d_i) \ (\text{INERTIAL}))\]

where $d_1 \leq d_0$ and $d_0 + d_3 < d_1 + d_2$. Let $D = d_0 + d_1 + d_2 + d_3$. Let $L$ be the module

\[(\text{STRUCT} \ (\text{CLK}) \ (D) \ (G_0 \ G_1 \ G_2 \ G_3) \ ((\text{CLK}) \ (\text{CLK} \ D) \ (S_1 \ S_3) \ (S_2 \ S_0)) \ ((S_0) \ (S_1) \ (S_2) \ (S_3))).\]

Let $p = (w_{\text{CLK}} w_0)$ be an input packet for $L$, and assume that

$$w_{\text{CLK}}(t) = \begin{cases} T & \text{for all } t \in [t_+, t_-) \\ F & \text{for all } t \in [t_-, t_f), \end{cases}$$

where $t_- > t_+ + D$ and $t_f > t_- + D$. Let $((w_0)(w_1)(w_2)(w_3)) = \text{sim}(L, p, t_f)$. Then $\hat{w}_2$ has a constant value $v$ on $[t_- + D, t_f)$. If $\hat{w}_0$ has a constant value $u$ on $[t_+, t_f)$, then $u = v$.

Proof: For each $t \in \mathbb{N}$, let $B_i = ((w_{0,i})(w_{1,i})(w_{2,i})(w_{3,i})) = \text{sim}(L, p, t)$. Then for $i = 0, \ldots, 3$, $w_i = w_{i,t_f}$. Let $t_0 = t_- + d_0$. For each $t \geq t_0$, the following results may be derived from Lemmas 3.1 and 2.12:

(a) $\hat{w}_{0,t}$ has the constant value $F$ on $[t_+ + d_0, t_0)$;
(b) $\hat{w}_{3,t}$ has the constant value $T$ on $[t_+ + d_0 + d_3, t_0 + d_3)$;
(c) $\hat{w}_{0,t}$ has the constant value $T$ on $[t_0, t_f + d_0)$;
(d) $\hat{w}_{1,t}$ has the constant value $T$ on $[t_- + d_1, t_f + d_1)$.

In particular, for each $t \geq t_0$, $\hat{w}_{0,t}$ and $\hat{w}_{1,t}$ are both constant on $[t_0, t_f)$.

By Lemma 2.12,

$$\text{sim}(G_2, (w_{1,t} \ w_{3,t}), t)$$

and

$$\text{sim}(G_3, (w_{0,t} \ w_{2,t}), t).$$

We shall apply Lemma 4.3 to both $G_2$ and $G_3$. 

33
We shall show that for some $t_1 \in [t_0, t_0 + D)$ and some $v \in B$, $\hat{w}_{2,t_1}(t_1) = v$ and $\hat{w}_{3,t_1}(t_1) = \text{not} l(v)$. Let $w_{2,t_0}(t_0) = v_2$ and $w_{3,t_0}(t_0) = v_3$. We consider the following cases:

Case 1: $v_3 = \text{not} l(v_2)$. In this case, we take $t_1 = t_0$ and $v = v_2$.

Case 2: $v_3 = v_2$. By Lemma 4.3(b),

$$w_{2,t_0} = \text{cons}((\text{not} l(v_2), t_2), \text{hist}(w_{2,t_0}, t_0)),$$

where $t_0 < t_2 \leq t_0 + d_2$, and

$$w_{3,t_0} = \text{cons}((\text{not} l(v_2), t_3), \text{hist}(w_{3,t_0}, t_0)),$$

where $t_0 < t_3 \leq t_0 + d_3$.

Subcase 2a: $t_3 < t_2$. Here, $t_{\text{next}}(t_0, p, B_{t_0}, L) = t_3$. By Lemma 2.7,

$$\hat{w}_{2,t_2}(t_3) = \hat{w}_{2,t_0}(t_3) = v_2$$

and

$$\hat{w}_{3,t_3}(t_3) = \hat{w}_{3,t_0}(t_3) = \text{not} l(v_2).$$

Thus, we have $t_1 = t_3$ and $v = v_2$.

Subcase 2b: $t_2 < t_3$. In this case, $t_{\text{next}}(t_0, p, B_{t_0}, L) = t_3$, and we have

$$\hat{w}_{2,t_2}(t_2) = \hat{w}_{2,t_0}(t_2) = \text{not} l(v_2)$$

and

$$\hat{w}_{3,t_3}(t_2) = \hat{w}_{3,t_0}(t_2) = v_2.$$ 

In this case, $t_1 = t_2$ and $v = \text{not} l(v_2)$.

Subcase 2c: $t_2 = t_3$. We have

$$\hat{w}_{2,t_2}(t_2) = \hat{w}_{3,t_2}(t_2) = \text{not} l(v_2).$$

By Lemma 4.3(b),

$$w_{2,t_2} = \text{cons}((v_2, t_2 + d_2), w_{2,t_0}),$$

and

$$w_{3,t_2} = \text{cons}((v_2, t_2 + d_3), w_{3,t_0}).$$

It follows from our hypotheses that $d_3 < d_2$. Hence,

$$\hat{w}_{2,t_2 + d_3}(t_2 + d_3) = \text{not} l(v_2)$$

and

$$\hat{w}_{3,t_2 + d_3}(t_2 + d_3) = v_2.$$

Thus, $t_1 = t_2 + d_3$ and $v = \text{not} l(v_2)$.

Now, by Lemma 4.3(a), $w_{2,t_1} = \text{hist}(\hat{w}_{2,t_1}, t_1)$ and $w_{3,t_1} = \text{hist}(\hat{w}_{3,t_1}, t_1)$. Hence, $t_{\text{next}}(t_1, p, B_{t_1}, L) \geq t_f$. It follows that for any $t' \in [t_1, t_f)$, $B_{t'} = B_{t_1}$, and in particular, $w_{2,t_f}(t') = w_{2,c}(t') = w_{2,t_1}(t') = v$. Thus, $w_{2,t_f}$ has the constant value $v$ on $[t_1, t_f] \supseteq [t_0 + D, t_f)$.

34
Finally, suppose that $\dot{w}_0$ has a constant value $u$ on $[t_+, t_f)$. Then $\dot{w}_1(t) = \text{not}(u)$ for $t \in [t_+ + d_1, t_+ + d_1)$. Since $\dot{w}_2(t) = T$ on $[t_+ + d_0 + d_3, t_+ + d_3)$, the combinational value corresponding to $S_2$ is $u$ on the intersection of these intervals, $[\max(t_+ + d_1, t_+ + d_0 + d_3), \min(t_- + d_1, t_+ + d_3)]$. Thus, by Lemma 3.1, $\dot{w}_2(t) = u$ for $t \in [\max(t_+ + d_1 + d_2, t_+ + d_0 + d_3 + d_2), \min(t_- + d_1 + d_2, t_+ + d_3 + d_2)]$. In particular, $\dot{w}_2(t) = u$ for $t \in [t_0, t_0 + d_3 + d_2)$. Thus, $v_2 = u$. Moreover, Subcases 2b and 2c, in which $\dot{w}$ assumes the value $\text{not}(v_2)$ at some point in this interval, are eliminated. In the remaining cases, $v = v_2 = u$.

In order to avail ourselves of the results of [15], we must restate Lemma 4.4 in terms of Moore's function $\text{det}$. If $V$ is a generalized bit vector and $\text{oracle}$ is a bit vector, then $\text{det}(V, \text{oracle})$ is the bit vector $V'$ defined as follows:

1. If $V = \text{NIL}$, then $V' = \text{NIL}$; otherwise:
2. If $\text{car}(V) \in B$, then $V' = \text{cons}($\text{car}(V), $\text{det}($\text{cdr}(V), $\text{oracle})$); otherwise:
3. If $\text{oracle} = \text{NIL}$, then $V' = \text{cons}(T, \text{det}($\text{cdr}(V), $\text{oracle})$); otherwise:
4. $V' = \text{cons}($\text{car}(\text{oracle}), $\text{det}(\text{cdr}(V), $\text{cdr}(\text{oracle})$)).

**Lemma 4.5** Let $p = (w_{\text{CLK}} w_0)$ be an input packet for $\text{dlatch}$, where $w_{\text{CLK}}$ is an $n$-cycle pulse based at $t_0$ with high $h > 7000$, low $l > 7000$, and period $\pi = h + l$, and $w_0$ is a quasi-smooth $n$-cycle waveform based at $t_0$ with value list $V$ and period $\pi$. Let

$$((w_0)(w_1)(w_2)(w_3)) = \text{sim}(\text{dlatch}, p, t_f),$$

where $t_f \geq t_0 + n\pi$. Then for some bit vector $\text{oracle}$, $w_2$ is a stable $n$-cycle waveform based at $t_0$ with setup $\ell - 7000$, value list $\text{det}(V, \text{oracle})$, and period $\pi$.

**Proof:** We induct on $n$. For $n = 0$, the statement is vacuous. For $n > 0$, we may assume that $w_2$ is a stable $(n - 1)$-cycle waveform based at $t_0 + \pi$ with setup $\ell - 7000$, value list $\text{det}(\text{cdr}(V), \text{oracle}')$, and period $\pi$. By Lemma 4.4, $\dot{w}_2$ has a constant value $v$ on $[t_0 + h + 7000, t_0 + \pi] = [t_0 + \pi - (l - 7000), t_0 + \pi]$ and if $\text{car}(V) \neq 0$, then $\text{car}(V) = v$. If $\text{car}(V) = 0$, then let $\text{oracle} = \text{cons}(v, \text{oracle}')$; otherwise, let $\text{oracle} = \text{oracle}'$. In

Figure 7: (a) $\text{dlatch}$ (b) $\text{bpm}$
either case, \( w_2 \) is a stable \( n \)-cycle waveform based at \( t_0 \) with setup \( \ell = 7000 \), value list \( \det(V, \text{oracle}) \), and period \( \pi \). □

4.4 The Main Theorem

In Section 5, we shall apply the results of this section to a circuit \( \text{bpm} \), consisting of two sequential submodules, \( \text{sndr} \) and \( \text{rcvr} \), and a \( \text{dlatch} \): According to the definitions that we shall present later, \( \text{sndr} \) has 9 data inputs and one registered output, \( \text{SOUT} \), while \( \text{rcvr} \) has one data input, \( \text{SIN} \), and 9 outputs. The circuit \( \text{bpm} \), which is diagrammed in Fig. 7, is defined as follows:

\[
(\text{STRUCT} \\
(\text{CLKS} \ \text{RSTS} \ \text{CLKR} \ \text{RSTR} \ \text{SEND} \ \text{I}0 \ \text{I}1 \ \text{I}2 \ \text{I}3 \ \text{I}4 \ \text{I}5 \ \text{I}6 \ \text{I}7) \\
(\text{DONE} \ 00 \ 01 \ 02 \ 03 \ 04 \ 05 \ 06 \ 07) \\
(\text{sndr} \ \text{dlatch} \ \text{rcvr}) \\
((\text{CLKS} \ \text{RSTS} \ \text{SEND} \ \text{I}0 \ \text{I}1 \ \text{I}2 \ \text{I}3 \ \text{I}4 \ \text{I}5 \ \text{I}6 \ \text{I}7) \\
(\text{CLKR} \ \text{SOUT}) \\
(\text{CLKR} \ \text{RSTS} \ \text{LOUT})) \\
((\text{SOUT}) \\
(\text{LOUT}) \\
(\text{DONE} \ 00 \ 01 \ 02 \ 03 \ 04 \ 05 \ 06 \ 07)))
\]

The following theorem summarizes our results on asynchrony, as they pertain to the module \( \text{bpm} \). The theorem refers to Moore's function \( \text{asynch} \), which is defined as follows:

Let \( V \) and \( \text{oracle} \) be bit vectors and let \( t_s, t_r, r_s, r_r \in \mathbb{N} \) such that \( r_s > 0, r_r > 0, \) and \( t_s < t_r < t_s + r_s \). Then

\[
\text{asynch}(V, t_s, t_r, r_s, r_r, \text{oracle}) = \det(\text{warp}(\text{smooth}(T, V), t_s, t_r, r_s, r_r), \text{oracle}).
\]

Theorem 4.1 Let \( p = (w_{\text{CLKS}} \ w_{\text{RSTS}} \ w_{\text{CLKR}} \ w_{\text{RSTR}} \ w_{\text{SEND}} \ w_0 \ldots \ w_7) \) be an input packet for \( \text{bpm} \), where

(a) \( (w_{\text{CLKS}} \ w_{\text{RSTS}} \ w_{\text{SEND}} \ w_0 \ldots \ w_7) \) is an admissible \( n_s \)-cycle input packet for \( \text{sndr} \) based at \( b_s \) with value matrix \( V \) and period \( \pi_s \);

(b) \( w_{\text{CLKR}} \) is an admissible \( (n_r + 2) \)-cycle pulse for \( \text{rcvr} \) based at \( b_r \) with high \( h > 7000 \), low \( \ell > 7000 + \text{setup(\text{SIN,rcvr})} \), and period \( \pi_r = h + \ell \);

(c) \( w_{\text{RSTR}} \) is an admissible \( (n_r + 1) \)-cycle reset waveform for \( \text{rcvr} \) based at \( b_r \) with period \( \pi_r \).

Let \( t_r = b_r + \pi_r \). Assume that \( b_s + 2\pi_s \leq t_r \leq b_s + (n_s + 2)\pi_s \leq t_r + n_r \pi_r \). Choose \( j \) so that \( b_s + j\pi_s \leq t_r < b_s + (j + 1)\pi_s \) and let \( t_s = b_s + j\pi_s \). Assume \( sv(j - 2, \text{SOUT}, V, \text{sndr}) = T \).

Let \( U = (sv(j - 1, \text{SOUT}, V, \text{sndr}) \ldots \ sv(n_s, \text{SOUT}, V, \text{sndr})) \). Let \( w_{\text{LOUT}} \) be the waveform for \( \text{LOUT} \) determined by \( \text{sim}(\text{bpm}, p, t_f) \), where \( t_f \geq t_r + n_r \pi_r \). Then for some bit vector \( \text{oracle} \), \( (w_{\text{CLKS}} \ w_{\text{RSTS}} \ w_{\text{SEND}} \ w_0 \ldots \ w_7) \) is an admissible input packet for \( \text{rcvr} \) based at \( b_r \) with value matrix \( \det(\text{warp}(\text{smooth}(T, V), t_s, t_r, r_s, r_r), \text{oracle}) \) and period \( \pi_r \).

36
Proof: Let $w_{SOUT}$ be the waveform for $SOUT$ determined by $sim(bpm,p,t_f)$. According to Lemma 4.1, $w_{SOUT}$ is a quasi-smooth waveform based at $t_s$ with value list $smooth(T,U)$ and period $\pi_s$. It follows from Lemma 4.2 that $w_{SOUT}$ is also a quasi-smooth waveform based at $t_r$ with value list $warp(smooth(T,U),t_s,t_r,\pi_s,\pi_r)$ and period $\pi_r$. Finally, by Lemma 4.5, $w_{SOUT}$ is a stable waveform based at $t_r$ with setup $\ell - 7000 > setup(SIN,rcvr)$, value list

\[
det(warp(smooth(T,U),t_s,t_r,\pi_s,\pi_r),oracle) = asynch(U,t_s,t_r,\pi_s,\pi_r,oracle)),
\]

for some oracle, and period $\pi_r$. \qed

5 Biphase Mark

Moore's formulation [15] of the biphase mark protocol is based on two functions, $send$ and $recv$, which represent the computations performed by the sender and the receiver, respectively. After presenting the definitions of these functions, we shall implement them in the design of the sequential modules $sndr$ and $rcvr$. Then, using a theorem of Moore in combination with results of Section 4, we shall show that the circuit $bpm$ achieves communication between these modules.

5.1 Sending

The function $send$ returns a bit vector that represents an encoding of a given input bit vector $msg$. Each bit of $msg$ is encoded as a bit vector called a cell, computed as the value of $cell(x,n,k,b)$, where $b$ is the bit of $msg$ to be encoded, $x$ is the final bit of the preceding cell, and $n$ and $k$ are parameters of the protocol. A cell consists of two subcells, each of which is a uniform bit vector: a mark subcell of length $n$, followed by a code subcell of length $k$. The mark subcell is intended as a signal to the receiver that a new cell has been entered: each of its bits is $notl(x)$. The code subcell is the region in which the receiver is expected to look for information from which it will derive the value $b$ of the encoded bit: if $b = T$, then each bit of this subcell is $x$; if $b = \_$, each bit is $notl(x)$.

The definition of $cell$ requires three auxiliary functions. First, the subcells are constructed by the function $listn$: for any $n \in \mathbb{N}$ and any $x$, $listn(n,x)$ is the uniform vector $(x \ldots x)$ of length $n$. Next, the two subcells are combined by the function $app$: for any two lists $L = (a_1 \ldots a_n)$ and $M = (b_1 \ldots b_m)$, $app(L, M) = (a_1 \ldots a_n b_1 \ldots b_m)$. Finally, the bit occurring in the code subcell is determined by the Boolean function $equal$, where $equal(x,y) = T$ iff $x = y$, i.e., $equal(x,y) = notl(xor2(x,y))$.

Now, we may define

\[
cell(x,n,k,b) = app(listn(n,notl(x)),listn(k,equal(x,b))),
\]

and $cells(x,n,k,msg)$ is defined as

1. $NIL$, if $msg = NIL$;
2. $app(cell(x,n,k,car(msg)),cells(equal(x,car(msg)),n,k,cdr(msg)))$, if $msg \neq NIL$. 

37
The protocol includes the convention that the value $T$ is transmitted until the encoded message is sent. Thus, the encoded bit vector constructed by $send$ includes "pads" consisting arbitrarily many copies of $T$ on both sides of the cells. The arguments of $send$ include the lengths $p_1$ and $p_2$ of these pads:

$$send(msg, p_1, n, k, p_2) = \text{app}(\text{listn}(p_1, T), \text{app}(\text{cells}(T, n, k, msg), \text{listn}(p_2, T))).$$

5.2 Receiving

Next, we define $recv(i, x, j, L)^1$, which may be shown, under suitable assumptions, to be the inverse of $send$. This function recovers a bit of the encoded message from each cell by first detecting the beginning of the mark subcell, and then reading and decoding a bit at a predetermined location within the cell, which has been calculated to lie within the code subcell. Its arguments are interpreted as follows: $i$ is the number of bits of the original message yet to be recovered, $x$ is the last bit to have been read (from the preceding cell), $j$ is the location within the cell of the bit to be read, and $L$ is the remaining input stream.

The beginning of a new cell is detected by the function $\text{scan}(x, L)$, which successively removes bits from the beginning of the list $L$ until a value different from $x$ is found. The recursive definition follows:

1. If $L = \text{NIL}$, then $\text{scan}(x, L) = \text{NIL}$; otherwise:
2. If $\text{car}(L) = x$, then $\text{scan}(x, L) = \text{scan}(x, \text{cdr}(L))$; otherwise:
3. $\text{scan}(x, L) = L$.

We shall require one other auxiliary function: If $n \in \mathbb{N}$ and $L$ is a list, then $\text{cdrn}(n, L)$ is defined to be

1. $L$, if $n = 0$;
2. $\text{cdrn}(n - 1, \text{cdr}(L))$, if $n > 0$.

Finally, we define $recv(i, x, j, L)$ to be the bit vector $msg$, where

1. If $i = 0$, then $msg = \text{NIL}$; otherwise:
2. Let $S = \text{scan}(x, L)$. If $\text{length}(S) \leq k$, then $msg = \text{NIL}$; otherwise:
3. Let $b = \text{nth}(k + 1, S)$ and $L' = \text{cdrn}(k + 1, S)$. If $b = x$, then $msg = \text{cons}(T, recv(i - 1, b, j, L'))$; otherwise:
4. $msg = \text{cons}(F, recv(i - 1, b, j, L'))$.

---

1 For technical reasons, we shall slightly modify Moore's original definition of this function. Our modification does not affect the validity of any of his results.
5.3 Moore's Theorem

Moore has proved a statement of correctness of the protocol for certain values of the parameters. The lengths of the mark and code subcells generated by send are taken to be $n = 5$ and $k = 13$, respectively. The index of the bit read by recv following the detection of an edge is $j = 10$, i.e., the eleventh bit after the edge is sampled. The theorem also depends on an assumption concerning the proximity of the two clock periods:

**Theorem 5.1 (Moore)** Let $\pi_s > 0$, $\pi_r > 0$, and $17\pi_s \leq 18\pi_s \leq 19\pi_r$. Let $t_s \leq t_r < t_s + \pi_s$. Let $msg$ be a bit vector of length $k$. Then for any bit vector oracle and any numbers $p_1$ and $p_2$,

$$recv(k, T, 10, asynch(send(msg, p_1, 5, 13, p_2), t_s, t_r, \pi_s, \pi_r, oracle)) = msg.$$ 

We shall apply Moore's theorem to the specification of the circuit bpm. The sequential submodules sndr and rcvr of bpm remain to be defined. As we present the definitions of the these modules and their components, which are diagrammed in Figs. 8−12, we shall derive characterizations of their behavior that are analogous to Propositions 3.1 and 3.2. The proofs of these results are based on straightforward calculations and have all been mechanically checked. Therefore, the details of these proofs are omitted here.

5.4 Basic Components

The message that is transmitted from sndr to rcvr will consist of eight bits. It is stored (by both sndr and rcvr) in a shift register, shift8, which is constructed from eight copies of the following 3-port cell, port3:

(STRUCT
 (CLK RST SHIFT SIN LOAD DIN)
 (Q)
 (edff nand2 nand2 or2 nand2)
 ((CLK RST S3 S4) (DIN LOAD) (SIN SHIFT) (LOAD SHIFT) (S1 S2))
 ((Q QW) (S1) (S2) (S3) (S4)))

The behavior of port3 may be derived easily from that of edff (Proposition 3.1):

**Proposition 5.1** Let $\Sigma$ and $V = (shift \ sin \ load \ din)$ be a state and a data vector for port3. Assume that shift and load are not both $T$. Then

$$nv(Q, V, \Sigma, port3) = \Sigma;$$

$$next(V, \Sigma, port3) = \begin{cases} 
\text{sin} & \text{if shift} = T \text{ and load} = \text{F} \\
\text{din} & \text{if shift} = \text{F} \text{ and load} = T \\
\Sigma & \text{if shift} = \text{F} \text{ and load} = \text{F}.
\end{cases}$$

The register shift8 is defined as follows:
Proposition 5.2 Let $\Sigma = (\sigma_0 \ldots \sigma_7)$ and $V = (\text{load shift sin d_0 \ldots d_7})$ be a state and a data vector for shift8. Assume that shift and load are not both $T$. Then

$$\nu(Q_i, V, \Sigma, \text{shift8}) = \sigma_i, i = 0, \ldots, 7;$$

$$\text{next}(V, \Sigma, \text{shift8}) = \begin{cases} (\sin \sigma_0 \ldots \sigma_6) & \text{if shift} = T \text{ and load} = F \\ (d_0 \ldots d_7) & \text{if shift} = F \text{ and load} = T \\ \Sigma & \text{if shift} = F \text{ and load} = F. \end{cases}$$

In order to describe the shifting operation that is performed by shift8, we define, for any $b \in B$ and any bit vector $V$, 

40
shift(b, V) = \begin{cases} 
\text{NIL} & \text{if } V = \text{NIL} \\
\text{cons}(b, shift(\text{car}(V), \text{cdr}(V))) & \text{if } V \neq \text{NIL}.
\end{cases}

Thus, \( shift(sin, (\sigma_0 \ldots \sigma_f)) = (sin \sigma_0 \ldots \sigma_f) \).

In addition to \( dff \) and \( edff \), we shall require two other versions of the flip-flop. The first of these, \( cdff \), has an input CLR, which may be used to override the other data input \( D \) and reinitialize the state:

\text{Proposition 5.3} \ Let \( \Sigma \) and \( V = (\text{clr} \ d) \) be a state and a data vector for \( cdff \). Then

\[ nv(Q, V, \Sigma, cdff) = \Sigma \] and \( nv(QN, V, \Sigma, cdff) = \text{not}(\Sigma) \);

\[ \text{next}(V, \Sigma, cdff) = \begin{cases} 
F & \text{if } \text{clr} = T \\
D & \text{if } \text{clr} = F.
\end{cases} \]

The second, \( cedff \), is a combination of \( edff \) and \( cdff \):

\text{Proposition 5.4} \ Let \( \Sigma \) and \( V = (\text{clr} \ end) \) be a state and a data vector for \( cedff \). Then

\[ nv(Q, V, \Sigma, cedff) = \Sigma \] and \( nv(QN, V, \Sigma, cedff) = \text{not}(\Sigma) \);

\[ \text{next}(V, \Sigma, cedff) = \begin{cases} 
F & \text{if } \text{clr} = T \\
D & \text{if } \text{clr} = F \text{ and } \text{en} = T \\
\Sigma & \text{if } \text{clr} = F \text{ and } \text{en} = F.
\end{cases} \]
Using \texttt{cedff}, we construct the following 5-bit counter, \texttt{count5}:

\begin{center}
\begin{verbatim}
(STRUCT
(Clk Rst Clr En)
(Q0 Q1 Q2 Q3 Q4)
(cedff cedff cedff cedff cedff
 and2 and2 and2 xor2 xor2 xor2 xor2)
((Clk Rst Clr En Qn0)
(Clk Rst Clr En X1)
(Clk Rst Clr En X2)
(Clk Rst Clr En X3)
(Clk Rst Clr En X4)
(Q0 Q1) (A1 Q2) (A2 Q3) (Q0 Q1) (Q2 A1) (Q3 A2) (Q4 A3))
((Q0 Qn0) (Q1 Qn1) (Q2 Qn2) (Q3 Qn3) (Q4 Qn4)
(A1) (A2) (A3) (X1) (X2) (X3) (X4)))
\end{verbatim}
\end{center}

**Proposition 5.5** Let $\Sigma = (\sigma_0 \ldots \sigma_4)$ and $V = (clr\ en)$ be a state and a data vector for \texttt{count5}. Then

$$nv(Q_i, V, \Sigma, count5) = \sigma_i, \ i = 0, \ldots, 4;$$

$$next(V_0, \Sigma, count5) = \left\{ \begin{array}{ll}
listn(5, F) & \text{if } clr = T \\
inc(cnt) & \text{if } clr = F \text{ and } en = T \\
\Sigma & \text{if } clr = F \text{ and } en = F.
\end{array} \right.$$ 

For convenience in representing states of both \texttt{count3} and \texttt{count5}, we define, for $k \in \mathbb{N}$ and $n \in \mathbb{N}$,

$$bv_k(n) = \left\{ \begin{array}{ll}
listn(k, F) & \text{if } n = 0 \\
inc(bv_k(n - 1)) & \text{if } n > 0.
\end{array} \right.$$
Thus, \( b_{vk}(n) \) is the \( k \)-bit vector that represents the number \( n \).

We shall also require a combinational module, the following 5-bit comparator \( \text{comp}S \):

\[
\begin{align*}
\text{STRUCT} & \quad (C0 \ B0 \ C1 \ B1 \ C2 \ B2 \ C3 \ B3 \ C4 \ B4) \\
(\text{MATCH}) & \quad \text{MATCH} \\
(xor2 \ xor2 \ xor2 \ xor2 \ xor5) & \quad (C0 \ B0 \ (C1 \ B1) \ (C2 \ B2) \ (C3 \ B3) \ (C4 \ B4) \ (S1 \ S2 \ S3 \ S4 \ S5))
\end{align*}
\]

This module simply determines whether two given 5-bit vectors are equal, i.e.,

\[
\text{cv}(\text{MATCH}, (c_0 b_0 c_1 b_1 \ldots c_4 b_4), \text{comp}S) = \begin{cases} 
\top & \text{if } (c_0 \ldots c_4) = (b_0 \ldots b_4) \\
\bot & \text{if not.}
\end{cases}
\]

### 5.5 The Sender

The action of \( \text{sndr} \) is controlled by the submodule \( \text{scount} \), which is defined as follows:

\[
\begin{align*}
\text{STRUCT} & \quad (\text{CLK} \ \text{RST} \ \text{STOP} \ \text{BIT}) \\
(\text{MARK} \ \text{CODE}) & \quad (\text{MARK} \ \text{CODE}) \\
(\text{cdff} \ \text{count5} \ \text{or2} \ \text{or2} \ \text{to} \ \text{to} \ \text{f0} \ \text{f0} \ \text{comp}5 \ \text{comp}5) & \quad (\text{cdff} \ \text{count5} \ \text{or2} \ \text{or2} \ \text{to} \ \text{to} \ \text{f0} \ \text{f0} \ \text{comp}5 \ \text{comp}5) \\
((\text{CLK} \ \text{RST} \ \text{STOP} \ \text{S1}) \ (\text{CLK} \ \text{RST} \ \text{S2} \ \text{Q}) \ (\text{BIT} \ \text{Q}) \ (\text{STOP} \ \text{BIT}) \ () \ () & \quad ((\text{CLK} \ \text{RST} \ \text{STOP} \ \text{S1}) \ (\text{CLK} \ \text{RST} \ \text{S2} \ \text{Q}) \ (\text{BIT} \ \text{Q}) \ (\text{STOP} \ \text{BIT}) \ () \ ())
\end{align*}
\]

A state of \( \text{scount} \) is a list \((\text{on} \ \text{cnt})\) of two components, corresponding to the two sequential submodules, \( \text{cdff} \) and \( \text{count}5 \). As long as both data inputs are \( \mathcal{F} \), the value of \( \text{on} \) remains constant. While \( \text{on} = \mathcal{T} \), \text{cnt} is incremented repeatedly; while \( \text{on} = \mathcal{F} \), \text{cnt} remains unchanged. If either input is \( \mathcal{T} \), then \( \text{on} \) is set accordingly and \( \text{cnt} \) is reset to \( b_{vs}(0) \). The output values are both determined by \( \text{cnt} \):

**Proposition 5.6** Let \( \Sigma = (\text{on} \ \text{cnt}) \) and \( V = (\text{stop} \ \text{bit}) \) be a state and a data vector for \( \text{scount} \). Then

\[
\nu(\text{MARK}, V, \Sigma, \text{scount}) = \begin{cases} 
\mathcal{T} & \text{if } \text{cnt} = b_{vs}(4) \\
\mathcal{F} & \text{if } \text{cnt} \neq b_{vs}(4);
\end{cases}
\]

\[
\nu(\text{CODE}, V, \Sigma, \text{scount}) = \begin{cases} 
\mathcal{T} & \text{if } \text{cnt} = b_{vs}(17) \\
\mathcal{F} & \text{if } \text{cnt} \neq b_{vs}(17);
\end{cases}
\]

\[
\text{next}(V, \Sigma, \text{scount}) = \begin{cases} 
(\mathcal{F} b_{vs}(0)) & \text{if } \text{stop} = \mathcal{T} \\
(\mathcal{T} b_{vs}(0)) & \text{if } \text{stop} = \mathcal{F} \text{ and } \text{bit} = \mathcal{T} \\
(\mathcal{T} \ \text{inc}(\text{cnt})) & \text{if } \text{stop} = \text{bit} = \mathcal{F} \text{ and } \text{on} = \mathcal{T} \\
(\mathcal{F} \ \text{cnt}) & \text{if } \text{stop} = \text{bit} = \mathcal{F} \text{ and } \text{on} = \mathcal{F}.
\end{cases}
\]

The definition of \( \text{sndr} \) is as follows:
This module has two modes of operation. In one mode, it waits dormantly for the SEND input to become $T$. When this occurs, the current values of the other eight data inputs are loaded into the shift register, the state of the flip-flop $edff$ (which determines the output value) changes, and the controller $scount$ begins counting. This mode is described by the following:

**Proposition 5.7** Let $V = (s_{do} \ldots d_T)$ be a data vector for $sndr$, and let $\Sigma = (\sigma_1 \sigma_2 \sigma_3 \sigma_4)$ be a state of $sndr$, where $\sigma_1 = (on \ cnt)$. Assume that $on = F$ and $cnt = bs_5(0)$. Let $\Sigma' = next(V, \Sigma, sndr)$.

(a) If $s = T$, then $\Sigma' = ((T \ bs_5(0)) (d_0 \ldots d_T) \ \sigma_3 \ not1(\sigma_4))$;

(b) If $s = F$, then $\Sigma' = \Sigma$.

In the other mode of operation, the register contents are encoded and transmitted. Each register bit is encoded as a cell consisting of a 5-bit mark subcell and a 13-bit code subcell, as measured by $scount$. The number of cells that have been transmitted is recorded as the contents of $count3$. At the end of each mark subcell, this number is incremented. At the end of each code subcell, the $scount$ counter is reset and the register contents are shifted:
Proposition 5.8 Let $V = (s_0, \ldots, s_7)$ be a data vector for sndr, and let $\Sigma = (\sigma_1, \sigma_2, \sigma_3, \sigma_4)$ be a state of sndr, where $\sigma_1 = (on, cnt)$ and $\sigma_2 = (q_0, \ldots, q_7)$. Assume that $s = F$ and $on = T$. Let $\Sigma' = \text{next}(V, \Sigma, \text{sndr})$.

(a) If $cnt = bv_5(4)$ and $\sigma_3 = bv_3(7)$, then

$$\Sigma' = ((F \cdot bv_5(0)) \cdot \sigma_2 \cdot \text{inc}(\sigma_3) \cdot \text{xor}^2(q_7,\sigma_4)).$$

(b) If $cnt = bv_5(4)$ and $\sigma_3 \neq bv_3(7)$, then

$$\Sigma' = ((T \cdot bv_5(5)) \cdot \sigma_2 \cdot \text{inc}(\sigma_3) \cdot \text{xor}^2(q_7,\sigma_4)).$$

(c) If $cnt = bv_5(17)$, then

$$\Sigma' = ((T \cdot bv_5(0)) \cdot \text{shift}(F, \sigma_2) \cdot \sigma_3 \cdot \text{not}^1(\sigma_4)).$$

(d) If $cnt \neq bv_5(4)$ and $cnt \neq bv_5(17)$, then

$$\Sigma' = ((T \cdot \text{inc}(cnt)) \cdot \sigma_2 \cdot \sigma_3 \cdot \sigma_4).$$

Our main theorem on sndr is the following specification:

Proposition 5.9 Let $V = (V_{\text{END}}, V_0, \ldots, V_7)$ be a list of bit vectors, each of length $n \geq 144$. Let $m = n - 144$. Assume that for $j = 1, \ldots, n$,

$$nth(j, V_{\text{END}}) = \begin{cases} T & \text{if } j = m \\ F & \text{if } j \neq m. \end{cases}$$

Let $d_i = nth(m, V_i)$, for $i = 0, \ldots, 7$. Let $sv_j = sv(j, \text{SGUT}, V, \text{sndr})$, for $j = 1, \ldots, n$. Then $(sv_1, \ldots, sv_n) = \text{send}(d_7, \ldots, d_0, m, 5, 13, 0)$.

Proof: Let $\Sigma_j = \text{state}(j, V, \text{sndr})$, $j = 0, \ldots, n$. By Proposition 5.7(b), for $j = 0, \ldots, m$,

$$\Sigma_j = \Sigma_0(\text{sndr}) = (\langle F \cdot bv_5(0) \rangle \cdot \text{listn}(8, F) \cdot \text{bv}_3(0) \cdot F)$$

and hence $(sv_1, \ldots, sv_m) = \text{listn}(m, T)$. It remains to show that

$$(sv_{m+1}, \ldots, sv_n) = \text{cells}(T, 5, 13, (d_7, \ldots, d_0)).$$

By Proposition 5.7(a),

$$\Sigma_{m+1} = (\langle T \cdot bv_5(0) \rangle \cdot (d_0, \ldots, d_7) \cdot \text{bv}_3(0) \cdot T).$$

We shall show that for all $k, 0 \leq k \leq 7$, if

$$\Sigma_{m+1+18k} = (\langle T \cdot bv_5(0) \rangle \cdot \text{app}(\text{listn}(k, F), (d_0, \ldots, d_{7-k})) \cdot \text{bv}_3(k) \cdot x),$$

then

$$(sv_{m+1+18k}, \ldots, sv_n) = \text{cells}(x, 5, 13, (d_{7-k}, \ldots, d_0)).$$

The proposition will follow from this result upon setting $k = 0.$
The proof is by induction on \( 7 - k \). In the base case, \( k = 7 \), our assumption is that
\[
\Sigma_{m+1+18k} = \Sigma_{m+127} = ((T \text{bv}_0(0)) \text{ app}(\text{listn}(7, \mathcal{F}),(d_0)) \text{ bv}_3(7) x).
\]
By Proposition 5.8(d), for \( \ell = 0, \ldots, 4 \),
\[
\Sigma_{m+127+\ell} = ((T \text{bv}_0(\ell)) \text{ app}(\text{listn}(7, \mathcal{F}),(d_0)) \text{ bv}_3(7) x),
\]
and by Proposition 5.8(a),
\[
\Sigma_{m+127+5} = \Sigma_{m+132} = ((T \text{bv}_0(0)) \text{ app}(\text{listn}(7, \mathcal{F}),(d_0)) \text{ bv}_3(0) \text{ xor}(d_0, x)).
\]
By Proposition 5.7(b), \( \Sigma_{m+132+\ell} = \Sigma_{m+132} \) for \( \ell = 0, \ldots, 12 \). It follows that
\[
(\text{svm}_{m+127} \ldots \text{sv}_n) = \text{app}(\text{listn}(5, \text{not}(1(x))),\text{listn}(13,\text{equal}(d_0, x)))
= \text{cell}(x,5,13,d_0)
= \text{cells}(x,5,13, (d_0)).
\]
In the inductive case, \( k < 7 \), we again have, for \( \ell = 0, \ldots, 4 \),
\[
\Sigma_{m+1+18k+\ell} = ((T \text{bv}_0(\ell)) \text{ app}(\text{listn}(k, \mathcal{F}),(d_0 \ldots d_{7-k})) \text{ bv}_3(k) x)
\]
by Proposition 5.8(d). By Proposition 5.8(b) and (d), for \( \ell = 5, \ldots, 17 \),
\[
\Sigma_{m+1+18k+\ell} = ((T \text{bv}_0(\ell)) \text{ app}(\text{listn}(k, \mathcal{F}),(d_0 \ldots d_{7-k})) \text{ bv}_3(k+1) \text{ xor}(d_{7-k}, x)).
\]
Thus, \( (\text{svm}_{m+1+18k} \ldots \text{sv}_{m+1+18k+17}) \) is
\[
\text{app}(\text{listn}(5, \text{not}(1(x))),\text{listn}(13,\text{equal}(d_{7-k}, x))) = \text{cell}(x,5,13,d_{7-k}).
\]
By Proposition 5.8(c), \( \Sigma_{m+1+18(k+1)} \) is
\[
((T \text{bv}_0(0)) \text{ app}(\text{listn}(k+1, \mathcal{F}),(d_0 \ldots d_{7-(k+1)})) \text{ bv}_3(k+1) \text{ equal}(d_{7-k}, x)).
\]
It follows from our inductive hypothesis that
\[
(\text{sv}_{m+1+18(k+1)} \ldots \text{sv}_n) = \text{cells}(\text{equal}(d_{7-k}, x),5,13,(d_{7-(k+1)} \ldots d_0)),
\]
and hence \( (\text{sv}_{m+1+18k} \ldots \text{sv}_n) \) is
\[
\text{app}(\text{cell}(x,5,13,d_{7-k}),\text{cells}(\text{equal}(d_{7-k}, x),5,13,(d_{7-(k+1)} \ldots d_0))
= \text{cells}(x,5,13,(d_{7-k} \ldots d_0)). \quad \Box
\]

5.6 The Receiver

Its action of the receiver is controlled by a submodule, rcount, which is defined as follows:
The functionality of rcount is similar to that of scount. A state is again a list 
(on cnt) of two components, corresponding to the two sequential submodules, cdff and 
count5. As long as both data inputs are $F$, the value of on remains constant. While 
on = $T$, cnt is incremented repeatedly; while on = $F$, cnt remains unchanged. If STOP 
is $T$, then on and cnt are reset to $F$ and $bv_5(0)$; otherwise, if START is $T$, then on is set 
to $T$. The output value is determined by comparing cnt with $bv_5(9)$:

**Proposition 5.10** Let $\Sigma = (on \ cnt)$ and $V = (stop \ start)$ be a state and a data vector 
for rcount. Then

$$nu(BIT, V, \Sigma, rcount) = \begin{cases} 
T & \text{if cnt} = bv_5(9) \\
F & \text{if cnt} \neq bv_5(9);
\end{cases}$$

$$next(V, \Sigma, rcount) = \begin{cases} 
(F \ bv_5(0)) & \text{if stop} = T \\
(T \ inc(cnt)) & \text{if stop} = F \text{ and start} = on = T \\
(T \ cnt) & \text{if stop} = on = F \text{ and start} = T \\
(T \ inc(cnt)) & \text{if stop} = start = F \text{ and on} = T \\
(T \ cnt) & \text{if stop} = start = on = F.
\end{cases}$$

The definition of rcvr is as follows:
Like sndr, rcvr has two modes of operation. In the first mode, it waits for an edge, i.e., a change in input. This is detected by comparing the input with the state of the flip-flop edff, which is the negation of the most recently read value. In this mode, the controller rcount is turned off. When an edge is detected, rcount is turned on and its counter is reset:

**Proposition 5.11** Let \( V = (\text{sin}) \) be a data vector for \( \text{rcvr} \), and let \( \Sigma = (\sigma_1 \sigma_2 \sigma_3 \sigma_4 \sigma_5) \) be a state of \( \text{rcvr} \), where \( \sigma_1 = (\text{on cnt}) \). Assume that \( \text{on} = \mathcal{F} \), \( \text{cnt} = \text{bvs}(0) \), and \( \sigma_5 = \mathcal{F} \). Let \( \Sigma' = \text{next}(V, \Sigma, \text{rcvr}) \).

(a) If \( \text{sin} = \sigma_2 \), then \( \Sigma' = ((\mathcal{F} \text{bvs}(0)) \text{not1}(\text{sin}) \text{bvs}(0)) \text{shift}(\text{xor2}(\sigma_2, \text{sin}), \sigma_4) \mathcal{T}) \);

(b) If \( \text{sin} \neq \sigma_2 \), then \( \Sigma' = \Sigma \).

In its second mode, the receiver counts until it reaches the input bit to be sampled. At this point, the appropriate value is shifted into the register shift8, the bit counter count3 is incremented, the current input value is stored in edff, and rcount is turned off. When the eighth bit has been computed, the state of dff is altered to indicate termination:

**Proposition 5.12** Let \( V = (\text{sin}) \) be a data vector for \( \text{rcvr} \), and let \( \Sigma = (\sigma_1 \sigma_2 \sigma_3 \sigma_4 \sigma_5) \) be a state of \( \text{rcvr} \), where \( \sigma_1 = (\text{on cnt}) \). Assume that \( \text{on} = \mathcal{T} \) and \( \sigma_5 = \mathcal{F} \). Let \( \Sigma' = \text{next}(V, \Sigma, \text{rcvr}) \).

(a) If \( \text{cnt} = \text{bvs}(9) \) and \( \sigma_3 = \text{bvs}(7) \), then

\[
\Sigma' = ((\mathcal{F} \text{bvs}(0)) \text{not1}(\text{sin}) \text{bvs}(0)) \text{shift}(\text{xor2}(\sigma_2, \text{sin}), \sigma_4) \mathcal{T});
\]

(b) If \( \text{cnt} = \text{bvs}(9) \) and \( \sigma_3 \neq \text{bvs}(7) \), then

\[
\Sigma' = ((\mathcal{F} \text{bvs}(0)) \text{not1}(\text{sin}) \text{inc}(\sigma_3) \text{shift}(\text{xor2}(\sigma_2, \text{sin}), \sigma_4) \mathcal{F});
\]

(c) If \( \text{cnt} \neq \text{bvs}(9) \), then \( \Sigma' = ((\mathcal{T} \text{inc}(\text{cnt})) \sigma_2 \sigma_3 \sigma_4 \mathcal{F}) \).

The specification of \( \text{rcvr} \) is given by the following lemma. For its proof, we require the following definition: If \( L \) and \( M \) are two bit vectors, then

\[
\text{push}(L, M) = \begin{cases} 
M & \text{if } L = \text{NIL} \\
\text{push}(\text{cdr}(L), \text{shift}(\text{car}(L), M)) & \text{if } L \neq \text{NIL}.
\end{cases}
\]

Thus, if \( L = (x_1 \ldots x_{\ell}) \) and \( M = (y_1 \ldots y_m) \), where \( \ell \leq m \), then

\[
\text{push}(L, M) = (x_{\ell} \ldots x_1 y_1 \ldots y_{m-\ell}).
\]

48
Proposition 5.13 Let \( \mathcal{V} = (V) \), where \( V \) is a bit vector of length \( n \). Assume that
\[
\text{length}(\text{recv}(8, T, 10, V)) = 8.
\]
Then for some \( m \), \( 1 \leq m \leq n \),
\[
sv(j, \text{DONE}, \mathcal{V}, \text{rcvr}) = \begin{cases} 
T & \text{if } j = m \\
\mathcal{F} & \text{if } j < m.
\end{cases}
\]
For \( i = 1, \ldots, 7 \), let \( d_i = sv(m, 0i, \mathcal{V}, \text{rcvr}) \). Then
\[
(d_7 \ldots d_0) = \text{recv}(8, T, 10, V).
\]

Proof: Let \( V = (v_1 \ldots v_n) \). For \( j = 0, \ldots, n \), let \( V_j = (v_{j+1} \ldots v_n) \) and
\[
\Sigma_j = \text{state}(j, \mathcal{V}, \text{rcvr}) = ((\text{on}_j \text{cnt}_j) \text{flg}_j \text{bits}_j \text{reg}_j \text{done}_j).
\]
We shall prove the following generalization of the desired result:
Suppose that for some \( j \), \( \text{on}_j = \mathcal{F} \), \( \text{cnt}_j = \text{bs}_0(0) \), \( \text{done}_i = \mathcal{F} \) for all \( i < j \), and
\[
\text{length}(\text{recv}(8 - b, \text{not}(\text{flg}_j), 10, V_j)) = 8 - b,
\]
where \( \text{bits}_j = \text{bs}_3(b) \). Then for some \( m > j \), \( \text{done}_i = \mathcal{F} \) for all \( i < m \), \( \text{done}_m = \mathcal{T} \), and
\[
\text{reg}_m = \text{push}(\text{recv}(8 - b, \text{not}(\text{flg}_j), 10, V_j), \text{reg}_j).
\]
The proposition will then follow from the case \( j = 0 \).

First note that according to our assumption,
\[
\text{recv}(8 - b, \text{not}(\text{flg}_j), 10, V_j) \neq \text{NIL},
\]
and hence, \( \text{scan}(\text{not}(\text{flg}_j), V_j) = V_k \) for some \( k \), \( j \leq k < n - 10 \). Thus, \( v_i = \text{not}(\text{flg}_j) \) for \( i = j + 1, \ldots, k \), and \( v_{k+1} = \text{flg}_j \). From the definition of \( \text{recv} \), we have
\[
\text{recv}(8 - b, \text{not}(\text{flg}_j), 10, V_j) = \text{cons}(\text{xor}2(\text{flg}_j, v_{k+1}), \text{recv}(7 - k, v_{k+1}, 10, V_{k+1})),
\]
and hence,
\[
\text{length}(\text{recv}(7 - b, v_{k+1}, 10, V_{k+1}))) = 7 - b.
\]
By Proposition 5.11, \( \Sigma_i = \Sigma_j \) for \( i = j, \ldots, k \), and
\[
\Sigma_{k+1} = ((\mathcal{T} \text{bs}_5(0)) \text{flg}_j \text{bits}_j \text{reg}_j \mathcal{F}).
\]
By Proposition 5.12(c), for \( i = 0, \ldots, 9 \),
\[
\Sigma_{k+1+i} = ((\mathcal{T} \text{bs}_5(i)) \text{flg}_j \text{bits}_j \text{reg}_j \mathcal{F}).
\]
The proof is by induction on \( 7 - b \). Consider first the base case, \( b = 7 \). By Proposition 5.12(a),
\[
\Sigma_{k+11} = ((\mathcal{F} \text{bs}_5(0)) \not\text{not}(v_{k+11}) \text{bs}_3(0) \text{shift}(\text{xor}2(\text{flg}_j, v_{k+11}), \text{reg}_j) \mathcal{T})).
\]
Here, the result holds for \( m = k + 11 \), since
Now suppose that \( b < 7 \), and assume that the claim holds with \( b \) replaced with \( b + 1 \). By Proposition 5.12(a),

\[
\Sigma_{k+11} = ((\mathcal{F} \text{ bool}(0)) \ \text{not1}(v_{k+11}) \ \text{bool}(b + 1) \ \text{shift}(\text{xor2}(flg_j, v_{k+11}), reg_j)) \ \mathcal{F}.
\]

We may conclude that for some \( m > k + 11 \), \( \text{done}_i = \mathcal{F} \) for all \( i < m \), \( \text{done}_m = \mathcal{T} \), and\( \text{reg}_m = \text{push}(\text{recv}(7 - b, v_{k+11}, 10, V_{k+11}), \text{shift}(\text{xor2}(flg_j, v_{k+11}), reg_j)) \)

\[
= \text{push}(\text{cons}(\text{xor2}(flg_j, v_{k+11}), \text{recv}(7 - b, v_{k+11}, 10, V_{k+11})), \text{reg}_j)
\]

\[
= \text{push}(\text{recv}(8 - b, \text{not1}(flg_j), 10, V_j), \text{reg}_j).
\]

5.7 The Main Theorem

Finally, we present our main result concerning the circuit \( \text{bpm} \). We assume that the two clock input waveforms are admissible pulses for \( \text{sndr} \) and \( \text{rcvr} \), respectively, with periods that conform to the constraints imposed by Moore's theorem, and that the other inputs are well-behaved with respect to the clocks, as required by Theorem 3.1. We also assume that the \( \text{SEND} \) input has the value \( T \) on exactly one cycle, during which an 8-bit message is read from the other data inputs. This message is then encoded and transmitted by \( \text{sndr} \), and received, decoded, and output by \( \text{rcvr} \). As stated in the theorem, the completion of this process is signalled by the output \( \text{DONE} \): when its value first becomes \( T \), the other outputs display the decoded message.

**Theorem 5.2** Let \( p_{in} = (w_{\text{CLKS}} w_{\text{RSTS}} w_{\text{CLKR}} w_{\text{RSTR}} w_{\text{SEND}} w_0 \ldots w_7) \) be an input packet for \( \text{bpm} \), where

(a) \( (c\text{LKS} w_{\text{RSTS}} w_{\text{SEND}} w_0 \ldots w_7) \) is an admissible \( n_s \)-cycle input packet for \( \text{sndr} \) based at \( b_s \) with value matrix \( V_s = (V_{\text{SEND}} V_0 \ldots V_7) \) and period \( \pi_s \);

(b) \( w_{\text{CLKR}} \) is an admissible \( (n_r + 2) \)-cycle pulse for \( \text{rcvr} \) based at \( b_r \) with high \( h > 7000 \), low \( \ell > 7000 + \text{setup}(\text{SIN}, \text{rcvr}) \), and period \( \pi_r = h + \ell \);

(c) \( w_{\text{RSTR}} \) is an admissible \( (n_r + 1) \)-cycle reset waveform for \( \text{rcvr} \) based at \( b_r \) with period \( \pi_r \).

Assume \( 17\pi_r \leq 18\pi_s \leq 19\pi_r \). Suppose that for some \( m_s, 1 \leq m_s \leq n_s - 144 \),

\[
\text{nth}(j, V_{\text{SEND}}) = \begin{cases} 
T & \text{if } j = m_s \\
\mathcal{F} & \text{if } j \neq m_s, 1 \leq j \leq n_s;
\end{cases}
\]

For \( i = 0, \ldots, 7 \), let \( d_i = \text{nth}(m_s, V_i) \). Let \( t_r = b_r + \pi_r \). Assume that \( b_s + 2\pi_s \leq t_r \leq b_s + (m_s + 2)\pi_s \) and \( b_s + (n_s + 2)\pi_s \leq t_r + n_r\pi_r \).

Let \( p_{out} = \text{outp}(\text{bpm}, \text{sim}(\text{bpm}, p_{in}, t_f)) \), where \( t_f \geq t_r + n_r\pi_r \). Then \( p_{out} \) is a stable \( n_r \)-cycle packet based at \( t_r + \pi_r \) with value matrix \( V_r \) and period \( \pi_r \), for some \( V_r = (V_{\text{DONE}} V_0 \ldots V_7) \). For some \( m_r, 1 \leq m_r \leq n_r \),

\[
\text{nth}(j, V_{\text{DONE}}) = \begin{cases} 
T & \text{if } j = m_r \\
\mathcal{F} & \text{if } j \neq m_r, 1 \leq j \leq n_r;
\end{cases}
\]

and for \( i = 0, \ldots, 7 \), \( \text{nth}(m_r, V_0) = d_i \).
Proof: We may assume, without loss of generality, that \( n_s = m_s + 144 \). For \( j = 0, \ldots, n_s \), let \( sv_j = sv(j, \text{SOUT}, \nu_s, \text{sndr}) \). By Proposition 5.9,

\[
(sv_1 \ldots sv_{n_s}) = \text{send}((d_7 \ldots d_0), m_s, 5, 13, 0).
\]

Since \( sv_0 = \mathcal{T} \), we have \( sv_j = \mathcal{T} \) for all \( j \leq m_s \).

Fix \( j \) so that \( b_s + j\pi_s \leq t_r < b_s + (j + 1)\pi_s \), and let \( t_s = b_s + j\pi_s \). Then \( 2 \leq j \leq m_s + 2 \), and hence \( sv_{j-2} = \mathcal{T} \). Let

\[
S = (sv_{j-1} \ldots sv_{n_s}) = \text{send}((d_7 \ldots d_0), m_s - j + 2, 5, 13, 0)
\]

and let \( w_{\text{LOUT}} \) be the waveform for \( \text{LOUT} \) determined by \( \text{sim}(b_p, p, t_f) \). By Theorem 4.1, \( (w_{\text{CLK}}, w_{\text{ASTR}}, w_{\text{LOUT}}) \) is an admissible input packet for \( \text{rcvr} \) based at \( b_r \) with value matrix \( A \) and period \( \pi_r \), where

\[
A = \text{asynch}(U, t_s, t_r, \pi_s, \pi_r, \text{oracle})
\]

for some bit vector \( \text{oracle} \).

Let \( V_r = (V_{\text{DONE}} V_{0s} \ldots V_{0t}) \), where

\[
V_{\text{DONE}} = (sv(1, \text{DONE}, (A), \text{rcvr}) \ldots sv(n_r, \text{DONE}, (A), \text{rcvr}))
\]

and for \( i = 0, \ldots, 7 \),

\[
V_{0i} = (sv(1, 0_i, (A), \text{rcvr}) \ldots sv(n_r, 0_i, (A), \text{rcvr})).
\]

By Theorem 3.1, \( p_{out} \) is a stable \( n_r \)-cycle packet based at \( b_r + \pi_r + \pi_r = t_r + \pi_r \) with value matrix \( V_r \) and period \( \pi_r \).

According to Moore's Theorem, \( \text{recv}(8, \mathcal{T}, 10, A) = (d_7 \ldots d_0) \). But then, by Proposition 5.13, there exists \( m_r \) such that \( 1 \leq m_r \leq n_r \),

\[
nth(j, V_{\text{DONE}}) = \begin{cases} 
\mathcal{T} & \text{if } j = m_r \\
\mathcal{F} & \text{if } j \neq m_r, 1 \leq j \leq n_r,
\end{cases}
\]

and

\[
(nth(m_r, V_{0r}) \ldots nth(m_r, V_{0t})) = (d_7 \ldots d_0).
\]

Thus, for \( i = 0, \ldots, 7 \), \( nth(m_r, V_{0i}) = d_i \). \( \square \)

6 NASA's Reliable Computing Platform

The goal of NASA's RCP project is an implementation of a provably correct operating system that provides the application software developer a mechanism for dispatching periodic tasks on a fault-tolerant computing base that appears as a single ultra-reliable processor. The RCP may be modeled at four levels of abstraction:

1. The uniprocessor model;
2. The fault-tolerant synchronous replicated model;

51
(3) The fault-tolerant asynchronous replicated model;

(4) The hardware/software implementation.

At the second level, fault-tolerance is achieved by voting results computed by the replicated processors, which operate on the same sensor inputs, and are assumed to behave synchronously. A verified version of this model was reported in Task 1 [1].

At the third level, the assumptions of the synchronous model must be discharged. This requires (a) a mechanism for achieving synchronization among the clocks that drive the replicated processors and (b) a protocol for asynchronous communication. These were addressed in Tasks 2 [22] and 3 [15], respectively.

Final realization of the RCP at the hardware level requires an appropriate hardware description language that will allow the integration of these previous results in an implementable design. This was the primary motivation for the present effort. Thus, we have designed a language that provides for the modeling of asynchronous circuits, at a sufficiently low level to allow straightforward implementation. In addition, we have demonstrated a methodology for deriving and verifying comprehensive descriptions of the behavior of these circuits.

Our verification of the simple biphase mark circuit defined in Section 5 is a first step toward a verified RCP implementation. We would like to apply the same techniques, along with our previous results on Byzantine agreement and clock synchronization, to create a realistic implementation of a fault-tolerant circuit, verified at a greater level of detail than has been previously possible.

References


Appendix: Nqthm Formalization

A  Language Definition

; S-EXPRESSIONS

; Some basic definitions (the first 5 are from J's asynchrony file):

(defun listn (n value)
  (if (zerop n)
      nil
      (cons value
        (listn (sub1 n) value))))

(defun cdrn (n lst)
  (if (zerop n)
      lst
      (cdrn (sub1 n) (cdr lst))))

(defun nth (n lst)
  (car (cdrn n lst)))

(defun boolp (x)
  (or
   (equal x t)
   (equal x f)))

(defun bv_ (x)
  (if (nlistp x)
      (equal x nil)
      (and (boolp (car x))
        (bv_ (cdr x)))))

(defun bvpn (x n)
  (if (zerop n)
      (equal x ())
      (and (boolp (car x))
        (bvpn (cdr x) (sub1 n))))

(defun plistp (l)
  (if (listp l)
      (plistp (cdr l))
      (equal l ())))

(defun firstn (n l)
  (if (zerop n)
      ()
      (cons (car l) (firstn (sub1 n) (cdr l)))))

; Boolean terms and their evaluation:

(defun arities ()
  '((t0 0) (f0 0)
    (not1 1)
    (and2 2) (or2 2) (nand2 2) (nor2 2) (xor2 2)
    (and3 3) (or3 3) (nand3 3) (nor3 3) (xor3 3)
    (and4 4) (or4 4) (nand4 4) (nor4 4) (xor4 4)
    (and5 5) (or5 5) (nand5 5) (nor5 5) (xor5 5)))

54
(defn elemp (fn)
  (assoc fn (arities)))

(defn arity (fn)
  (cdr (assoc fn (arities))))

(defn termp$ (flg x l)
  (if (equal flg 'list)
      (if (listp x)
        (and (termp$ t (car x) l)
             (termp$ 'list (cdr x) l))
        t)
      (if (listp x)
        (and (elemp (car x))
             (equal (length (cdr x)) (arity (car x)))
             (termp$ 'list (cdr x) l))
        (member x l))))

(defn apply0 (fn)
  (case fn
    (tO t)
    (fO f)
    (otherwise f)))

(defn apply1 (fn x)
  (case fn
    (not1 (not x))
    (otherwise f)))

(defn apply2 (fn x y)
  (case fn
    (and2 (and x y))
    (or2 (or x y))
    (nand2 (not (and x y)))
    (nor2 (not (or x y)))
    (xor2 (not (equal x y)))
    (otherwise f)))

(defn apply3 (fn x y z)
  (case fn
    (and3 (and x y z))
    (or3 (or x y z))
    (nand3 (not (and x y z)))
    (nor3 (not (or x y z)))
    (xor3 (not (equal x (not (equal y z)))))
    (otherwise f)))

(defn apply4 (fn w x y z)
  (case fn
    (and4 (and w x y z))
    (or4 (or w x y z))
    (nand4 (not (and w x y z)))
    (nor4 (not (or w x y z)))
    (xor4 (not (equal w (not (equal x (not (equal y z)))))
             (otherwise f)))

(defn apply5 (fn v w x y z)
  (case fn
    (and5 (and v w x y z)))

55
(or5 (or v w x y z))
(nand5 (not (and v w x y z)))
(nor5 (not (or v w x y z)))
(xor5 (not (equal v (not (equal w (not (equal x (not (equal y z))))))))
(otherwise f))

(defn eval (x a)
  (if (listp x)
    (case (arity (car x))
      0 (apply0 (car x))
      1 (apply1 (car x) (eval (cadr x) a))
      2 (apply2 (car x) (eval (cadr x) a) (eval (caddr x) a))
      3 (apply3 (car x) (eval (cadr x) a) (eval (caddr x) a) (eval (cadddr x) a))
      4 (apply4 (car x) (eval (cadr x) a) (eval (caddr x) a) (eval (cadddr x) a) (eval (caddddr x) a))
      5 (apply5 (car x) (eval (cadr x) a) (eval (caddr x) a) (eval (cadddr x) a) (eval (caddddr x) a) (eval (cadddddd x) a))
      (otherwise f))
    (car (assoc x a))))

;; We define an "extended number" to be a number or F. (F represents infinity.) The following operations are defined on this set:

(defn e= (x y)
  (if x
    (if y
      (if (lessp x y) x y)
      x)
    y))

(defn e< (x y)
  (if x
    (if y
      (if (lessp x y) y x)
      x))

(defn eadd1 (x)
  (if x
    (add1 x)
    x))

(defn eplus (x y)
  (if y
    (if y
      y
      x)))

56
A waveform is a list ((v₀ . t₀) ... (v₁ . t₁) (v₀ . t₀)) of "events", each of which associates a Boolean value vᵢ with a time tᵢ at which the value is to be assumed by the associated signal. We require that 0 ≤ t₀ < t₁ < ... < tₙ and v₀ ≠ v₁ ≠ ... vₙ:

(DEFN WAVEP (W) 
  (IF (LISTP W) 
    (AND (BOOLP (CAAR W)) 
      (IF (LISTP (CDR W)) 
        (AND (WAVEP (CDR W)) 
          (NUMBERP (CDAR W)) 
          (LESP (CADR W) (CDAR W)) 
          (NOT (EQUAL (CADDR W) (CAAR W)))) 
        (AND (EQUAL (CDAR W) 0) 
          (EQUAL (CDR W) ()))))
  ))

A packet is a list of waveforms:

(DEFN PACKETP (L N) 
  (IF (ZEROP N) 
    (EQUAL 1 ()) 
    (AND (LISTP L) 
      (WAVEP (CAR L)) 
      (PACKETP (CDR L) (SUB1 N))))

The value of a signal at a given time is computed from its waveform as follows:

(DEFN WAVAL (WAVE TIME) 
  (IF (LISTP WAVE) 
    (IF (LESSP TIME (CDAR WAVE)) 
      (WVAL (CDR WAVE) TIME) 
      (CAAR WAVE))
  ))

(DEFN PVAL (PACKET TIME) 
  (IF (LISTP PACKET) 
    (CONS (WVAL (CAR PACKET) TIME) 
      (PVAL (CDR PACKET) TIME))
  ))

Histories:

(DEFN WHIST (WAVE TIME) 
  (IF (LISTP WAVE) 
    (IF (LESSP TIME (CDAR WAVE)) 
      (WHIST (CDR WAVE) TIME) 
      WAVE)
(defn phist (packet time)
  (if (listp packet)
    (cons (whist (car packet) time)
      (phist (cdr packet) time))
    ())))

;; To determine whether some waveform of a packet acquires a new value
;; at a given time:

(defun wnewp (wave time)
  (if (listp wave)
    (if (lessp time (cdar wave))
      (wnewp (cdr wave) time)
      (equal time (cdar wave)))
    f))

(defun pnevp (packet time)
  (if (listp packet)
    (or (wnewp (car packet) time)
      (pnevp (cdr packet) time))
    f))

;; The basic propagation functions:

(defun trans (w v tv)
  (if (listp w)
    (if (lessp (cdar w) tv)
      (if (equal (tsar w) v)
        w
        (cons (cons v tv) w))
      (trans (cdr w) v tv))
    f))

(defun inert (w v t0 tv)
  (if (listp w)
    (if (equal (wval w t0) v)
      (whist w t0)
      (if (lessp (cdar w) tv)
        (if (equal (caar w) v)
          (cons (car w) (whist w t0))
        (cons (cons v tv) (whist w t0)))
      (inert (cdr w) v t0 tv)))
    f))

;; A behavioral module is a list M = (BEHAV I O R P D), where
;; I is a list of litatoms, the inputs of M
;; O is a list of litatoms, the outputs of M
;; R is a list of elementary Boolean terms over I, corresponding to the outputs
;; D is a list of delays corresponding to the outputs
;; P is a list of modes (TRANS or INERT) corresponding to the outputs
(defn type (mod)
  ;a litatom
  (car mod))

(disable type)

(defn behavp (m) (equal (type m) 'behav))

(defn i (mod)
  ;a list of litatoms
  (cadr mod))

(disable i)

(defn o (mod)
  ;a list of litatoms
  (caddr mod))

(disable o)

(defn ni (mod)
  (length (i mod)))

(defn no (mod)
  (length (o mod)))

(defn r (mod)
  ;a list of Boolean terms
  (cadddr mod))

(defn d (mod)
  ;a list of positive numbers
  (caddddr mod))

(disable d)

(defn p (mod)
  ;a list of litatoms
  (cadddddr mod))

(disable p)

(defn distinct-symbols (l)
  (if (listp l)
      (and (litatom (car l))
           (not (member (car l) (cdr l)))
           (distinct-symbols (cdr l)))
      t))

(defn check-modes (modes)
  (if (listp modes)
      (and (member (car modes) '(trans inert))
           (check-modes (cdr modes)))
      t))

(defn check-delays (delays)
  (if (listp delays)
      (and (not (zerop (car delays)))
           (check-delays (cdr delays)))
      t))
(defn check-behav (m)
  (and (distinct-symbols (append (i m) (o m)))
    (equal (length (r m)) (length (o m)))
    (equal (length (d m)) (length (o m)))
    (check-delays (d m))
    (equal (length (p m)) (length (o m)))
    (check-modes (p m))))

(defn post-event (w v t0 mode delay)
  (case mode
    (trans (trans w v (plus t0 delay)))
    (inert (inert w v t0 (plus t0 delay)))
    (otherwise f)))

(defn post-events (packet outs pval t0 modes delays m)
  (if (listp packet)
    (cons (post-event (car packet)
                   (eval (car outs)
                         (pairlist (i m) pval))
               t0
               (car modes)
               (car delays))
    (post-events (cdr packet)
                 (cdr outs)
                 pval
t0
               (cdr modes)
               (cdr delays)
               m))
  )

;;The semantics of behavioral modules are defined by a function EXEC of
;;four arguments: (1) a module M, (2) an input packet INP, (3) an output packet
;;OUTP, and (4) a time TO. The value returned is the result of updating OUTP
;;by “executing” M on the input INP at time TO:
(被告 exec (m imp outp t0)
  (post-events outp (r m) (pval inp t0) t0 (p m) (d m) m))

;;Gates are modeled as behavioral modules with inertial delay:
(defn t0 ()
  '(behav () (t ((t0)) (2000) (inert))))

(defn f0 ()
  '(behav () (f ((f0)) (2000) (inert))))

(defn not1 ()
  '(behav (a) (b) ((not1 a)) (2000) (inert))))

(defn and2 ()
  '(behav (a b) (c) ((and2 a b)) (2000) (inert))))

(defn or2 ()
  '(behav (a b) (c) ((or2 a b)) (2000) (inert))))
(defn nand2 ()
  '(behav (a b) (c) ((nand2 a b)) (2000) (inert)))

(defn nor2 ()
  '(behav (a b) (c) ((nor2 a b)) (2000) (inert)))

(defn xor2 ()
  '(behav (a b) (c) ((xor2 a b)) (2000) (inert)))

(defn and3 ()
  '(behav (a b c) (d) ((and3 a b c)) (2000) (inert)))

(defn or3 ()
  '(behav (a b c) (d) ((or3 a b c)) (2000) (inert)))

(defn nand3 ()
  '(behav (a b c) (d) ((nand3 a b c)) (2000) (inert)))

(defn nor3 ()
  '(behav (a b c) (d) ((nor3 a b c)) (2000) (inert)))

(defn xor3 ()
  '(behav (a b c) (d) ((xor3 a b c)) (2000) (inert)))

(defn and4 ()
  '(behav (a b c d) (e) ((and4 a b c d)) (2000) (inert)))

(defn or4 ()
  '(behav (a b c d) (e) ((or4 a b c d)) (2000) (inert)))

(defn nand4 ()
  '(behav (a b c d) (e) ((nand4 a b c d)) (2000) (inert)))

(defn nor4 ()
  '(behav (a b c d) (e) ((nor4 a b c d)) (2000) (inert)))

(defn xor4 ()
  '(behav (a b c d) (e) ((xor4 a b c d)) (2000) (inert)))

(defn and5 ()
  '(behav (a b c d e) (g) ((and5 a b c d e)) (2000) (inert)))

(defn or5 ()
  '(behav (a b c d e) (g) ((or5 a b c d e)) (2000) (inert)))

(defn nand5 ()
  '(behav (a b c d e) (g) ((nand5 a b c d e)) (2000) (inert)))

(defn nor5 ()
  '(behav (a b c d e) (g) ((nor5 a b c d e)) (2000) (inert)))

(defn xor5 ()
  '(behav (a b c d e) (g) ((xor5 a b c d e)) (2000) (inert)))

;;********************************************************************
;;
;;  STRUCTURAL MODULES
;;
;;********************************************************************
a structural module is a list \( M = (\text{STRUCT I O S LI LO}) \), where

- \( I \) is a list of (global) inputs
- \( O \) is a list of (global) outputs
- \( S \) is a list of submodules
- \( LI \) is a list of local inputs: each member of \( LI \) is a list representing the inputs to the corresponding submodule
- \( LO \) is a list of local outputs: each member of \( LO \) is a list representing the outputs to the corresponding submodule

```
(defun structp (m) (equal (type m) 'struct))

(defun s (m)
  ;; a list of modules
  (cadddr m))

(disable s)

(defun li (m)
  ;; a list of lists of litatoms
  (caddddr m))

(disable li)

(defun lo (m)
  ;; a list of lists of litatoms
  (cadddddr m))

(disable lo)

(defun lookup1 (key keys list)
  (if (listp keys)
      (if (member key (car keys))
          (car list)
          (lookup1 key (cdr keys) (cdr list)))
      f))

(defun find-lo (out m)
  (lookup1 out (lo m) (lo m)))

(defun find-s (out m)
  (lookup1 out (lo m) (s m)))

(defun find-li (out m)
  (lookup1 out (lo m) (li m)))

(defun lookup (key keys list)
  (if (listp keys)
      (if (equal key (car keys))
          (lookup (cadr keys) (cadr list))
          f))

(defun find-o (out m)
  (lookup out (find-lo out m) (o (find-s out m))))

(defun match-inputs (subins subs)
  (if (listp subs)
      f))
```
(and (listp subins)
 (equal (length (car subins)) (length (car subs)))
 (match-inputs (cdr subins) (cdr subs))))
)

(defn match-outputs (subouts subs)
 (if (listp subs)
 (and (equal (length (car subouts)) (length (car subs)))
 (match-outputs (cdr subouts) (cdr subs)))
 t))

(defn appears (x l)
 (if (listp l)
 (or (member x (car l))
 (appears x (cdr l)))
 t)
)

(defn all-appear (1 m)
 (if (listp l)
 (and (appears (car l) m)
 (all-appear (cdr l) m))
 t)
)

(defn lists-all-appear (ls m)
 (if (listp ls)
 (and (all-appear (car ls) m)
 (lists-all-appear (cdr ls) m))
 t)
)

(defn none-appear (1 m)
 (if (listp l)
 (and (not (appears (car l) m))
 (none-appear (cdr l) m))
 t)
)

(defn all-distinct-symbols (ls)
 (if (listp ls)
 (and (distinct-symbols (car ls))
 (none-appear (car ls) (cdr ls))
 (all-distinct-symbols (cdr ls)))
 t)
)

(defn check-struct (m)
 (and (equal (length (li m)) (length (s m)))
 (match-inputs (li m) (s m))
 (equal (length (lo m)) (length (s m)))
 (match-outputs (lo m) (s m))
 (all-appear (o m) (lo m))
 (lists-all-appear (li m) (cons (i m) (lo m)))
 (all-distinct-symbols (cons (i m) (lo m))))
)

(prove-lemma lessp-count-submodules (rewrite)
 (implies (equal (type m) 'struct)
 (equal (lessp (count (s m)) (count m)) t))
 (enable s type))
)

(defn modulep$ (flag m)
 (if (equal flag 'list)
 (if (listp m)

(and (modulep (car m))
  (modulep 'list (cdr m))
  (equal m ()))
(case (type m)
  (struct (and (check-struct m)
    (modulep 'list (s m)))
    (behav (check-behav m))
    (otherwise ())))

(prove-lemma plistp-s ()
  (implies (modulep 'list s)
    (plistp s)))

(defun modulep (m)
  (modulep t m))

(prove-lemma plistp-s-m (rewrite)
  (implies (and (structp m) (modulep m))
    (plistp (s m)))
  (use (plistp-s (s (s m)))))

;; For a given structural module M, a bundle is an object that consists of
;; a waveform corresponding to each output of each behavioral component of M

(defun bundlep (flag b m)
  (if (equal flag 'list)
    (if (listp m)
      (and (bundlep t (car b) (car m))
       (bundlep 'list (cdr b) (cdr m)))
      (equal b ()))
    (if (structp m)
      (bundlep 'list b (s m))
      (packetp b (no m))))

(defun bundlep (b m) (bundlep t b m))

;; An output packet for M may be extracted from a bundle for M as follows:

(defun select-wave (key signals packets)
  (if (listp packets)
    (if (member key (car signals))
      (lookup key (car signals) (car packets))
      (select-wave key (cdr signals) (cdr packets))
    ())
  ())

(defun select-packet (keys signals packets)
  (if (listp keys)
    (cons (select-wave (car keys) signals packets)
      (select-packet (cdr keys) signals packets))
    ()))

(defun outp (flag m b)
  (if (equal flag 'list)
    (if (listp m)
      (cons (outp t (car m) (car b))
        (outp flag (cdr m) (cdr b)))
      ()))

64
(case (type m)
  (struct (select-packet (o m) (lo m) (outp$ 'list (s m) b)))
  (behav b)
  (otherwise f))))

(defn outp (m b) (outp$ t m b))

;;;;A list of input packets for the submodules of M may be extracted from
;;;;an input packet and a bundle for M as follows:

(defn input-packet (ins p b m)
  (select-packet ins
   (cons (i m) (lo m))
   (cons p (outp$ 'list (s m) b))))

(defn input-packets (ins p b m)
  (if (listp ins)
    (cons (input-packet (car ins) p b m)
      (input-packets (cdr ins) p b m))
    (listp)))

(defn inps (m p b)
  (input-packets (li m) p b m))

;;;;The semantics of structural modules are defined by a function STEP of
;;;;four arguments: (1) a module M, (2) an input packet P for M, (3) a bundle
;;;;B for M, and (4) a time T0. The value is the result of updating B by executing
;;;;each behavioral component of M for which some input acquires a new value
;;;;at time T0:

(defn step$ (flag m p b t0)
  (if (equal flag 'list)
    (if (listp m)
      (cons (step$ t (car m) (car p) (car b) t0)
        (step$ 'list (cdr m) (cdr p) (cdr b) t0))
      (case (type m)
        (struct (step$ 'list (s m) (inps m p b) b t0))
        (behav (if (pnevp p t0) (exec m p b t0) b))
        (otherwise f))))

(defn step (m p b t0) (step$ t m p b t0))

;;examples:

(defn adder2 ()
  `(struct (a b c) (1 b)
    ((a b) (a t1) (b t1) (t2 t3) (c t4) (t5 t4) (c t5) (t6 t1) (t7 t6))
    ((t1) (t2) (t3) (t4) (t5) (t6) (t7) (b) (1))))

(defn dff ()
  `(struct (clk rst d) (q qn)
    ((not1) (and2) (nand2) (nand2) (nand3) (nand2) (nand2) (nand2) (nand2))
    ((rst) (rn d) (b2 b1) (a1 clk) (b1 clk b2) (a2 dd) (b1 qn) (q a2))
    ((rn) (dd) (a1) (b1) (a2) (b2) (q) (qn) (q))})
(defn fnand2 ()
  "((behav (a b) (c) ((nand2 a b)) (1000) (inert)))"
)

(defn dlatch ()
  "(struct (clk d) (s2)
         ,(not1) ,(nand2) ,(nand2) ,(fnand2))
  ( ((clk) (clk d) (s1 s3) (s0 s2))
   ( (s0) (s1) (s2) (s3)))"
)

*************************************************************************
;;
SIMULATION
*************************************************************************

;;;;The top-level simulation function SIM takes three arguments: (1) a module
;;;;M, (2) an input packet P for M, and (3) a termination time TF. The value
;;;;returned is the bundle produced by simulating M with input P over the
;;;;interval from 0 to TF.

;;;;The time at which each simulation cycle occurs is computed by the function
;;;;TNEXT. Its arguments are (1) the time TO of the last simulation cycle,
;;;;(2) the input packet P, (3) the current bundle B, and (4) the module M.
;;;;The value returned is the time of the earliest event occurring in either
;;;;P or B that is later than TO, if such an event exists, and F otherwise.

(defun tnextw (wave to)
  (if (listp wave)
      (if (lessp to (cdar wave))
          (if (lesseq to (cdadr wave))
              (tnextw (cdr wave) to)
              (cdar wave))
          f)
      f))

(defun tnextp (p to)
  (if (listp p)
      (emin (tnextw (car p) to)
           (tnextp (cdr p) to))
      f))

(defun tnextb$ (flag bun m to)
  (if (equal flag 'list)
      (if (listp m)
          (emin (tnextb$ t (car bun) (car m) to)
                (tnextb$ 'list (cdr bun) (cdr m) to))
          f)
      (case (type m)
            (struc (tnextb$ 'list bun (s m) to))
            (behav (tnextp bun to))
            (otherwise f))))

(defun tnext (to p b m)
  (emin (tnextp p to) (tnextb$ t b m to)))

;;;;The function RUN is the guts of the simulator. Its arguments are
;;;;(1) a module M, (2) an input packet P, (3) an initial bundle B,
;;;;(4) an initial time TO, and (5) a termination time TF. It simulates
;;;;M over the interval from TO to TF, repeatedly calling STEP.
(prove-lemma lessp-tnextw (rewrite)
  (implies (tnextw p t0) (lessp t0 (tnextw p t0))))

(prove-lemma lessp-tnextp (rewrite)
  (implies (tnextp p t0) (lessp t0 (tnextp p t0))))

(prove-lemma lessp-tnext-b (rewrite)
  (implies (tnextb$ flag b m t0) (lessp t0 (tnextb$ flag b m t0))))

(prove-lemma lessp-tnext (rewrite)
  (implies (tnext p b m) (lessp t0 (tnext p b m))))

(defun run (m p b t0)
  (let ((tnext (tnext t0 p b m)))
    (if (and tnext (leq tnext tf))
      (run m p (step m p b tnext) tnext tf)
      ((lessp (difference tf t0))))
    ))

;; SIM calls RUN with an initial time T0 = 0 and an initial bundle that
;; is computed by first associating the trivial waveform ((F . 0)) with
;; each signal of M, and then executing every behavioral component of M:

(defun wO 0 '((,f ,0)))

(defun bO$ (flg m)
  (if (equal flg 'list)
    (if (listp m)
      (cons (bO$ t (car m)) (bO$ 'list (cdr m)))
      ()
    )
    (case (type m)
      (struct (bO$ 'list (s m)))
      (behav (listn (no m) (wO)))
      (otherwise f)))))

(defun bO (m) (bO$ t m))

(defun init$ (flg m p)
  (if (equal flg 'list)
    (if (listp m)
      (cons (init$ t (car m) (car p)) (init$ 'list (cdr m) (cdr p)))
      ()
    )
    (case (type m)
      (struct (init$ 'list (s m) (insps m p (bO $ m)))
      (behav (exec m p (bO m) 0))
      (otherwise f)))))

(defun init (m p)
  (init$ t m p))

(defun sim (m p tf)
  (run m p (init m p) 0 tf))
B Properties of the Simulator

;;;---------------------------------------------------------------------
;;; WAVEFORMS AND PROPAGATION
;;;---------------------------------------------------------------------

;;; The value of a waveform at any time is a Boolean:
(prove-lemma boolp-wval (rewrite)
  (implies (wavep w)
    (boolp (wval w t0)))))

;;; The value of a packet at any time is a bit vector:
(prove-lemma bvp-pval (rewrite)
  (implies (packetp p n)
    (bvpn (pval p t0) n))
    ((disable boolp)))

;;; Any history of a waveform is a waveform:
(prove-lemma wavep-whist (rewrite)
  (implies (wavep w)
    (wavep (whist w t0)))))

(prove-lemma listp-whist (rewrite)
  (implies (wavep w)
    (listp (whist w t0)))))

;;; The history of a waveform w w.r.t. at time T0 has the same value
;;; at T0 as W:
(prove-lemma whist-value (rewrite)
  (equal (wval (whist w t0) t0)
    (wval w t0)))))

(prove-lemma wval-caar-whist (rewrite)
  (implies (wavep w)
    (equal (wval w t0) (caar (whist w t0)))))

(disable wval-caar-whist)

(prove-lemma leq-cdar-whist-t0 (rewrite)
  (implies (wavep w)
    (not (lessp t0 (cdar (whist w t0))))))

(prove-lemma lessp-cdar-whist (rewrite)
  (implies (and (wavep w)
    (not (equal (wval w t0) (caar w))))
    (lessp (cdar (whist w t0)) (cdar w)))))

;;; The history of W w.r.t. T0 has a constant value for all T1 > T0:
(prove-lemma wval-whist (rewrite)
  (implies (and (wavep w)
    (leq t0 t1))
    (equal (wval (whist w t0) t1)
    (caar (whist w t0)))))

68
prove-lemma leq-cdar-whist (rewrite)
   (not (lessp t0 (cdar (whist w t0))))

(prove-lemma leq-cdar-whist-t0-revrite (rewrite)
   (implies (and (wavep w)
      (lessp t0 tv))
   (equal (lessp (cdar (whist w t0)) tv t))
   ((use (leq-cdar-whist-t0))))

;; Both propagation functions, TRANS and INERT, transform waveforms:
(prove-lemma wavep-trans (rewrite)
   (implies (and (wavep w)
      (boolp v)
      (not (zerop t0)))
   (wavep (trans w v t0)))

(prove-lemma wavep-inert (rewrite)
   (implies (and (wavep w)
      (boolp v)
      (lessp t0 tv))
   (wavep (inert w v t0 tv))
   ((induct (inert w v t0 tv))
   (disable boolp)
   (enable wval-caar-whist)))

;; Both propagation functions are "nonretroactive", i.e., do not alter the history of a waveform w.r.t. the current time:
(prove-lemma trans-nonretroactive (rewrite)
   (implies (and (wavep wave)
      (lessp t0 t1))
   (equal (whist (trans wave val t1) t0)
   (whist wave t0)))

(prove-lemma inert-nonretroactive (rewrite)
   (implies (and (wavep wave)
      (lessp t0 tv))
   (equal (whist (inert wave val t0 tv) t0)
   (whist wave t0))
   ((induct (inert wave val t0 tv))
   (disable boolp)
   (enable wval-caar-whist)))

;; The predicate WCONP determines whether a waveform W has a constant value V over a time interval (T1,T2):
(defun wconp (v v tl t2)
   (if (liatp w)
      (if (leeep (cdar v) t2)
       (and (leq (cdar v) tl)
       (equal (caar v) v))
       (wconp (cdr v) v tl t2))
   (f))

(prove-lemma wval-wconp (rewrite)
   (implies (and (wconp v v t1 t2)
      (wavep v)
      (leq t1 tp))
(lessp tp t2))
  (equal (vval w tp) v)))

;;; The waveform \(\text{TRANS } W \ V \ TV\) has the constant value \(V\)
;;; for all \(T2 \geq TV\):

(prove-lemma wcomp-trans-1 (rewrite)
  (implies (and (wavep w)
                (not (zerop tv))
                (lessp tv t2))
           (wcomp (trans \(W \ V\) tv) v tv t2)))

;;; The waveform \(\text{INERT } W \ V \ TO \ TV\) has the constant value \(V\)
;;; for all \(T2 \geq TV\):

(prove-lemma wcomp-inert-1 (rewrite)
  (implies (and (wavep w)
                (lessp t0 tv)
                (lessp tv t2))
           (wcomp (inert \(W \ V\) tv0 tv) v tv t2))

          ((enable \(vval\)-caar-whist))))

;;; If \(W\) has the constant value \(U\) over \([T1, T2)\), where \(T1 \leq T2 \leq TV\),
;;; then so does \(\text{TRANS } W \ V \ TV\):

(prove-lemma wcomp-trans-2 (rewrite)
  (implies (and (wavep w)
                (wcomp \(W \ U\) t1 t2)
                (leq t1 t2)
                (leq t2 tv)
                (not (zerop t2)))
           (wcomp (trans \(W \ V\) tv) u t1 t2)))

;;; If \(W\) has the constant value \(U\) over \([T1, T2)\), where
;;; \(T1 \leq T0 \leq T2 \leq TV\), then so does \(\text{INERT } W \ V \ TO \ TV\):

(prove-lemma wcomp-inert-2 (rewrite)
  (implies (and (wavep w)
                (wcomp \(W \ U\) t1 t2)
                (lessp t0 tv)
                (leq t1 t0)
                (leq t0 tv)
                (leq t2 tv))
           (wcomp (inert \(W \ V\) t0 tv) u t1 t2)))

;;; Both propagation functions are "idempotent" in the following sense:

(prove-lemma trans-trans (rewrite)
  (implies (and (wavep w)
                (leq tv1 tv2))
           (equal (trans (trans \(W \ V\) tv1) v tv2)
                  (trans \(W \ V\) tv1))))

(prove-lemma inert-inert (rewrite)
  (implies (and (wavep w)
                (lessp t01 tv1) (lessp t02 tv2)
                (lessp t01 t02) (lessp tv1 tv2))
           (equal (inert (inert \(W \ V\) t01 tv1) v t02 tv2)
                  (inert \(W \ V\) t01 tv1))))
((induct (inert w v t0 tvi))
  (enable wval-caar-whist)))

(disable trans)

(disable inert)

;;-----------------------------------------------------------------------
;; BEHAVIORAL MODULES
;;-----------------------------------------------------------------------

;; Execution of a behavioral module depends only on the current value of
;; the input (i.e., it is independent of both past and future input):

(prove-lemma exec-comb (rewrite)
  (implies (equal (pval pi to) (pval p2 t0))
    (equal (equal (exec m pi pout t0) (exec m p2 pout t0)) t0))
)

(prove-lemma exec-nonret-1 ()
  (implies (and (check-delays d) (equal (length d) n) (check-modes pm) (equal (length pm) n) (packetc pout n))
    (equal (phiest (post-events pout r inv t0 pm d m) t0) (phiest pout t0)))))

;; Execution is "nonretroactive", i.e., does not alter the history of
;; the output packet:

(prove-lemma exec-nonretroactive (rewrite)
  (implies (and (modulep m) (behavp m) (packetc pout (no m)) (equal (phiest (exec m pin pout t0) t0) (phiest pout t0))
    (use (exec-nonret-1 (d (d m)) (pm (p m)) (n (no m)) (r (r m)) (inv (pval pin t0))))))

(prove-lemma exec-idem-1 ()
  (implies (and (check-delays d) (equal (length d) n) (check-modes pm) (equal (length pm) n) (packetc pout n) (lessp to ti))
    (equal (post-events (post-events pout r inv t0 pm d m) r inv ti pm d m) (post-events pout r inv t0 pm d m))))

;; Execution is "idempotent" in the following sense:

(prove-lemma exec-idempotent (rewrite)
  (implies (and (modulep m) (behavp m) (packetc pout (no m)) (lessp to ti) (equal (pval pin t0) (pval pin ti)))
    (equal (exec m pin (exec m pin pout t0) ti) (exec m pin pout t0))
    (use (exec-idem-1 (d (d m)) (pm (p m)) (n (no m)) (r (r m)) (inv (pval pin t0))))))

71
We shall prove that under normal conditions, execution always produces a valid output packet. We must first show that evaluation of a Boolean term always produces a Boolean value:

(prove-lemma boolp-apply0 (rewrite)
  (boolp (apply0 fn))
)

(prove-lemma boolp-apply1 (rewrite)
  (boolp (apply1 fn x))
)

(prove-lemma boolp-apply2 (rewrite)
  (boolp (apply2 fn x y))
)

(prove-lemma boolp-apply3 (rewrite)
  (boolp (apply3 fn x y z))
)

(prove-lemma boolp-apply4 (rewrite)
  (boolp (apply4 fn w x y z))
)

(prove-lemma boolp-apply5 (rewrite)
  (boolp (apply5 fn w w x y z))
)

(prove-lemma boolp-eval-list (rewrite)
  (implies (listp x)
    (boolp (eval x a))
  )
)

(prove-lemma boolp-eval-nil (rewrite)
  (implies (and (termp t term i) (nilp term)
    (bvpn pval (length i)))
    (boolp (eval term (pairlist i pval)))))

(prove-lemma boolp-eval (rewrite)
  (implies (and (termp t term i) (bvpn pval (length i)))
    (boolp (eval term (pairlist i pval))))
)

(defn ppe-induct (d pm r pout n)
  (if (zerop n)
    t
    (ppe-induct (cdr d) (cdr pm) (cdr r) (cdr pout) (sub1 n)))
)

(prove-lemma packetp-post-events ()
  (implies (and (check-delays d) (equal (length d) n)
    (check-modes pm) (equal (length pm) n)
    (termp t list r (i m)) (equal (length r) n)
    (bvpn inv (length (i m)))
    (packetp pout n))
    (packetp (post-events pout r inv t0 pm d m) n))
)

(prove-lemma packetp-exec (rewrite)
  (implies (and (modulep m)
    (behev p m)
    (packetp pin (length (i m))))
)
(packetp pout (length (o m)))
(packetp (exec m pin pout t0) (length (o m)))
(use (packetp-post-events
(d (d m)) (pm (p m)) (r (r m)) (n (no m)) (inv (pval pin t0)))))

;;;;; We extend the notion of "history" to bundles in the natural way:
(defn bhist$ (flag b m t0)
 (if (equal flag 'list)
  (if (listp m)
  (cons (bhist$ t (car b) (car m) t0)
  (bhist$ 'list (cdr b) (cdr m) t0))
  (if (structp m)
  (bhist$ 'list b (s m) t0)
  (phist b t0))))
)

(defn bhist (b m t0)
 (bhist$ t b m t0))

(prove-lem_a stepi-nonrst ()
 (implies 
 (and 
 (modulep$ flag m)
 (bundlep$ flag b m))
 (equal (bhist$ flag (steps flag m p b t0) m t0)
 (bhist$ flag b m t0)))
 ((disable exec))
)

;;;; We extend the notion of "history" to bundles in the natural way:

(prove-lemma step-nonret ()
 (implies (and (modulep m)
 (bundlep b m))
 (equal (bhist (step m p b t0) m t0)
 (bhist b m t0)))
 ((use (step$-nonret (flag t)))))

(prove-lemma whist-lookup (rewrite)
 (implies (equal (phist p1 t0) (phist p2 t0))
 (equal (equal (whist (lookup z v p1) t0)
 (whist (lookup z v p2) t0))
 t)))

(defn phist$ (flag p t0)
 (if (equal flag 'list)
  (if (listp p)
  (cons (phist$ t (car p) t0)
  (phist$ 'list (cdr p) t0))
  ()
  (phist p t0)))
)

(prove-lemma whist-select-wave (rewrite)
 (implies (equal (phist$ 'list p1 t0)
(prove-lemma phist-select-packet (rewrite)
  (implies (equal (phist 'list pl t0)
                   (phist 'list p2 t0))
           (equal (equal (phist (select-packet subouts pl) t0)
                      (phist (select-packet subouts p2) t0))
                   t))))

(prove-lemma phist$-outp$ (rewrite)
  (implies (equal (bhist$ flag b1 m t0) (bhist$ flag b2 m t0))
           (equal (equal (pbist pl t0)
                         (phist p2 t0))
                    t)))

(prove-lemma history-outp-submodules (rewrite)
  (implies (and (structp m)
                (equal (bhist b1 m t0)
                       (bhist b2 m t0))
                (equal (equal (phist 'list (outp 'list (s m) b1) t0)
                           (phist 'list (outp 'list (s m) b2) t0))
                        t))
           (disable input-packet)
           (induct (input-packets li imp s m)))))

(prove-lemma phist$-inps-2 (rewrite)
  (implies (and (structp m)
                (equal (phist flag pX t0)
                       (phist flag p2 t0))
                (not (equal flag 'list))
                (equal (equal (pbis_$ 'list (inps m pl b) t0)
                           (phist$ 'list (inps m p2 b) t0))
                        t))
           (prove-lemma phist$-input-packets (rewrite)
  (implies (and (structp m)
                (equal (phist p1 t0)
                       (phist p2 t0))
                (equal (equal (phist$ 'list (input-packets li1 pl b1 m) t0)
                           (phist$ 'list (input-packets li1 p2 b2 m) t0))
                        t))
           (prove-lemma phist-input-packet (rewrite)
  (implies (equal (bhist b1 m t0)
                   (bhist b2 m t0))
           (equal (equal (phist pl t0)
                         (phist p2 t0)))
           (prove-lemma phist-input-packets (rewrite)
  (implies (and (structp m)
                (equal (phist$ 'list pl b) t0)
                (phist$ 'list p2 t0))
                (equal (equal (phist$ 'list (input-packets li1 pl b1 m) t0)
                              (phist$ 'list (input-packets li1 p2 b2 m) t0))
                           t))
           (prove-lemma phist-input-packet (rewrite)
  (implies (equal (phist pl t0)
                   (phist p2 t0))
           (equal (equal (phist$ 'list pl b) t0)
                   (phist$ 'list p2 t0))
           (prove-lemma whist-vnewp (rewrite)
  (implies (and (vnevp el t0)
                (not (equal el 'list))
                (equal (equal (phist$ 'list pl b) t0)
                       (phist$ 'list p2 b2 m) t0))
           (disable input-packet)
           (induct (input-packets li imp s m))))

(prove-lemma phist$-inps-2 (rewrite)
  (implies (and (structp m)
                (equal (phist flag pl t0)
                       (phist flag p2 t0))
                (not (equal flag 'list))
                (equal (equal (phist$ 'list (inps m pl b) t0)
                              (phist$ 'list (inps m p2 b) t0))
                           t))
           (prove-lemma phist-input-packets (rewrite)
  (implies (and (structp m)
                (equal (phist$ 'list pl b) t0)
                (phist$ 'list p2 t0))
                (equal (equal (phist$ 'list (input-packets li1 pl b1 m) t0)
                              (phist$ 'list (input-packets li1 p2 b2 m) t0))
                           t))
           (prove-lemma whist-vnewp (rewrite)
  (implies (and (vnevp w1 t0)
(equal (whist w1 t0) (whist w2 t0))

(wnevp w2 t0))

(defun list-2-induct (x y)
  (if (listp x)
      (list-2-induct (cdr x) (cdr y))
      t))

(prove-lemma phist-pnepv (rewrite)
  (implies (and (pnevp pl t0)
                 (equal (phist pl t0) (phist p2 t0)))
           (pnevp p2 t0))
  ((induct (list-2-induct pl p2))))

(prove-lemma pval-phist ()
  (equal (pval (phist p t0) t0)
         (pval p t0)))

(prove-lemma equal-phist-pval (rewrite)
  (implies (equal (phist pl t0) (phist p2 t0))
           (equal (equal (pval pl t0) (pval p2 t0))
                  t))
  ((use (pval-phist (p pl)) (pval-phist (p p2))))

(defun sn-induct (flag m b pi p2)
  (if (equal flag 'list)
      (if (listp m)
          (and (sn-induct t (car m) (car b) (car pl) (car p2))
               (sn-induct flag (cdr m) (cdr b) (cdr pl) (cdr p2))
          t)
      (if (structp m)
          (sn-induct 'list (s m) b (inps m pl b) (inps m p2 b))
          t))
  ((disable exec)
   (induct (sn-induct flag m b pl p2))))

(prove-lemma step-nonpred-1 ()
  (implies (and (modulep flag m)
                 (bundlep flag b m)
                 (equal (phist flag pl t0) (phist flag p2 t0))
                 (equal (step flag pl b t0) (step flag p2 b t0)))
           (induct (sn-induct flag m b pl p2))))

:; Unlike EXEC, STEP depends in general on the history (and not merely
:; the current values) of the input. However, STEP is "nonpredictive";
:; i.e., independent of future input:

(prove-lemma step-nonpredictive (rewrite)
  (implies (and (modulep m)
                 (bundlep b m)
                 (equal (phist pl t0) (phist p2 t0))
                 (equal (step pl b t0) (step p2 b t0))
                 t))
  ((use (step-nonpred-1 (flag t))))

:; INPACKETP tests whether P is a valid input packet for M:

(defun inpacketp (p m)
  (packetp p (length (i m))))

75
(defn inpacketp$ (flag p m)
  (if (equal flag 'list)
    (if (listp m)
      (and
        (inpacketp (car p) (car m))
        (inpacketp$ 'list (cdr p) (cdr m)))
      t)
    (inpacketp p m)))

(prove-lemma wavep-lookup (rewrite)
  (implies (and (packetp w n)
    (equal (length v) n)
    (member z v))
    (wavep (lookup z v w))))

(defn packetp$ (flag p n)
  (if (equal flag 'list)
    (if (listp n)
      (and (packetp (car p) (car n))
        (packetp$ 'list (cdr p) (cdr n)))
      t)
    (packetp p n)))

(defn lengths (flag I)
  (if (equal flag 'list)
    (if (listp I)
      (cons (length (car I)) (lengths 'list (cdr I)))
      o)
    (length I)))

(prove-lemma vavep-select-vave (rewrite)
  (implies (and (packetp$ 'list p ns)
    (equal (lengths 'list subouts) ne)
    (appears z subouts))
    (wavep (select-vave z subouts p)))))

(prove-lemma packetp$-select-packet (rewrite)
  (implies (and (packetp$ 'list p ns)
    (equal (lengths 'list subouts) ns)
    (all-appear sours subouts))
    (packetp (select-packet sours subouts p) (length subouts))))

(defn no$ (flag mod)
  (if (equal flag 'list)
    (if (listp mod)
      (cons (no (car mod))
        (no$ 'list (cdr mod)))
      ()
      (no mod)))

(prove-lemma match-outputs-length$ (rewrite)
  (implies (and (match-outputs x y)
    (equal (length x) (length y)))
    (equal (length$ 'list x) (no$ 'list y))))

(prove-lemma packetp-output-packet-1 (rewrite)
  (implies (and (packetp$ 'list (outp$ 'list s (s mod))
    (no$ 'list (s mod)))))

76
(structp mod)
(modulep mod)
    (packetp (select-packet (o mod)
    (lo mod)
    (outp$ 'list s (s mod))
    (no mod)))

(prove-lemma packetp$-outp$ (rewrite)
    (implies (and (modulep$ flag m)
    (bundlep$ flag b m))
    (packetp$ flag
    (outp$ flag a b)
    (no$ flag m))
    ((disable packetp)))

(prove-lemma packetp$-outp$-list (rewrite)
    (implies (and (structp mod)
    (modulep mod)
    (bundlep b mod))
    (packetp$ 'list
    (outp$ 'list s mod b)
    (no$ 'list s mod)))

(prove-lemma packetp-length (rewrite)
    (implies (packetp p n)
    (packetp p (length p))))

(prove-lemma packetp$-cons-inp-outs (rewrite)
    (implies (and (structp m)
    (modulep m)
    (bundlep b m)
    (inpacketp p m))
    (packetp$ 'list
    (cons p (outp$ 'list s m b)
    (cons (length p) (lengthS 'list (lo m))))
    ((disable bundlep)))

(prove-lemma packetp$-select-packet-2 (rewrite)
    (implies (and (packetp$ 'list p (lengthS 'list p))
    (equal (lengthS 'list subouts) (lengthS 'list p))
    (all-appear subouts subouts))
    (packetp (select-packet subouts subouts p)
    (length subouts)))

(prove-lemma length-select-packet (rewrite)
    (equal (length (select-packet x y z))
    (length x)))

(prove-lemma length-packet (rewrite)
    (implies (packetp p n)
    (equal (length p) (fix n)))

(prove-lemma length-outp (rewrite)
    (implies (and (bundlep b m)
    (modulep m))
    (equal (length (outp$ t m b))
    (no m))
    ((expand (outp$ t m b))))

77
(prove-lemma length$-outp$ (rewrite)
  (implies (and (bundlep$ flag b m)
                (modulep$ flag m))
           (equal (length$ flag (outp$ flag m b))
                  (no$ flag m))))

(prove-lemma length-lo ()
  (implies (and (modulep m)
                (structp m))
           (equal (no$ 'list (s m))
                  (length$ 'list (lo m))))))

(prove-lemma packetp-input-packet (rewrite)
  (implies (and (modulep m)
                (bundlep b m)
                (inpacketp p m)
                (all-appear ins (cons (i m) (lo m))))
           (packetp (input-packet ins p b m) (length ins)))
  ((disable packetp$ length-packet)
   (use (length-lo)
        (length-packet (m (length (i m))))))
  (expand (bundlep$ t b m))))

(prove-lemma packetsp-input-packets (rewrite)
  (implies (and (modulep m)
                (bundlep b m)
                (inpacketp p m)
                (lists-all-appear li (cons (i m) (lo m))))
           (packetp$ 'list
                    (input-packets li p b m)
                    (lengthS 'list li)))
  ((disable input-packet)
   (induct (input-packets li p b m))))

(defn ni$ (flag m)
  (if (equal flag 'list)
      (if (listp m)
       (cons (hi (car m))
             (ni$ 'list (cdr m)))
       (ni m))
  (ni m))

(prove-lemma inpacketp$-packetp$ ()
  (equal (inpacketp$ 'list p s)
         (packetp$ 'list p (ni$ 'list s))))

(prove-lemma match-inputs-length$ (rewrite)
  (implies (and (match-inputs x y)
                (equal (length x) (length y)))
           (equal (lengthS 'list x) (ni$ 'list y))))

(prove-lemma packetp$-li ()
  (implies (and (modulep m)
                (structp m))
           (equal (inpacketp$ 'list p (s m))
                  (packetp$ 'list p (lengthS 'list (li m))))
           ((use (inpacketp$-packetp$ (s (s m))))))

78
(prove-lemma inpacketp$-inps (rewrite)
  (implies (and (structp m)

                (modulep m)

                (bundlep b m)

                (inpacketp p m))

  ((expand (modulep$ t m))

   (disable match-inputs-length$)

   (use (packetp$-li (p (inps m p b))))))

(prove-lemma bundlep$-step$ ()
  (implies (and (modulep$ flag m)

                (inpacketp$ flag p m)

                (bundlep$ flag b m))

  ((bundlep$ flag (step$ flag m p b t0) m))

  ((disable exec check-behav inps))))

;; Under normal conditions, STEP always produces a valid bundle:

(prove-lemma bundlep-step (rewrite)
  (implies (and (modulep m)

            (inpacketp p m))

  (bundlep (step m p b t0) m))

  ((use (bundlep$-step$ (flag t)))))

;;********************************************************************************
;; SIMULATION
;;********************************************************************************

(prove-lemma whist-whist ()
  (implies (leq t0 t1)

    (equal (whist w t0)

            (whist (whist w t1) t0))))

(prove-lemma equal-whist-leq (rewrite)
  (implies (and (equal (whist w1 t1) (whist w2 t1))

            (leq t0 t1))

    (equal (equal (whist w1 t0) (whist w2 t0))

            t))

    ((use (whist-whist (w w1)) (whist-whist (w w2))))

(prove-lemma equal-phist-leq (rewrite)
  (implies (and (equal (phist b1 t1) (phist b2 t1))

            (leq t0 t1))

    (equal (equal (phist b1 t0) (phist b2 t0))

            t))

    ((induct (list-2-induct b1 b2))))

(prove-lemma equal-bhist$-leq ()
  (implies (and (equal (bhist$ flag b1 m t1) (bhist$ flag b2 m t1))

            (leq t0 t1))

    (equal (bhist$ flag b1 m t0) (bhist$ flag b2 m t0))))

(prove-lemma equal-bhist-leq (rewrite)
  (implies (and (equal (bhist b1 m t1) (bhist b2 m t1))

            (leq t0 t1))

    (equal (bhist$ flag b1 m t0) (bhist$ flag b2 m t0)));
(equal (equal (bhist b1 m t0) (bhist b2 m t0))
  t))
((use (equal-bhist$-leq (flag t)))))

;; RUN is "nonretroactive", i.e., does not alter the history of the
;; bundle B w.r.t. the initial time T0:

(prove-lemma run-nonretroactive (rewrite)
  (implies (and (modulep m)
    (bundlep b m)
    (inpacketp p m))
    (equal (bhist (run m p b t0 tf) m t0)
      (bhist b m t0))
    ((disable step bundlep modulep inpacketp bhiet)))

(prove-lemma tnextv-tnextw (rewrite)
  (implies (and (lessp tp (tnextv w t0))
    (wavep w)
    (leq t0 tp))
    (equal (tnextw w tp) (tnextw w t0)))

(prove-lemma leq-tnextv-cdar (rewrite)
  (implies (and (wavep w)
    (lessp t0 (cdar w))
    (not (lessp (cdar w) (tnextw w t0))))

(prove-lemma tnextv-tnextw-2 (rewrite)
  (implies (and (tnextv w tp)
    (wavep w)
    (leq t0 tp))
    (not (lessp (tnextv w tp) (tnextw w t0) )))

(prove-lemma tnextp-true (rewrite)
  (implies (and (not (lessp tp t0))
    (tnextp p tp))
    (tnextp p t0))

(prove-lemma tnextv-ttrue (rewrite)
  (implies (and (not (lessp tp t0))
    (tnextv w tp))
    (tnextv w t0))

(prove-lemma tnextp-tnextp (rewrite)
  (implies (and (packetp p m)
    (lessp tp (tnextp p t0))
    (leq t0 tp))
    (equal (tnextp p tp) (tnextp p t0))
    ((disable tnextw wavep)))

(prove-lemma tnextb$-true (rewrite)
  (implies (and (not (lessp tp t0))
    (tnextb$ flag b m tp))
    (tnextb$ flag b m t0)))

(prove-lemma tnextb$-tnextb$ (rewrite)
  (implies (and (modulep$ flag m)
    (bundlep$ flag b m)
    (lessp tp (tnextb$ flag b m t0))
    (leq t0 tp))
    (equal (bhist (run m p b t0 tf) m t0)
      (bhist b m t0))
    ((disable step bundlep modulep inpacketp bhiet)))

80
(equal (tnextb$ flag b m tp) (tnextb$ flag b m tO)))

(prove-lemma lessp-emin ()
  (implies (and x y (lessp m (emin x y))
    (and (lessp m x) (lessp m y)))))

(prove-lemma tnext-tnext (rewrite)
  (implies (and (modulep m)
    (bundlep b m)
    (inpacketp p m)
    (lessp tp (tnext tO p b m))
    (leq tO tp))
    (equal (tnext tp p b m) (tnext tO p b m))
    (use (lessp-emin (x (tnextb$ t b m tO)) (y (tnextp p tO)) (m tp))))

(prove-lemma tnext-true (rewrite)
  (implies (and (not (lessp tp tO))
    (tnext tp p b m))
    (tnext tO p b m))

;; This lemma provides for the decomposition of a simulation interval
;; into two subintervals:

(prove-lemma run-run ()
  (implies (and (modulep m)
    (bundlep b m)
    (inpacketp p m)
    (leq tO tp) (leq tp tf))
    (equal (run m p b tO tf)
      (run m p (run m p b tO tp) tp tf))
    (disable step bundlep modulep)
    (induct (run m p b tO tf))
    (expand (run m p b tp tf) (run m p b tO tp))))

;; Under normal conditions, RUN always produces a valid bundle:

(prove-lemma bundlep-run (rewrite)
  (implies (and (modulep m)
    (bundlep b m))
    (bundlep (run m p b tO tf) m))
    (disable modulep bundlep step inpacketp tnext)))

C  Synchronous Sequential Circuits

;;****************************************************************************
;;                        COMBINATIONAL MODULES
;;****************************************************************************

;; We begin with the relatively simple class of "combinational" modules.
;; The definition of this class depends on a function SLEVEL$$, which
;; computes the maximum length from any input signal to a given signal
;; of an arbitrary module. The definition of SLEVEL$$ is difficult to
;; establish for two reasons: (1) we allow arbitrarily deep hierarchical
;; module definitions, and (2) the desired maximum path length may not exist,
;; i.e., the signal may lie on a structural loop, which must be effectively detected.

(defvar unionl (l)
  (if (listp l)
      (union (car l) (unionl (cdr l)))
      ()))

(defvar signals (mod)
  (unionl (cons (1 mod) (lo mod))))

(defvar delete (x l)
  (if (listp l)
      (if (equal x (car l))
          (cdr l)
          (cons (car l) (delete x (cdr l))))
      ()))

(defvar subbagp (l m)
  (if (listp l)
      (and (member (car l) m)
           (subbagp (cdr l) (delete (car l) m))
           t)
      ()))

(defvar subsetp (l m)
  (if (listp l)
      (and (member (car l) m)
           (subsetp (cdr l) m))
      ()))

(prove-lemma length-delete (rewrite)
  (implies (member x l)
           (equal (length (delete x l))
                  (subl (length l))))))

(prove-lemma member-delete (rewrite)
  (implies (and (member x l)
                (not (equal x y)))
           (member x (delete y l))))

(prove-lemma lessp-length-subbagp ()
  (implies (and (subbagp l m)
                (member x m)
                (not (member x l)))
           (lessp (length l) (length m))))

(prove-lemma subsetp-delete (rewrite)
  (implies (and (subsetp l m)
                (not (member x l)))
           (subsetp l (delete x m))))

(prove-lemma subsetp-subbagp (rewrite)
  (implies (and (distinct-symbols l)
                (subsetp l m))
           (subbagp l m))
  (((induct (subbagp l m))))

(prove-lemma lessp-length-subset (rewrite)
  (implies (and (subsetp l m)
                (distinct-symbols l)
                (subsetp l m))
           (subbagp l m))))
(distinct-symbols 1)
(member x m)
(not (member x l)))
(lesssp (length l) (length m)))
((use (lesssp-length-subbagp)))

(defun index (s lo)
  (if (listp lo)
      (if (member s (car lo))
          0
          (addl (index s (cdr lo)))))
  t))

(defun slevel$$ (flag out m bad q)
  ;;(SLEVEL$$ T OUT M.() Q) is the length of the longest path to OUT that does not
  ;;pass through any of the first Q submodules of M
  (if (= flag 'list)
      (if (listp out)
          (amax (slevel$$ t (car out) m bad q)
            (slevel$$ 'list (cdr out) m bad q))
          0)
      (if (or (member out (i m))
              (lesssp (index out (lo m)) q))
          0
          (if (and (not (member out bad))
                   (distinct-symbols bad)
                   (member out (signals m))
                   (subsetp bad (signals m)))
            (addl (slevel$$ 'list (find-li out m) m (cons out bad) q))
            t))
      (ord-lessp (lex (list (difference (length (signals m)) (length bad))
                         (count out)))))))

;;SDEPTH returns the maximum SLEVEL$$ of all signals of M:
(defun sdepth (m q)
  (slevel$$ 'list (signals m) m () q))

;;The final argument of SLEVEL$$ will be relevant to our analysis of
;;sequential modules. For the present purpose, we take it to be 0.
;;We may now define "combinational module":
(defun combp$ (flag m)
  (if (equal flag 'list)
      (if (listp m)
          (and (combp$ t (car m))
               (combp$ 'list (cdr m)))
          t)
      (if (modulep m)
          (case (type m)
                  (struct (and (sdepth m 0) (combp$ 'list (s m))))
                  (behav t)
                  (otherwise f))
          f))
  (defn combp (m) (combp$ t m))

;;Now that SLEVEL$$ has been defined, we may use it to define a simpler
;;version, SLEVEL$, which will be easier to use. The purpose of this
;: function is to provide a recursion scheme for various functions
;: pertaining to combinational and sequential modules.
;: The definition will take some work:

(prove-lemma member-slevel$$(rewrite)
 (implies (and (member s l)
 (slevel$$ 'list l m bad q))
 (slevel$$ t s m bad q)))

(prove-lemma subsetp-slevel$$(rewrite)
 (implies (and (subsetp s l)
 (slevel$$ 'list l m bad q))
 (slevel$$ 'list s m bad q)))

(prove-lemma signals-slevel$$(rewrite)
 (implies (and (subsetp s (signals m))
 (slevel$$ 'list s m () q))
 (use (subsetp-slevel$$(l (signals m)) (bad ()))))

(prove-lemma leq-slevel$$-cdr (rewrite)
 (implies (and (sdepth m q) (listp st (subsetp s (signals m)))
 (equal (leqsp (slevel$$ 'list s m () q)
 (slevel$$ 'list (cdr s) m () q))
 f))
 (use (signals-slevel$$())
 (expand (slevel$$ 'list s m () q))))

(prove-lemma leq-slevel$$-car (rewrite)
 (implies (and (sdepth m q) (listp e) (subsetp s (signals m)))
 (equal (leqsp (slevel$$ 'list e m () q)
 (elevel$$ t (car s) m (7 q))
 f))
 (use (signals-slevel$$())
 (expand (slevel$$ 'list s m () q))))

(defun ss-induct (flag s m bad1 bad2 q)
 (if (equal flag 'list)
 (if (listp s)
 (and (ss-induct t (car s) m bad1 bad2 q)
 (ss-induct 'list (cdr s) m bad1 bad2 q))
 t)
 (if (or (member s (i m))
 (lessp (index s (lo m)) q))
 t
 (if (and (not (member s bad2))
 (distinct-symbols bad2)
 (member s (signals m))
 (subsetp bad2 (signals m)))
 (as-induct 'list (find-li s m) m (cons s bad1) (cons s bad2) q)
 t))
 (ord-lessp (lex (list (difference (length (signals m)) (length bad2))
 (count s)))))

(defun sublistp (l m)
 (if (listp l)
 (if (listp m)
 (if (equal (car l) (car m))
 (sublistp (cdr l) (cdr m))
(sublistp l (cdr m))

(prove-lemma distinct-symbols-sublistp (rewrite)
  (implies (and (distinct-symbols m)
    (sublistp l m))
    (distinct-symbols l)))

(prove-lemma sublistp-subsetp (rewrite)
  (implies (and (sublistp l m)
    (subsetp m p))
    (subsetp l p)))

(prove-lemma sublistp-member (rewrite)
  (implies (and (sublistp l m)
    (member x l))
    (member x m)))

(disable sublistp-member)

(prove-lemma slevel$$-sublistp ()
  (implies (and (slevel$$ flag s m bad2 q)
    (sublistp bad1 bad2))
    (equal (slevel$$ flag s m bad1 q)
      (slevel$$ flag s m bad2 q)))
  ((induct (as-induct flag s m badl bad2 q))
    (enable sublistp-member)))

(prove-lemma slevel$$-nil (rewrite)
  (implies (slevel$$ flag s m (list b) q)
    (equal (slevel$$ flag s m (list b) q)
      (slevel$$ flag s m () q)))
  ((use (slevel$$-sublistp (badl ()) (bad2 (list b))))))

(prove-lemma slevel$$-list-find-li (rewrite)
  (implies (and (sdepth m q)
    (member s (signals m))
    (not (member s (i m))))
    (slevel$$ 'list (lookupl s (lo m) (li m)) m (list s) q))
  ((use (member-slevel$$ (I (signals m)) (bad ()())))
    (disable member-slevel$$)))

(prove-lemma slevel$$-list-find-li-nil (rewrite)
  (implies (and (sdepth m q)
    (member s (signals m))
    (not (member s (i m))))
    (not (lessp (index s (lo m)) q)))
    (slevel$$ 'list (lookupl s (lo m) (li m)) m () q))
  ((use (slevel$$-list-find-li-nil)))

(prove-lemma lessp-slevel$$-find-li (rewrite)
  (implies (and (sdepth m q)
    (not (equal flag 'llst))
    (member s (signals m))
    (not (member s (i m)))
    (not (lessp (index s (lo m)) q)))
    (equal (lessp (slevel$$ 'list (lookupl s (lo m) (li m)) m () q))
    85)

85
(slevel$$ flag s m () q))
t())
((expand (slevel$$ flag s m () q))))

(defn slevel$ (flag s m q)
  (if (sdepth m q)
    (if (equal flag 'list)
      (if (subsetp s (signals m))
        (if (listp s)
          (max (slevel$ t (car s) m q)
            (slevel$ 'list (cdr s) m q))
        0)
      0)
    (if (member s (signals m))
      (if (or (member s (i m))
        (lessp (index s (lo m)) q))
        (add1 (slevel$ 'list (find-li s m) m q)))
      f))
  f)
)((ord-lessp (lex (list (slevel$$ flag s m () q) (count s))))))

(prove-lemma leq-slevel$-cdr (rewrite)
  (implies (and (sdepth m q) (listp s) (subsetp s (signals m)))
    (equal (lessp (slevel$ 'list s m q)
                  (slevel$ 'list (cdr s) m q))
      f)))

(prove-lemma leq-slevel$-car (rewrite)
  (implies (and (sdepth m q) (listp s) (subsetp s (signals m)))
    (equal (lessp (slevel$ 'list s m q)
                  (slevel$ t (car s) m q))
      f)))))

(prove-lemma lessp-slevel$-find-li (rewrite)
  (implies (and (sdepth m q)
                (not (equal flag 'list))
                (member s (signals m))
                (not (member s (i m)))
                (not (lessp (index s (lo m)) q)))
    (equal (lessp (slevel$ 'list (lookupl s (lo m) (li m)) m q)
                 (slevel$ flag s m q))
      t)))))

(prove-lemma comp$-sdepth (rewrite)
  (implies (and (structp m) (combp m))
    (sdepth m 0)))

(prove-lemma lessp-count-lookup (rewrite)
  (implies (lessp (count s) (count m))
    (equal (lessp (count (lookupl x y s)) (count m))
      t)))))

;;CVECP determines whether V is a valid input vector for M:
(defn cvvecp (v m)
  (bvpn v (ni m)))

;;Each signal of a combinational module is naturally associated
:with a certain Boolean function of the inputs. This function is computed as follows:

\[
\begin{align*}
\text{(defn } & \text{cv} \text{ (flag s v m)} \\
\text{  ; (equal flag 'list)} \\
\text{      (if (and (combp m) (structp m) (subsetp s (signals m)))} \\
\text{      (if (listp s) \\
\text{         (cons (cv t (car s) v m) \\
\text{          (cv 'list (cdr s) v m)))} \\
\text{         (}}} \\
\text{      (if (behaviorp m) \\
\text{         (lookup s (o m) (pairlist (i m) v))} \\
\text{         (if (and (combp m) (member s (signals m))) \\
\text{            (if (member s (i m)) \\
\text{               (lookup s (i m) v) \\
\text{               (cv t (find-o s m) \\
\text{                (cv 'list (find-li s m) v m) \\
\text{                (find-s s m)))} \\
\text{               f)} \\
\text{            (f)))) \\
\text{            ((ord-lessp (lex (list (count m) (slevel flag s m 0) (count s))))))} \\
\text{      (defn } \text{cv} \text{ (s v m)} \\
\text{      (cv t s v m))} \\
\text{      (defn } \text{dcmin} \text{ (flag s m)} \\
\text{      (if (equal flag 'list) \\
\text{        (if (and (combp m) (structp m) (subsetp s (signals m))) \\
\text{          (if (listp s) \\
\text{             (emin (dcmin t (car s) m) \\
\text{              (dcmin 'list (cdr s) m)))} \\
\text{             (f)) \\
\text{          (if (behaviorp m) \\
\text{             (lookup s (o m) (d m))} \\
\text{             (if (and (combp m) (member s (signals m))) \\
\text{                (if (and (structp m) (member s (signals m))) \\
\text{                    (if (member s (i m)) \\
\text{                       0 \\
\text{                       (eplus (dcmin t (find-o s m) (find-s s m) \\
\text{                           (dcmin 'list (find-li s m) m)))} \\
\text{                       f)) \\
\text{                    (f)))) \\
\text{                    ((ord-lessp (lex (list (count m) (slevel flag s m 0) (count s))))))} \\
\text{      (defn } \text{dcmin} \text{ (s m) (dcmin t s m))} \\
\text{      (defn } \text{dcmax} \text{ (flag s m)} \\
\text{      (if (equal flag 'list) \\
\text{        (if (and (combp m) (structp m) (subsetp s (signals m))) \\
\text{          (if (listp s) \\
\text{             (emax (dcmax t (car s) m) \\
\text{             (dcmax 'list (cdr s) m))))))))}
\end{align*}
\]
(dcmax$ 'list (cdr s) m))
0)
(f)
(if (behavp m)
 (lookup s (o m) (d m))
 (if (and (combp m) (member s (signals m)))
 (if (and (structp m) (member s (signals m)))
 (if (member s (i m))
 0
 (eplus (dcmax$ t (find-o s m) (find-s s m))
 (dcmax$ 'list (find-l s m) m)))
(f)
((ord-lessp (lex (list (count a) (slevel fla s a 0) (count s))))))

(defun dcmax (s m) (dcm_ t sm))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%% SEQUENTIAL MODULES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

;We shall define a class of synchronous sequential circuits, using the ;flip-flop DFF as the primitive state-holding device. The recursive ;definition will require that for some Q > 0, the first Q submodules ;of a sequential module M (other than DFF) are sequential and the rest ;are all combinational. For any module M, we define the parameter ;(Q M) as follows:

(defun q$ (mods)
 (if (listp mods)
  (if (combp (car mods))
    0
    (addl (q$ (cdr mods))))
  0))

(defun q (m)
 (q$ (s m)))

(prove-lemma leq-q$ ()
 (leq (q$ s) (length s)))

(prove-lemma lessp-count-firstn ()
 (implies (and (plistp l) (leq q (length l)))
  (leq (count (firstn q l)) (count l)))
 ((induct (firstn q l))))

(prove-lemma lessp-count-first-q (rewrite)
 (implies (and (modulep m) (structp m))
  (equal (lessp (count (firstn q$ (s m)) (s m))
  (count m))
  t))
 ((use (lessp-count-firstn (q (q m)) (l (s m))))
 (lessp-count-submodules)
 (leq-q$ (s (s m)))
 (disable lessp-count-submodules)))

;A path is "combinational" if it passes through only combinational ;components. A signal is "native" if it is not connected to any
global input by a combination path:

```lisp
(defun nativep$ (flag s m)
  (if (sdepth m (q m))
      (if (equal flag 'list)
          (if (subsetp s (signals m))
              (if (listp s)
                  (and (nativep$ t (car s) m)
                      (nativep$ 'list (cdr s) m))
                    t)
              f)
      (if (equal m (dff))
          (member s (o m))
          (if (member s (i m))
              f
          (if (lesseq (index s (lo m)) (q m))
              t
          (nativep$ (find-li s m))))
        f))
    f)

(defun nativep (s m) (nativep$ t s m))

(defun check-seq-li (clk rst li)
  (if (listp li)
      (and (equal elk (cast li))
          (equal rst (cedar li))
          (not (member elk (cddar li)))
          (not (member rst (cddar li)))
          (check-seq-li clk rst (cdr li)))
    t))

(defun check-comb-li (clk rst li)
  (if (listp li)
      (and (not (member clk (car li)))
          (not (member rst (car li)))
          (check-comb-li clk rst (cdr li)))
    t))

A sequential module other than DFF has Q sequential submodules, Q > 0,
with the rest combinational. It has at least two inputs. The first
and second inputs are by convention the clock and the reset. The clock
(resp., reset) is connected the the clock (resp., reset) input of each
sequential submodule, and not to any other submodule input. No combinational
loops are permitted. Finally, all outputs are required to be native signals:

(defun seqp$ (flag m)
  (if (equal flag 'list)
      (if (listp m)
          (and (seqp$ t (car m))
              (seqp$ 'list (cdr m)))
        t)
  (if (and (modulep m) (structp m))
      (or (equal m (dff))
          (and (geq (ni m) 2)
              (not (zerop (q m)))
              (seqp$ 'list (firstn (q m) (s m))))))
```

89
(check-seq-li (car (i m)) (cadr (i m)) (firstn (q m) (li m)))
(check-comb-li (car (i m)) (cadr (i m)) (cdrn (q m) (li m)))
(sdepth m (q m))
(nativep$ 'list (o m) m))
((lessp (count m)))))

(defun seqp (m) (seqp$ t m))

(prove-lemma lessp-count-car-s (rewrite)
  (implies (structp m)
    (equal (lessp (count (car (s m))) (count m))
       t))
  ((use (lessp-count-submodules))
    (disable lessp-count-submodules)))

(prove-lemma modulep-seqp (rewrite)
  (implies (seqp m)
    (modulep$ t m)))

(prove-lemma seqp-sdepth (rewrite)
  (implies (and (seqp m) (not (equal m (dff))))
    (sdepth m (q$ (s m))))
  ((disable sdepth dff q$)))

(prove-lemma seqp-structp (rewrite)
  (implies (seqp m) (equal (type m) 'struct)))

;; A native signal S of M is "registered" if either (a) M = DFF and S is an
;; output of M, or (b) M <> DFF and S is associated with a registered output
;; of a sequential submodule of M:

(defun regp (s m)
  (if (seqp m)
    (if (equal m (dff))
      (member s (o m))
      (and (lessp (index e (lo m)) (q m))
        (regp (find-o s m) (find-s s m))))
    f))

;; A "state" of a sequential module is a structure that associates a
;; Boolean value with each flip-flop:

(defun statep$ (flag state m)
  (if (equal flag 'list)
    (if (listp m)
      (and (statep$ t (car state) (car m))
        (statep$ 'list (cdr state) (cdr m)))
      (equal state ()
        (if (and (modulep m) (structp m))
          (if (equal m (dff))
            (boolp state)
            (if (equal (q m) 1)
              (statep$ t state (car (s m)))
              (statep$ 'list state (firstn (q m) (s m)))))))
    (statep$ t state m))
  (statep$ t state m))
(defn find-state (s state m)
  (if (equal (q m) i)
    state
    (lookup\s (lo m) state)))

(disable sdepth)

;; A state determines a "resultant value" for each native signal:

(defn rv$ (flag s state m)
  (if (seqp m)
    (if (equal flag 'list)
      (if (and (subsetp s (signals m)) (not (equal m (diff))))
        (if (listp s)
          (cons (rv$ t (car s) state m)
            (rv$ 'list (cdr s) state m))
          ()
        )
      )
    )
  )
)

(if (member s (signals m))
  (if (member s (i m))
    f
    (if (equal m (diff))
      (if (equal s 'q) state (not state))
      (if (lessp (index s (lo m)) (q m))
        (rv$ t (find-o s m) (find-state s state m) (find-s s m))
        (cv (find-o s m) (rv$ 'list (find-li s m) state m) (find-s s m)))
      )
    )
  )
)

((ord-lessp (lex (list (count m) (slevel$ flag s m (q m)) (count s)))
  f))

(defn rv (s state m) (rv$ t s state m))

;; A "data vector" associates a Boolean value with each data input:

(defn svecp (x m)
  (bvpn x (difference (hi m) 2)))

;; A state and a data vector determine a "sequential value" for each signal
;; (other than the clock and reset inputs):

(defn sv$ (flag s v state m)
  (if (seqp m)
    (if (equal flag 'list)
      (if (and (subsetp s (signals m)) (not (equal m (diff))))
        (if (listp s)
          (cons (sv$ t (car s) v state m)
            (sv$ 'list (cdr s) v state m))
          ()
        )
      )
    )
  )
)

(if (member s (signals m))
  (if (member s (i m))
    (lookup s (cddr (i m)) v)
    (if (or (equal m (diff))
      (lessp (index s (lo m)) (q m)))
      (rv s state m)
    )
  )
)

(cv (find-o s m) (sv$ 'list (find-li s m) v state m) (find-s s m)))

f)

f)
((ord-lessp (lex (list (slevel$ flag s m (q m)) (count s)))))

(defn sv (s v state m)
  (sv$ t s v state m))

(defn svl (li v state m)
  (sv$ 'list (cddr li) v state m))

(defn svll (s v state m)
  (if (listp s)
      (cons (svl (car s) v state m)
            (svll (cddr s) v state m))
      ()))

;; NEXT computes a new state from a state and a data vector:

(defn next$ (flag v state m)
  (if (equal flag 'list)
      (if (listp m)
          (cons (next$ t (car v) (car state) (car m))
                (next$ 'list (cddr v) (cddr state) (cddr m)))
          (if (seqp m)
              (if (equal m (diff))
                  (car v)
                  (if (equal (q m) 1)
                      (next$ t (svl (car (li m)) v state m) state (car (s m)))
                      (next$ 'list (svl (firstn (q m) (li m)) v state m)
                                   state
                                   (firstn (q m) (s m))))))
      ()))

(defn next (v state m)
  (next$ t v state m))

;; Each native signal is associated with a minimum and a
;; maximum delay, which determine an interval during which the
;; signal's value may change following a rising edge:

(defn dsmin$ (flag s m)
  (if (seqp m)
      (if (equal flag 'list)
          (if (and (subsetp s (signals m)) (not (equal m (diff))))
              (if (listp s)
                  (dmin (dsmin$ t (car s) m)
                        (dsmin$ 'list (cddr s) m))
                f)
              (if (member s (signals m))
                  (if (member s (i m))
                      0
                      (if (equal m (diff))
                          4000
                          (if (lessp (index s (lo m)) (q m))
                              (dmin$ t (find-o s m) (find-s s m))
                              (plus (dmin (find-o s m) (find-s s m))
                                    (dmin$ 'list (find-li s m) m))))))
          f))
  )

92
f)

((ord-lessp (lex (list (count m) (slevel$ flag s m (q m)) (count s))))))

(defun dsmin (s m) (dsmin$ t s m))

(defun dsmax$ (flag s m)
  (if (seqp m)
    (if (equal flag 'list)
      (if (and (subseqp s (signals m)) (not (equal m (dff))))
        (if (listp s)
          (dsmax$ (car s) m)
           (dsmax$ 'list (cdr s) m)
          0)
        (if (member s (signals m))
          (if (member s (i m))
            0
            (if (equal m (dff))
              6000
              (if (lessp (index s (lo m)) (q m))
                (dsmax$ (find-o s m) (find-s s m))
                (dsmax$ 'list (find-li s m)))))))
      f))
  ((ord-lessp (lex (list (count m) (slevel$ flag s m (q m)) (count s))))))

(defun dsmax (s m) (dsmax$ t s m))

;;The definition of "setup" times requires some work:

(defun setup-comb (sigs setups m)
  (if (listp sigs)
    (if (zerop (car setups))
      (setup-comb (cdr sigs) (cdr setups) m)
      (emax (eplus (dcmax (car sigs) m) (car setups))
            (setup-comb (cdr sigs) (cdr setups) m))))

(defun collect-i (s li i)
  (if (listp li)
    (if (equal s (car li))
      (cons (car li) (collect-i s (cdr li) (cdr i)))
      (collect-i s (cdr li) (cdr i)))
    ()))

(defun collect-li (s li m)
  (if (listp li)
    (if (member s (car li))
      (cons (collect-i s (car li) (i (car m))
            (collect-li s (cdr li) (cdr m))
            (collect-li s (cdr li) (cdr m)))
    ()))

(defun collect-lo (s li lo)
  (if (listp li)
    (if (member s (car li))
      (cons (car lo) (collect-lo s (cdr li) (cdr lo)))
      (collect-lo s (cdr li) (cdr lo))))

93
(defn slevel (m) (slevel$ 'list (signals m) m (q m)))

(defn sm_x (m) (slevel$ 'lLat (signals m) m (q m)))

(prove-lemma leq-slevel-member ()
  (implies (and (subsetp 1 (signals m))
               (member s 1))
             (leq (slevel$ t s m q)
                  (slevel$ 'list 1 m q))))

(prove-lemma subsetp-cdr (rewrite)
  (implies (subsetp 1 (cdr m))
            (subsetp 1 m)))

(prove-lemma subsetp-l-1 (rewrite)
  (subsetp l 1))

(prove-lemma leq-slevel-smax ()
  (implies (member s (signals m))
            (leq (slevel s m) (smax m))
            (use (leq-slevel-member (1 (signals m)) (q (q m)))
                 (disable signals q slevel$)))

(defn m0 (s m) (add1 (difference (smax m) (slevel s m))))

(defn ml (s m) (if (listp s)
                    (max (m0 (car s) m)
                         (ml (cdr s) m))
                    0))

(defn m4 (s m) (if (listp s)
                    (max (ml (car s) m)
                         (m4 (cdr s) m))
                    0))

(defn setup-mess (flag s m)
  (case flag
    0 (m0 s m)
    1 (ml s m)
    3 (ml s m)
    4 (m4 s m)
    (otherwise f)))

(defn attachedp (x y i li lo)
  (if (zerop i)
      (and (member x (car li))
           (member y (car lo))
           (attachedp x y (sub1 i) (cdr li) (cdr lo)))
      (prove-lemma member-union (rewrite)
       (implies (member x m) (member x (union 1 m)))))

(prove-lemma member-union (rewrite)
  (implies (member x m) (member x (union 1 m))))

94
(prove-lemma attached-unionl ()
  (implies (attachedp x y i li lo)
    (member y (unionl lo))))

(prove-lemma attachedp-member-signals (rewrite)
  (implies (attachedp x y i (li m) (lo m))
    (member y (signals m)))
  ((use (attached-unionl (li (li m)) (lo (lo m))))))

(prove-lemma member-unienl-appears (rewrite)
  (implies (not (appears x lo))
    (not (member x (unionl lo))))))

(prove-lemma none-appear-member-unionl ()
  (implies (and (none-appear in lo)
    (member x (unionl lo))
    (not (member x in))))))

(prove-lemma attachedp-not-member-i (rewrite)
  (implies (and (attachedp x y i (li m) (lo m))
    (check-struct m))
    (not (member y (i m))))
  ((use (attached-unionl (li (li m)) (lo (lo m))))
    (none-appear-member-unionl (in (i m)) (lo (lo m)) (x y))))

(prove-lemma none-appear-not-attached (rewrite)
  (implies (and (member y car)
    (none-appear car cdr))
    (not (attachedp x y i li cdr)))
  ((use (attached-unionl (lo cdr))
    (none-appear-member-unionl (in car) (lo cdr) (x y))))

(prove-lemma attachedp-index ()
  (implies (and (attachedp x y i li lo)
    (all-distinct-symbols lo))
    (equal (index y lo) (fix i))))

(prove-lemma attachedp-index-rewrite (rewrite)
  (implies (and (attachedp x y i (li m) (lo m))
    (check-struct m))
    (equal (index y (lo m)) (fix i)))
  ((use (attachedp-index (li (li m)) (lo (lo m))))))

(prove-lemma attachedp-member-lookupl ()
  (implies (and (attachedp x y i li lo)
    (all-distinct-symbols lo))
    (member x (lookupl y lo li))))

(prove-lemma attachedp-member-find-li (rewrite)
  (implies (and (attachedp x y i (li m) (lo m))
    (check-struct m))
    (member x (find-li y m)))
  ((use (attachedp-member-lookupl (li (li m)) (lo (lo m))))))

(prove-lemma appears-member-unionl (rewrite)
  (implies (appears x 1)
    (member x (unionl li))))

95
(prove-lemma all-appear-subsetp-unionl (rewrite)
  (implies (all-appear li l)
            (subset li (unionl l))))

(prove-lemma subsetp-lookupl ()
  (implies (lists-all-appear li l)
            (subsetp (lookupl y lo li) (unionl l))))

(prove-lemma subsetp-find-li (rewrite)
  (implies (check-struct m)
            (subsetp (find-li y m) (signals m)))
  ((use (subsetp-lookupl (li (li m)) (lo (lo m)) (l (cons (i m) (lo m))))))))

(prove-lemma attached-lessp-slevel$ ()
  (implies (and (adepth m q)
                 (modulep m)
                 (structp m)
                 (attachedp x y i (li m) (lo m))
                 (leq q i))
            (lessp (slevel$ t x m q)
                   (slevel$ t y m q)))
  ((disable adepth find-li signals index slevel$ check-struct attachedp)
   (use (slevel$ (flag t) (s x)))
   (slevel$ (flag t) (s y)))
  (expand (modulep$ t m))))

(prove-lemma lessp-mO ()
  (implies (and (seqp m)
                 (not (equal m (dff)))
                 (attachedp x y i (li m) (lo m))
                 (leq (q m) i))
            (lessp (mO y m) (mO x m)))
  ((use (attached-lessp-slevel$ (q (q m)))
         (leq-slevel-smax (s y)))
   (disable modulep attachedp slevel$ adepth smax q signals dff *))
  (normalize attachedp-alt)
  (expand (modulep$ t m))))

(prove-lemma not-zerop-mO ()
  (not (zerop (mO x m))))

(disable mO)

(prove-lemma attachedp-alt ()
  (implies (and (member x (car (cdrn ili)))
                 (member y (car (cdrn ilo))))
            (attachedp x y i li lo))
  (normalize attachedp-alt)
  (expand (modulep$ t m)))

(prove-lemma lessp-mO-revrite (rewrite)
  (implies (and (seqp m)
                 (not (equal m (dff)))
                 (leq (q m) i))
            (member x (car (cdrn i li)))
            (member y (car (cdrn i lo))))
  (normalize attachedp-alt)
  (expand (modulep$ t m))))

(prove-lemma lessp-mi ()
  (implies (and (seqp m)
                 (not (equal m (dff)))
                 (leq (q m) i))
            (member x (car (cdrn i li)))
            (member y (car (cdrn i lo))))
  (normalize attachedp-alt)
  (expand (modulep$ t m)))
(not (equal m (dff)))
(leq (q m) i)
(member x (car (cdrn i (li m))))
(subsetp ys (car (cdrn i (lo m))))
(equal (lessp (m1 ys m) (m0 x m)) t))
((disable attachedp dff *1-diff seqp member q)
 (INDUCT (LENGTH YS))
 (use (not-zero-mp-m0))))
(prove-lemma lessp-m1-m0 (rewrite)
 (implies (and (seqp m)
 (leq (q m) i)
 (member x (car (cdrn i (li m))))
 (equal (lessp (m1 (car (cdrn i (lo m))) m)
 (m0 x m)) t))
 ((disable attachedp dff *1-diff seqp member q)
 (use (lessp-m1 (ys (car (cdrn i (lo m))))))))))
(prove-lemma cdr-cdrn (rewrite)
 (equal (cdr (cdrn r (lo m))) (cdrn (addl r) (li m))))
(defn lm4-induct (r m)
 (if (lessp r (length (li m)))
 (lm4-induct (addl r) m)
 t)
 ((lessp (difference (length (li m)) r))))
(prove-lemma nlistp-cdrn (rewrite)
 (implies (leq (length l) n)
 (not (listp (cdrn n i))))))
(prove-lemma lessp-m4 ()
 (implies (and (seqp m)
 (not (equal m (dff)))
 (leq (q m) r)
 (leq r (length (li m))))
 (equal (lessp (m4 (collect-lo s (cdrn r (li m))) (cdrn r (lo m))) m)
 (m0 s m)) t))
 ((disable dff *1-diff seqp q cdrn m1)
 (induct (lm4-induct r m))
 (use (not-zero-mp-m0 (z s))))))
(prove-lemma equal-length-li-s ()
 (implies (seqp m)
 (equal (length (li m)) (length (s m))))
 ((expand (seqp$ t m) (modulep$ t m))))
(prove-lemma lessp-m4-rewrite (rewrite)
 (implies (and (seqp m)
 (not (equal m (dff)))
 (equal (lessp (m4 (collect-lo s (cdrn (q$ (s m)) (li m))) (cdrn (q$ (s m)) (lo m))) m)
 (m0 s m)) t))
 ((disable dff *1-diff seqp q$ cdrn m1 m4 collect-lo)
 (use (lessp-m4 (r (q m))))
 (leq-q$ (s (s m))))

97
(prove-lemma leq-count-collect-lo ()
  (implies (and (equal (length li) (length s))
               (plistp s))
    (leq (count (collect-lo x li s)) (count s)))
  ((induct (collect-lo x li s))))

(prove-lemma leq-count-cdrn ()
  (implies (plistp s)
    (leq (count (cdrn q s)) (count s))))

(prove-lemma equal-length-cdrn (rewrite)
  (implies (equal (length x) (length y))
    (equal (equal (length (cdrn q x)) (length (cdrn q y)))
      t)))

(prove-lemma plistp-cdrn-q (rewrite)
  (implies (and (leq q (length s))
                (plistp s))
    (plistp (cdrn (q$ s))))
  ((use (plistp-cdrn-q (q (q$ s)))
        (leq-q$)))

(prove-lemma lessp-count-collect-lo (rewrite)
  (implies (and (seqp m)
                (not (equal m (dff))))
    (equal (lessp (count (collect-lo x
                          (cdrn (q$ (s m)) (li m)))
                 (cdrn (q$ (s m)) (s m))))
      (count m))
  ((use (leq-count-collect-lo (li (cdrn (q m) (li m)))
                               (s (cdrn (q m) (s m))))
      (leq-count-cdrn (s (s m)) (q (q m)))
      (equal-length-li-s)
      (lessp-count-submodules)))
  (disable modulep$ diff *l*dff lessp-count-submodules))

(prove-lemma length-firstn (rewrite)
  (equal (length (firstn q x)) (fix q)))

(prove-lemma plistp-firstn (rewrite)
  (plistp (firstn q 1)))

(prove-lemma lessp-count-collect-lo-firstn (rewrite)
  (implies (and (seqp m)
                 (not (equal m (dff))))
    (equal (lessp (count (collect-lo x
                          (firstn (q$ (s m)) (li m)))
                 (firstn (q$ (s m)) (s m))))
      (count m))
  ((use (lessp-count-first-q)
        (equal-length-li-s)
        (leq-count-collect-lo (li (firstn (q m) (li m)))
                               (s (firstn (q m) (s m))))))
  (disable lessp-count-first-q diff *l*dff modulep$)))
(prove-lemma leq-m0 (rewrite)
  (implies (listp x)
    (equal (lessp (mi x m) (m0 (car x) m)) f)))

(prove-lemma leq-cdr-m1 (rewrite)
  (implies (listp x)
    (equal (lessp (mi x m) (mi (cdr x) m)) f)))

(prove-lemma leq-m1 (rewrite)
  (implies (listp x)
    (equal (lessp (mi x m) (mi (car x) m)) f)))

(prove-lemma leq-cdr-m4 (rewrite)
  (implies (listp x)
    (equal (lessp (m4 x m) (m4 (cdr x) m)) f)))

;; Each input other than the clock is associated with a "setup time",
;; which represents the duration over which the signal is required to
;; hold constant prior to a rising edge:

(disable seqp)
(disable dff)
(disable *i=dff)

(defun setups (flag x m)
  (case flag
    (0 (if (listp x)
         (if (equal m (dff))
             (case x
               (rst 8000)
               (d 6000)
               (d 6000)
               (otherwise f))
             (max (setup$ 2
                   (collect-li x (firstn (q m) (li m)) (firstn (q m) (s m)))
                   (collect-lo x (firstn (q m) (li m)) (firstn (q m) (s m))))
                   (setup$ 4
                   (collect-lo x (cdrn (q m) (li m)) (cdrn (q m) (lo m)))
                   (collect-lo x (cdnr (q m) (li m)) (cdnr (q m) (s m))))
                   f))
         (1 (if (listp x)
         (max (setup$ 0 (car x) m)
         (setup$ 1 (cdr x) m))
         0))
     (2 (if (listp m)
         (max (setup$ 1 (car x) (car m))
         (setup$ 2 (cdr x) (cdr m)))
         0))
     (3 (if (listp x)
         (cons (setup$ 0 (car x) m)
         (setup$ 3 (cdr x) m))
         ()))
     (4 (if (listp x)
         (cons (setup$ 3 (car x) m)
         (setup$ 4 (cdr x) m))
         ()))
     (5 (if (listp m)
         (max (setup-comb (o (car m)) (car x) (car m))))
         0))
;; Finally, we define three parameters pertaining to the behavior of the clock input, called the "clock high", the "clock low", and the "minimum period". These represent the minimum durations between a rising edge and the next falling edge, a falling edge and the next rising edge, and successive rising edges, respectively:

(defn high (m)
  (high m)
)

(defn low (m)
  (low m))

(defn setups-plus-delays (setups outs sub)
  (if (listp outs)
    (max (plus (dsmax (car outs) sub)
              (car setups))
          (setups-plus-delays (cdr setups) (cdr outs) sub))
    0))

(defn p3 (s lo m)
  (if (listp s)
    (max (setups-plus-delays setup s lo m) (o (car s)) (car s))
    (setup 5 (cdr x) (cdr m)))

(otherwise f))

((ord-lesp (lex (list (count m) (setup-meas flag x m) (count x))))))

(enable seqp)
(enable dfp)
(enable *ledff)

(defn setup (s m)
  (setup 0 s m))

;; Finally, we define three parameters pertaining to the behavior of the clock input, called the "clock high", the "clock low", and the "minimum period". These represent the minimum durations between a rising edge and the next falling edge, a falling edge and the next rising edge, and successive rising edges, respectively:

(defn high (flag m)
  (if (equal flag 'list)
    (if (listp m)
      (max (high t (car m))
           (high 'list (cdr m)))
      0)
    (if (seqp m)
      (if (equal m (dff))
        4000
        (high 'list (firstn (q m) (s m))))))

(defn low (flag m)
  (if (equal flag 'list)
    (if (listp m)
      (max (low t (car m))
           (low 'list (cdr m)))
      0)
    (if (seqp m)
      (if (equal m (dff))
        6000
        (low 'list (firstn (q m) (s m))))))

(defn setups-plus-delays (setups outs sub)
  (if (listp outs)
    (max (plus (dsmax (car outs) sub)
               (car setups))
         (setups-plus-delays (cdr setups) (cdr outs) sub))
    0))

(defn p3 (s lo m)
  (if (listp s)
    (max (setups-plus-delays setup 3 (car lo m) (o (car s)) (car s))
...
(p3 (cdr s) (cdr lo m))

(defn per$ (flag m)
  (if (equal flag 'list)
      (if (listp m)
        (max (per$ t (car m))
            (per$ 'list (cdr m))))
      0)
    (if (seqp m)
        (if (equal m (diff))
          10000
          (max (per$ 'list (firstn (q m) (s m)))
                (max (setup$ 3 (cdr (i m)) m)
                      (p3 (firstn (q m) (s m)) (firstn (q m) (lo m)) m)))
        f))))

(defn per (m) (per$ t m))

(disable seqp-structp)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;; COMPUTATIONS ON COMBINATIONAL MODULES
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;Whenever a combinational module is introduced, we derive all its of
;;relevant properties and then disable its definition. This procedure
;;is automated by means of several macros, which we define in this section.

;;First, for the sake of efficiency, we derive some rewrite rules that allow
;;us to disable various definitions:

(prove-lemma bvpn-rewrite-1 (rewrite)
  (implies (not (zerop n))
    (equal (bvpn x n)
      (and (boolp (car x))
            (bvpn (cdr x) (sub1 m))))))

(prove-lemma bvpn-rewrite-2 (rewrite)
  (implies (zerop n)
    (equal (bvpn x n)
      (equal x ()))))

(disable bvpn)

(prove-lemma compb-rewrite-1 (rewrite)
  (implies (listp m)
    (equal (compb 'list m)
          (and (combp (car m))
               (compb 'list (cdr m))))))

(prove-lemma compb-rewrite-2 (rewrite)
  (implies (nlistp m)
            (compb 'list m)))

(prove-lemma compb-rewrite-3 (rewrite)
  (implies (and (modulep m) (structp m))
(equal (combp t m)
  (and (adepth m 0) (combp 'list (s m))))

(prove-lemma combp-modulep (rewrite)
  (implies (combp m) (modulep m)))

(disable combp)

(disable combp$)

(prove-lemma match-inputs-rewrite-1 (rewrite)
  (implies (listp subs)
    (equal (match-inputs subins subs)
      (and (listp subins)
        (equal (length (car subins)) (ni (car subs)))
        (match-inputs (cdr subins) (cdr subs))))))

(prove-lemma match-inputs-rewrite-2 (rewrite)
  (implies (nlistp subs)
    (match-inputs subins subs)))

(prove-lemma match-outputs-rewrite-1 (rewrite)
  (implies (listp subs)
    (equal (match-outputs subouts subs)
      (and (equal (length (car subouts)) (no (car subs)))
        (match-outputs (cdr subouts) (cdr subs))))))

(prove-lemma match-outputs-rewrite-2 (rewrite)
  (implies (nlistp subs)
    (match-outputs subouts subs)))

(disable match-inputs)

(disable match-outputs)

(prove-lemma modulep$-rewrite-1 (rewrite)
  (implies (structp m)
    (equal (modulep$ t m)
      (and (equal (length (li m)) (length (s m)))
        (match-inputs (li m) (s m))
        (equal (length (lo m)) (length (s m)))
        (match-outputs (lo m) (s m))
        (all-appear (o m) (lo m))
        (lists-all-appear (li m) (cons (i m) (lo m)))
        (all-distinct-symbols (cons (i m) (lo m)))
        (modulep$ 'list (s m)))))

(prove-lemma modulep$-rewrite-2 (rewrite)
  (implies (listp m)
    (equal (modulep$ 'list m)
      (and (modulep (car m))
        (modulep$ 'list (cdr m))))))

(prove-lemma modulep$-rewrite-3 (rewrite)
  (modulep$ 'list ()
  (disable modulep)

(disable modulep$)
(prove-lemma slevel$$-rewrite-1 (rewrite)
  (implies (listp out)
    (equal (slevel$$ 'list out m bad q)
      (emax (slevel$$ t (car out) m bad q)
        (slevel$$ 'list (cdr out) m bad q)))))

(prove-lemma slevel$$-rewrite-2 (rewrite)
  (implies (nlistp out)
    (equal (slevel$$ 'list out m bad q) 0)))

(prove-lemma slevel$$-rewrite-3 (rewrite)
  (implies (or (member out (i m))
    (lessp (index out (lo m)) q))
    (equal (slevel$$ t out m bad q) 0)))

(prove-lemma slevel$$-rewrite-4 (rewrite)
  (implies (and (not (member out (i m)))
    (not (lessp (index out (lo m)) q))
    (not (member out bad))
    (distinct-symbols bad)
    (member out (signals m))
    (subsetp bad (signals m)))
    (equal (slevel$$ t out m bad q)
      (eaddl (slevel$$ 'list (find-li out m) m (cons out bad) q))))

(disable slevel$$)

(prove-lemma cv$-rewrite-1 (rewrite)
  (implies (and (combp m) (structp m) (subsetp s (signals m)) (listp s))
    (equal (cv$ 'list s v m)
      (cons (cv (car s) v m)
        (cv$ 'list (cdr s) v m))))

(prove-lemma cv$-rewrite-2 (rewrite)
  (implies (and (combp m) (structp m) (nlistp s))
    (equal (cv$$ 'list s v m) 0)))

(prove-lemma cv$-rewrite-3 (rewrite)
  (implies (and (combp m) (structp m) (member s (signals m)))
    (equal (cv$ t s v m)
      (if (member s (i m))
        (lookup s (i m) v)
        (cv (find-o s m)
          (cv$ 'list (find-li s m) v m)
          (find-s s m))))))

(prove-lemma cv$-rewrite-4 (rewrite)
  (implies (beavp m)
    (equal (cv$ t s v m)
      (eval (lookup s (o m) (r m)) (pairlist (i m) v)))))

(prove-lemma cv-rewrite (rewrite)
  (equal (cv s v m) (cv$ t s v m)))

(disable cv)

(disable cv$)
(prove-lemma dcmin$-rewrite-1 (rewrite)
  (implies (and (combp m) (structp m) (subsetp s (signals m)) (listp s))
    (equal (dcmin$ 'list s m)
      (emin (dcmin (car s) m)
        (dcmin$ 'list (cdr s) m)))))

(prove-lemma dcmin$-rewrite-2 (rewrite)
  (implies (and (combp m) (structp m) (nlistp s))
    (equal (dcmin$ 'list s m) 0)))

(prove-lemma dcmin$-rewrite-3 (rewrite)
  (implies (and (combp m) (structp m) (member s (signals m)))
    (equal (dcmin$ t s m)
      (if (member s (i m))
        0
        (eplus (dcmin (find-o s m) (find-s s m))
          (dcmin$ 'list (find-li s m) m))))))

(prove-lemma dcmin$-rewrite-4 (rewrite)
  (implies (behavp m)
    (equal (dcmin$ t s m)
      (lookup s (o m) (d m))))))

(prove-lemma dcmin-rewrite (rewrite)
  (equal (dcmin s m) (dcmin$ t s m)))

(disable dcmin$)

(disable dcmin)

(prove-lemma dcmax$-rewrite-1 (rewrite)
  (implies (and (combp m) (structp m) (subsetp s (signals m)) (listp s))
    (equal (dcmax$ 'list s m)
      (emax (dcmax (car s) m)
        (dcmax$ 'list (cdr s) m)))))

(prove-lemma dcmax$-rewrite-2 (rewrite)
  (implies (and (combp m) (structp m) (nlistp s))
    (equal (dcmax$ 'list s m) 0)))

(prove-lemma dcmax$-rewrite-3 (rewrite)
  (implies (and (combp m) (structp m) (member s (signals m)))
    (equal (dcmax$ t s m)
      (if (member s (i m))
        0
        (eplus (dcmax (find-o s m) (find-s s m))
          (dcmax$ 'list (find-li s m) m)))))

(prove-lemma dcmax$-rewrite-4 (rewrite)
  (implies (behavp m)
    (equal (dcmax$ t s m)
      (lookup s (o m) (d m))))))

(prove-lemma dcmax-rewrite (rewrite)
  (equal (dcmax s m) (dcmax$ t s m)))

(disable dcmax$)

(disable dcmax)
(prove-lemma lookup-rewrite (rewrite)
  (implies (listp i)
    (equal (lookup s i v)
      (if (equal s (car i))
        (car v)
        (lookup s (cdr i) (cdr v))))))

(disable lookup)

(prove-lemma lookup1-rewrite (rewrite)
  (implies (listp i)
    (equal (lookup1 s i v)
      (if (member s (car i))
        (car v)
        (lookup1 s (cdr i) (cdr v))))))

(disable lookup1)

;; For each gate, we establish its components, prove that it is a
;; combinational module, derive its basic parameters, and then disable its
;; definition:

(defun hyphen (x y)
  (intern (format () "-" A x y)))

(defun ex (m)
  (intern (format () "#1" A m)))

(defun dogate (m i o r d cv)
  "(and (print-and-prove , (hyphen m 'type) (rewrite))
    (equal (type , (m)) 'behav)
    ((enable type)))
  (print-and-prove , (hyphen m 'i) (rewrite)
    (equal (i , (m)) ',i)
    ((enable i)))
  (print-and-prove , (hyphen m 'o) (rewrite)
    (equal (o , (m)) ',o)
    ((enable o)))
  (print-and-prove , (hyphen m 'r) (rewrite)
    (equal (r , (m)) ',r)
    ((enable r)))
  (print-and-prove , (hyphen m 'd) (rewrite)
    (equal (d , (m)) ',d)
    ((enable d)))
  (print-and-prove , (hyphen m 'p) (rewrite)
    (equal (p , (m)) ',p)
    ((enable p)))
  (print-and-prove , (hyphen m 'modulep) (rewrite)
    (modulep , (m)))
  (print-and-prove , (hyphen m 'combp) (rewrite)
    (combp , (m)))
  (print-and-prove , (hyphen m 'cv) (rewrite)
    (equal (cv , v , (m)))
    ((enable cv)))")
(print-and-prove ,(hyphen m 'dmin) (rewrite)
   (equal (dmin ',o (,m)) ,d))
(print-and-prove ,(hyphen m 'dmax) (rewrite)
   (equal (dmax ',o (,m)) ,d))
(disable ,m)
(disable ,(ex m)))))

dogate t0 () t (t0) 2000 t)
(dogate f0 () f (f0) 2000 f)
(dogate not1 (a) b (not1 a) 2000
   (not (car v)))
(dogate and2 (a b) c (and2 a b) 2000
   (and (car v) (cadr v)))
(dogate or2 (a b) c (or2 a b) 2000
   (or (car v) (cadr v)))
(dogate nand2 (a b) c (nand2 a b) 2000
   (not (and (car v) (cadr v))))
(dogate fnand2 (a b) c (nand2 a b) 1000
   (not (and (car v) (cadr v))))
(dogate nor2 (a b) c (nor2 a b) 2000
   (not (or (car v) (cadr v))))
(dogate xor2 (a b) c (xor2 a b) 2000
   (not (equal (car v) (cadr v))))
(dogate and3 (a b c d) e (and3 a b c) 2000
   (and (car v) (cadr v) (caddr v)))
(dogate or3 (a b c d) e (or3 a b c) 2000
   (or (car v) (cadr v) (caddr v)))
(dogate nand3 (a b c d) e (nand3 a b c) 2000
   (not (and (car v) (cadr v) (caddr v))))
(dogate nor3 (a b c d) e (nor3 a b c) 2000
   (not (or (car v) (cadr v) (caddr v))))
(dogate xor3 (a b c d) e (xor3 a b c) 2000
   (not (equal (car v) (cadr v)) (not (equal (cadr v) (caddr v))))))
(dogate and4 (a b c d) e (and4 a b c d) 2000
   (and (car v) (cadr v) (caddr v) (cadddr v)))
(dogate or4 (a b c d) e (or4 a b c d) 2000
   (or (car v) (cadr v) (caddr v) (cadddr v)))
(dogate nand4 (a b c d) e (nand4 a b c d) 2000
   (not (and (car v) (cadr v) (caddr v) (cadddr v))))
(dogate nor4 (a b c d) e (nor4 a b c d) 2000
   (not (or (car v) (cadr v) (caddr v) (cadddr v))))
(dogate xor4 (a b c d) e (xor4 a b c d) 2000
  (not (equal (car v) (not (equal (cadr v) (not (equal (cadadr v) (caddddr v)))))))))

(dogate and5 (a b c d e) g (and5 a b c d e) 2000
  (and (car v) (cadr v) (cadadr v) (caddddr v)))

(dogate or5 (a b c d e) g (or5 a b c d e) 2000
  (or (car v) (cadr v) (cadadr v) (caddddr v)))

(dogate nand5 (a b c d e) g (nand5 a b c d e) 2000
  (not (and (car v) (cadr v) (cadadr v) (caddddr v))))

(dogate xor5 (a b c d e) g (xor5 a b c d e) 2000
  (not (equal (car v) (not (not (not (equal (cadadr v) (caddddr v))))))))

;; The same is done for every combinational structure at the time of its definition. We illustrate with the structure ADDER2:

(prove-lemma type-adder2 (rewrite)
  (equal (type (adder2)) 'struct
         (enable type)))

(prove-lemma i-adder2 (rewrite)
  (equal (i (adder2)) '(a b c)
         (enable i)))

(prove-lemma o-adder2 (rewrite)
  (equal (o (adder2)) '(l h)
         (enable o)))

(prove-lemma s-adder2 (rewrite)
  (equal (s (adder2))
         (enable s)))

(prove-lemma li-adder2 (rewrite)
  (equal (li (adder2))
         '((a b) (a t1) (a t1) (t2 t3) (c t4) (t5 t4) (c t5) (t5 t1) (t7 t6))
         (enable li)))

(prove-lemma lo-adder2 (rewrite)
  (equal (lo (adder2))
         '(((t1) (t2) (t3) (t4) (t5) (t6) (t7) (h) (l))
         (enable lo)))

(disable adder2)

(disable *adder2)

(prove-lemma modulep-adder2 (rewrite)
  (modulep (adder2))
  (use (modulep (m (adder2))))))

107
(prove-lemma combp-adder2 (rewrite)
  (combp (adder2))
  ((enable sdepth))
  (use (combp (m (adder2)))))

(prove-lemma cv-adder2-1 (rewrite)
  (implies (cvecp v (adder2))
    (equal (cv 'l v (adder2))
      (not (equal (car v) (not (equal (cadr v) (caddr v)))))))

(prove-lemma cv-adder2-h (rewrite)
  (implies (cvecp v (adder2))
    (equal (cv 'h v (adder2))
      (if (car v) (or (cadr v) (caddr v)) (and (cadr v) (caddr v))))))

(prove-lemma adder2-dcmin-1 (rewrite)
  (equal (dcmin 'l (adder2)) 4000))

(prove-lemma adder2-dcmax-1 (rewrite)
  (equal (dcmax 'l (adder2)) 12000))

(prove-lemma adder2-dcmin-h (rewrite)
  (equal (dcmin 'h (adder2)) 4000))

(prove-lemma adder2-dcmax-h (rewrite)
  (equal (dcmax 'h (adder2)) 10000))

(defun make-s (subs)
  (if (consp subs)
    (cons (list (caar subs)) (make-s (cdr subs)))))

(defun make-li (subs)
  (if (consp subs)
    (cons (cadar subs) (make-li (cdr subs)))))

(defun make-lo (subs)
  (if (consp subs)
    (cons (caddar subs) (make-lo (cdr subs)))))

;; We use the following macro to introduce new combinational structures:
(defmacro defcomb (m i o arest subs)
  (let ((s (make-s subs)) (li (make-li subs)) (lo (make-lo subs)))
    (and (defn ,m ()
      (list 'struct ',i ',o (list ,s) ',li ',lo))
      (print-and-prove ,(hyphen m 'type) (rewrite)
        (equal (type ,m) 'struct)
        ((enable type)))
      (print-and-prove ,(hyphen m 'i) (rewrite)
        (equal (i ,m) ',i)
        ((enable i)))
      (print-and-prove ,(hyphen m 'o) (rewrite)
        (equal (o ,m) ',o)
        ((enable o)))))
We establish a similar procedure for deriving the relevant properties of a sequential module before disabling its definition.

First, we derive the basic properties of DFF:

(prove-lemma not-combp-dff (rewrite)
  (not (combp (dff)))
  ((enable *lnotl *leand2 *l*nand2 *lsnand3)))

(prove-lemma modulep-dff (rewrite)
  (modulep (dff))
  ((enable *lnotl *l*and2 *l*nand2 *l*and3)))

(prove-lemma type-dff (rewrite)
  (equal (type (dff)) 'struct)
  ((enable type)))

(prove-lemma i-dff (rewrite)
  (equal (i (dff)) 'clk rat d))
  ((enable i)))

(prove-lemma o-dff (rewrite)
  (equal (o (dff)) 'q x)
  ((enable o)))

(prove-lemma seqp-dff (rewrite)
  (seqp (dff))
  ((enable *lnotl *l*and2 *l*nand2 *l*and3)))

(prove-lemma rv-rewrite (rewrite)
  (equal (rv s state m) (rv$ t s state m)))

(prove-lemma rv-dff-q (rewrite)
  (equal (rv 'q state (dff)) state))
((enable *1*not1 *1*and2 *1*nand2 *1*nand3 i lo)))

(prove-lemma rv-dff-qn (rewrite)
(equal (rv 'qn state (dff)) (not state))
((enable *1*not1 *1*and2 *1*nand2 *1*nand3 i lo)))

(prove-lemma next-dff (rewrite)
(equal (next v state (dff)) (car v))
((enable *1*not1 *1*and2 *1*nand2 *1*nand3)))

(disable dff)

(disable *1+dff)

;; Next, we derive some rewrite rules that allow us to disable various
;; function definitions:

(defun sc-induct (m)
  (if (structp m)
    (if (equal m (dff))
      t
    (sc-induct (car (s m))))
  t))

(prove-lemma combp-car-s (rewrite)
  (implies (and (structp m)
    (combp m)
    (listp (s m)))
    (combp (car (s m))))
  ((enable combp combp$)
    (expand (combp$ t m))))

(prove-lemma seqp$-car-s (rewrite)
  (implies (and (seqp m)
    (not (equal m (dff))))
    (seqp$ t (car (s m))))
  ((expand (seqp$ t m) (firstn (q$ (s m)) (s m))))

(prove-lemma seq-combp (rewrite)
  (implies (seqp m) (not (combp m)))
  ((induct (sc-induct m))))

(prove-lemma nativep$-rewrite-1 (rewrite)
  (implies (and (sdepth m (q m))
    (subse tp s (signals m))
    (listp s))
    (equal (nativep$ 'list s m)
      (and (nativep$ t (car s) m)
       (nativep$ 'list (cdr s) m))))

(prove-lemma nativep$-rewrite-2 (rewrite)
  (implies (and (sdepth m (q m))
    (alist s))
    (nativep$ 'list s m))

(prove-lemma nativep$-rewrite-3 (rewrite)
  (implies (and (sdepth m (q m))
    (not (equal m (dff)))
    (member s (signals m))
    (not (member s (i m)))))
(equal (nativep$ t s m)
  (if (lessp (index s (lo m)) (q m))
    t
    (nativep$ 'list (find-li s m) m))))

(disable nativep$)

(prove-lemma firstn-rewrite-1 (rewrite)
  (implies (not (zerop n))
    (equal (firstn n) (cons (car 1) (firstn (subl n) (cdr 1)))))))

(prove-lemma firstn-rewrite-2 (rewrite)
  (implies (zerop n)
    (equal (firstn n) ()')))}

(disable firstn)

(prove-lemma seqp$-rewrite-1 (rewrite)
  (implies (listp m)
    (equal (seqp$ 'list m)
      (and (seqp (car m))
        (seqp$ 'list (cdr m))))))}

(prove-lemma seqp$-rewrite-2 (rewrite)
  (implies (nlistp m)
    (seqp$ 'list m))))

(prove-lemma seqp$-rewrite-3 (rewrite)
  (implies (and (modulep m) (structp m) (not (equal m (dff))))
    (equal (seqp$ t m)
      (and (geq (ni m) 2)
        (not (zerop (q m)))
        (seqp$ 'list (firstn (q m) (s m)))
        (check-seq-li (car (i m)) (cadr (i m)) (firstn (q m) (li m)))
        (check-comb-li (car (i m)) (cadr (i m)) (cdadr (q m) (li m)))
        (adepth m (q m))
        (nativep$ 'list (o m) m)))))

((expand (seqp$ t m))))

(disable seqp$)

(disable seqp)

(prove-lemma rv$-rewrite-1 (rewrite)
  (implies (and (seqp m)
    (subsetp s (signals m))
    (listp s)
    (not (equal m (dff)))))
    (equal (rv$ 'list s state m)
      (cons (rv (car s) state m)
        (rv$ 'list (cadr s) state m))))

(prove-lemma rv$-rewrite-2 (rewrite)
  (implies (and (seqp m)
    (nlistp s)
    (not (equal m (dff))))
    (equal (rv$ 'list s state m) ()')))
\begin{verbatim}
(prove-lemma rv$-rewrite-3 (rewrite)
  (implies (and (seqp m)
   (not (equal m (diff)))
   (member s (signals m))
   (not (member s (i m))))
   (equal (rv t s state m)
   (if (lessp (index s (lo m)) (q m))
     (rv (find-o s m) (find-state s state m) (find-s s m))
     (cv (find-o s m) (rv$ 'list (find-li s m) state m) (find-s s m))))))

(disable rv$)

(disable rv)

(prove-lemma sv$-rewrite-1 (rewrite)
  (implies (and (seqp m)
   (subsetp s (signals m))
   (listp s)
   (not (equal m (diff))))
   (equal (sv$ 'list s v state m)
   (cons (sv$ t (car s) v state m)
   (sv$ 'list (cdr s) v state m))))

(prove-lemma sv$-rewrite-2 (rewrite)
  (implies (and (seqp m)
   (nlistp s)
   (not (equal m (diff))))
   (equal (sv$ 'list s v state m) ())))

(prove-lemma sv$-rewrite-3 (rewrite)
  (implies (and (seqp m)
   (not (equal m (diff)))
   (member s (signals m)))
   (equal (sv$ t s v state m)
   (if (member s (i m))
     (lookup s (cddr (i m)) v)
     (if (lessp (index s (lo m)) (q m))
     (rv s state m)
     (cv (find-o s m) (sv$ 'list (find-li s m) v state m) (find-s s m))))))

((disable member)))

(disable sv$)

(prove-lemma next$-rewrite-1 (rewrite)
  (implies (listp m)
   (equal (next$ 'list v state m)
   (cons (next$ 'list v state m)
   (next$ (car v) (car state) (car m))
   (next$ 'list (cdr v) (cdr state) (cdr m))))))

(prove-lemma next$-rewrite-2 (rewrite)
  (implies (nlistp m)
   (equal (next$ 'list v state m) ())))

(prove-lemma next$-rewrite-3 (rewrite)
  (implies (and (seqp m) (not (equal m (diff))))
  (equal (next$ t v state m)
  (if (equal (q m) 1)
    (next (svl (car (li m)) v state m) state (car (s m)))

112
\end{verbatim}
(next$ 'list
  (sull (firstn (q m) (li m)) v state m)
  state
  (firstn (q m) (s m)))))
  ((disable dff)))

(disable next$)

(disable next$)

(prove-lemma q$s-rewrite-1 (rewrite)
  (implies (listp mods)
    (equal (q$s mods)
      (if (combp (car mods))
        0
        (add1 (q$s (cdr mods)))))
    )))

(prove-lemma q$s-rewrite-2 (rewrite)
  (implies (alistp mode)
    (equal (q$s mode) 0)))

(disable q$s)

(disable q)

(prove-lemma statep$-rewrite-1 (rewrite)
  (implies (listp m)
    (equal (statep$ 'list state m)
      (and (statep (car state) (car m))
        (statep$ 'list (cdr state) (cdr m)))))
    ))

(prove-lemma statep$-rewrite-2 (rewrite)
  (implies (alistp m)
    (equal (statep$ 'list state m)
      (equal state ()))))))

(prove-lemma statep$-rewrite-3 (rewrite)
  (implies (and (modulep m) (structp m) (not (equal m (dff))))
    (equal (statep$t state m)
      (if (equal (q m) 1)
        (statep state (car (s m))
          (statep$ 'list state (firstn (q m) (s m)))))))
    )))

(prove-lemma statep-dff-rewrite (rewrite)
  (equal (statep state (dff))
    (boolp state))
  ((disable boolp)))

(disable statep$)

(disable statep)

(prove-lemma regp-rewrite (rewrite)
  (implies (seqp m)
    (equal (regp s m)
      (if (equal m (dff))
        (member s (o m))
        (and (lessp (index s (lo m)) (q m))
          (regp (find-o s m) (find-s s m)))))
      )))
(disable regp)

(prove-lemma dsmin-rewrite-1 (rewrite)
  (implies (and (seqp m) (subsetp s (signals m)) (not (equal m (dff))) (listp s))
    (equal (dsmin 'list s m)
      (dsmin t (car s) m)
      (dsmin 'list (cdr s) m))))

(prove-lemma dsmin-rewrite-2 (rewrite)
  (implies (and (seqp m) (subsetp s (signals m)) (not (equal m (dff))) (nlistp s))
    (equal (dsmin 'list s m) t)))

(prove-lemma dsmin-rewrite-3 (rewrite)
  (implies (and (seqp m) (not (member s (i m)))
    (not (equal m (dff))))
    (equal (dsmin t s m)
      (if (leesp (index s (lo m)) (q m))
        (dsmin (find-o s m) (find-s s m))
        (dsmin 'list (find-li s m) m))))

(prove-lemma dsmin-rewrite-4 (rewrite)
  (implies (and (member s (signals (dff)))
    (not (member s (i (dff)))))
    (equal (dsmin t s m) 4000)))

(prove-lemma dsmin-rewrite (rewrite)
  (equal (dsmin s m) (dsmin t s m)))

(disable dsmin$)

(disable dsmin)

(prove-lemma dff-dsmin-q (rewrite)
  (equal (dsmin 'q (dff)) 4000)
  ((enable *l*dff *i.hand2 *1.1o *l*i *l*nand3 *l*notl)))

(prove-lemma dff-dsmin-qn (rewrite)
  (equal (dsmin 'qn (dff)) 4000)
  ((enable *l*dff *l*nand2 *i*lo *1*i *1*nand3 *1*notl)))

(prove-lemma dmsmax-replace-1 (rewrite)
  (implies (and (seqp m) (subsetp s (signals m)) (not (equal m (dff))) (listp s))
    (equal (dmsmax 'list s m)
      (dmsmax t (car s) m)
      (dmsmax 'list (cdr s) m))))

(prove-lemma dmsmax-replace-2 (rewrite)
  (implies (and (seqp m) (subsetp s (signals m)) (not (equal m (dff))) (nlistp s))
    (equal (dmsmax 'list s m) 0)))

(prove-lemma dmsmax-replace-3 (rewrite)
  (implies (and (seqp m)
    (not (member s (i m)))
    (not (equal m (dff)))))

114
(equal (dsmax$ t s m)
  (if (lessp (index s (lo m)) (q m))
    (dsmax (find-o s m) (find-s s m))
    (eplus (dcmax (find-o s m) (find-s s m))
      (dsmax$ 'list (find-li s m))))))

(prove-lemma dsmax$-rewrite-4 (rewrite)
  (implies (and (member s ($signals (dff))
      (not (member s (i (dff))))))
    (equal (dsmax$ t s (dff)) 6000)))

(prove-lemma dsmax-revrite (rewrite)
  (equal (dsmax s m) (dsmax$ t s m)))

(disable dsmax$)

(disable dsmax)

(prove-lemma dff-dsmax-q (rewrite)
  (equal (dsmax 'q (dff)) 6000)
  ((enable *i=dff *i=nand2 *i=lo *i=i *i=nand3 *i=not1))

(prove-lemma dff-dsmax-qn (rewrite)
  (equal (dsmax 'qn (dff)) 6000)
  ((enable *i=dff *i=nand2 *i=lo *i=i *i=nand3 *i=not1))

(prove-lemma setup-revrite (rewrite)
  (equal (setup s m) (setup$ 0 s m)))

(disable setup)

(prove-lemma dff-setup-rst (rewrite)
  (equal (setup 'rst (dff)) 8000))

(prove-lemma dff-setup-d (rewrite)
  (equal (setup 'd (dff)) 6000))

(prove-lemma setupS-revrite-1 (rewrite)
  (implies (and (seqp m) (not (equal m (dff))))
    (equal (setup$ 0 x m)
      (emax (setup$ 2
        (collect-li x (firstn (q m) (li m)) (firstn (q m) (s m)))
        (collect-lo x (firstn (q m) (li m)) (firstn (q m) (s m)))
        (setup$ 5
          (setup$ 4
            (collect-lo x (cdrn (q m) (li m)) (cdrn (q m) (lo m)))
            (collect-lo x (cdrn (q m) (li m)) (cdrn (q m) (s m)))))

(prove-lemma setup$-revrite-2 (rewrite)
  (implies (listp x)
    (equal (setup$ 1 x m)
      (emax (setup (car x) m)
        (setup$ 1 (cdr x) m))))

(prove-lemma setup$-revrite-3 (rewrite)
  (implies (nlistp x)

115
(equal (setup$ 1 x m) 0))

(prove-lemma setup$-rewrite-4 (rewrite)
  (implies (listp m)
    (equal (setup$ 2 x m)
      (setup$ 2 (cdr x) (cdr m))))))

(prove-lemma setup$-rewrite-5 (rewrite)
  (implies (listp m)
    (equal (setup$ 2 x m) 0))

(prove-lemma setup$-rewrite-6 (rewrite)
  (implies (listp x)
    (equal (setup$ 3 x m)
      (cons (setup (car x) m)
        (setup$ 3 (cdr x) m))))))

(prove-lemma setup$-rewrite-7 (rewrite)
  (implies (listp x)
    (equal (setup$ 3 x m) ()

(prove-lemma setup$-rewrite-8 (rewrite)
  (implies (listp x)
    (equal (setup$ 4 x m)
      (setup$ 4 (cdr x) m))))))

(prove-lemma setup$-rewrite-9 (rewrite)
  (implies (listp x)
    (equal (setup$ 4 x m) ()

(prove-lemma setup$-rewrite-10 (rewrite)
  (implies (listp m)
    (equal (setup$ 5 x m)
      (setup$ 5 (cdr x) (cdr m))))))

(prove-lemma setup$-rewrite-11 (rewrite)
  (implies (listp m)
    (equal (setup$ 5 x m) 0))

(disable setup$)

(prove-lemma setup-comb-rewrite-1 (rewrite)
  (implies (listp sigs)
    (equal (setup-comb sigs setups m)
      (if (zerop (car setups))
        (setup-comb (cdr sigs) (cdr setups) m)
        (setup-comb (cdr sigs) (cdr setups) m))))))

(prove-lemma setup-comb-rewrite-2 (rewrite)
  (implies (listp sigs)
    (equal (setup-comb sigs setups m) 0))

(prove-lemma collect-i-rewrite-1 (rewrite)
  (implies (listp li)
(equal (collect-i s li i)
    (if (equal s (car li))
        (cons (car li) (collect-i s (cdr li) (cdr i)))
        (collect-i s (cdr li) (cdr i))))

(prove-lemma collect-i-rewrite-2 (rewrite)
    (implies (nlistp li)
        (equal (collect-i s li i) 0)))

(prove-lemma collect-li-rewrite-1 (rewrite)
    (implies (listp li)
        (equal (collect-li s li m)
            (if (member s (car li))
                (cons (collect-i s (car li) (car m))
                (collect-li s (cdr li) (cdr m)))
                (collect-li s (cdr li) (cdr m))))))

(prove-lemma collect-li-rewrite-2 (rewrite)
    (implies (nlistp li)
        (equal (collect-li s li m) 0)))

(prove-lemma collect-lo-rewrite-1 (rewrite)
    (implies (listp li)
        (equal (collect-lo s li lo)
            (if (member s (car li))
                (cons (car lo) (collect-lo s (cdr li) (cdr lo)))
                (collect-lo s (cdr li) (cdr lo))))))

(prove-lemma collect-lo-rewrite-2 (rewrite)
    (implies (nlistp li)
        (equal (collect-lo s li lo) 0)))

(prove-lemma high-rewrite (rewrite)
    (equal (high m) (high$ t m)))

(disable high)

(prove-lemma high$-rewrite-1 (rewrite)
    (implies (listp m)
        (equal (high$ 'list m)
            (max (high (car m))
                (high$ 'list (cdr m))))))

(prove-lemma high$-rewrite-2 (rewrite)
    (implies (nlistp m)
        (equal (high$ 'list m) 0)))

(prove-lemma high$-rewrite-3 (rewrite)
    (implies (and (seqp m) (not (equal m (dff)))))
        (equal (high$ t m)
                (high$ 'list (firstn (q m) (s m)))))))

(prove-lemma dff-high-rewrite (rewrite)
    (equal (high (dff)) 4000))

(disable high$)

(prove-lemma low-rewrite (rewrite)
    (equal (low m) (low$ t m)))

117
(disable low)
(prove-lemma low$-rewrite-1 (rewrite)
(implies (listp m)
  (equal (low$ 'list m)
    (max (low (car m))
      (low$ 'list (cdr m))))))
(prove-lemma low$-rewrite-2 (rewrite)
(implies (nlistp m)
  (equal (low$ 'list m) 0)))
(prove-lemma low$-rewrite-3 (rewrite)
(implies (and (seqp m) (not (equal m (iff))))
  (equal (low t m)
    (low$ 'list (firstn (q m) (s m))))))
(prove-lemma diff-low-rewrite (rewrite)
(equal (low (diff)) 6000))
(disable low$)
(prove-lemma setups-plus-delays-rewrite-1 (rewrite)
(implies (listp outs)
  (equal (setups-plus-delays setups outs sub)
    (max (plus (dsmax (car outs) sub)
      (car setups))
      (setups-plus-delays (cdr setups) (cdr outs) sub))))
(prove-lemma setups-plus-delays-rewrite-2 (rewrite)
(implies (nlistp outs)
  (equal (setups-plus-delays setups outs sub) 0)))
(prove-lemma p3-rewrite-1 (rewrite)
(implies (listp s)
  (equal (p3 s Io m)
    (max (setups-plus-delays (setup$ 3 (car 1o) m) (o (car s)) (car s))
      (p3 (cdr s) (cdr 1o) m))))
(prove-lemma p3-rewrite-2 (rewrite)
(implies (nlistp s)
  (equal (p3 s Io m) 0))
(prove-lemma per-rewrite (rewrite)
  (equal (per m) (per$ t m)))
(disable per)
(prove-lemma per$-rewrite-1 (rewrite)
(implies (listp m)
  (equal (per$ 'list m)
    (max (per (car m))
      (per$ 'list (cdr m))))))
(prove-lemma per$-rewrite-2 (rewrite)
(implies (nlistp m)
  (equal (per$ 'list m) 0))

118
(prove-lemma per-diff-rewrite (rewrite)
  (equal (per (dff)) 10000))
(prove-lemma per$-rewrite-3 (rewrite)
  (implies (and (seqp m) (not (equal m (dff))))
    (equal (per$ t m)
      (max (per$ 'list (firstn (q m) (s m))
        (max (setup$ 3 (cdr (i m)) m)
          (p3 (firstn (q m) (s m)) (firstn (q m) (lo m)) m))))))
  (disable per$)
  ;;Finally, we define the following macro, which we use to define
  ;;sequential modules and derive their properties:
(defmacro defseq (m q i o _rest subs)
  (let ((s (make-s subs))
        (li (make-li subs))
        (io (make-lo subs)))
    '(and
      (defn ,m 0
        (list 'struct ',i ',o
          (list ,Os) ',li ',io))
      (print-and-prove ,(hyphen m 'type) (rewrite)
        (equal (type (,m)) 'struct)
        ((enable type)))
      (print-and-prove ,(hyphen m 'i) (rewrite)
        (equal (i (,m)) ',i)
        ((enable i)))
      (print-and-prove ,(hyphen m 'o) (rewrite)
        (equal (o (,m)) ',o)
        ((enable o)))
      (print-and-prove ,(hyphen m 's) (rewrite)
        (equal (s (,m)) (list ,@s))
        ((enable s)))
      (print-and-prove ,(hyphen m 'li) (rewrite)
        (equal (li (,m)) ',li)
        ((enable li)))
      (print-and-prove ,(hyphen m 'lo) (rewrite)
        (equal (lo (,m)) ',lo)
        ((enable lo)))
      (print-and-prove ,(hyphen m 'not-dff) (rewrite)
        (not (equal (,m) (dff))))
        ((enable dff)))
      (disable ,m)
      (disable ,(ex m))
      (print-and-prove ,(hyphen m 'modulep) (rewrite)
        (modulep (,m))
        ((use (modulep (m (,m))))))
      (print-and-prove ,(hyphen m 'q) (rewrite)
        (equal (q (,m)) ',q)
        ((use (q (m (,m))))))
      (print-and-prove ,(hyphen m 'sdepth) (rewrite)
        (sdepth (,m) ',q)
        ((use (sdepth (m (,m)) (q ,q))))
      (print-and-prove ,(hyphen m 'seq) (rewrite)
        (seqq (,m))
        ((use (seqq (m (,m)))))))))

;;*******************************************************
;;
;; BPM
;;*******************************************************
We illustrate our methodology with a pair of circuits, RCVR and SNDR, which achieve asynchronous communication via the biphase mark protocol. The definitions of these circuits are presented below.

Each combinational component is defined via DEFCOMB. For each of its outputs, three lemmas are proved, establishing the values of the functions RV, DCMIN, and DCMAX.

Each sequential component is defined via DEFSEQ. For each output, a lemma is proved pertaining to RV. For each input, a lemma is proved, giving the setup time. Other lemmas give the period and characterize the behavior of STATEP and NEXT:

```
(defun cdff (rst clear d) (q qn)
  (notl (clear) (cn))
  (and2 (d cn) (dcn)))
(prove-lemma cdff-statep
  (equal (statep state (cdff))
    (boolp (state (m (cdff))))))
(prove-lemma rv-cdff-q
  (equal (rv 'q state (cdff)) state))
(prove-lemma rv-cdff-qn
  (equal (rv 'qn state (cdff)) (not state)))
(prove-lemma next-cdff
  (implies (svecp v (cdff))
    (equal (next v state (cdff))
      (if (car v) f (cadr v)))))
(prove-lemma cdff-setup-rst
  (equal (setup 'rst (cdff)) 8000))
(prove-lemma cdff-setup-clear
  (equal (setup 'clear (cdff)) 10000))
(prove-lemma cdff-setup-d
  (equal (setup 'd (cdff)) 8000))
(prove-lemma cdff-per
  (equal (per (cdff)) 10000))
```

```
(defun edff (enable d) (q qn)
  (not1 (enable) (s1))
  (and2 (s1 q) (s2))
  (and2 (d enable) (s3))
  (and2 (s2 s3) (s4)))
(prove-lemma edff-statep
  (equal (statep (edff) state))
  ((use (statep (m (edff))))))
```
(equal (statep state (edff))
  (boolp state))
  ((use (statep (m (edff))))))

(prove-lemma rv-edff-q (rewrite)
  (equal (rv 'q state (edff)) state))

(prove-lemma rv-edff-qn (rewrite)
  (equal (rv 'qn state (edff)) (not state)))

(prove-lemma next-edff (rewrite)
  (implies (and (svecp v (edff))
    (statep state (edff)))
    (equal (next v state (edff))
      (if (car v) (cadr v) state))
  ((use (next (m (edff))))
    (statep (m (edff))))))

(prove-lemma edff-setup-rst (rewrite)
  (equal (setup 'rst (edff)) 8000))

(prove-lemma edff-setup-enable (rewrite)
  (equal (setup 'enable (edff)) 12000))

(prove-lemma edff-setup-d (rewrite)
  (equal (setup 'd (edff)) 10000))

(prove-lemma edff-per (rewrite)
  (equal (per (edff)) 16000))

(defseq ecddff l
  (clk rst clear enable d) (q qn)
  (d (clk rst s5) (q qn))
  (not1 (enable) (s1))
  (not1 (clear) (s2))
  (nand3 (q s1 s2) (s3))
  (nand2 (d s2 enable) (s4))
  (nand2 (s3 s4) (s5)))

(prove-lemma ecddff-statep (rewrite)
  (equal (statep state (ecddff))
    (boolp state))
  ((use (statep (m (ecddff))))))

(prove-lemma rv-ecddff-q (rewrite)
  (equal (rv 'q state (ecddff)) state))

(prove-lemma rv-ecddff-qn (rewrite)
  (equal (rv 'qn state (ecddff)) (not state)))

(prove-lemma next-ecddff (rewrite)
  (implies (and (svecp v (ecddff))
    (statep state (ecddff)))
    (equal (next v state (ecddff))
      (if (car v) f (if (cadr v) (caddr v) state))
  ((use (next (m (ecddff))))
    (statep (m (ecddff))))))
(prove-lemma ecdff-setup-rst (rewrite)
  (equal (setup 'rst (ecdff)) 8000))

(prove-lemma ecdff-setup-clear (rewrite)
  (equal (setup 'clear (ecdff)) 12000))

(prove-lemma ecdff-setup-enable (rewrite)
  (equal (setup 'enable (ecdff)) 12000))

(prove-lemma ecdff-setup-d (rewrite)
  (equal (setup 'd (ecdff)) 10000))

(prove-lemma ecdff-per (rewrite)
  (equal (per (ecdff)) 16000))

(defseq port3 1
  (clk rst shift sin load din) (q)
  (edff (clk rst s3 s4) (q qn))
  (nand2 (din load) (s1))
  (nand2 (sin shift) (s2))
  (or2 (load shift) (s3))
  (nand2 (s1 s2) (s4)))

(prove-lemma port3-statep (rewrite)
  (equal (statep state (port3))
    (boolp state))
  ((use (statep (m (port3))))))

(prove-lemma rv-port3-q (rewrite)
  (equal (rv 'q state (port3)) state))

(prove-lemma next-port3-1 (rewrite)
  (implies (and (svecp v (port3))
    (statep state (port3))
    (not (car v)))
    (equal (next v state (port3))
      (if (caddr v) (cadr v) state))
    ((use (next (m (port3))))
      (statep (m (port3))))))

(prove-lemma next-port3-2 (rewrite)
  (implies (and (svecp v (port3))
    (statep state (port3))
    (not (caddr v)))
    (equal (next v state (port3))
      (if (car v) (cadr v) state))
    ((use (next (m (port3))))
      (statep (m (port3))))))

(prove-lemma port3-setup-rst (rewrite)
  (equal (setup 'rst (port3)) 8000))

(prove-lemma port3-setup-shift (rewrite)
  (equal (setup 'shift (port3)) 14000))

(prove-lemma port3-setup-sin (rewrite)
  (equal (setup 'sin (port3)) 14000))
(prove-lemma port3-setup-load (rewrite)
  (equal (setup 'load (port3)) 14000))

(prove-lemma port3-setup-din (rewrite)
  (equal (setup 'din (port3)) 14000))

(prove-lemma port3-per (rewrite)
  (equal (per (port3)) 16000))

(defseq shift8
  (clk rst load shift sin d0 d1 d2 d3 d4 d5 d6 d7)
  (q0 q1 q2 q3 q4 q5 q6 q7)
  (port3 (clk rst shift sin load d0) (q0))
  (port3 (clk rst shift q0 load d1) (q1))
  (port3 (clk rst shift q1 load d2) (q2))
  (port3 (clk rst shift q2 load d3) (q3))
  (port3 (clk rst shift q3 load d4) (q4))
  (port3 (clk rst shift q4 load d5) (q5))
  (port3 (clk rst shift q5 load d6) (q6))
  (port3 (clk rst shift q6 load d7) (q7)))

(prove-lemma shift8-statep (rewrite)
  (equal (statep state (shift8))
        (bvpn state 8))
  (use (statep (m (shift8))))
  (disable boolp)))

(prove-lemma rv-shift8-q0 (rewrite)
  (equal (rv 'q0 state (shift8)) (car state)))

(prove-lemma rv-shift8-q1 (rewrite)
  (equal (rv 'q1 state (shift8)) (cadr state)))

(prove-lemma rv-shift8-q2 (rewrite)
  (equal (rv 'q2 state (shift8)) (caddr state)))

(prove-lemma rv-shift8-q3 (rewrite)
  (equal (rv 'q3 state (shift8)) (cadddr state)))

(prove-lemma rv-shift8-q4 (rewrite)
  (equal (rv 'q4 state (shift8)) (cadddddr state)))

(prove-lemma rv-shift8-q5 (rewrite)
  (equal (rv 'q5 state (shift8)) (caddddddr state)))

(prove-lemma rv-shift8-q6 (rewrite)
  (equal (rv 'q6 state (shift8)) (cadddddddr state)))

(prove-lemma rv-shift8-q7 (rewrite)
  (equal (rv 'q7 state (shift8)) (caddddddddr state)))

(defn shift (sin 1)
  (if (listp 1)
      (cons sin (shift (car 1) (cdr 1)))
      ()))

(prove-lemma shift-rewrite-1 (rewrite)
  (implies (boolp (car 1))
            ())))
(equal (shift s 1)
(cons s (shift (car 1) (cdr 1))))

(prove-lemma shift-rewrite-2 (rewrite)
(implies (null p)
(equal (shift s 1) ()�)
)

(disable shift)

(prove-lemma cons-car-nil (rewrite)
(implies (equal (cdr u) ())
(equal (cons (car u) ()) u))

(disable cons-car-nil)

(prove-lemma next-shiftS-1 (rewrite)
(implies (and (sweep v (shift8))
(statep state (shift8))
(not (car v)))
(equal (next v state (shift8))
(if (cadr v) (shift (caddr v) state) state)))

(use (next (m (shift8))))
(enable cons-car-nil)
(disable boolp))

(prove-lemma next-shiftS-2 (rewrite)
(implies (and (svecp v (shift8))
(statep state (shift8))
(not (cadr v)))
(equal (next v state (shift8))
(if (car v) (cdddr v) state)))

(use (next (m (shift8))))
(enable cons-car-nil)
(disable boolp))

(prove-lemma shiftS-setup-ret (rewrite)
(equal (setup 'rst (shift8)) 8000))

(prove-lemma shiftS-setup-shift (rewrite)
(equal (setup 'shift (shift8)) 14000))

(prove-lemma shiftS-setup-sin (rewrite)
(equal (setup 'sin (shift8)) 14000))

(prove-lemma shiftS-setup-load (rewrite)
(equal (setup 'load (shift8)) 14000))

(prove-lemma shiftS-setup-d0 (rewrite)
(equal (setup 'd0 (shift8)) 14000))

(prove-lemma shiftS-setup-d1 (rewrite)
(equal (setup 'd1 (shift8)) 14000))

(prove-lemma shiftS-setup-d2 (rewrite)
(equal (setup 'd2 (shift8)) 14000))

(prove-lemma shiftS-setup-d3 (rewrite)
(124)
(equal (setup 'd3 (shiftS)) 14000))
(prove-lemma shiftS-setup-d4 (rewrite)
(equal (setup 'd4 (shiftS)) 14000))
(prove-lemma shiftS-setup-d5 (rewrite)
(equal (setup 'd5 (shiftS)) 14000))
(prove-lemma shiftS-setup-d6 (rewrite)
(equal (setup 'd6 (shiftS)) 14000))
(prove-lemma shiftS-setup-d7 (rewrite)
(equal (setup 'd7 (shiftS)) 14000))
(prove-lemma shiftS-per (rewrite)
(equal (per (shiftS)) 20000))
(defcomb comp5 (c0 b0 c1 b1 c2 b2 c3 b3 c4 b4) (match)
xor2 (c0 b0) (s1))
xor2 (c1 b1) (s2))
xor2 (c2 b2) (s3))
xor2 (c3 b3) (s4))
xor2 (c4 b4) (s5))
nor5 (s1 s2 s3 s4 s5) (match)))
(prove-lemma cv-comp5 (rewrite)
(let ((c0 (car v)) (b0 (cadr v))
(c1 (caddr v)) (b1 (cadddr v))
(c2 (caddddr v)) (b2 (caddddddr v))
(c3 (caddddddr v)) (b3 (cadddddddr v))
(c4 (cadddddddr v)) (b4 (caddddddddr v))))
(implies (cvecp v (comp5))
(equal (cv 'match v (comp5))
(equal (list b0 b1 b2 b3 b4) (list c0 c1 c2 c3 c4))))
((disable boolp)))
(defseq count3 3
(clk rst enable) (q0 q1 q2)
edff (clk rst enable q0) (q0 q0))
edff (clk rst enable s3) (q1 qn1))
edff (clk rst enable s2) (q2 qn2))
(and2 (q0 q1) (s1))
xor2 (s1 q2) (s2))
xor2 (q0 q1) (s3))
(prove-lemma countp-statep (rewrite)
(equal (statep state (count3)) (bvpn state 3))
((use (statep (m (count3))))
(disable boolp)))
(prove-lemma rv-count3-q0 (rewrite)
(equal (rv 'q0 state (count3)) (car state)))
(prove-lemma rv-count3-q1 (rewrite)
(equal (rv 'q1 state (count3)) (cadr state)))
(prove-lemma rv-count3-q2 (rewrite)
(equal (rv 'q2 state (count3)) (caddr state)))

(defun modinc (n)
  (if (listp n)
      (if (car n)
          (cons f (modinc (cdr n)))
          (cons t (cdr n)))
      n))

(prove-lemma modinc-rewrite-1 (rewrite)
  (implies (not (car n))
            (equal (modinc n)
                   (cons t (cdr n))))))

(prove-lemma modinc-rewrite-2 (rewrite)
  (implies (and (boolp (car n)) (car n))
            (equal (modinc n)
                   (cons f (modinc (cdr n))))))

(prove-lemma modinc-rewrite-3 (rewrite)
  (implies (nilistp n)
            (equal (modinc n) n)))

(disable modinc)

(prove-lemma next-count3 (rewrite)
  (implies (statep state (count3))
            (equal (next v state (count3))
                   (if (car v)
                       (modinc state)
                       state)))

(prove-lemma count3-setup-rst (rewrite)
  (equal (setup 'rst (count3)) 8000))

(prove-lemma count3-setup-enable (rewrite)
  (equal (setup 'enable (count3)) 12000))

(prove-lemma count3-per (rewrite)
  (equal (per (count3)) 20000))

(defseq count5 5
  (clk rst clear enable) (q0 q1 q2 q3 q4)
  (ecdff (clk rst clear enable qn0) (q0 qn0))
  (ecdff (clk rst clear enable x1) (q1 qn1))
  (ecdff (clk rst clear enable x2) (q2 qn2))
  (ecdff (clk rst clear enable x3) (q3 qn3))
  (ecdff (clk rst clear enable x4) (q4 qn4))
  (and2 (q0 q1) (a1))
  (and2 (a1 q2) (a2))
  (and2 (a2 q3) (a3))
  (xor2 (q0 q1) (x1))
  (xor2 (q2 a1) (x2))
  (xor2 (q3 a2) (x3))
  (xor2 (q4 a3) (x4)))

(prove-lemma count5-statep (rewrite)
(equal (statep state (countS))
(bvpn state 5))
((use (statep (m (countS))))
(disable boolp)))

(prove-lemma rv-countS-q0 (rewrite)
(equal (rv 'q0 state (countS)) (car state)))

(prove-lemma rv-countS-q1 (rewrite)
(equal (rv 'q1 state (countS)) (cadr state)))

(prove-lemma rv-countS-q2 (rewrite)
(equal (rv 'q2 state (countS)) (caddr state)))

(prove-lemma rv-countS-q3 (rewrite)
(equal (rv 'q3 state (countS)) (cadddr state)))

(prove-lemma rv-countS-q4 (rewrite)
(equal (rv 'q4 state (countS)) (caddddr state)))

(prove-lemma next-countS (rewrite)
(implies (statep state (countS))
(equal (next v state (countS))
(if (car v)
(listn s f)
(if (cadr v)
(modinc state)
state))))
((use (next (m (countS))))))

(prove-lemma countS-setup-rst (rewrite)
(equal (setup 'rst (countS)) 8000))

(prove-lemma countS-setup-clear (rewrite)
(equal (setup 'clear (countS)) 12000))

(prove-lemma countS-setup-enable (rewrite)
(equal (setup 'enable (countS)) 12000))

(prove-lemma countS-per (rewrite)
(equal (per (countS)) 24000))

(defseq rcount 2
(clk rst stop start) (bit)
(cdff (clk rst stop s)) (q qn))
(countS (clk rst stop q) (q0 q1 q2 q3 q4))
(or2 (start q) (s1))
(to () (t))
(f0 () (f))
(comp5 (t q0 f q1 f q2 t q3 f q4) (bit)))

(prove-lemma rcount-statep (rewrite)
(equal (statep state (rcount))
(and (boolp (car state))
(bvpn (cadr state) 5)
(equal (cddr state) ()))))
((use (statep (m (rcount))))
(disable boolp)))

127
(prove-lemma rv-rcount-bit (rewrite)
  (implies (statep state (rcount))
    (equal (rv 'bit state (rcount))
      (equal (cadr state) (list t f t f)))))

(prove-lemma next-rcount (rewrite)
  (implies (statep state (rcount))
    (equal (next v state (rcount))
      (if (car v)
        (list t (lietn 5 f))
        (list (if (cadr v) t (car state))
          (if (car state)
            (modinc (cadr state))
            (cadr state)))))))

(prove-lemma rcount-eetup-rst (rewrite)
  (equal (setup 'rst (rcount)) 8000))

(prove-lemma rcount-eetup-stop (rewrite)
  (equal (setup 'stop (rcount)) 12000))

(prove-lemma rcount-eetup-start (rewrite)
  (equal (setup 'start (rcount)) 10000))

(prove-lemma rcount-per (rewrite)
  (equal (per (rcount)) 24000))

(defseq scount 2
  (clk rst stop bit) (mark code)
  (cdff (clk rst stop si) (q qn))
  (count5 (clk rst s2 q) (q0 q1 q2 q3 q4))
  (or2 (bit q) (si))
  (or2 (stop bit) (s2))
  (t0 () (t))
  (f0 () (f))
  (comp5 (f q0 f q1 t q2 f q3 f q4) (mark))
  (comp5 (t q0 f q1 f q2 f q3 t q4) (code)))

(prove-lemma scount-statep (rewrite)
  (equal (statep state (scount))
    (and (boolp (car state))
      (bvpn (cadr state) 5))
    (equal (cddr state) ())))

(prove-lemma rv-scount-mark (rewrite)
  (implies (statep state (scount))
    (equal (rv 'mark state (scount))
      (equal (cadr state) (list f f t f f)))))

(prove-lemma rv-scount-code (rewrite)
(implies (statep state (scount))
  (equal (rv 'code state (scount))
  (equal (cadr state) (list t f f f t)))
  ((disable bvnp boolp)))

(prove-lemma next-scount (rewrite)
  (implies (statep state (scount))
  (equal (next v state (scount))
  (if (car v)
    (list f (listn 5 f))
    (if (cadr v)
      (list t (listn 5 f))
      (if (car state)
        (list (car state) (modinc (cadr state)))
        state)))))
  (use (next (m (scount)))
  (boolp (x (car state))))
  (disable boolp bvnp-rewrite-1 bvnp-rewrite-2)))

(prove-lemma scount-setup-rst (rewrite)
  (equal (setup 'rst (scount)) 8000))

(prove-lemma scount-setup-stop (rewrite)
  (equal (setup 'stop (scount)) 14000))

(prove-lemma scount-setup-bit (rewrite)
  (equal (setup 'bit (scount)) 14000))

(prove-lemma scount-per (rewrite)
  (equal (per (scount)) 24000))

(defseq rcvr 5
  (clk rst sin) (d0 d1 d2 d3 d4 d5 d6 d7 done)
  (edff (clk rst bit n1) (q qn))
  (count (clk rst bit n2) (bit))
  (count3 (clk rst bit) (q0 q1 q2))
  (shift8 (clk rst f bit x f f f f f f f) (d0 d1 d2 d3 d4 d5 d6 d7))
  (diff (clk rst a) (done denen))
  (noti (sin) (n1))
  (noti (x) (n2))
  (xor2 (sin q) (x))
  (and4 (q0 q1 q2 bit) (a))
  (to 0 (_)))

(prove-lemma rcvr-statep (rewrite)
  (equal (statep state (rcvr)))
  (and (boolp (car state))
    (statep (cadr state) (rcount))
    (bvnp (caddr state) 3)
    (bvnp (caddr state) 8)
    (boolp (cadddr state))
    (equal (cdddddr state) ()))
    (use (statep (m (rcvr)))))
    (disable boolp bvnp-rewrite-1 bvnp-rewrite-2)))

(prove-lemma rv-rcvr-d0 (rewrite)
  (implies (statep state (rcvr))
  (equal (rv 'd0 state (rcvr))
  129
(caadddr state)))
((disable bvpn boolp)))

(prove-lemma rv-rcvr-d1 (rewrite)
  (implies (statep state (rcvr))
    (equal (rv 'd1 state (rcvr))
      (caadddr state)))
  ((disable bvpn boolp)))

(prove-lemma rv-rcvr-d2 (rewrite)
  (implies (statep state (rcvr))
    (equal (rv 'd2 state (rcvr))
      (caddaddadr state)))
  ((disable bvpn boolp)))

(prove-lemma rv-rcvr-d3 (rewrite)
  (implies (statep state (rcvr))
    (equal (rv 'd3 state (rcvr))
      (caddaddaddadr state)))
  ((disable bvpn boolp)))

(prove-lemma rv-rcvr-d4 (rewrite)
  (implies (statep state (rcvr))
    (equal (rv 'd4 state (rcvr))
      (caddaddaddaddadr state)))
  ((disable bvpn boolp)))

(prove-lemma rv-rcvr-d5 (rewrite)
  (implies (statep state (rcvr))
    (equal (rv 'd5 state (rcvr))
      (caddaddaddaddaddadr state)))
  ((disable bvpn boolp)))

(prove-lemma rv-rcvr-d6 (rewrite)
  (implies (statep state (rcvr))
    (equal (rv 'd6 state (rcvr))
      (caddaddaddaddaddaddadr state)))
  ((disable bvpn boolp)))

(prove-lemma rv-rcvr-d7 (rewrite)
  (implies (statep state (rcvr))
    (equal (rv 'd7 state (rcvr))
      (caddaddaddaddaddaddadr state)))
  ((disable bvpn boolp)))

(prove-lemma rv-rcvr-done (rewrite)
  (implies (statep state (rcvr))
    (equal (rv 'done state (rcvr))
      (caadddr state)))
  ((disable bvpn boolp)))

(prove-lemma next-rcvr-1 (rewrite)
  (implies (and (statep state (rcvr))
    (swcp v (rcvr))
    (equal (car v (rcvr)) (list f (listn 5 f)))
    (equal (cadddr state) f))
  (equal (next v state (rcvr))
    (if (equal (car v) (car state))
      (list (car state)

130
(list t (listn 5 f))
(caddr state)
(cadddr state)
(f)
(state)))
((use (next (m (rcvr)))))
(boolp (x (car state))))
(disable boolp bvpn-rewrite-1 bvpn-rewrite-2)
(enable cons-car-nil)))
(prove-lemma bvp3-t (rewrite)
   (implies (and (bvpn v 3)
   (car v)
   (cadr v)
   (caddr v)
   (cadddr v))
   (equal (equal v (list t t t)) t)))
(prove-lemma next-rcvr-2 (rewrite)
   (implies (and (statep state (rcvr))
   (sweep v (rcvr))
   (equal (caadr state) t)
   (equal (cadddr state) f))
   (equal (next v state (rcvr))
   (if (equal (cadadr state) (list t f t f))
   (list (not (car v))
   (list f (listn 5 f))
   (modinc (caddr state))
   (shift (not (equal (car v) (car state))) (caddr state))
   (equal (caddr state) (list t t t)))
   (list (car state))
   (list (modinc (caddr state)))
   (caddr state)
   (caddr state)
   f)))))
((use (next (m (rcvr)))))
(boolp (x (car state))))
(disable boolp bvpn-rewrite-1 bvpn-rewrite-2)
(enable cons-car-nil)))
(prove-lemma rcvr-setup-rst (rewrite)
   (equal (setup 'rst (rcvr)) 8000))
(prove-lemma rcvr-setup-sin (rewrite)
   (equal (setup 'sin (rcvr)) 16000))
(prove-lemma rcvr-per (rewrite)
   (equal (per (rcvr)) 24000))
(defun sndr 4
  (clk rst send d0 d1 d2 d3 d4 d5 d6 d7) (sout)
  (scount (clk rst a4 o2) (mark code))
  (shift8 (clk rst send code f d0 d1 d2 d3 d4 d5 d6 d7) (q0 q1 q2 q3 q4 q5 q6 q7))
  (count3 (clk rst mark) (c0 c1 c2))
  (edff (clk rst o3 sout) (q sout))
  (or2 (code send) (o2))
  (and2 (q? mark) (a2))
  (and4 (mark c0 c1 c2) (a4))
  (or3 (a2 send code) (o3))
(prove-lemma sndr-statep (rewrite)
  (equal (sndr state (sndr))
    (and (statep (car state) (scount))
      (bvpn (cdr state) 8)
      (bvpn (cadadr state) 3)
      (boolp (cadddr state))
      (equal (cddddr state) ()))))
  ((use (statep (m (sndr)))))
  (disable boolp bvpn-rewrite-1 bvpn-rewrite-2)))

(prove-lemma rv-sndr-sout (rewrite)
  (implies (state D
    state
    (sndr))
    (equal (rv 'sour
      state (sndr))
    (not (cadddr state))))
  ((disable bvpn boolp)))

(prove-lemma regp-sndr-sout (rewrite)
  (regp 'sout (sndr)))

(prove-lemma boolp-car-listp (rewrite)
  (implies (boolp (car v))
    (listp v)))
  (disable boolp-car-listp)

(prove-lemma equal-list-4 (rewrite)
  (implies (and (equal a (car s))
    (equal b (cadadr s))
    (equal c (cadddr s))
    (equal d (cddddr s))
    (equal () (cdddddr s))
    (equal (equal (list a b c d) s)
      t)))

(prove-lemma next-sndr-1 (rewrite)
  (implies (and (statep state (sndr))
    (svecp v (sndr))
    (equal (car state) (list f (listn 5 f)))))
  (equal (next v state (sndr))
    (if (car v)
      (list (list t (listn 5 f))
        (list (cadadr v)
          (cadr v)
          (cadddr v)
          (cddddr v)
          (cdddddr v)
          (cddddddr v)
          (cddddddd r v)
          (cddddddddr v)
          (cadddr state)
          (not (cadr state))))
    (state)))
  ((use (next (m (sndr)))))
  (boolp (x (cadadr state)))
  (disable boolp)
  (enable cons-car-nil boolp-car-listp)))
(prove-lemma next-sndr-2 (rewrite)
  (implies (and (statep state (sndr))
    (svecp v (sndr))
    (not (car v))
    (equal (caar state) t))
  (equal (next v state (sndr))
    (if (equal (cadar state) (list f f t f)) ; mark
      (if (equal (caddr state) (list t t t)) ; 8th bit
        (list (list f (listn 5 f))
          (cadr state)
          (list f f)
          (if (caddddddddadr state)
            (not (caddr state))
            (caddr state))))
      (list (list t (modinc (cadar state)))
        (cadr state)
        (modinc (caddr state))
        (if (caddddddddadr state)
          (not (caddr state))
          (caddr state)))))
  (if (equal (cadar state) (list t f f t)) ; code
    (list (list t (listn 5 f))
      (shift f (cadr state))
      (cadr state)
      (not (caddr state))
      (list (list t (modinc (cadar state)))
        (cadr state)
        (caddr state)
        (caddr state))))))
  ((use (next (m (sndr)))
    (boolp (x (caddr state)))
    (disable boolp)
    (enable cons-car-nil boolp-car-listp)))

(prove-lemma sndr-setup-rst (rewrite)
  (equal (setup 'rst (sndr)) 8000))

(prove-lemma sndr-setup-send (rewrite)
  (equal (setup 'send (sndr)) 16000))

(prove-lemma sndr-setup-d0 (rewrite)
  (equal (setup 'd0 (sndr)) 14000))

(prove-lemma sndr-setup-d1 (rewrite)
  (equal (setup 'd1 (sndr)) 14000))

(prove-lemma sndr-setup-d2 (rewrite)
  (equal (setup 'd2 (sndr)) 14000))

(prove-lemma sndr-setup-d3 (rewrite)
  (equal (setup 'd3 (sndr)) 14000))

(prove-lemma sndr-setup-d4 (rewrite)
  (equal (setup 'd4 (sndr)) 14000))

(prove-lemma sndr-setup-d5 (rewrite)
  (equal (setup 'd5 (sndr)) 14000))

(prove-lemma sndr-setup-d6 (rewrite)
(equal (setup 'd6 (sndr)) 14000))

(prove-lemma sndr-setup-d7 (rewrite)
 (equal (setup 'd7 (sndr)) 14000))

(prove-lemma sndr-per (rewrite)
 (equal (per (sndr)) 26000))
**ABSTRACT**

We present a mathematical definition of a hardware description language (HDL) that admits a semantics-preserving translation to a subset of VHDL. Our HDL includes the basic VHDL propagation delay mechanisms and gate-level circuit descriptions. We also develop formal procedures for deriving and verifying concise behavioral specifications of combinational and sequential devices. The HDL and the specification procedures have been formally encoded in the computational logic of Boyer and Moore, which provides a LISP implementation as well as a facility for mechanical proof-checking. As an application, we design, specify, and verify a circuit that achieves asynchronous communication by means of the biphase mark protocol.