Technical Report

UCAM-CL-TR-132
ISSN 1476-2986

Number 132

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

Miriam Ellen Leeser

April 1988

This technical report is based on a dissertation submitted December 1987 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Queens' College.

ISSN 1476-2986

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

Miriam Ellen Leeser
Queens' College
Cambridge University

Abstract

The structure of circuits is specified with Prolog; their function and timing behavior is specified with interval temporal logic. These structural and behavioral specifications are used to formally verify the functionality of circuit elements as well as their timing characteristics. A circuit is verified by deriving its behavior from the behavior of its components. The derived results can be abstracted to functional descriptions with timing constraints. The functional descriptions can then be used in proofs of more complex hardware circuits.

Verification is done hierarchically, with transistors as primitive elements. Transistors are modeled as switch-level devices with delay. In order to model delay, the direction of signal flow through each transistor must be assigned. This is done automatically by a set of Prolog routines which also determine the inputs and outputs of each circuit component.

Interval temporal logic descriptions are expressed in Prolog and manipulated using PALM: Prolog Assistant for Logic Manipulation With PALM, the user specifies rewrite rules and uses these rules to manipulate logical terms. In the case of reasoning about circuits, PALM is used to manipulate the temporal logic descriptions of the components to derive a temporal logic description of the circuit.

These techniques are demonstrated by applying them to several commonly used complementary metal oxide semiconductor (CMOS) structures. Examples include a fully complementary dynamic latch and a 1-bit adder. Both these circuits are implemented with transistors and exploit 2-phase clocking and charge sharing. The 1-bit adder is a sophisticated full adder implemented with a dynamic CMOS design style. The derived timing and functional behavior of the 1-bit adder is abstracted to a purely functional behavior which can be used to derive the behavior of an arbitrary n-bit adder.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The increasing complexity of integrated circuits has generated interest in new methods for the computer aided design of these circuits. One such method is formal hardware verification — using techniques based on mathematical logic to formally prove that a circuit correctly implements its behavioral specification. This technique is applicable at many different levels of implementation and specification. For example, it can be proved that a correctly connected group of combinational gates implements an adder; that an adder, registers, and control logic implement an arithmetic logic unit (ALU); and that an ALU, memory busses, registers, microprogram section, and control logic correctly implement a central processing unit. Research has been done into behavioral verification at these levels. Investigations have also been done into relating the various levels with abstraction, and into formally describing the relative timing of signals. Recent research, however, has not concentrated on implementation details of concern to real designers. These details include the implementation of logic gates with transistors, and the timing considerations due to delays through these transistors. My research addresses the issues of formal verification applied to this low level of the design hierarchy: relating transistor level implementations of circuits to their functional specification, and formally deriving the timing behavior of these circuits.

1.2 Specifying Hardware

I am interested in specifying and verifying hardware at the level of implementations of cells in a complementary metal oxide semiconductor (CMOS) design library. These cells are then used as building blocks when constructing larger integrated circuit designs.

Typically, the behavior of such cells is specified informally in English, their implementation given by a layout, and their timing specified with equations and timing diagrams. I describe their behavior and timing using interval temporal logic (ITL) and their implementation using Prolog.

I assume that hardware is described modularly and hierarchically. A chip is made up of a series of functional blocks which in turn are made up of functional blocks, and so forth, until the transistor level description is reached. Each level need not contain circuit elements all of the same type. For example, adders and transistors may be mixed at the same level of description.

This hierarchy does not imply that a designer is constrained to use a top-down refinement design style. Design is usually a combination of top-down refinement and bottom-up composition. No matter how the design is attacked, the design can be described in a hierarchy of design levels, and each adjacent pair of levels can be related formally.

1.3 Why Formal Hardware Verification?

Formal hardware verification has several advantages over conventional methods such as simulation for verifying circuits. With formal verification, signals are manipulated symbolically. By proving that an implementation meets its specification, the designer is sure that it has that behavior for all cases; in simulation the designer can only be sure of the behavior for the cases tested. Another advantage of verification is the ability to exploit modularity. A functional block such as an adder need only be proved once no matter how many times it is used in the circuit. In addition, a portion of the design can be proved to meet its specification before the rest of the design is complete. Thus errors can be caught early in the design process. Modularity also allows small changes to a design to be handled easily since only those portions of the design which have been altered need to be reverified.

These advantages do not imply that formal verification will replace simulation. Formal verification can be viewed as another tool in the designer's tool box which gives added confidence in the correctness of designs. A drawback of verification is that one can never have complete confidence in the specification. It is useful to simulate a specification to check that it exhibits the required behavior.

A circuit is verified by showing that its behavior can be derived from the mathematical models of the behavior of the components which make it up. If the mathematical model does not capture a physical reality of the circuit, that physical reality will not be captured in the proof. For example, a proof system whose primitives are combinational elements without delay cannot derive delay characteristics of a circuit. Similarly, if charge-sharing is not modeled, charge-sharing bugs will not be detected. In these cases other tools such as timing analyzers and design rule checkers are required.

1.4 Adding Timing Analysis to Functional Verification

Existing tools for timing analysis do not use formal techniques. The advantage of using formal methods for timing is that they allow timing models to be related to the higher level behavioral models of circuit elements using abstraction. The same descriptions of circuit implementations can be used to do timing as well as other analysis, and different levels of description can be related formally. The designer, therefore, has increased confidence that various aspects of the design are correct, since different tools are all working on the same representation of the design.

I derive low level timing details such as delay and set-up and hold times from interval temporal logic (ITL) [Mos83] descriptions of circuit components. The timing analysis is done in conjunction with behavioral verification since delay, for example, depends on the function of the circuit as well as the timing characteristic of its components.

In addition, I express constraints on the behavior of inputs and outputs of a circuit component in interval temporal logic. These constraints, which express such conditions as when the inputs must be stable, are verified using the same formal techniques. They are verified when the component is composed with other components to form a larger functional block.

1.5 Synchronous CMOS Circuits

The techniques for reasoning about circuits presented in this thesis can be applied to a wide range of technologies and design styles. My examples are restricted to synchronous Complementary Metal Oxide Semiconductor (CMOS) circuits [WE85]. I chose this class of circuits for several reasons. CMOS has become an important commercial technology. MOS lends itself to switch-level analysis. In many CMOS designs, the ratios of transistor sizes is not important as it is in nMOS. (This is not the case with such designs as RAM cells; I do not consider these.) Synchronous circuits lend themselves to a hierarchical, systematic design style, and avoid many of the problems which are associated with asynchrony. In addition, it is easier to reason about synchronous systems formally, and to formulate timing constraints.

I do not consider, in detail, the design and timing issues which arise from layout. Layout is an important aspect of VLSI design, and a significant proportion of delay in CMOS circuits is due to interconnect. The methods for deriving the timing behavior of a circuit can be extended to model circuits whose delay characteristics are derived from information extracted from the layout. A wire can be modeled as a simple case of a combinational element with delay.

The circuits discussed are synchronous CMOS circuits where all feedback loops are broken by clocks and there are no reconvergent branches in combinational logic. There are many practical VLSI designs which fall into this class of circuits. Within this framework I consider several different CMOS design styles, including full complementary CMOS and dynamic CMOS.

1.6 Contributions

The main contributions of this thesis are:

- Automatically deriving characteristics of a circuit from the schematic. These characteristics include direction of signal flow through transistors and direction of ports of components. The direction of a port may be *in*, *out*, or *bidirectional.*
- Extending ITL to model capacitive effects which arise in MOS circuits. One such capacitive effect is charge storage on an undriven node.
- Using extended ITL to reason formally about the timing properties of circuits at the transistor level. The behavior of a transistor is modeled in ITL as a switch with an associated delay. These delays are used to derive the length of the different cycles in clocked circuits.
- Expressing and reasoning about constraints in ITL. Constraints include set-up and hold times, and constraints on when a signal must be stable.

1.7 Organization

In the next chapter, I present related work in hardware validation. I discuss languages and systems for specifying and verifying circuit behavior. I present both informal and formal approaches, and emphasize methods which model timing behavior.

Chapter 3 demonstrates how to specify circuit schematics in Prolog, as well as presenting tools for manipulating these Prolog specifications. These tools use the structure of the circuit to derive characteristics such as which ports are inputs and which are outputs. These characteristics are then used when proving other properties of the circuit.

Chapter 4 gives an introduction to interval temporal logic (ITL). Chapter 5 describes how ITL descriptions are used in verifying function and timing of circuits as well as reasoning about circuit constraints. In Chapter 6, I describe my Prolog system for rewriting logic equations, and show how this system is used for manipulating ITL equations.

In Chapter 7, I present several examples including a complementary CMOS dynamic latch and a 1-bit adder. Both these circuits are implemented with transistors and exploit 2-phase clocking and charge sharing. The 1-bit adder is a sophisticated full adder implemented with a dynamic CMOS design style. I use ITL and the system presented in Chapter 6 to derive the function and timing behavior of these circuits and to reason about constraints. In addition I derive constraints on the duration of the different clocking phases. I also show how the function of the 1-bit adder can be abstracted, and how the abstract behavior can be used in a proof of an n-bit adder.

Finally I present conclusions. I summarize the salient aspects of my approach, discuss issues and improvements, and consider how my approach could be incorporated into a computer aided design system for VLSI.

Chapter 2

Related Work

In this chapter, I present languages and systems which have been developed to aid designers in specifying and verifying their hardware designs. The emphasis is on tools which give designers information about the timing behavior of their circuits. First the current tools available for design validation are presented. These include simulators and timing verifiers. These tools have several drawbacks in that they do not exploit hierarchy and only validate behavior for specified inputs. Recent advances in these informal methods attempt to redress these drawbacks.

Formal verification methods for hardware overcome the drawbacks of simulation. In addition, they provide a means to formally relate the different levels of detail at which a circuit is described, and allow for incremental, hierarchical validation. Thus, a designer has more confidence that the design being validated is the same as that being fabricated. I discuss several representations and systems for formally reasoning about hardware. Here I emphasize systems which use higher-order logic or temporal logic to reason about circuits, and compare the advantages and disadvantages of these two formalisms.

2.1 Current Methods for Hardware Validation

Two widely used methods for validating a design are simulation and timing verification. In simulation, a designer validates the behavior of a design by specifying the inputs and checking that the resulting outputs exhibit the desired behavior. In MOS timing verification, the timing of a circuit is derived by assuming that all transistors are conducting and then calculating the path with the longest delay through the circuit. These techniques usually operate on circuit descriptions which are flat rather than hierarchical. One advantage of these approaches is that they are widely used. As a result of being used with real designs, circuit models tend to be accurate and algorithms used are efficiently implemented.

2.1.1 Simulation

Most simulators used to verify VLSI circuits are either switch-level or mixed-mode. With switch-level simulation [Bry81], large networks of transistors are modeled as nodes connected by switches which have an associated conductance strength. Logic states on nodes are represented by signals which are modeled as values and strengths. A circuit is simulated by applying inputs specified by the designer to a model of the circuit and calculating the resulting outputs. When a steady state is reached, the result is reported and a new set of inputs is driven through the circuit. Every transition in the simulation takes one unit of time. This use of timing makes computation easier, but bears little relation to the time a circuit actually requires to settle. In addition, simulation only models the behavior of the circuit for the inputs specified. In general, the number of input combinations required to fully simulate a circuit is exponential in the number of inputs. Thus, simulating a circuit completely is often impractical, even for simple circuits.

Mixed-mode simulators are capable of simulating circuits which are described at several different levels of detail. A commonly used mixed-mode simulator is MOTIS [AB*80], [CL*84], which simulates circuits described at the transistor level, logic gate level and functional level. MOTIS also handles several different timing models including unit delay, multiple delay and low level timing. With a multiple delay model, different delays are used for different circuit components and for propagating rising and falling pulses. With a low level timing model, voltages are treated as continuous and time is modeled as a sequence of discrete time steps. Mixed-mode simulators exploit some hierarchy in the description of circuits, but still require the user to provide inputs and interpret the outputs.

2.1.2 Timing Verification

Two widely used MOS timing verifiers are TV [Jou83] and Crystal [Ous83]. These timing verifiers are similar to switch-level simulators in that they operate on a transistor level description of the circuit and model transistors as ideal switches. The difference is that simulators simulate the behavior of a circuit for a given set of inputs, while timing verification is independent of input values. The object of timing verification is to identify the slowest paths through a circuit and to determine the maximum possible clock speed. First the direction of signal flow through each transistor is determined; then the delays through paths in the circuit are calculated assuming that all transistors are conducting. Direction of signal flow is determined either automatically or from hints from the designer. TV and Crystal differ in the way paths are traced through a network and in the way delays are calculated. TV uses breadth-first search to exhaustively search the entire circuit, and separately calculates minimum and maximum propagation delays. Crystal uses a depth-first search algorithm and provides an average propagation delay. Both handle different phases of clock cycles by specifying that different clocks cannot be active at the same time.

These timing verifiers report the slowest or *n* slowest paths to the user. A problem with timing verification is that it is frequently difficult for the user to identify how to improve the behavior of these slowest paths. Since the circuit structure is flat, timing verifiers do not recognize bottlenecks that affect several paths. Such bottlenecks would be more obvious from a hierarchical circuit structure. In addition, if several parallel paths exhibit the same timing behavior, they will all be reported. Another problem is that the entire circuit must be re-analyzed if a portion of the design is modified to improve the timing behavior. Jouppi [Jou87] presents an interactive timing assistant (IA) to help alleviate some of these problems. IA proposes design changes and provides incremental timing analysis.

A further drawback of timing verification is the separation of timing from function. The slowest paths predicted may never arise in practice. These paths would not be detected if timing and function were considered together. In addition, certain errors cannot be identified. For example, the relation between signals on different clock phases is not checked. Thus, timing problems which arise because signals do not behave according to certain timing constraints are not identified. In order to find these, some form of verification which incorporates timing and functional verification must also be done.

2.1.3 New Directions in Simulation and Timing Verification

New systems for design validation address some of the shortcomings discussed above. The most common changes are allowing symbolic inputs to simulators, and exploiting hierarchy in design validation.

MOSSYM [Bry85] is an extension of Bryant's switch-level simulator which allows the user to specify inputs symbolically as well as with boolean variables. An advantage of this approach is that the user can choose between conventional simulation, symbolic simulation or a hybrid. A disadvantage is that the user has to interpret the outputs.

Lin and Mead [LM86] present a hierarchical timing simulator. This approach is used to model synchronous circuits similar to those presented in this thesis. They assume that input and output ports of components have been determined before simulation begins, and assume that an input may only change once in a given clock period. These assumptions are not checked. Functional and timing behavior is validated. This involves simulating a component, and then replacing that component with its derived behavior. The behavior can be derived analytically as well as through simulation. One drawback is that design validation must be bottom up. In other words, a designer cannot specify a component at a high level, later give a more detailed representation, and show that these two representations are equivalent.

2.2 Formal Hardware Verification

In the previous section I presented some commonly used design validation techniques and discussed their advantages and disadvantages. I also presented some new techniques which redress some of these disadvantages.

In this section I describe formal methods for design verification. These methods address many of the disadvantages mentioned above. Hierarchical design is supported, and different levels of description can be formally related. Localized changes to the circuit require that only the altered components be re-verified. Inputs and outputs are represented symbolically rather than by values. Assumptions about designs can be expressed formally and verified along with verification of behavior. The major drawback of these systems is that they have not been widely used to verify real designs. As a result, the models of circuit behavior are not as well developed. In addition, some of these approaches have not been mechanized, which limits their acceptance by designers.

Since I am interested in the timing as well as the functional behavior of circuits, I discuss the way time is modeled. Specifically, I present formal methods which use higher-order logic and those which use temporal logic, and I discuss the relative merits of these two approaches. I also present other formalisms for specifying and verifying hardware, especially those which represent time or deal with low level details of circuit operation. First, I discuss two systems which use first-order logic.

2.2.1 First-Order Logic

Several approaches to formal hardware verification have been based on first-order logic. The approaches presented in this section, [Bar84], [Hun86], automate hardware verification and have been used to verify complex hardware designs. Drawbacks in the use of first-order logic include difficulty in expressing timing in a detailed manner, and difficulty in expressing abstraction between different levels of description. Systems which use higher-order logic do not experience these difficulties. Two such systems are presented in the next section.

Verify

Verify [Bar84] is a Prolog system which attempts to automatically prove that the behavior of the implementation of a digital system is equivalent to the specification. Verify uses Prolog as a hardware description language. The behavior of a component is specified as a finite state machine with next state and output equations. There is an implicit clock; at each clock tick the state variables and outputs are updated. Timing at a more realistic level is not considered. With Verify, circuits can be described and verified down to the transistor level. The transistor model employed is a simple unidirectional, switch-level model. Different levels of description can be related in a limited manner, but abstraction functions cannot be defined by the user.

Boyer-Moore Theorem Prover

Hunt [Hun86] uses the Boyer-Moore theorem prover to prove correct a 16-bit microprogrammed microprocessor implemented at the register-transfer level. The behavior of combinational circuits is described by recursively defined functions. Sequential circuits are modeled using self-recursive functions which have an explicit clock argument. These functions call themselves recursively once each clock tick. Thus time is discrete and behavior at any given level is related to the representation of a single clock. The theorem prover is mechanical; its operation is guided by heuristics. Since the logic used is first-order and there are no quantifiers, Hunt has difficulty formally relating clocks at different levels of description of the microprocessor. He overcomes this difficulty by using *oracles*: functions which guess the exact number of time steps a low level representation needs to execute to be equivalent to the next higher level specification. Such oracles cannot be represented in first-order logic. If higher-order logic is used as the specification language, abstraction between levels can be expressed directly, and such oracles are not required.

2.2.2 Higher-Order Logic

Higher-order logic provides several advantages over first-order logic, including a more direct way of expressing abstraction and a more elegant way of representing time with signals modeled as functions from time to values. I describe two important systems which employ higher-order logic to reason about hardware.

HOL

The HOL (Higher-Order Logic) system was developed by Mike Gordon for specifying and verifying hardware [Gor85b]. HOL is being used by many people at Cambridge University as well as elsewhere [CGM86]. Several large examples from real designs are being verified, including the VIPER microprocessor [Coh87]. My work has been influenced by the approach taken by HOL. A hardware component is described in HOL with a logical formula, components are composed by *and*ing them together, and internal connections are hidden using existential quantification. Verification proceeds by showing that a structure correctly implements a specification. This involves expanding the structure by the behavioral description of its parts, and deriving a description of behavior which implies the specified behavior. The HOL system mechanizes some aspects of the verification of circuits.

A major difference between my approach and the HOL approach is the way time is modeled. In temporal logic, time is implicit in the logical operators. In higher-order logic, time is usually represented by an explicit time variable. For example, the behavior of an inverter with input *In* and output *Out* and delay *m* is expressed in HOL by:

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

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

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

Note that in HOL, the signals *In* and *Out* are functions from time to values.

Herbert [Her86] uses HOL to verify the timing and function of digital circuits. Primitive components are gates with propagation delay such as the inverter described in HOL above. The proofs proceed by verifying the function and timing of a circuit using these timing level models, and then abstracting to a synchronous level model which expresses function only, subject to constraints on the timing behavior of the signals. The synchronous behavior of the inverter in HOL is:

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

The constraints are that *In* and *Out* are stable around *t*. Herbert uses this approach to derive the timing constraints on the inputs and outputs of a d-type flipflop built up from nor gates. Since the primitive components of his proofs are logic gates, Herbert doesn't consider the special requirements of verifying the timing of transistor level descriptions of circuits.

Dhingra [Dhi87] uses HOL to formally describe and verify the rules for combining components with the CLIC design style, a successor of the NORA design style [GDM83]. This work complements that done in my derivations since the components of the adder presented in Section 7.2 fit into the framework of the CLIC design style. Dhingra formally proves that well-defined inputs will produce well-defined outputs provided that components are connected according to the rules of the design style. He does not however model low level timing details of circuits. For example, his transistor models do not have delay. He introduces time into his descriptions by describing two-phase non-overlapping clocks where each phase has duration of unit length, and assumes the duration is long enough for correct circuit operation. This is a coarser level of detail than that which I use.

VERITAS

Hanna and Daeche also use higher-order logic to specify and reason about circuits [HD86a]. Their theory of time includes formalizations of time instants, durations and intervals, where an interval starts at a time instant and has a specified duration. As in HOL, time is explicitly part of the behavioral specification, and signals are functions from time to values. In [HD86b], a d-type flipflop similar to that presented by Herbert is verified, and similar timing constraints are formally derived.

2.2.3 Temporal Logic

Temporal logic has been used for specifying software, hardware and communications protocols. One of the earliest papers on using temporal logic for hardware [Boc82] describes specifying and manually verifying an implementation of a self-timed arbiter. In this example, temporal logic is used to specify the sequence rather than the relative timing of events. In the following, I discuss examples of temporal logic applied to circuit design in cases where the relative timing of events is modeled or where the verification technique has been automated.

ITL and Tempura

Moszkowski uses ITL to specify and reason about digital circuits [Mos83]. Since my work was motivated by his, the circuit descriptions are similar. A major difference is that the primitive components he uses to describe the behavior of larger components are delayless combinational elements and memory elements, while mine are MOS transistors with delay. Moszkowski presents a definition of pass transistor behavior, but does not model its detailed timing behavior and does not show how this description can be used in larger MOS circuits. Also, Moszkowski does not reason about different signal strengths in ITL. Another difference is the way constraints are handled. Moszkowski includes such constraints as set-up and hold times as part of the behavioral definition. His descriptions generally have one clock signal, so he does not relate the behavior of different clocks.

Tempura is an executable programming language based on a subset of ITL [Mos86]. The Tempura interpreter simulates this subset by finding values for the Tempura variables which result in the temporal logic formulas being valid. Moszkowski shows how this approach can be used to simulate ITL descriptions of circuits. Such simulation is useful for checking that the specification does in fact describe the desired behavior of the device. Modifying Tempura to handle my extensions to ITL, described in Section 4.4, would provide a useful tool for simulating my ITL specifications.

CTL and SML

A propositional temporal logic called CTL (Computation Tree Logic) has been used to specify the behavior of asynchronous and sequential circuits [BCDM86], [DC86]. The behavior of a circuit is specified in CTL and compared with the expected behavior of a gate level representation of the circuit. EMC (Extended Model Checker) checks the CTL specification against a finite state graph which represents the behavior of the gate level description of the circuit. In the case of asynchronous circuits, this finite state graph is generated by a preprocessor which combines flow-table models of the behavior of gates to form the state graph of the circuit being verified. No quantitative timing parameters are used; gates are assumed to have arbitrary delay. In the case of sequential circuits, there are two techniques for generating state graphs from the circuit. The first is to simulate the circuit using a mixed gate and switch-level simulator similar to MOSSIM [Bry81]. The simulator uses a unit delay model for all elements. The second is to compile the state graph from a hardware description language representation of the circuit. The language SML (State Machine Language) was developed for this purpose. These SML descriptions can also be used to generate different regular hardware structures such as PLAs, PALs or ROMs.

The main advantage of this approach is that it is mechanized. A major disadvantage is that the finite state graph which is generated is not hierarchical so its size grows rapidly as the complexity of the circuit grows. In addition, temporal logic is used only to specify the sequence of events. Thus the delay models used do not realistically capture the timing behavior of the circuits modeled.

LTTL and Tokio

LTTL (Linear Time Temporal Logic) is used to specify hardware at the register-transfer level and above [FTM83]. As with CTL, the examples presented use temporal logic to reason about the functional and sequential behavior rather than the low level timing behavior of circuits. Temporal logic assertions are made about a circuit, and these assertions are checked against a Prolog description of the operation of the circuit. This verification proceeds by expanding both the temporal logic and the Prolog description into behaviors of the current state and the next state; and then comparing these descriptions. Verification is done automatically. In addition, the logic programming language Tokio [FKTM86] executes LTTL specifications. The result is a simulation of a specification similar to that provided by Tempura. However, the underlying model of execution for Tokio is logic programming. Advantages and disadvantages of this work are similar to those for CTL. This approach has the additional benefits of exploiting hierarchy in circuit design and providing a tool to simulate the temporal logic specification. Disadvantages include the fact that low level timing considerations are not modeled.

ETL

Fusaoka et al. [FST84] use ETL (Extended Temporal Logic) to describe and reason about VLSI circuits. ETL, first presented by Wolper [Wol82], extends propositional temporal logic with operators for regular expressions. VLSI circuits are modeled down to the transistor level. The transistor model is unidirectional and specifies how the drain voltage behaves as a function of the source voltage when the gate signal rises or falls. Low level timing details modeled include the different delays associated with a rising pulse and a falling pulse being propagated through an inverter. A temporal logic description of the behavior of a circuit is derived by composing temporal logic descriptions of the behaviors of the components. The behavior of a circuit is validated by verifying that the specification of the input signals *and*ed with the temporal logic description of the circuit implies a specification of the output signals. Note that the user must specify the inputs and outputs, so the result resembles simulation more than formal verification.

As an example, a dynamic latch circuit similar to that discussed in Section 7.1.3 is presented [FST84]. Their circuit uses only one clock and charge storage is not modeled. The latch is modeled as having unit delay, and low level timing details of the components are suppressed. In addition, the specification of the input signals is not related to the state of the clock signal, even though the specified behavior is only true if a clock signal begins at the correct time. Some timing information about the output is derived in this example, but it is not very detailed and is also not related to the behavior of the clock.

2.2.4 Comparing Temporal Logic and Higher-Order Logic

Temporal logic and higher-order logic are formalisms which have been used to specify and verify hardware. Each formalism has certain advantages and disadvantages. In temporal logic there are no explicit time variables; time is implicitly part of the temporal operators. As a result, the specifications of behaviors are more succinct. In addition, there is no need to express such information as an order over time variables. In higher-order logic descriptions, time variables are explicit and signals are functions from time to values. Behavior specifications are frequently more complex. The advantage of higher-order logic is that it is not necessary to have special proof rules to deal with the temporal logic operators, so the proof system may be less complex.

Temporal logics can be embedded in higher-order logic. Gordon and Hale [GH87] have shown how ITL can be embedded in HOL. There are several advantages in doing this, including the ability to mix a temporal logic description of behavior with explicit time representations. In [Sub86a], timing behaviors are described by such a mixture of temporal and higher-order logic formalisms.

2.2.5 Other Formalisms for Verification

I discuss a few other systems for formally reasoning about systems. Specifically, I discuss systems which model low level characteristics of hardware. Such low level considerations include timing of combinational elements, geometric layout, and transistor level modeling.

Circal

Milne uses Circal to specify the structure and behavior of circuits [Mil86a]. Structure can be specified hierarchically. A component is specified by a name and a set of ports. Circal supports parallel composition of components and hiding of ports. When components are composed, similarly labeled ports are joined. Behavior is described by sequences of events on ports. The behavior of each component is specified as a finite state machine. Time may either be expressed implicitly or explicitly. If time is explicit, then the time variable must have a port at each device. In this case, input and output events may only occur when a time tick or time event occurs. Different grains of time may be modeled and these can be formally related. Specifications of behavior in Circal tend to be lengthy. For example, the behavior of a wire with an explicit representation of time and a one unit time delay requires four lines of description. Verification of device behavior proceeds by specifying the behavior of a device at an abstract level, specifying the behavior of the components of the device at a lower level, and proving that the two are equivalent.

μFP

μFP [She86], [She83] is a VLSI design language based on the functional programming language FP which captures both behavioral and geometric information about a circuit. The circuits described are regular array circuits whose behavior is either combinational or can be defined by a finite state machine. The μ operator introduces memory into the functional descriptions of circuits. Circuit behavior is described by functions from streams of inputs to streams of outputs. Circuits are manipulated hierarchically. A behavioral description of a circuit is transformed by applying algebraic laws. The resulting layout is therefore correct by construction. μFP successfully combines reasoning about the function and low level implementation details. However, the class of circuits which can be modeled is restricted since only synchronous circuits can be described. In addition, the model of time used is discrete, and all clock signals must be related to the same underlying clock. Thus, low level timing considerations are not expressed.

A Compositional Model of MOS Circuits

Winskel describes a compositional model for MOS circuits [Win87a]. This formal model is based on Bryant's lattice of voltage values which have different resistive and capacitive strengths. Circuit behaviors are modeled as static configurations where a static configuration is the set of steady states a circuit can adopt. A static configuration is characterized by voltage values and strengths, internal voltage sources and signal flow information. These static configurations place constraints on the way ports can interact with the environment. Two circuits can be composed if the ports which are connected in the process impose consistent constraints on the environment. In [Win87b], Winskel shows how his model can be formally related to Gordon's switch-level model which does not take into account resistive and capacitive signal strengths.

Winskel does not deal with time explicitly. Time may be introduced by viewing it as a sequence of static configurations where each static configuration has an associated time variable. The assumption is made that all input signal durations and clock periods are sufficiently long for the circuit to settle into a steady state before the environment changes.

Silica Pithecus

Silica Pithecus is a system for verifying the digital behavior of synchronous systems [Wei86]. Weise is mainly concerned with verifying low level aspects of the behavior of nMOS circuits. Verification proceeds by calculating the analog behavior of a circuit implementation, abstracting that behavior to the digital level, and comparing the result to the specification provided by the user. The system uses a bidirectional transistor model, and models resistive and capacitive strengths. Signals are functions from time to voltages and strengths. The system can detect charge sharing bugs, ratio bugs, threshold drops, and races and hazards. It does not handle capacitive coupling and general feedback in circuits. Timing verification is also not handled. The assumption is made that a circuit settles into a steady state before the inputs change. This system reasons about constraints on the behavior of signals in a manner similar to that which I use.

2.3 Conclusions

In this chapter, I have discussed informal and formal methods for reasoning about circuits. By using formal methods to validate a hardware design, several advantages are gained over informal methods. These advantages include the ability to control the complexity of verification through hierarchical analysis, and the ability to formally relate the different levels of description. With experience and further research, formal methods may also gain the advantages currently exhibited by informal methods: maturity and widespread acceptance.

My research differs from previous work in the level of hardware designs verified. I specify and verify function and timing of circuits built from MOS transistors modeled at a detailed level. This research complements much of the work described above.

In the rest of this thesis I describe how Prolog and ITL can be used to specify and formally verify the timing and functional behavior of CMOS circuits. In the next chapter, I describe how to specify circuits using Prolog, and how the specifications can be used to derive characteristics of these circuits. I then go on to describe how these specifications are used in formally reasoning about circuit behavior.

Prolog for Circuit Specification and Manipulation

Circuits can be specified, simulated and reasoned about using logic programming. In this chapter I describe the use of the logic programming language Prolog to represent the structure of circuits. These Prolog descriptions directly reflect the structure and hierarchy of a circuit as shown in a circuit schematic. Furthermore, important characteristics of circuits can be automatically derived from these specifications.

First I define some terminology and give a brief review of Prolog. Next I describe the specification method and present an example. I then describe tools I have written for manipulating these circuit descriptions. These tools include programs for automatically deriving the direction of signal flow through transistor networks and deriving port directions from the schematic. Finally I discuss related research into Prolog tools for hardware design.

3.1 Terminology

3.1.1 Specifying Circuits

A *circuit* is composed of a set of *components.* Components can be composed hierarchically, where components are specified in terms of constituent components. At the bottom of the hierarchy are *primitive* components. I describe CMOS circuits; the primitive components are *n*-type and *p*-type transistors and power and ground sources. There are no strict rules about the levels of hierarchy. One component may be made up of primitive as well as non-primitive components.

Each component has *ports* for external connections. A port may be an *input*, an *output*, or *bidirectional*. A *node* is a junction of ports. Nodes which are *external* to a component are formed by connecting one of the ports of that component to one or more ports of other components. Nodes which are *internal* to a component are formed by connections of the ports of the constituents of that component.

3.1.2 A Brief Review of Prolog

Logic programming is based on the Predicate Calculus. An introduction to Prolog (PROgramming in LOGic), a widely used logic programming language, is given in [CM84]. In this section I present some key concepts and terminology, especially those used to specify and manipulate circuits. I use Edinburgh Prolog syntax.

In Prolog, terms are constant symbols, variables or compound terms. *Constant symbols* are either integers, or written with an initial lower case letter, or a string of non-alphanumeric symbols. *Variables* are written with an initial upper case letter. An n-ary *compound term* is written as $f(t_1, ..., t_n)$ where f is a constant symbol called the *functor*, and each t_i $(1 \leq i \leq n)$ is a term called an *argument*.

A *list* is another commonly used compound term with special Prolog syntax. A list of length 0 is written []. A list of length n is written $[t_1, \ldots, t_n]$. t_1 is called the head of the list, and $t_2, \ldots, t_n$ are elements of a list called the tail. The list having head h and tail l can also be written $[h|l]$.

Formulas are written as Horn clauses, a subset of the formulas of Predicate Calculus. The Horn clause:

$$P :- Q_1, Q_2, \ldots, Q_q$$

is equivalent to the Predicate Calculus formula:

$$\forall x_1, \ldots, x_m \, (\exists y_1, \ldots, y_n \, Q_1 \wedge Q_2 \wedge \ldots \wedge Q_q) \quad \supset \quad P$$

where P and the Q_i $(1 \leq i \leq q)$ stand for compound terms, the x_j $(1 \leq j \leq m)$ are variables appearing in P and possibly in the Q_i, and the y_k $(1 \leq k \leq n)$ stand for variables appearing in the Q_i and not in P. P is the head of the Horn clause, the Q_i are the body. A clause with no body is called a unit clause.

Clauses are used as a program to which questions (or goals) can be posed. The strategy used by Prolog for the execution of goals combines a simple backtracking strategy together with a pattern matching algorithm known as unification. Two terms are matched according to the unification algorithm if there is a most general substitution for the variables in the terms such that the terms may be made equal. In logic programming, unification is a general purpose feature used for passing input and output parameters and for incremental construction of data structures. When using Prolog as a hardware description language, unification is used for propagating signal values through a circuit.

3.2 Specifying Circuits in Prolog

Clocksin [Clo87] presents and compares three different methods of specifying circuits in Prolog. These are the functional method, the extensional method and the definitional method. I give brief descriptions of the first two methods, and describe in more detail the method which I employ: the definitional method.

The Functional Method

In this method, a functional representation of circuits is used. An output of a component is described by a function applied to arguments which are the inputs to the component. These arguments may in turn be functions. For example, the component halfadd shown in Figure 3.1 is described as:

xor(A, B).
not(nand(A, B)).

This method has two disadvantages. First, only acyclic circuits can be specified. This excludes many practical circuits. Second, a separate expression must be used to represent each output of a circuit. In addition, functional specifications quickly become very complex, even for simple circuits.

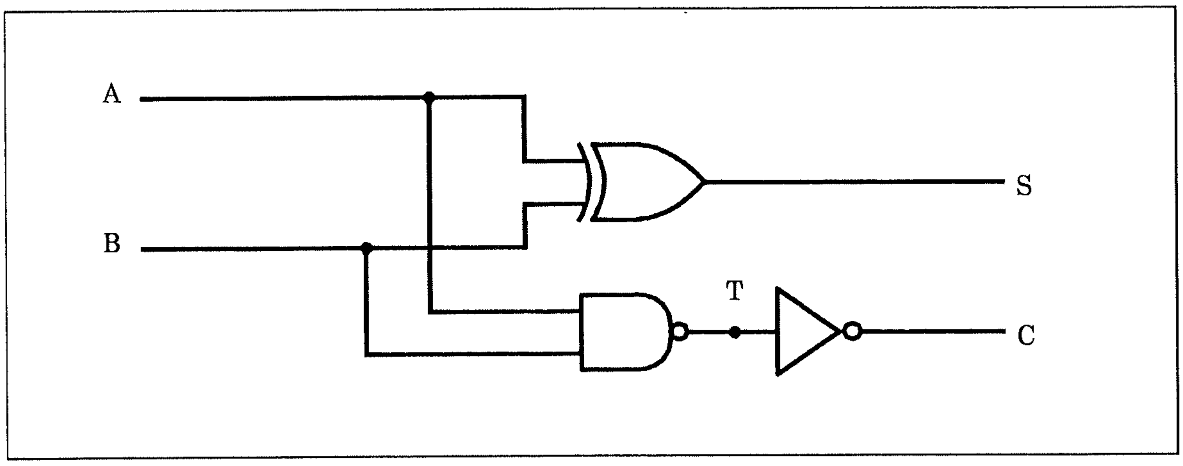


Figure 3.1: The Component halfadd

The Extensional Method

The extensional method represents each component and connection with a separate clause. Prolog constants are used to name connections. For example, the halfadd in Figure 3.1 component can be described using the relations **component** and **connect**. The arguments of **component** are the name of the component, a list of inputs and a list of outputs. The binary predicate **connect** describes connections between ports:

```
component(xor,[a, b], [c]).
component(nand,[a, b], [c]).
component(not,[a, b]).

connect(a,xor(a)).
connect(b,xor(b)).
connect(a,nand(a)).
connect(b,nand(b)).
connect(nand(c),not(a)).
connect(xor(c), s).
connect(not(b), c).
```

This method does not have the same drawbacks as the functional method. However, disadvantages arise since modules are not represented as a single term. It is therefore difficult to do certain kinds of circuit manipulations. In addition, hiding internal lines is cumbersome so expressing modularity is not straightforward.

The Definitional Method

In this method, a circuit is represented as a set of Horn clauses. A component with n ports is represented as a predicate of arity n whose head represents the component being defined. The body of the predicate is a composition of the constituent components which define the component. Constituents are composed with the comma connective. The order of the components in the body is not important. The ': –' connective of Prolog is reinterpreted to mean 'is defined by'. A node is represented by a unique, like-named variable. A node which is named by a variable not appearing in the head of the clause is an internal node.

The component halfadd depicted in Figure 3.1 is specified as:

$$\mathsf{halfadd}(A,B,S,C) :- \mathsf{xor}(A,B,S).\mathsf{nand}(A,B,T).\mathsf{invert}(T,C).$$

The variable T defines the 'hidden' node between the output of the nand gate and the input of the inverter. Figure 3.2 shows the definition of the nand gate in terms of primitive elements. A transistor is specified as $\mathsf{trans}(X,G,A,B)$ where X specifies the type of transistor (either n or p); G is the gate and A and B are the channel nodes (source and drain). Component nand is specified as:

$$\begin{array}{l}\mathsf{nand}(A,B,C) :- \\ \qquad \mathsf{pwr}(P).\mathsf{trans}(p,B,P,C).\mathsf{trans}(p,A,P,C). \\ \qquad \mathsf{trans}(n,A,C,T).\mathsf{trans}(n,B,T,G).\mathsf{gnd}(G).\end{array}$$

Note that the ports of these components do not have directions specified.

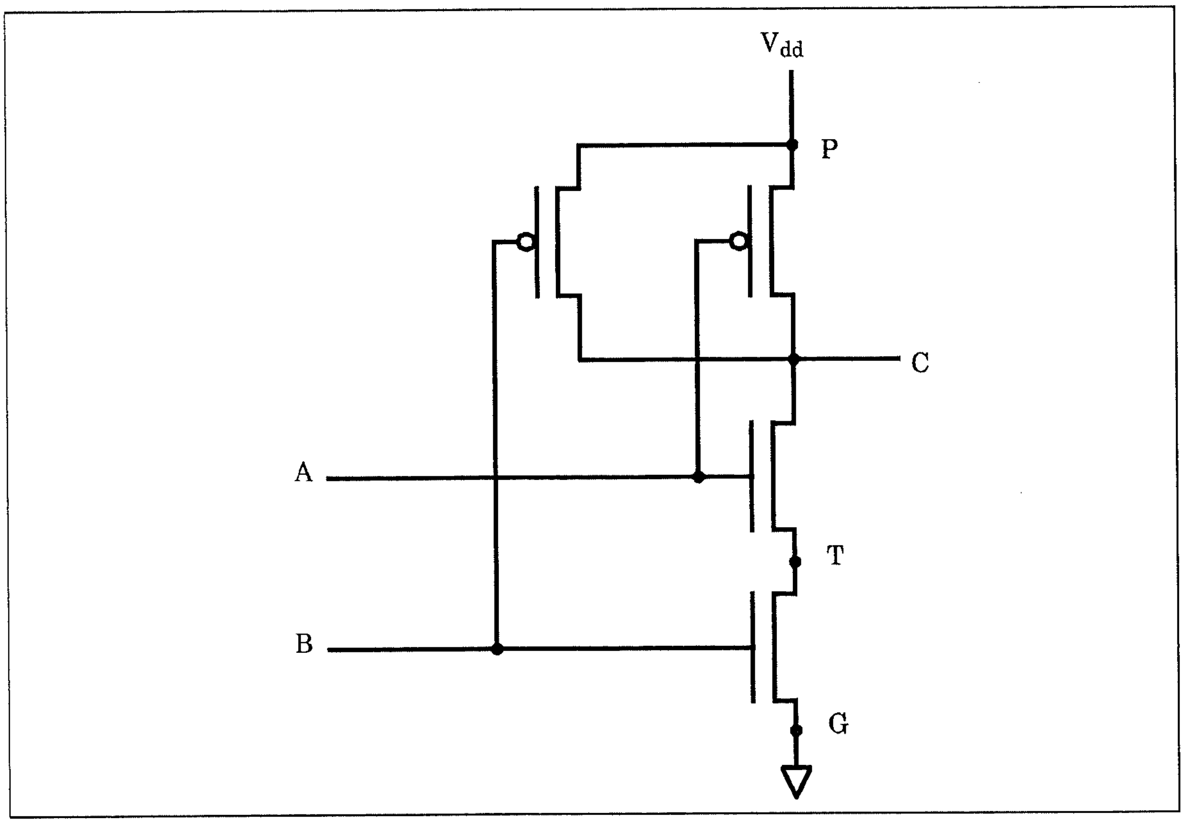


Figure 3.2: The Component nand

Advantages of the Definitional Method

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

Another advantage is that the specifications can be manipulated or directly executed by Prolog systems. Clocksin [Clo87] describes how Prolog specifications of circuits can be directly executed. In the next section I describe other tools for manipulating these specifications.

3.3 Tools for Manipulating Prolog Circuit Specifications

I have written several Prolog procedures for manipulating descriptions of MOS circuits. These tools analyze the circuits hierarchically. Examples of such tools include code to automatically determine the direction of signal flow and the direction of ports of components in hierarchically specified MOS circuits. In the next subsection I describe this code. Then, I describe work others have done in developing Prolog tools for circuit design.

Other procedures I have written identify all the components directly connected to a given node in a circuit and identify all the primitive components directly connected to a given node. I have also written routines to extract circuit components from a network of circuit elements. These tools are used with the techniques described in this thesis; their implementation is not described further.

3.3.1 Automatic Determination of Signal Flow through MOS Transistors Networks

Determining the signal flow through transistors is often a necessary precursor for further circuit analysis. For example, both Crystal [Ous83] and TV [Jou83] require flow analysis before they can proceed with timing analysis. Many formal methods for circuit verification [Bar84] and design simplification [Clo87] depend on signal flow being specified in advance. Some design verification methods do not require preliminary flow analysis, but proofs are easier to compute if the results of flow analysis are available [Gor85a]. One way to provide signal flow information is to manually specify the input and output roles of components' ports. Some systems [Bar84],[Ous83] rely on manual specifications entirely. However, automatic signal flow determination can reduce the designer's workload, particularly for the case of MOS transistors, which, due to the symmetry of source and drain nodes, are capable of conducting in different directions at different times. TV [Jou83] automatically determines signal flow in nMOS transistor networks free of bidirectional transistors and specification errors. TV uses nine rules to determine the direction of transistors. In addition to the general Kirchoff current law, TV uses information about the types of circuits used in the design methodology.

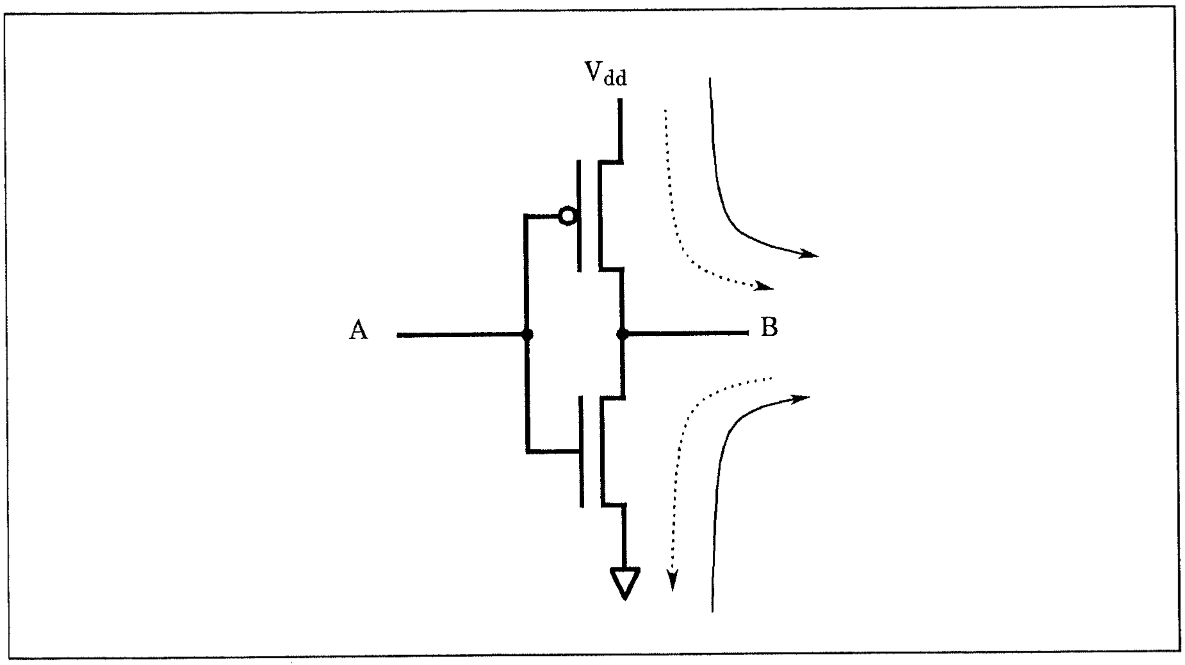


Figure 3.3: A CMOS inverter

The method described in this section was originally presented in [CL86]. It is intended for hierarchically specified MOS transistor networks that contain bidirectional transistors, and where there is no information about any particular design methodology. The method is data independent; it does not use knowledge of the values of inputs to a circuit. The analysis is static; it is only dependent on the circuit topology. In essence, we are determining the flow of information through a network assuming all transistors are turned on. Although it is unlikely that this configuration will occur in practice, we are able to determine those transistors whose direction will always remain the same. Transistors which can support signal flow in either direction are labeled bidirectional. Clearly we label some transistors bidirectional which, due to the inputs provided, will only propagate signals in one direction in practice. However, the algorithm is guaranteed to correctly find all bidirectional transistors, and will not incorrectly label any unidirectional transistor. This procedure will be able to determine the direction of a large percentage of transistors in most circuits. It will not provide much useful information about a design that relies heavily on bidirectional transistors; however, such designs are unusual.

Signal flow may be viewed as the propagation of logic levels, either *high* or *low*, through a network. Power and ground connections are viewed as sources of signal flow. Inputs to a component and outputs from a component are sources and sinks of signal flow, respectively. Note the difference between signal flow and the more traditional view of current flow where power acts as a source of current flow and ground acts as a sink. For example, consider the CMOS inverter in Fig 3.3. The dotted arrows show the direction of current flow through the transistors; the solid arrows show the direction of signal flow. Using the definition of signal flow we can derive the input node and output node of the inverter. This will be shown later.

The Method

The input to the direction finder is a circuit specification. Primitive components are *p*-type and *n*-type transistors and power and ground supplies. Ports of components may be sinks or sources of signal flow or both. The task of determining the direction of signal flow can be viewed as a consistent labeling problem [Mak77]. To solve the problem, it is necessary to associate with each transistor a label drawn from the set {*right, left*}. Refer to Figure 3.4. If the two non-gate terminals of a transistor are arbitrarily named A and B, the label *right* is assigned to a transistor for which terminal A is a sink of signal flow and terminal B is a source of signal flow. The label *left* is assigned to a transistor for which terminal A is a source of signal flow and terminal B is a sink of signal flow. For a circuit consisting of m transistors, the number of unconstrained labelings is 2^m. The problem is to assign labels to transistors such that a constraint is met: each node of a circuit must be connected to at least one sink and at least one source of signal flow. Moreover, if a node is directly connected to power or ground, then at most one source of signal flow (the power or ground itself) is allowed. This constraint is called *the Signal Law.* The Signal Law is a consequence of Kirchoff's current law. For a circuit containing bidirectional transistors, multiple solutions to the consistent labeling problem are admitted. In other words, there may be more than one possible labeling of a circuit that satisfies the Signal Law. A bidirectional transistor will be labeled *right* in one solution and *left* in another.

The direction of ports (whether they are used as inputs or outputs) is an additional constraint on the solution. With this method there is a choice of whether to attempt to derive the direction of ports automatically (at the risk of obtaining a weak solution) or to use a specified direction of ports to constrain the search for labelings of transistors.

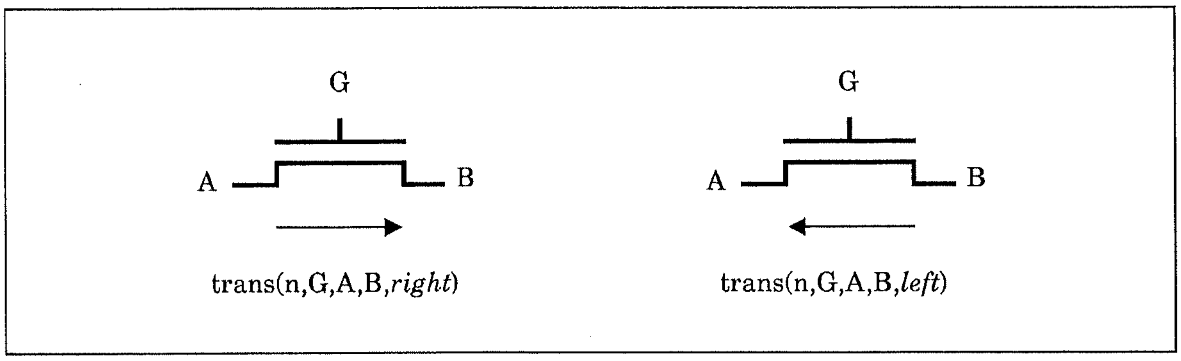


Figure 3.4: Transistor Labelings

Implementation

The method is implemented in Prolog, using depth-first search through the hierarchical circuit specification to exhaustively enumerate the directions of transistors. The method compounds the direction finding of transistors with establishing the direction of ports of non-primitive components. Unlike previous approaches [Jou83], the whole circuit is not flattened into a set of primitive components. First, the outermost component is decomposed into its constituents, then the direction of the ports of each of these components is established, and finally it is confirmed that the hidden nodes of the circuits obey the Signal Law. If a component is not primitive then the direction of its ports is established recursively.

The direction of ports of primitive components, in this case *n*-type and *p*-type transistors, are determined as follows. Transistors have three ports: gate, source, and drain. A transistor's gate terminal is always a sink of signal flow. If a transistor, specified as $\mathsf{trans}(n, A, B, C)$, is assigned the label *left* then B is its source and C is its drain. Similarly, if it is assigned the label *right* then B is its drain and C is its source. A transistor source is a source of signal flow; its drain is a sink. Either the label *left* or the label *right* is assigned to a transistor. This is a nondeterministic choice which may be reversed subsequently if it is found that the original choice leads to a violation of the Signal Law. Power and ground connections are always sources of signal flow.

At any level of the hierarchy we can identify the set C of components that make up the component at the next higher level of the hierarchy. Labels taken from the set $\{in, out\}$

are associated with each port of each component in C. The label *in* corresponds to the associated port acting as a source of signal flow to a node outside the component; the label *out* corresponds to the port acting as a sink of signal flow. The labels of ports connected to a given node are collected into a list associated with the node. A hidden node obeys the Signal Law if (a) its list contains at least one *in* and one *out*, and (b) if the node directly connects to a power or ground node, then it must contain only one *in*.

Only the hidden nodes of the enclosing component must satisfy the Signal Law; its ports are checked when the enclosing component itself becomes a component of a larger circuit. If at any level of the hierarchy a node list does not satisfy the Signal Law, Prolog will automatically backtrack to find a direction of the components which satisfies the Signal Law.

The effect of this procedure is to try both directions of all transistors until the Signal Law is satisfied. Further backtracking will find all possible solutions. Transistors whose directions are the same in all possible solutions are unidirectional, the remaining transistors are labeled bidirectional. When all solutions are found, each port of each component has an associated list of *in*s and *out*s. If all elements on the list are *in*s, then the port is an output; if all elements on the list are *out*s then it is an input. Otherwise, it is bidirectional. In addition, the number of elements on the list is the number of primitive components connected to that port. This information may be useful in subsequent analysis of the circuit for determining the capacitive loading of a node.

The hierarchy of the circuit specification can be exploited further. Once the directions of ports have been found for a type of component, the result can be stored in a library. This library would be consulted for each component when the procedure is invoked. Thus, direction setting needs to be determined only once for each type of component. In this context, trying both possible directions for all transistors in a circuit becomes reasonable for circuits with very large numbers of transistors. For example, suppose we wish to include a CMOS inverter in the library. Running the procedure on the inverter circuit once will set the direction of both transistors and establish the input and output. This result will then be stored, and each additional time an inverter is encountered, the input and output nodes are looked up in the library. Instead of four possible labelings for each inverter, only one is considered. The benefits become greater as the library and the complexity of the components in that library grow.

Examples

I will now show the result of applying this procedure to a few simple examples. These include a CMOS inverter, a static register which illustrates constituent components defined at different levels of hierarchy and the utility of a library, and a full adder which has bidirectional transistors.

Example 1. A CMOS inverter (Figure 3.3). This circuit is specified as:

$$\mathsf{invert}(A,B) :- \mathsf{pwr}(P),\ \mathsf{trans}(p,A,P,B),\ \mathsf{gnd}(G), \mathsf{trans}(n,A,B,G).$$

Since any connection to power or ground is an input, it is clear that the n-type transistor would be labeled *right*, and the p-type transistor would be labeled *left*. This corresponds to the arrows for signal flow shown in Figure 3.3. The resulting node list for A is $[A, out, out]$, telling us that A is the input to the component. The resulting node list for B is $[B, in, in]$ so B is the component's output. Backtracking would not find further solutions.

Example 2. A static register (Figure 3.5). This circuit is specified as:

```
sreg(A, B, Load) :-
        trans(n, Load, A, X),invert(X, Y),invert(Y, B),
        trans(n, Z, B, X),invert(Load, Z).
```

This simple component illustrates several key aspects of this approach. Internal connections such as nodes X, Y, and Z are hidden through use of Prolog variables which do not appear in the head of the clause. In addition, a circuit may be made up of constituent components described at different levels of hierarchy; here we mix primitive and non-primitive components. This example also illustrates the utility of a library of components. If the directions of the ports of the inverter are stored in a library, then only the direction of the two pass transistors need be considered. Thus only four possible label assignments are investigated.

From the method presented so far, it is impossible to ascertain a unique direction for the two pass transistors. There are three ways to deal with this problem. One is to leave the

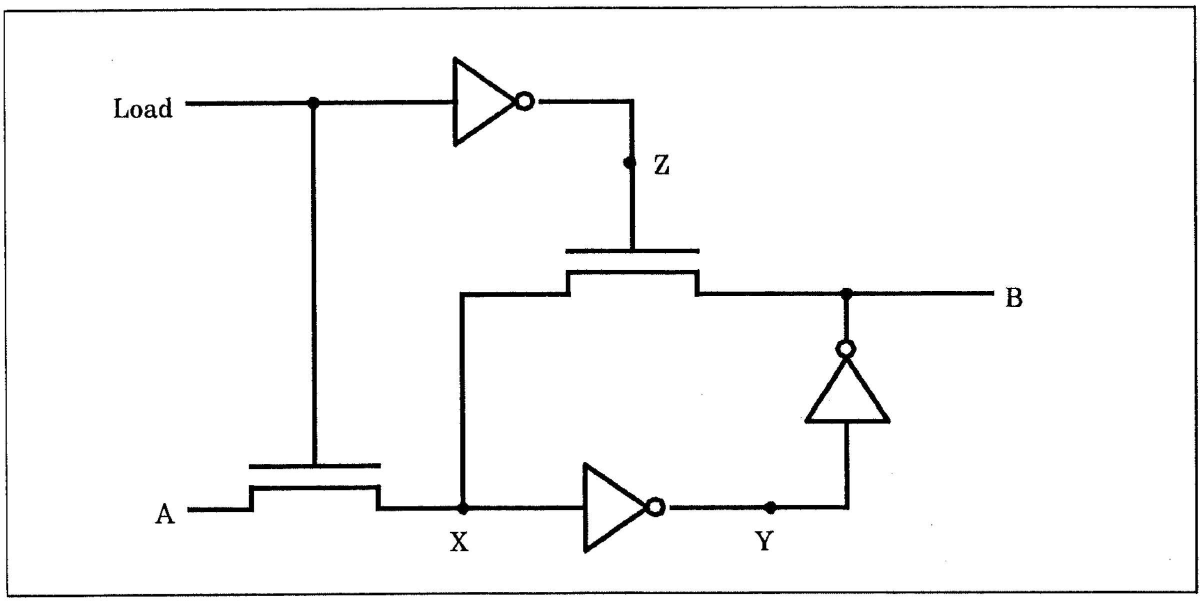


Figure 3.5: A Static Register

direction specified as bidirectional until the register is used in an enclosing component. Another is to use specifications of the directions of ports. For example, in the static register, the specification that A is an input leads to the solution of the direction of the transistor directly connected to it. However, the specification that B is an output does not imply a single direction for the other transistor. Another approach is to incorporate additional rules which recognize commonly occurring structures. I use a combination of these approaches. For the static register, both the direction of ports are specified by the user and additional rules are used.

Two additional rules are used by the direction finder. One recognizes that two transistors with gates which are the complements of one another and which each have a channel connected to the same node have the same direction with respect to that node. This rule explicitly recognizes a 2-to-1 multiplexer implemented with transistors, a configuration frequently found in designs. The second rule recognizes that both transistors in a transmission gate have the same direction. Note that the register is not implemented in fully complementary MOS. This can be rectified by replacing the pass transistors by transmission gates consisting of p-type and n-type transistors with the complementary gate and the same channel connections. The transmission gate rule ensures that such a circuit is analyzed the same way as that shown in Figure 3.5.

Exploiting hierarchy is still advantageous for a component which has several possible labelings of its ports. All of the different labelings are stored in the library when the circuit is analyzed, and the labeling which satisfies the constraints of the environment in which the component is used becomes the labeling for that component. Note that two instances of the same component in a circuit may have different labelings. In any case, the direction finder is applied to this component only once.

Example 3. A full adder. Figure 3.6 shows a full adder circuit built out of two components, a sumpart and a carrypart. Each of these components is itself built out of transistors.

The top level specification of the adder is:

```
adder (A, B, C, Sum, Carry) : -
        sumpart(A, B, C, NCA, Sum),
        carrypart(A, B, C, NCA, Carry).
```

The procedure is applied to this circuit. The sumpart consists of twelve transistors, six of whose direction cannot be established and are thus labeled as bidirectional. All of the transistors in the carrypart are unidirectional. The procedure also establishes that A, B, and C are always input ports and Sum and $Carry$ are always output ports despite the different possible internal configurations.

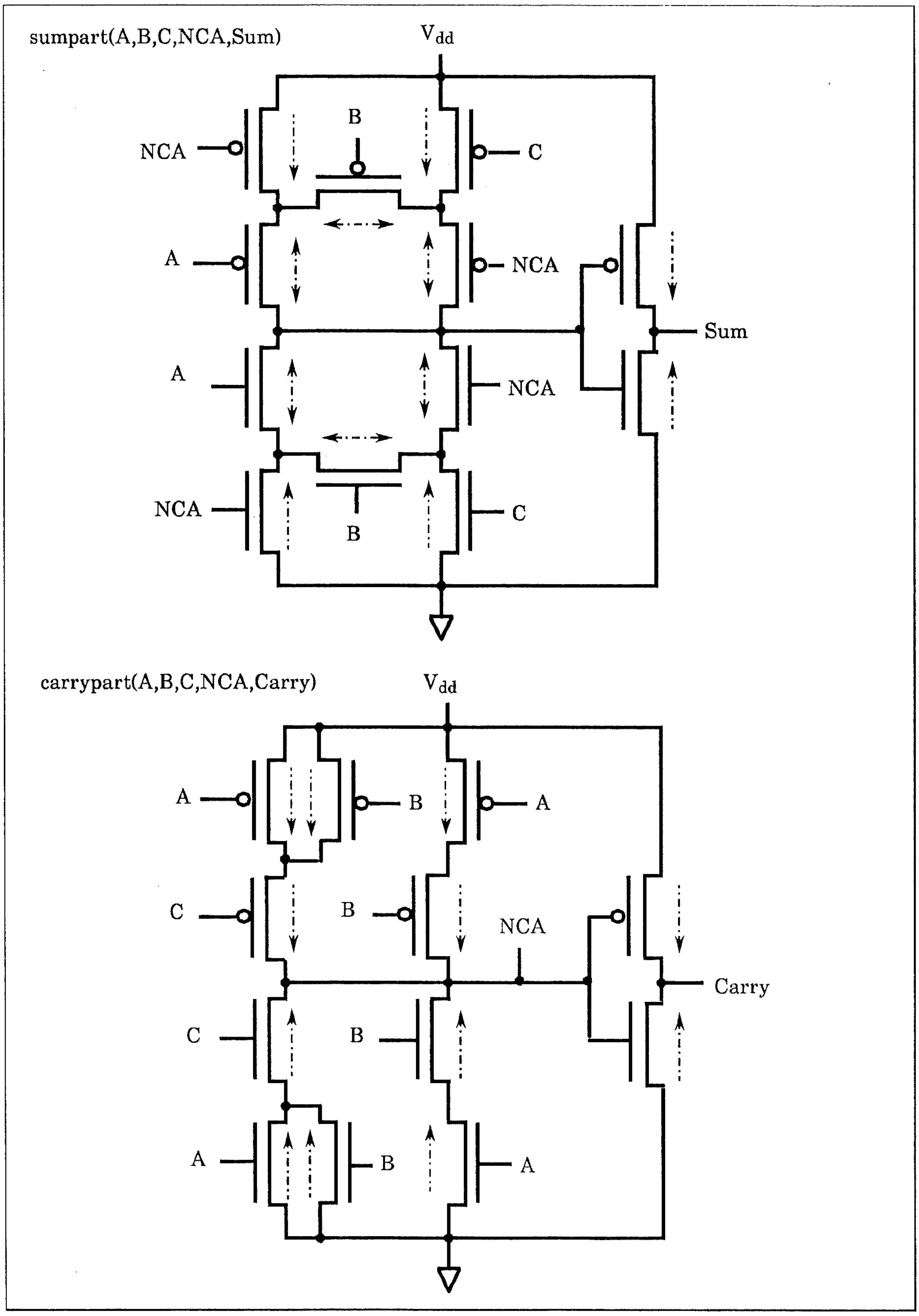


Figure 3.6: A Full Adder

Discussion

Signal flow detection is able to recognize certain classes of specification errors. Circuit specifications that do not admit any labeling of transistor directions must be suspect as incorrectly specified. Such specifications contain at least one internal node which contains either all sinks or all sources of signal flow, an impossible situation in practice.

Techniques exploiting combinatorial enumeration are always suspect as being inefficient. I have already mentioned the use of hierarchical specification of circuits to reduce the number of configurations to be examined. The number of different kinds of components in the circuit becomes the important factor instead of the total number of transistors. The following experiment illustrates the improvement in efficiency realized by hierarchical analysis. This is the case even when an improved backtracking algorithm is used.

A version of the signal flow analyzer was implemented which uses a dependency directed backtracking algorithm developed by M. P. Shanahan [Sha87]. This algorithm was compared with Prolog's built-in backtracking algorithm by running each on two versions of the full adder circuit shown in Figure 3.6. One version of the adder was flattened into primitive components; the other version was described hierarchically. The results of these experiments is shown in Table 3.1. The numbers were obtained using the *statistics* command available in C-Prolog run on a DEC Microvax II.

These experiments show that the dependency directed algorithm was much faster than Prolog's built-in backtracking algorithm at finding an assignment of transistor directions if a circuit was first flattened into its primitive components. The dependency directed backtracker was slightly slower in finding the first solution than the built-in backtracker for circuits which were analyzed hierarchically.

The results for the circuit analyzed hierarchically are much faster than for the flattened circuit, irrespective of the backtracking algorithm used. This illustrates the efficacy of hierarchical analysis.

1) FLAT CIRCUIT ANALYSIS

a) Built-in backtracker:	
one solution:	151.50 seconds
all solutions:	748.53 seconds
b) Dependency directed backtracker:	
one solution:	22.07 seconds
all solutions:	323.53 seconds

2) HIERARCHICAL CIRCUIT ANALYSIS

a) Built-in backtracker:	
one solution:	4.53 seconds
all solutions:	41.00 seconds
b) Dependency directed backtracker:	
one solution:	10.81 seconds
all solutions:	22.93 seconds

Table 3.1: Performance Results of Signal Flow with Different Backtracking Algorithms

3.3.2 Related Work: Prolog for Circuit Design

Many tools have been written which employ logic programming for computer aided design of electronic circuits. In this section I describe such tools which do not use formal methods. Work on formal methods applied to hardware design is discussed in Chapter 2. Here I discuss the application of logic programming to symbolic simulation, design transformations and automatic test pattern generation. These tools operate on gate level descriptions of circuits. I also discuss a silicon compiler project which employs transistor level descriptions.

Several researchers have implemented symbolic simulators which operate on Prolog specifications of circuits [Bat83], [Clo87], [Gul85], [Suz85]. All of these use the definitional method of specification, and all simulate circuits down to the gate level. The simulations by Gullischen [Gul85] can be run forward or backward but only operate on combinational circuits. Batten [Bat83], Clocksin [Clo87], and Suzuki [Suz85] simulate sequential circuits by manipulating streams of values. A stream of values on a wire is represented as a list. The next state and the outputs are represented as functions of the current state and the inputs. Suzuki uses a parallel implementation of Prolog to explicitly exploit the parallelism which occurs in a circuit. A global clock is explicitly modeled, and simulation is event-driven based on the value of that clock.

Clocksin also presents algorithms for gate assignment, circuit rewriting, and component specialization. All of these tools manipulate gate level descriptions of circuits. The purpose of gate assignment is to map a circuit onto a set of commercially available products, and then to estimate the cost of the resulting assembly. With circuit rewriting, a circuit is transformed into a different implementation with the same functional behavior. Specialization automatically removes redundant components from a circuit. Redundant components arise when a designer is designing with a standard design library. Specialization automatically adds cells to the library to customize that library for the current design. Another tool for design transformation of Prolog circuit specifications is presented by Shiu-Kai Chin [Chi86] in his PhD thesis. Chin shows how multiplier specifications can be automatically synthesized using equivalence transformations implemented in Prolog. The resulting synthesized circuits are gate level implementations of multipliers.

Logic programming has been used by Gupta [Gup86] and Svanæs and Aas [SA84] to automatically generate test patterns. Both of these approaches employ the definitional method to describe circuits at the gate level. Test pattern generation exploits simulation of descriptions of circuits. Faults are injected into a circuit and the result of simulation of the faulty circuit are compared to the results of simulation for the correctly implemented circuit. Gupta uses a hierarchical description of circuits where components which do not meet their functional behavior are expanded by their implementations. Horstmann [Hor83] uses Prolog in an expert system for design for testability. The system applies rules to check that certain criteria are met. If a rule fails, the circuit is transformed by adding circuitry to the original circuit. Horstmann specifies circuits with the extensional method.

All of the tools described above manipulate gate level descriptions of circuits. The Advanced Silicon Compiler in Prolog (ASP) project at the University of California at Berkeley manipulates transistor level descriptions of circuits. The aim of the ASP project is to implement a silicon compiler on a parallel machine which runs Prolog [DPS*87]. One module of ASP is the Prolog Timing Analyzer (PTA) [PD86] which calculates the delays of all nodes using a lumped RC delay model. PTA exploits a hierarchical description of the schematic. Another module, MOST, uses information from PTA and a simulated annealing algorithm to assign sizes to transistors in a VLSI schematic.

3.4 Conclusions

I have shown how Prolog can be used to specify and manipulate the structural representations of MOS circuits. I go on to describe reasoning about the behavior of circuits. In the next chapter I will introduce interval temporal logic. Subsequently I will show how temporal logic can be used to specify the behavior of circuits and how Prolog structural descriptions can be manipulated to derive the behavior of a circuit from its structure and the behavior of its components.

Chapter 4

Interval Temporal Logic

Interval temporal logic (ITL) is a development of classical linear time temporal logic due to Moszkowski [Mos83]. In this chapter I present a subset of ITL used to specify and reason about hardware.

Temporal logic is a formalism for reasoning about time which has been used for specifying and proving properties of software and hardware. Classical temporal logic adds operators for reasoning about time to the usual logical operators: $\wedge$ (and), $\vee$ (or), $\supset$ (implies), $\neg$ (not). Temporal operators are used to express properties about events in the future: $\Box$ (always), $\bigcirc$ (next), and $\Diamond$ (eventually). [1]

First I give an informal introduction to ITL. Formal definitions of some ITL operators are given in Section 4.2. Then I present more operators. Finally, I present my extensions to ITL. Moszkowski [Mos83] models circuits using boolean logic; I extend this approach by adopting a more detailed switch-level model. In this model, signals are described as pairs of booleans. The first member of the pair denotes the logical value of the signal and the second denotes the strength of the signal. I present operators for manipulating these signals. The operators presented in this chapter are summarized in Table 4.1. In the next chapter I use signals and the ITL operators presented here to specify and reason about the timing and logical behavior of circuits at the transistor level.

[1]The temporal logic operator $\Diamond$ does not appear again in this thesis. I specify necessary requirements for the behavior of a circuit; such requirements are specified using $\Box$. Fairness statements, as well as other statements which employ the $\Diamond$ operator, are not considered.

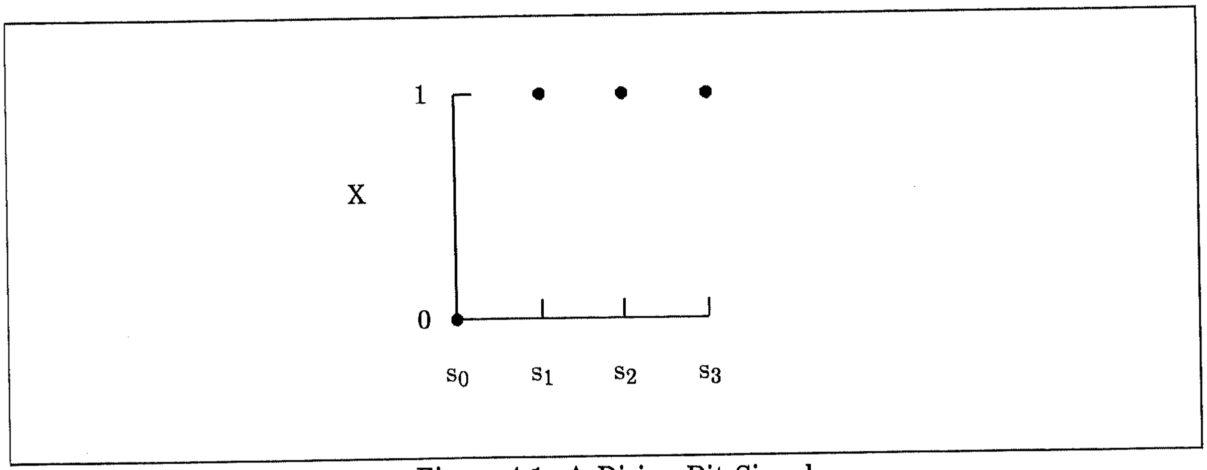


Figure 4.1: A Rising Bit Signal

4.1 An Informal Introduction to ITL

In linear time temporal logic (LTTL), time is a single, linear sequence of discrete events. ITL is a development of LTTL which allows reasoning about intervals. An ITL formula describes behavior on an interval of time. ITL operators include $\Box$ (always), $\bigcirc$ (next), and ; (chop). An intuitive description of these operators is given below. Their syntax and semantics are given in the following section.

An *interval* is a non-empty sequence of states. (The notation $\langle\rangle$ is used to delimit intervals.) A *state* can be viewed as an instantaneous snapshot of a system. The *length* of an interval is one less than the number of states it contains. An interval of length 0 is called an *empty* interval; it has one state and is just an instant of time. For example, in Figure 4.1, $\langle s_0, s_1, s_2, s_3\rangle$ and $\langle s_2\rangle$ are intervals. The state $\langle s_2\rangle$ is an *empty* interval.

Formulas are assigned truth values with respect to intervals. A formula w which contains no temporal operators is true on an interval if it is true in the first state of that interval. In Figure 4.1, the formula $X = 0$ is true on the interval $\langle s_0, s_1, s_2, s_3\rangle$.

$\Box\, w$ (always w) is true on $\langle s_0, s_1, \ldots s_n\rangle$ if w is true in every subinterval which finishes in the final state. In Figure 4.1, $\Box\, X = 1$ is true on the interval $\langle s_1, s_2, s_3\rangle$.[2]

[2]Moszkowski has 3 *always* operators in ITL, $\boxed{\scriptstyle a}$, $\boxed{\scriptstyle i}$, and $\boxed{\scriptstyle t}$. I use $\Box$ to stand for $\boxed{\scriptstyle t}$, and do not use the others. $\boxed{\scriptstyle t}$ corresponds directly to $\Box$ in LTTL.

Similarly, $\bigcirc w$ (next w) where w is a formula, is true on an interval $\langle s_0, s_1, \ldots, s_n \rangle$, if w is true on the sub-interval $\langle s_1, \ldots, s_n \rangle$. In Figure 4.1, the formula $\bigcirc X = 1$ is true on the interval $\langle s_0, s_1, s_2, s_3 \rangle$.

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

The formula ***skip*** is true of any interval of length one. It is frequently necessary to use skip in formulas employing the chop operator. For example, the signal X in Figure 4.1 rises from 0 to 1. The temporal logic formula

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

is true on the interval $\langle s_0, s_1, s_2, s_3 \rangle$. The chop operator splits an interval into two sub-intervals which share a common state. It is impossible for the signal X to be both 0 and 1 at the same time, so it is necessary to use ***skip*** to represent the interval of length one where X is changing. Here ***skip*** represents the interval $\langle s_0, s_1 \rangle$ which has length one.

Temporal operators for expressing such concepts as temporal equality and delay can be built out of formulas using the operators described above. In the next section I give formal definitions of the operators already described. Then I present more ITL operators.

4.2 Formal Definitions of Some ITL operators

4.2.1 Syntax

Legal ITL formulas are either atomic, or they are formed by combining other legal formulas or expressions. Expressions are composed of variables and function applications. In the following, upper case characters $(A, B, C, \ldots)$ are used to stand for variables, and (w, w_1, w_2) stand for well-formed formulas. Most variables represent bit signals, having value either 0 or 1.

Here are some of the operators with which formulas and expressions can be combined:

- Operators from classical logic: These include negation ($\neg w$), conjunction ($w_1 \wedge w_2$), and implication ($w_1 \supset w_2$).
- Existential and Universal quantifiers: $\exists A.w$ and $\forall A.w$ are well-formed formulas.
- Predicates: $p(e_1, \ldots, e_k)$ where $k \geq 0$ and $e_1, \ldots, e_k$ are expressions. Equality, $e_1 = e_2$, is one of the most commonly used predicates.
- Always: $\Box\, w$.
- Next: $\bigcirc\, w$.
- Chop: $w_1\, ;\, w_2$

Parentheses are used in formulas when the binding of logical operators is ambiguous or different from the default. I assume the following defaults for the binding of classical and temporal logic operators. The most binding are the unary operators $\neg$, *len*, *stb*, weaken and **strengthen**. The next most binding are the binary operators $\wedge$, $\vee$, and join. Less binding is $=$. Less binding still are the binary operators $\longrightarrow$, $\supset$, and $\approx$. The least binding are the unary operators $\Box$, $\bigcirc$, $\uparrow$, and $\downarrow$. Some of these operators have not been introduced yet. They will be presented later in this chapter.

4.2.2 Semantics

In this section, I present a more formal way to express what is meant for a formula to be true on an interval.

A Model is a pair $(\Sigma, \mathcal{M})$ consisting of a set of states $\Sigma = \{s, t, u, \ldots\}$ together with an interpretation or meaning function $\mathcal{M}$. $\mathcal{M}$ maps formulas and states to truth values. For example, $\mathcal{M}_s[\![(X = 1)]\!] = \mathit{true}$ signifies that the formula $(X = 1)$ is *true* in state s. In other words, the value of variable X in state s is 1.

The function $\mathcal{M}$ can be extended to mapping formulas and *intervals* to truth values. Note that a state is a special case of an interval: an interval with length zero. A formula is either true or false on an interval. If formula w is true on an interval σ, $\mathcal{M}_\sigma[\![w]\!] = \mathit{true}$, then σ satisfies w. This is written $\sigma \models w$. If w is true on all intervals then it is a valid formula. This is written $\models w$.

The semantics of the operators which were described informally are now given using the meaning function.

Non-temporal Operators

The negation of a formula $\neg w$, is true in an interval if and only if w is false in that interval:

$$\mathcal{M}_{s_0 \ldots s_n}[\![\neg w]\!] = \mathit{true} \quad \text{iff} \quad \mathcal{M}_{s_0 \ldots s_n}[\![w]\!] = \mathit{false}$$

The conjunction $w_1 \wedge w_2$ is true in an interval if w_1 and w_2 are both true in that interval:

$$\mathcal{M}_{s_0 \ldots s_n}[\![w_1 \wedge w_2]\!] = \mathit{true} \quad \text{iff} \quad \mathcal{M}_{s_0 \ldots s_n}[\![w_1]\!] = \mathit{true} \text{ and } \mathcal{M}_{s_0 \ldots s_n}[\![w_2]\!] = \mathit{true}$$

Other non-temporal operators such as implication ($\supset$) are interpreted similarly.

Temporal Operators

The formula $\Box\, w$ is true if w is true in every subinterval which finishes in the final state:

$$\mathcal{M}_{s_0 \ldots s_n}[\![\Box\, w]\!] = \mathit{true} \quad \text{iff} \quad \mathcal{M}_{s_i \ldots s_n}[\![w]\!] = \mathit{true} \text{ for all } i \le n$$

The formula $\bigcirc\, w$ is true on a non-empty interval if w is true from the next state on:

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

The formula w_1 ; w_2 is true in an interval iff the interval can be partitioned into two sub-intervals which share a common state s_i, such that w_1 is true in the first sub-interval, and w_2 is true in the second:

$$\mathcal{M}_{s_0 \ldots s_n}[\![w_1 ; w_2]\!] = \mathit{true} \quad \text{iff} \quad \mathcal{M}_{s_0 \ldots s_i}[\![w_1]\!] = \mathit{true} \text{ and } \mathcal{M}_{s_i \ldots s_n}[\![w_2]\!], \text{ for some } i, 0 \le i \le n.$$

The formula *len n* is true on an interval whose length is exactly n :

$$\mathcal{M}_{s_0 \ldots s_m}[\![\mathit{len}\ n]\!] = \mathit{true} \quad \text{iff} \quad m = n$$

Similarly, the formula $\mathit{len} > m$ is true on an interval whose length is greater than m.

4.3 More ITL Operators

The operators presented so far can be combined to define more ITL operators. Moszkowski [Mos83] defines many of these. The operators frequently used in my hardware descriptions are defined below.

Temporal Equality

Two signals A and B are temporally equal in an interval if they have the same values in all states. This is written $A \approx B$, and is defined:

$$A \approx B \quad \equiv_{\text{def}} \quad \Box\, A = B$$

For example in Figure 4.1:

$$\langle s_1, s_2, s_3 \rangle \models (X \approx 1)$$

Temporal Assignment

The formula $A \rightarrow B$ is true for an interval if the signal A's initial value equals B's final value. To define this I need the ITL construct $\mathit{fin}(w)$ which is true if w is true in the last state of an interval, and false otherwise. In this definition, c is a static variable; its value does not change on the interval:

$$A \rightarrow B \quad \equiv_{\text{def}} \quad \forall c.((A = c) \supset \mathit{fin}(B = c))$$

Temporal Stability

A signal A is stable in an interval if its value does not change in that interval. This is defined using the static variable c:

$$\mathit{stb}\ A \quad \equiv_{\text{def}} \quad \exists c.(A \approx c)$$

Delay

These operators can be used to state that one signal is a delayed version of another. In addition, quantitative information about that delay can be specified. The ITL formula to express signal B is signal A delayed by m unit intervals is:

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

4.4 Extension of ITL

Up to this point I have described ITL as presented in Moszkowski's PhD Thesis. In order to facilitate description of circuits at the transistor level, I have changed the definition of signals and extended the logical and temporal operators to handle them. In this section I discuss these changes.

4.4.1 Variables

I have intentionally refrained from specifying variable types. Moszkowski presents several, including static variables and signals. The type of a variable is usually obvious from context. Most of my variables are signals, which are described below. Variables which range over the natural numbers include m in $\mathit{len}\ m$. Static variables are only used for definitions and are explicitly described as static. A static variable is a signal whose value does not change on an interval.

4.4.2 Signals

I describe signals as $\{\mathit{value}, \mathit{strength}\}$ pairs. Each of the fields *value* and *strength* can have value 0 or 1. The *value* field represents the boolean logic value of a signal. The *strength* field represents the strength of a signal which may be either capacitive (0) or driven (1). The use of the strength field is explained in the next chapter.

4.4.3 Operators for Signals

The Logical Operators

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

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

Most temporal logic operators have the same semantics as described earlier. The exception is the temporal assignment operator, which is defined using $\Leftrightarrow$ instead of $=$. Temporal assignment assigns both strength and value. The new definition for $\rightarrow$ is:

$$A \rightarrow B \quad \equiv_{\text{def}} \quad \forall c.((A \Leftrightarrow c) \supset \mathit{fin}(B \Leftrightarrow c))$$

Here c is a static *signal* variable. Its value and strength fields have the same value throughout the interval.

The Functions weaken and strengthen

The functions weaken and strengthen operate only on the strength field of a signal. The weaken function takes a signal and returns a signal with the same value field whose strength is 0. The strengthen function is described similarly:

$$\forall S.\ \mathsf{weaken}\ \{V,S\} = \{V,0\}$$
$$\forall S.\ \mathsf{strengthen}\ \{V,S\} = \{V,1\}$$

Note that, in general, weaken and strengthen are not the inverse of each other:

$$\neg\forall a.\ \mathsf{weaken}(\mathsf{strengthen}(a)) \Leftrightarrow a$$
$$\neg\forall a.\ \mathsf{strengthen}(\mathsf{weaken}(a)) \Leftrightarrow a$$

The Join Operator

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

$$
\begin{array}{l}
\{V,1\} \sqcup \{W,0\} \Leftrightarrow \{V,1\} \\
\{W,0\} \sqcup \{V,1\} \Leftrightarrow \{V,1\} \\
A \sqcup A \Leftrightarrow A
\end{array}
$$

A New Property for Temporal Assignment

Temporal assignment allows one signal at the beginning of an interval to be assigned to another signal at the end of the interval. I extend this to allow more than one signal to be assigned to another signal. When this occurs, the join of the two signals is assigned to the resulting signal. This can be described in ITL:

$$\langle s_i \ldots s_n \rangle \models ((A \rightarrow B) \;\wedge\; (C \rightarrow B) \supset (A \sqcup C \rightarrow B))$$

Note that this is logically inconsistent with the semantics of $\rightarrow$ given above. There the property implied is:

$$\langle s_i \ldots s_n \rangle \models ((A \rightarrow B) \;\wedge\; (C \rightarrow B) \supset A \Leftrightarrow C) \qquad (4.1)$$

I do not however exploit this inconsistency. As will be seen in Chapter 6, the system I use to manipulate temporal logic formulas has its rules listed in a rule database. Equation 4.1 is not part of that database. In addition, I check that all formulas of the form $A \rightarrow B$ are considered when determining the value of B. All such formulas which are true on an interval with the same final state, s_n, may effect the value of B in that state. The new property for the join operator implies that the value of a node can always be changed by another signal contributing to that node. Eventually a node becomes hidden in a circuit description, so the number of signals which contribute to that node is limited.

In essence, I am combining the join function with temporal assignment. Others [Joy87], [Dhi87] have used explicit join components in their circuit representations. I decided not to use this approach for several reasons. First, join components must be represented in the circuit description so this description no longer directly reflects the structure of the circuit. Second, there is an explosion of signal names. For example, with my description joining two signals can be described by:

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

An equivalent description with an explicit join component would require five signal names instead of three:

$$(A \rightarrow B_1) \wedge (C \rightarrow B_2) \wedge (B = (B_1 \sqcup B_2))$$

Finally, if join components are added to nodes which are not yet hidden, additional signals may later be connected to the join component which would change the value of the output. This is equivalent, in my description, to needing to ensure that all contributing signals will be taken into account. The problem arises as a result of modeling circuits which may not be completely specified and not as a result of the way the join function is implemented.

4.4.4 Another Interpretation for Signal Strengths

The description of signals allows modeling of the capacitive effects of circuits. The two strengths, 0 and 1 can be viewed as capacitive and driven strengths, respectively. This is based on Bryant's more general lattice model [Bry81], which allows an arbitrary number of signal strengths. My signals can be extended to be equivalent to Bryant's lattice model by allowing natural numbers as strengths and by appropriately modifying the weaken and strengthen functions.

The lattice for two strength values is shown in Figure 4.2. Here values are taken from the set $\{1, 0, X\}$. An X signal results when two signals of the same strength but different values are joined. Z signifies the null signal. The figure shows a partial ordering on signals. The strength of signals can be viewed as a vertical chain which has the weakest strength at the bottom. The partial ordering on values is specified by the following equations, where $a \sqsubseteq b$ means a precedes b in the ordering:

$$\begin{aligned} 1 &\sqsubseteq X \\ 0 &\sqsubseteq X \end{aligned}$$

My descriptions do not make explicit use of X and Z. If a join occurs which would result in an X value, the corresponding formula cannot be simplified. As a result, I cannot detect the effects of propagating X and Z values through a circuit. I can however detect designs which will cause nodes to go into the X state provided that the inputs are well defined.

4.5 Conclusions

In this chapter I have presented an extended subset of ITL for transistor level circuit description. The constructs which have been introduced are summarized in Table 4.1. In the next chapter I will use this extended subset of ITL to specify and reason about the behavior of circuits at the transistor level.

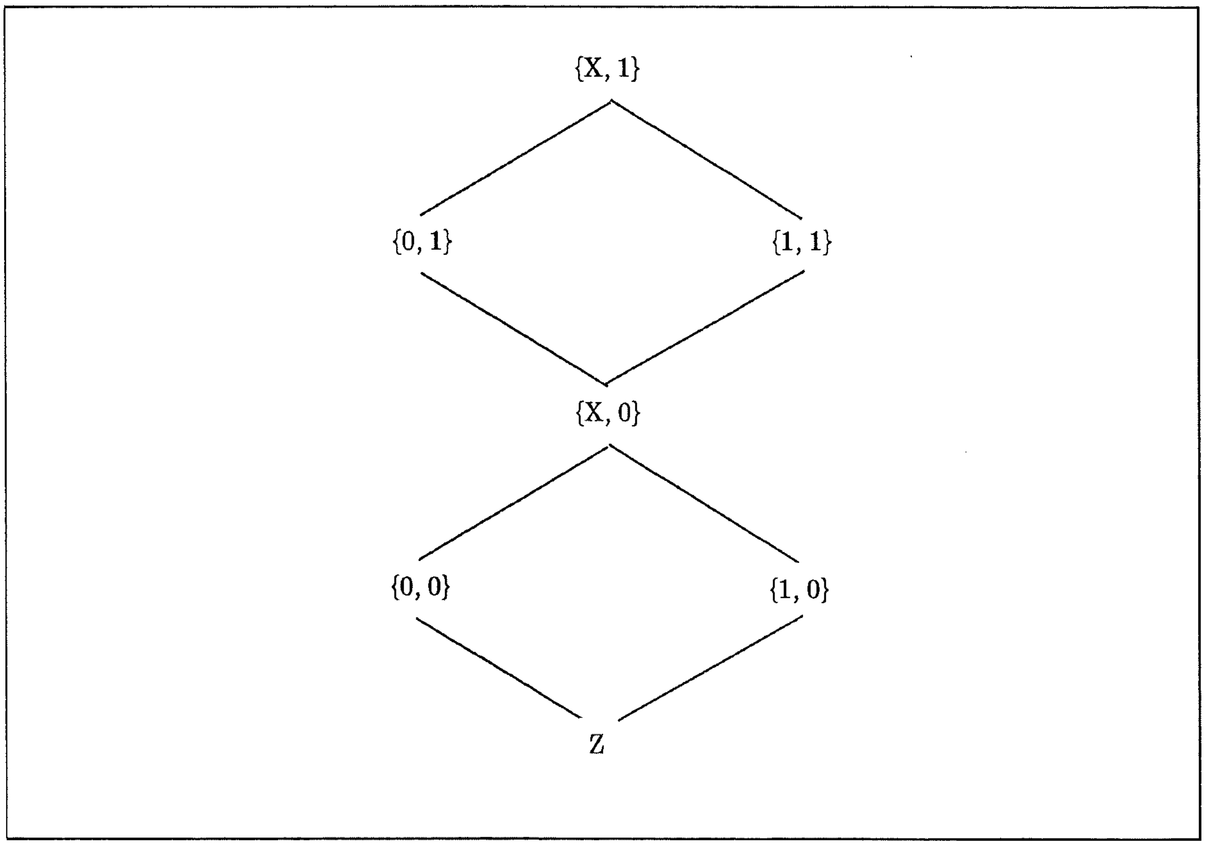


Figure 4.2: The Lattice Model for Signals

Operator Name	Formula	Interpretation/Definition
always	$\Box\, w$	$\mathcal{M}_{s_0 \ldots s_n}[\![\Box\, w]\!] = \mathit{true}$ iff $\mathcal{M}_{s_i \ldots s_n}[\![w]\!] = \mathit{true}$ for all $i \leq n$
next	$\bigcirc\, w$	$\mathcal{M}_{s_0 \ldots s_n}[\![\bigcirc\, w]\!] = \mathit{true}$ iff $n \geq 1$ and $\mathcal{M}_{s_1 \ldots s_n}[\![w]\!] = \mathit{true}$
chop	$w_1; w_2$	$\mathcal{M}_{s_0 \ldots s_n}[\![w_1; w_2]\!] = \mathit{true}$ iff $\mathcal{M}_{s_0 \ldots s_i}[\![w_1]\!] = \mathit{true}$ and $\mathcal{M}_{s_i \ldots s_n}[\![w_2]\!]$ for some i $0 \leq i \leq n$
fin	$\mathit{fin}\ w$	$\mathcal{M}_{s_0 \ldots s_n}[\![\mathit{fin}\ w]\!] = \mathit{true}$ iff $\mathcal{M}_{s_n}[\![w]\!] = \mathit{true}$
length	$\mathit{len}\ n$	$\mathcal{M}_{s_0 \ldots s_m}[\![\mathit{len}\ n]\!] = \mathit{true}$ iff $m = n$
skip	skip	$\mathit{skip} \equiv_{\text{def}} \mathit{len}\ 1$
temporal equality	$A \approx B$	$A \approx B \equiv_{\text{def}} \Box(A = B)$
temporal assignment	$A \rightarrow B$	$A \rightarrow B \equiv_{\text{def}} \forall c.((A \Leftrightarrow c) \supset \mathit{fin}(B \Leftrightarrow c))$ for c static
temporal stability	$\mathit{stb}\ A$	$\mathit{stb}\ A \equiv_{\text{def}} \exists c.(A \approx c)$ for c static
delay	$A\, \mathrm{del}^m\, B$	$A\, \mathrm{del}^m\, B \equiv_{\text{def}} \Box\, \mathit{len}\ m \supset (A \rightarrow B)$
signal equivalence	$\{V_1, S_1\} \Leftrightarrow \{V_2, S_2\}$	$\{V_1, S_1\} \Leftrightarrow \{V_2, S_2\} \equiv_{\text{def}} V_1 = V_2 \wedge S_1 = S_2$
weaken	weaken$\{V, S\}$	weaken $\{V, S\} \Leftrightarrow \{V, 0\}$
strengthen	strengthen$\{V, S\}$	strengthen $\{V, S\} \Leftrightarrow \{V, 1\}$
join	$V_1 \sqcup V_2$	$\{V_1, 1\} \sqcup \{V_2, 0\} \Leftrightarrow \{V_1, 1\}$ $\{V_2, 0\} \sqcup \{V_1, 1\} \Leftrightarrow \{V_1, 1\}$ $A \sqcup A \Leftrightarrow A$

Table 4.1: Extended ITL Operators

Chapter 5

Using ITL to Reason about the Behavior of Circuits

In Chapter 3 I discussed how Prolog is used to specify the structure of circuits. In Chapter 4 I presented some constructs of extended ITL. In this chapter I show how extended ITL is used to model the behavior of circuits, and show with examples how these models are manipulated. First, I discuss my models for the behavior of circuit elements. These models were chosen to satisfy the following criteria:

- Expressibility in ITL
- Ability to express characteristics of hardware
- Ease of manipulation

The models describe two types of information:

- The dependence of outputs on inputs (function and timing)
- Constraints on the inputs

Constraints ensure that the model accurately captures the behavior of the device. If the constraints hold, the function and timing information is valid. Constraints include such timing information as set-up and hold times. The implicit form of the correctness statement for the behavior of a circuit is:

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

Constraints are described and reasoned about with the same logic used for the timing and functional behavior of circuits. However, constraints on the outputs of a circuit are considered separately from behavior.

I consider a restricted class of CMOS circuits. These are synchronous circuits where all feedback loops are broken by clocks and there are no branches in combinational logic which merge together again. As I shall show, many timing constraints are satisfied by this class of circuits.

In the first section I discuss how behavioral models are used in hierarchical verification. Then I present models for circuit elements. These include combinational elements, transistors, and memory elements. Examples are provided which illustrate how these models are manipulated in reasoning about circuit behavior. Note that the term *gate* is used sometimes to refer to the electrode of a transistor and sometimes to refer to a combinational element. The term's meaning should be clear from context.

5.1 Hierarchical Verification

I use ITL to reason about the function and timing behavior of circuits. In this chapter I present temporal logic models for the behavior of circuits, and show with examples how these behaviors are manipulated. Specifically, I show that the formula describing the implementation of a circuit logically implies the formula describing the specification. This approach was inspired by [Gor85a] and [Bar84]. In general it proceeds as follows. A circuit implementation is described as the composition of circuit elements. These elements are composed with the logical operator $\wedge$ (*and*). The elements are expanded by their behavioral descriptions. These descriptions are then manipulated using rules of mathematical logic to show that the implementation implies the specification of the behavior of the circuit.

This verification is done hierarchically. The behavior of a circuit component is verified by checking that it is implied by the composition of the behavior of its components. The verified behavior is then used as the description of the circuit component when that component is used later in the design process. This is called behavioral abstraction.

One advantage of hierarchical verification is that a circuit component needs to be verified only once, no matter how many times it is used in the circuit. A component is reused simply by *renaming* its ports to their connections. I use Prolog predicates to describe behavior, thus renaming happens automatically as a result of unification.

When components are composed hierarchically, internal lines are hidden. In the Prolog specifications, a variable is hidden if it appears in the body of a clause but not in the head of that clause. These variables are effectively ***existentially quantified.***

In addition, I describe constraints in temporal logic. These constraints are composed and reasoned about in a similar manner to behavioral specifications. When a circuit element is composed with others to form a larger circuit component, some of its ports may become inaccessible to other parts of the circuit. Constraints on these ports must be met. Other constraints which are not met become constraints of the larger component.

In the next few sections I describe how different circuit elements are modeled, and use examples to show how these models are used in hierarchical verification. Composing circuit elements, renaming, hiding external lines, and abstracting the behavior of components are all used in these examples. For each example, I show how constraints as well as behaviors are manipulated. These examples employ Prolog notation. Upper case letters denote Prolog variables. Lower case letters denote logical variables. The Prolog notation '_' is used for anonymous variables.

5.2 Modeling Combinational Elements

My main objective is to model timing as well as functional behavior of circuit elements. I model the timing behavior of all combinational elements as delay. Breuer and Friedman [BF76] present many hardware delay models, and Moszkowski [Mos83] shows how several of these can be expressed in ITL. I will use Breuer and Friedman's transport delay, which Moszkowski has defined in ITL. Note that Moszkowski does not use delay to model combinational elements. The definition of transport delay in ITL was given in the previous chapter:

$$A\ \mathrm{del}^m\ B \quad \equiv_{\mathbf{def}} \quad \Box\, len\ m \supset (A \rightarrow B)$$

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

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

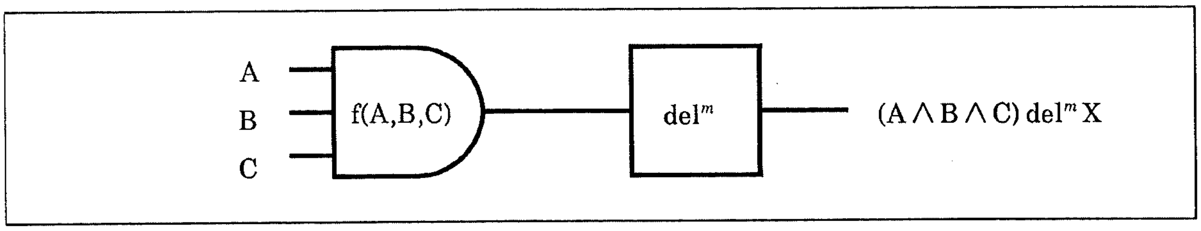


Figure 5.1: Model of a 3-Input and Gate with Delay

Here $\mathsf{f}(A, B, \ldots)$ is a function of the inputs which contains no temporal operators. This can be generalized for a combinational cell with several outputs $(X, Y, \ldots)$, which is described by composing similar descriptions:

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

For example, the 3-input and gate shown in Figure 5.1 is described by:

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

The delay is viewed as being lumped at the output. Note that m represents the worst case delay from inputs to output for this gate. A more accurate model may represent the delay as being between the two extremes of best and worst case. This can be expressed in ITL:

$$A\ \mathrm{vardel}^{m,n}\ B \quad \equiv_{\mathrm{def}} \quad \exists i\ m \leq i \leq n.(A\ \mathrm{del}^i\ B)$$

For simplicity, I will use transport delay rather than variable delay.

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

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

In my descriptions, the output receives a delayed function of the inputs; Moszkowski refers to this as functional delay. Functional composition applies:

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

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

$$\begin{array}{l} \mathsf{well\text{-}behaved}(A) \quad \equiv_{\mathrm{def}} \\ \qquad \Box(\uparrow\downarrow A \supset \mathit{len} > g) \wedge \\ \qquad \Box(\downarrow\uparrow A \supset \mathit{len} > g) \end{array}$$

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

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

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

$$\mathsf{well\text{-}behaved}(A) \wedge A\,\mathrm{del}^m\, B \quad \supset \quad \bigcirc^m(\mathsf{well\text{-}behaved}(B))$$

Here $\bigcirc^m$ is an abbreviation for m $\bigcirc$ operators applied in sequence. This can be defined recursively:

$$\begin{array}{l} \bigcirc^1 \mathrm{x} = \bigcirc \mathrm{x} \\ \bigcirc^m \mathrm{x} = \bigcirc^{m-1} \bigcirc \mathrm{x} \end{array}$$

Note that the definitions above rely on the assumption that all unit intervals have the same duration. I assume this throughout the thesis.

5.3 Modeling Transistors

I use switch-level models [Bry81] to describe circuit primitives: n-type and p-type transistors. A bidirectional model without delay for the n-type transistor shown in Figure 5.2 is:

$$\Box G = \{1,\_\} \supset (S \Leftrightarrow D)$$

The '_' is a Prolog anonymous variable. Since the '=' operator only examines the value field of the gate signal, it does not matter what the strength field is. This formula expresses that when the value of the voltage on the gate is high, the source and drain are connected.

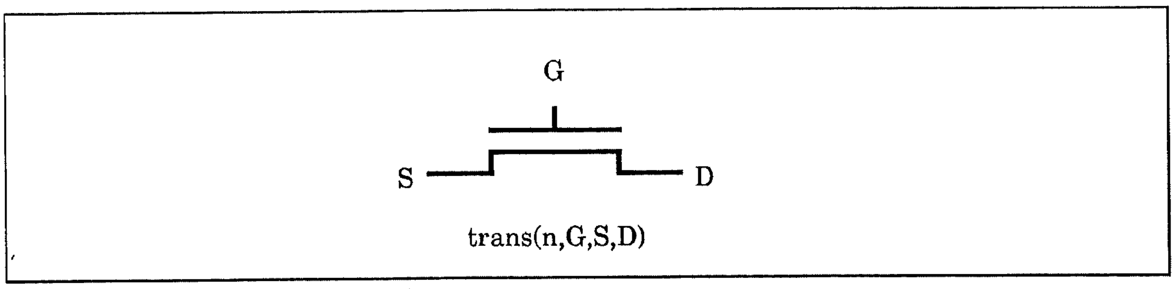


Figure 5.2: An n-type Transistor

There is no distinction between source and drain. The formula says nothing about the behavior of the switch when the gate is low. The similar model for a p-type transistor is:

$$\Box G = \{0, \_\} \supset (S \Leftrightarrow D)$$

From now on, I will only present models for n-type transistors. The models for the complementary p-type transistors are analogous.

I extend the simple switch-level model with timing information. In order to do this I also must specify directionality. Directions of transistors are derived using the Prolog program described in Section 3.3.1. When the gate of the transistor is high, the signal flows from source to drain. Source and drain nodes are not symmetrical. If the model is constrained such that the source is stable when the gate changes, the delay, m, from source to drain can be expressed in ITL:

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

This model is subject to the constraints:

$$\Box(G \approx \{1, \_\} \supset \textit{stb}\ S) \wedge$$
$$\mathsf{well\text{-}behaved}(G)$$

5.3.1 Example 1: a CMOS Inverter

As a simple example, consider the CMOS inverter shown in Figure 5.3. Its behavior is specified in ITL by:

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

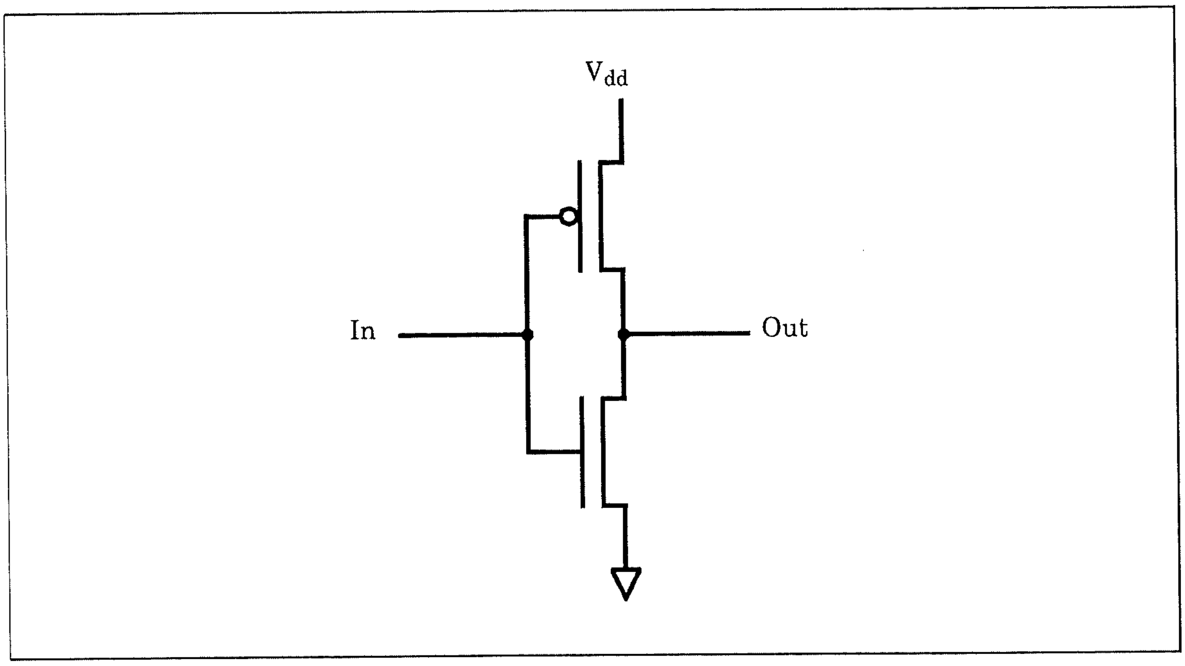


Figure 5.3: A CMOS Inverter

Here strengthen specifies that the output of the inverter is always driven. This is true whether or not the input is driven. Note that I use Prolog variables for the node names. When this behavior is used as the definition of a component of a cell, these nodes are instantiated to the names of the wires to which they are connected.

The implementation of the inverter is specified in ITL by:

$$\begin{aligned}&\mathsf{invert\text{-}struct}(In, Out, m) \quad \equiv_{\text{def}}\\ &\qquad \mathsf{ntrans}(In, \{0,1\}, Out, m) \wedge \mathsf{ptrans}(In, \{1,1\}, Out, m)\end{aligned}$$

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

I proceed by logically manipulating the implementation to derive the behavior. The PALM (Prolog Assistant for Logic Manipulation) system described in Chapter 6 was developed to mechanize these manipulations. The outline of the derivation of behavior for the inverter is given below. The output of the PALM system for this example is given in Appendix B.1. First I derive the behavior of the inverter, then I derive the constraints.

Deriving Behavior

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

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

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

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

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

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

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

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

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

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

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

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

Combining the cases results in the behavioral description:

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

This equation is the same as the definition for behavior given at the start of the section. I have shown that the specified behavior for the inverter can be deduced from its structure and the behavior of its components. Note that in this derivation I used case analysis in a temporal formula. In general, case analysis will not work in this framework. Case analysis over any variable in a temporal logic formula would require enumerating all possible values of that variable in all possible states. If infinite intervals are permitted, this is clearly impossible. I therefore restrict the temporal logic formulas to which I apply case analysis. These restrictions are discussed in Section 6.1.1.

Deriving Constraints

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

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

Since $stb\{1,1\}$ and $stb\{0,1\}$ are always true, this reduces to the single constraint for the inverter:

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

5.3.2 Pass Transistors and Transmission Gates

The circuits I reason about include steering elements as well as combinational elements. In steering logic, the steering element is used as a switch to conditionally connect two nodes together. Steering logic is appropriate when the logic function can be conceptualized as signals being conditionally steered through a network. Steering elements include nMOS

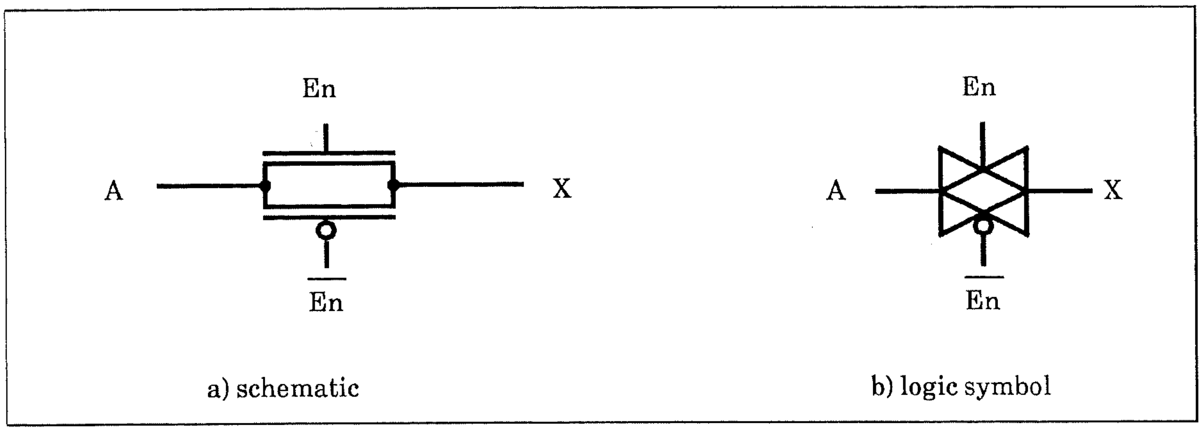


Figure 5.4: A CMOS Transmission Gate

pass transistors and CMOS transmission gates. A pass transistor has neither channel node connected directly to power or ground. In CMOS a *p*-type and an *n*-type transistor are connected with common source and drain connections to form a transmission gate as shown in Figure 5.4.

Whenever the *n*-type transistor of the transmission gate is on, the *p*-type transistor is on as well. Due to threshold drops, the transmission of a logical '1' is degraded when passed through an *n*-type device, while the transmission of a logical '0' is degraded when passed through a *p*-type device. Both types of devices are used in a transmission gate so that both 0's and 1's can be transmitted without degradation. My model for a transistor does not represent threshold effects. In this model, a transmission gate behaves equivalently to an *n*-type pass transistor and is treated as such. The *p*-type transistor is considered redundant. The model used is therefore the same as the *n*-type transistor model presented earlier. Note that the models used by Bryant [Bry81] and Weise [Wei86] also do not represent threshold drops.

5.4 Modeling Charge Storage

The transistor model used above does not specify the behavior of the output node when the transistor is off. Because of the capacitance in MOS circuits, nodes which become isolated as a result of transistors turning off retain their previous driven value for as long as several milliseconds. I refer to this phenomenon as charge storage. There are many

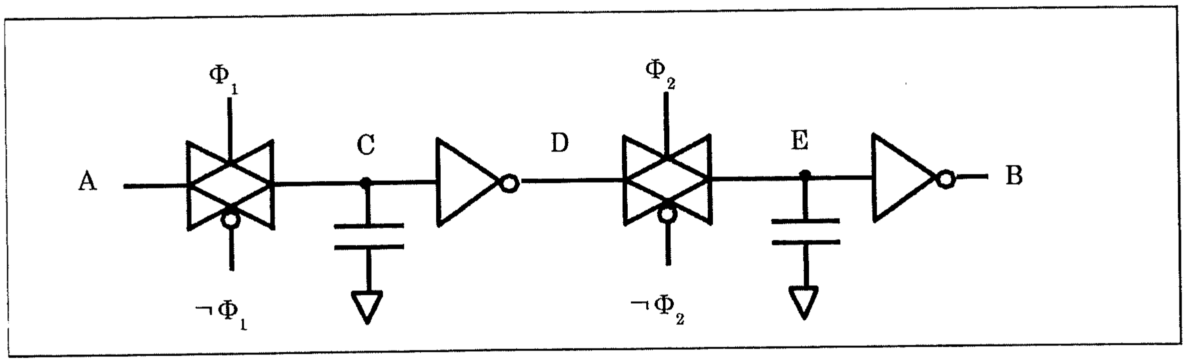


Figure 5.5: A CMOS Shift Register with Explicit Capacitances

sources of capacitance in a MOS circuit including capacitance due to interconnect and capacitance on the gate and channel nodes of transistors.

Charge storage is frequently exploited in MOS circuit design. Circuits which make use of this phenomenon include the CMOS shift register in Figure 5.5. The charge storage in this circuit is largely due to the input capacitance of the inverters. This in turn is due to the gate capacitance of the transistors which make up the inverter. This capacitance is shown explicitly in the diagram. The shift register shifts the value of the input charge stored at node A to the output at node B as follows. When the first transmission gate turns on, the input capacitance of the first inverter (at node C) is charged. When the transmission gate turns off, that charge is isolated from the source of the transmission gate at node A. The capacitance at node C retains its charge until it is discharged or leaks away due to leakage currents. This charge can drive the input of the inverter but has lower drive capability than a driven node which is not isolated from power or ground. When the second transmission gate is on, the inverter drives the source of that gate at node D, and the charge is switched through to the input capacitance of the second inverter on node E. The charge on node E drives the input of the second inverter. Thus the output is driven at node B.

When the first transmission gate is off and the second is on, the second part of the of the circuit is isolated from the input. When the second transmission gate is off, the output (node B) is isolated and retains its value until the next time the input capacitance of the second inverter is charged, or until the present charge leaks away. Due to the 2-phase clocking scheme (described in Section 5.5) the two transmission gates are never on at

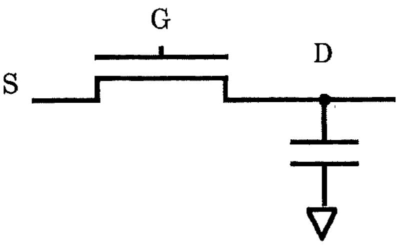


Figure 5.6: A Pass Transistor with Explicit Capacitance

the same time. The charge is stored on the gates of the transistors which make up the inverters; isolation of the charge is a result of having a transmission gate connected to those gates.

I model pass transistors and transmission gates using the temporal logic equations presented in the previous section. I model charge storage by connecting an explicit 'ideal capacitance' to the drain node of the transistor as shown in Figure 5.6. As in the shift register example, this capacitance is largely due to the gates of the transistors being driven. However, the effects I am interested in modeling only influence the circuit when a node is isolated. This in turn only occurs when a switch is open. Therefore, I model this capacitance on the drain of a transistor. I assume the drain drives a capacitive load. In actual fact, this load usually arises from the capacitance on the gates of the transistors being driven.

The capacitor is described in ITL by:

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

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

$$\begin{array}{l} \Box((G = \{1, \_\}) \land \mathit{len}\ m \supset (S \rightarrow D)) \land \\ \Box\, \mathit{len}\ 1 \supset (\mathsf{weaken}\ D \rightarrow D) \end{array}$$

This states that m time units after the transistor turns on, node D is driven. If the transistor is on and S is a strong signal, then this overrides the memory due to the capacitor. When the transistor is off, the memory due to the capacitor retains the last driven value on the node. Note that the overriding effect is due to the property of temporal assignment discussed in Section 4.4.

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

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

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

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

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

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

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

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

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

The constraints on this behavior are:

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

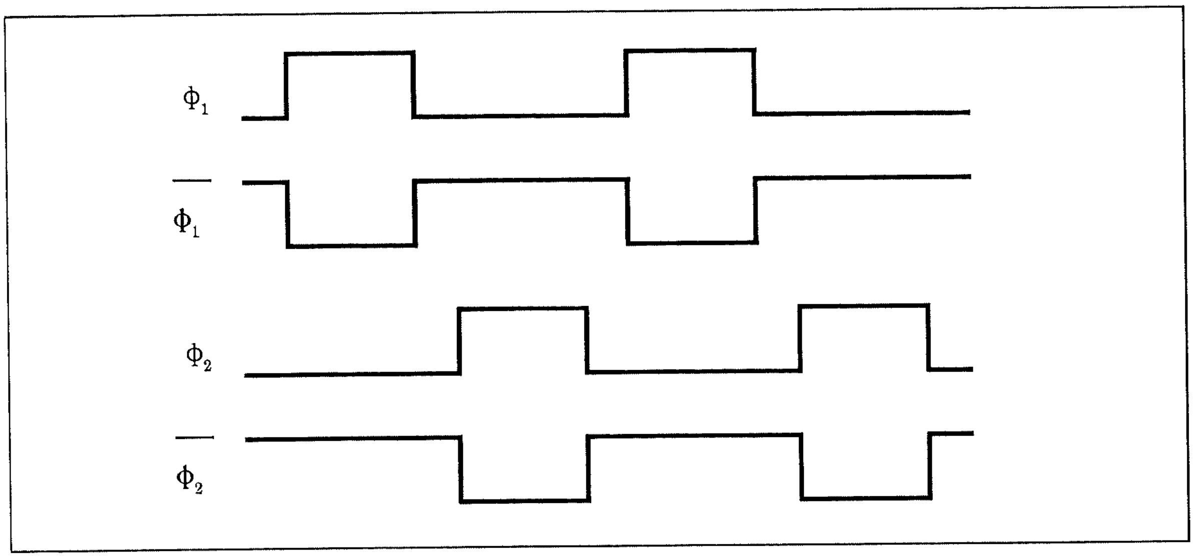


Figure 5.7: 2-Phase Non-overlapping Clocks

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

5.5 Clocking

Various clocking strategies for CMOS circuits are discussed in section 5.4 of [WE85]. The circuits presented in this thesis employ a 2-phase clocking strategy with non-overlapping clocks. This is referred to in [WE85] as pseudo 2-phase clocking because in reality four different clock phases are used by the circuit. These phases $(\Phi_1, \overline{\Phi}_1, \Phi_2, \overline{\Phi}_2)$ are shown in Figure 5.7.

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

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

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

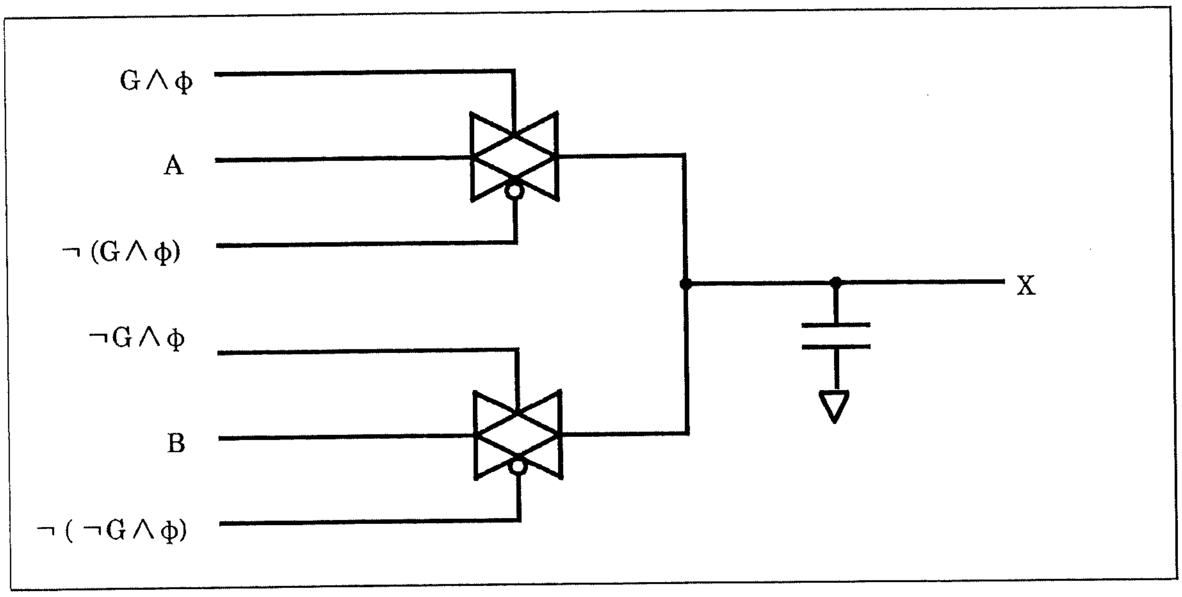


Figure 5.8: A CMOS Clocked 2-to-1 Multiplexer

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

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

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

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

In this section I derive the behavior and constraints of the CMOS clocked 2-to-1 multiplexer shown in Figure 5.8. This circuit behaves like a 2-to-1 multiplexer when the clock(Φ) is high, and retains its previous value when the clock is low. The example makes use of charge storage and clocking models discussed in previous sections.

Deriving Behavior

The behavior of the multiplexer is described in ITL as:

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

This behavior definition uses the conditional:

$$\mathit{if}\; a\; \mathit{then}\; b\; \mathit{else}\; c \quad \equiv_{\textsf{def}} \quad (a \supset b) \;\wedge\; (\neg a \supset c)$$

The full derivation of behavior using the PALM system is given in Appendix B.2. An outline of this derivation follows. As usual, it begins with the description of the implementation of the circuit.

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

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

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

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

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

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

Deriving Constraints

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

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

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

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

This stronger constraint states that if G is stable whenever Φ is true, then the resulting signal is well-behaved. I still need to show that X is driven often enough. This is a direct consequence of the stronger constraint since, when Φ is true either G or $\neg G$ is true. Thus, the output is driven once every clock cycle. The remaining two constraints are the constraints 2 and 4 above. Thus the constraints for this device are:

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

5.6 Conclusions

I have shown how Prolog and ITL can be used to derive the behavior of a circuit from its implementation and to reason about constraints on that behavior. These derivations were done by manipulating the behavior of the components of the circuits using the PALM system. PALM is described in more detail in the next chapter.

Chapter 6

PALM: Prolog Assistant for Logic Manipulation

PALM (Prolog Assistant for Logic Manipulation) is a general purpose rule rewriting system which provides several facilities for proof checking. The user can use the system interactively, or she can write a script to run a series of manipulations. The operations described below are available in either mode. In both modes, the user specifies which operations to apply.

PALM manipulates terms, rules, and definitions. A term is a logical formula, and is represented as a Prolog term. The system maintains a stack onto which the user can push and pop terms. Rules are schemata for rewriting terms. Prolog variables in rules are matched to sub-formulas of terms using the unification algorithm. Rules are provided by the user. Definitions are abbreviations; these are also provided by the user.

This chapter gives an introduction to PALM. Figure 6.1 gives a brief summary of commands available in PALM. The next section discusses these PALM comands in more detail. In the following section I discuss the PALM rules used to reason about circuits.

6.1 General Purpose Features

PALM provides commands which allow the user to manipulate terms, rules, and definitions, apply rules to terms, and do more complex manipulations. These commands are described below. PALM also provides other aids to the user such as the commands **help**, **halt**, and **undo**.

The PALM commands for manipulating terms are:

enter-term	Enter a new term which becomes the current term.
disp-term	Display the current term.
push-term	Save the current term on the stack.
pop-term	Pop the top term off the stack, and set it to the current term.
split-term	The current term must be in the form $A \wedge B$. (If not an error message is printed.) The current term is split into two terms, A and B. The current term is set to A and B is pushed onto the stack. The user is asked where the term should be split.
swap-term	Swap the current term with the top of the stack.
and-term	Pop the top term off the stack and *and* it with the current term.
swap-top	Swap the top two terms on the stack. Does not affect the current term.
list-terms	List all terms on the stack.

The PALM commands for manipulating rules are:

enter-rule	Enter a new rule. The user is prompted for the rule name and the rule. The new rule is asserted in the database.
disp-rule	Display a specified rule. The user is prompted for the rule name.
op-rule	List all rules in the database whose main logical operator is *Op*. The user is prompted for *Op*.
list-rules	List all rules in the database.

The PALM commands for manipulating definitions are:

enter-def	Enter a new definition. The user is prompted for the name and the definition. The new definition is asserted in the database.
disp-def	Display a specified definition. The user is prompted for the name.
list-defs	List all definitions in the database.
expand-def	The user is prompted for the definition name. The first occurrence of this name is replaced with its definition in the current term.
expand-all	All occurrences of names which have been *defined* are replaced by their definitions in the current term.
shrink-def	The user is prompted for the definition name. The first occurrence of the definition is replaced with its name in the current term.

Figure 6.1: PALM Commands (continued on next page)

The PALM commands for applying rules to terms are:

apply	Apply named rule to the current term. The user is prompted for the name of the rule.
rec-apply	Same as apply, but finds all the possible ways a term can be rewritten with the named rule. Displays the possibilities one by one, and the user chooses which displayed term becomes the current term.
suc-apply	Same as apply, but applies the same rule successively. The user is prompted for the number of times to apply the rule.

The compound PALM commands are:

replace	Replace every occurrence of an expression in the current term with a new expression. The user is prompted for the expressions. The system creates a sub-goal of proving the two expressions are equivalent. The current term is saved on the stack.
end-replace	If the current term is the constant *true* then pops the top of the stack and sets it to the current term. This is the term with replacements. If the current term is not the constant *true*, an error message is printed.
dnf-rep	Replace an expression in the current term with its disjunctive normal form.
case-anal	Perform case analysis on an expression in the current term. The user is prompted for the variable to which case analysis is applied and the possible values that variable can have. The current term becomes the term with the case analysis variable assuming its first value.
next-val	The current term becomes the term to which case analysis is being applied with the case analysis variable assuming its next value.
end-case	Combine the results of case analysis to form the new current term. If all cases have not been considered an error message is printed.

The miscellaneous PALM commands are:

help	List all PALM commands.
call	Call a Prolog predicate. The user is prompted for the predicate.
undo	Undo the last PALM command.
halt	Exit the PALM system and return to Prolog.

Figure 6.1: PALM Commands

Manipulating Terms

PALM provides commands for manipulating terms and saving them on a stack. The *current* term is the term being manipulated. Initially, the current term is set to the empty list: '[]'. A user can enter a new term or display the current term. She can list all the terms on the stack, push the current term onto the stack, and pop the top term off the stack to make it the current term. The current term may be swapped with the top of the stack, and the two top terms on the stack may be swapped.

PALM also provides two compound commands which allow the user to save part of the current term on the stack. The current term is split into two parts; one will become the current term and the other will be saved on the stack. This split will only occur if the current term is of the form $A \wedge B$ where A and B are themselves terms. A becomes the new current term and B is pushed onto the stack. In general, the current term is of the form $A_1 \wedge A_2 \wedge \ldots \wedge A_n$, where the A_i are terms. The command **split-term** queries the user where to split the current term. The first part of the term becomes the new current term, and the second part is pushed on the stack. The command **and-term** does the reverse of **split-term**. The top term is popped off the stack and *and*ed with the current term to form the new current term.

Manipulating Rules

Rules are schemata for rewriting terms. A Prolog variable in a rule stands for a term. PALM exploits Prolog unification to match rules to terms. Rules are asserted in a database which is manipulated by the system. There are no built-in rules; all rules are provided by the user. The PALM commands described in this section show how rules are entered and examined. Applying rules to terms is described in a later section.

Each rule has a name associated with it in the database. The user enters a rule by specifying the name and the rule. She can display a rule of a certain name or display all rules in the database. All rules with the same main logical operator can also be displayed. For example, rules in the database which have have $=$ as their main logical operator are shown in Figure 6.2. Note that I use $\{1,\_\}$ and $\{0,\_\}$ to represent the truth values *true* and *false* respectively. This is done for convenience, in order to have fewer rules in the database.

```
option : op-rule.
operator : = .
rule and-true is: $(A \wedge B = \{1,\_\}) = (A = \{1,\_\}) \wedge (B = \{1,\_\})$
rule falsity is: $(\{0,\_\} = \{1,\_\}) = \{0,\_\}$
rule identity is: $(A = A) = \{1,\_\}$
rule comm-equals is: $(A = B) = (B = A)$
rule equals-false is: $(A = \{0,\_\}) = \neg A$
rule equals-true is: $(A = \{1,\_\}) = A$
```

Figure 6.2: Rules for the Equals Operator '='

Manipulating Definitions

Definitions are abbreviations provided by the user. These are asserted in a definition database. Definitions are similar to rules in that Prolog variables stand for terms and unification is used to match definitions to terms. A definition is identified by a name.

The user can enter a new definition, display a named definition, or list all definitions in the database. The **expand** command allows the user to replace the head of a definition with its body. Prolog unification is used to match variables in the definition. The user can either specify the name of the definition to be expanded, or can ask for all definitions whose head matches part of the current term to be expanded by their bodies. A user can also ask that the body of a named definition be replaced by its head. This is carried out only if the body can be unified with a sub-term of the current term.

I use definitions for circuit elements and for temporal logic operators. Definitions for the circuit primitives ntrans (n-type transistor), ptrans (p-type transistor), and cap (capacitor), and the definition of the temporal logic operator del (delay) are shown in Figure 6.3. These are equivalent to the definitions discussed in Chapter 4.

Applying Rules to Terms

PALM provides three commands for applying rules to terms. The command **apply** applies a rule to the current term to yield the new current term. The command **rec-apply** finds

definition ntrans is:
  ntrans$(A, B, C, D) \equiv_{\text{def}} \Box(A = \{1,_\}) \wedge \mathit{len}\ D \supset (B \rightarrow C)$
definition ptrans is:
  ptrans$(A, B, C, D) \equiv_{\text{def}} \Box(A = \{0,_\}) \wedge \mathit{len}\ D \supset (B \rightarrow C)$
definition cap is:
  cap$(A) \equiv_{\text{def}} \Box\, \mathit{len}\ 1 \supset (\mathit{weaken}\ A \rightarrow A)$
definition del is:
  del$(A, B, C) \equiv_{\text{def}} \Box\, \mathit{len}\ A \supset (B \rightarrow C)$

Figure 6.3: Some Definitions

all the possible ways of applying the rule. The new terms are displayed and the user specifies which one to use. The command **suc-apply** successively applies a rule to the current term. The user specifies the number of times to apply the rule.

A rule in PALM is a schema for rewriting a term. Rules consist of a left-hand side, a right-hand side, and an operator. The operator should be $=$, $\Leftrightarrow$, or $\supset$ (implies), although PALM does not enforce this. If the operator is $=$ or $\Leftrightarrow$ then rewriting is equivalent to the rule of inference of substitution for equality. Most rules in my database fall into this category. The operator $\supset$ corresponds to substitution for implication. This is valid due to the monotonicity of implication. It is not, however, reversible.

Internally, the left hand side and the right hand side of a rule are represented as binary trees. The root of the tree is an operator and the left child and right child are sub-terms corresponding to the first and second operands. Operators are either unary or binary. For a unary operator, the second operand is represented as the empty list([]). For example, my rule database has two rules for the associativity of the binary operator $\wedge$. Figure 6.4 shows the trees for the left and right-hand sides of the rule **assoc-andl**:

$$A \wedge B \wedge C = A \wedge (B \wedge C)$$

The Prolog representation of these trees are also shown. In Prolog a tree is represented as a list of three elements whose first element is the root of the tree and whose second and third elements are the left and right children respectively. Each element is itself either a tree or a simple term, which represents a leaf node.

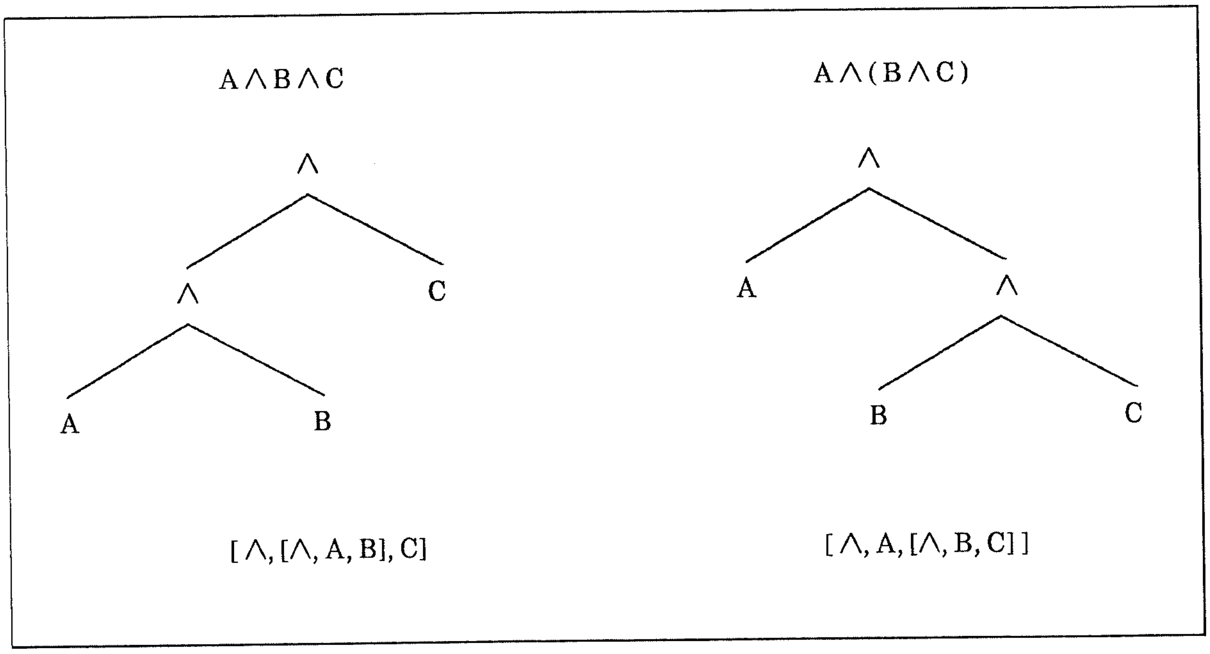


Figure 6.4: Trees which Represent the Rule **assoc-andl**

PALM represents terms internally as trees in the same way as it represents rules. When the user applies a rule to a term, the system tries to match the left-hand side of the rule to a sub-tree of the term using unification. If a match is found, the matching sub-tree is replaced with the right-hand side of the rule with the appropriate substitutions. The term is related to the new term by the operator of the rule.

6.1.1 Compound Commands

In addition to simply applying a rule, PALM provides commands which may have several effects. Examples of such commands are the **split-term** and **and-term** commands described above. These commands alter the stack as well as the current term. Other complex commands are described in this section. These commands allow the user to replace part of one term with another, apply case analysis or replace part of the current term with its disjunctive normal form.

Replacements

When manipulating logic equations, the user frequently wishes to replace one expression with another. PALM facilitates this by providing two commands, **replace** and **end-replace**. The **replace** command is used to specify that every occurrence of an expression in the current term ($Expr1$) be replaced by another expression ($Expr2$). PALM does these replacements and then pushes the new term onto the stack. A new goal is formed to prove that the two expressions are equivalent. Thus the new current term has the form: $Expr1 = Expr2$. The command **end-replace** checks to see if the current term is the logical constant *true*. In other words, it checks that the terms being replaced are equivalent. If so, the top term is popped off the stack, making the term with replacements the new current term.

Case Analysis

Case analysis is a useful technique in hardware verification. It allows the user to consider all the terms which result from a variable in the current term being instantiated to all of its possible values.

The PALM system provides three commands, **case-anal**, **next-val**, and **end-case**, to facilitate case analysis. The command **case-anal** queries the user for the name of the variable to do case analysis over, and for its possible values. Since PALM does not have types associated with variables, the system does not check that all possible values are included. If the current term is Q and the variable over which case analysis is being done is a, then PALM generates a term for each possible value of a. Suppose a is boolean and its possible values are taken from the set $\{true, false\}$. Then PALM generates two terms, saves one and sets the new current term to the other. The new current term in this example would be:

$$(a = true) \supset Q[true/a]$$

where $Q[true/a]$ denotes Q with every occurrence of a replaced by *true*.

The command **next-val** saves the current term and makes the new current term the next term generated by case analysis. In our example the current term would become:

$$(a = \mathit{false}) \supset Q[\mathit{false}/a]$$

next-val continues to generate terms as long as there are more values of a to consider. The command **end-case** puts together the terms generated by case analysis, as described below.

The logical justification of case analysis follows. Consider a boolean variable a which has possible values taken from the set $\{\mathit{true}, \mathit{false}\}$. Assume the current term is Q, and that Q contains no temporal operators. Since a is boolean, the following equation is true:

$$((a = \mathit{false}) \vee (a = \mathit{true})) = \mathit{true}$$

For a variable a which is not boolean, it is easy to construct a statement analogous to the one above over all the possible values of a, provided that the variable can only take a finite number of values. Any formula Q is equivalent to the implication: $\mathit{true} \supset Q$. Using the inference rule of substitution for equality we can replace true in this implication with the left-hand side of the equality above. This yields the formula:

$$((a = \mathit{false}) \vee (a = \mathit{true})) \quad \supset \quad Q$$

This is equivalent to:

$$((a = \mathit{false}) \supset Q) \;\wedge\; ((a = \mathit{true}) \supset Q)$$

The PALM system gives the user these terms one at a time split at the $\wedge$. The user then performs logical manipulations on them, to derive:

$$((a = \mathit{false}) \supset Q') \;\wedge\; ((a = \mathit{true}) \supset Q'')$$

The command **end-case** takes the various terms from case analysis and *ands* them together. If the two resulting formulas Q' and Q'' are equivalent, then PALM does the following.

First the rule *or-introl*, $(A \supset B) \wedge (C \supset B) = ((A \vee C) \supset B)$, is applied to yield:

$$((a = \mathit{false}) \vee (a = \mathit{true})) \supset Q'$$

The rule *or-bool* tells us that the left hand side is equivalent to *true*. The resulting term is $\mathit{true} \supset Q'$. Since $(\mathit{true} \supset Q) = Q$, we can derive Q'. PALM automatically applies these rules when the command **end-case** is invoked.

The discussion above assumes that the formula Q contains no temporal operators. I wish to do case analysis over temporal formulas. Case analysis over any variable in a temporal formula would require enumerating all possible values of that variable in all possible states. In general this is very complex. For a logic which allows intervals of infinite length, it is impossible.

Case analysis is only permitted for temporal formulas of the form $\Box Q$, where Q contains no temporal operators, and only if certain other conditions hold. The formula describes an interval of time. Case analysis may be done over a variable a if it is only examined in the first state of the interval described by the formula. For example, in the inverter circuit presented in Section 5.3.1, I do case analysis over the variable In which is only examined in the first state of the interval. Thus I only need consider possible values of In in that state. Case analysis is not done for other temporal formulas.

Disjunctive Normal Form

PALM can replace a subterm in the current term with its disjunctive normal form. This is frequently convenient when working with complicated logical formulas. The command **dnf** invokes a series of Prolog predicates and does not use rewrite rules. This command illustrates the ease with which the user can interface Prolog packages to the PALM system.

In the 1-bit adder example discussed in Chapter 7.5, **dnf** is invoked to replace the expression:

$$\begin{array}{l}\neg(a \wedge b) \wedge (\neg(a \wedge c) \wedge \neg(b \wedge c)) \vee \\ \neg(\neg a \wedge \neg b \wedge \neg c) \wedge \neg(\neg c \wedge \neg b \wedge \neg a)\end{array}$$

The disjunctive normal form returned is:

$$
\begin{array}{l}
a \wedge b \wedge c \vee (a \wedge \neg b \wedge \neg c) \vee (\neg a \wedge a \wedge \neg b) \vee \\
(\neg a \wedge a \wedge \neg c) \vee (\neg a \wedge a \wedge \neg b \wedge \neg c) \vee \\
(\neg a \wedge b \wedge \neg c) \vee (\neg a \wedge \neg b \wedge b) \vee (\neg a \wedge \neg b \wedge c) \vee \\
(\neg a \wedge \neg b \wedge b \wedge \neg c) \vee (\neg a \wedge \neg b \wedge \neg c \wedge c) \vee \\
(\neg a \wedge \neg c \wedge c) \vee (\neg b \wedge b \wedge \neg c) \vee (\neg b \wedge \neg c \wedge c)
\end{array}
$$

The resulting formula is longer but much easier to manipulate than the original formula. Note that the **dnf** commands sorts variables into alphabetical order and removes duplicates. It does not, however, remove terms in the form $A \wedge \neg A$.

6.1.2 PALM Could be More Rigorous

I have described the general purpose features of PALM, a tool for manipulating logical formulas. PALM remembers formulas, automatically applies rules, and handles other bookkeeping tasks. The aim in developing PALM was to provide a tool which allowed the user to quickly explore the results of manipulating formulas.

There are several ways in which the PALM system could be more rigorous. For example, PALM could start with a small set of axioms and require that additional rules be derived from these. At present, a user can enter any rule she wishes. If the user enters an incorrect rule, the results derived may not be valid.

In addition, a type system could be added. Variables in PALM do not have types, so there is no type checking. As a result of a lack of types, case analysis cannot automatically generate the possible values of a variable. With a type system, case analysis would be more soundly based.

Systems which provide a more formal approach to logic manipulation include HOL [Gor85b] and VERITAS [HD86a]. My approach is not inconsistent with the approach used by these more formal systems. For example, the rules I use to rewrite formulas can be checked for consistency by generating them from a small set of axioms.

6.2 Rules for Reasoning about Circuits

In this section I discuss the use of the rule database of PALM for reasoning about circuits. The rules are divided into four categories: basic logic rules, rules for signal strengths, rules for interval temporal logic, and rules for combining circuit elements. These rules are listed in Appendix A.

Basic logic rules examine the value field of a signal only, and are familiar from boolean logic. These include rules such as the commutativity and associativity of the operator $\wedge$ (*and*), and DeMorgan's laws. Most variables are signals. For convenience, I represent the truth values *true* and *false* as the signals $\{1, \_\}$ and $\{0, \_\}$ respectively.

Rules for signal strengths examine the strength field of a signal only. These include rules for strengthen and weaken, and properties of the $\sqcup$ (join) operator described in Section 4.4.2.

Very few rules are required for temporal logic operators:

$$\begin{aligned}
&(A \rightarrow B) \wedge (C \rightarrow B) = (A \sqcup C \rightarrow B)\\
&(\Box A) \wedge (\Box B) = (\Box A \wedge B)\\
&(\Box A \wedge B) = (\Box A) \wedge (\Box B)
\end{aligned}$$

The first rule is the property of the temporal assignment operator discussed in section 4.4. The last two rules are properties of the $\Box$ operator from linear time temporal logic (LTTL). These and many similar rules are proved from an axiomatization of LTTL in [MP83].

Rules for Combining Circuit Elements

The temporal logic equations for describing the behavior of logic elements and transistors all have the same general form. For example, an element with input X, output Y, and enable A has the form:

$$\Box(((A = \{1, \_\}) \wedge \mathit{len}\ M) \quad \supset \quad (\mathsf{strengthen}(X) \rightarrow Y))$$

A similar clocked element with clock Φ has the general form:

$$\Box(((A = \{1, \_\}) \wedge \Phi = \{1, \_\} \wedge \mathit{len}\ M) \quad \supset \quad (\mathsf{strengthen}(X) \rightarrow Y))$$

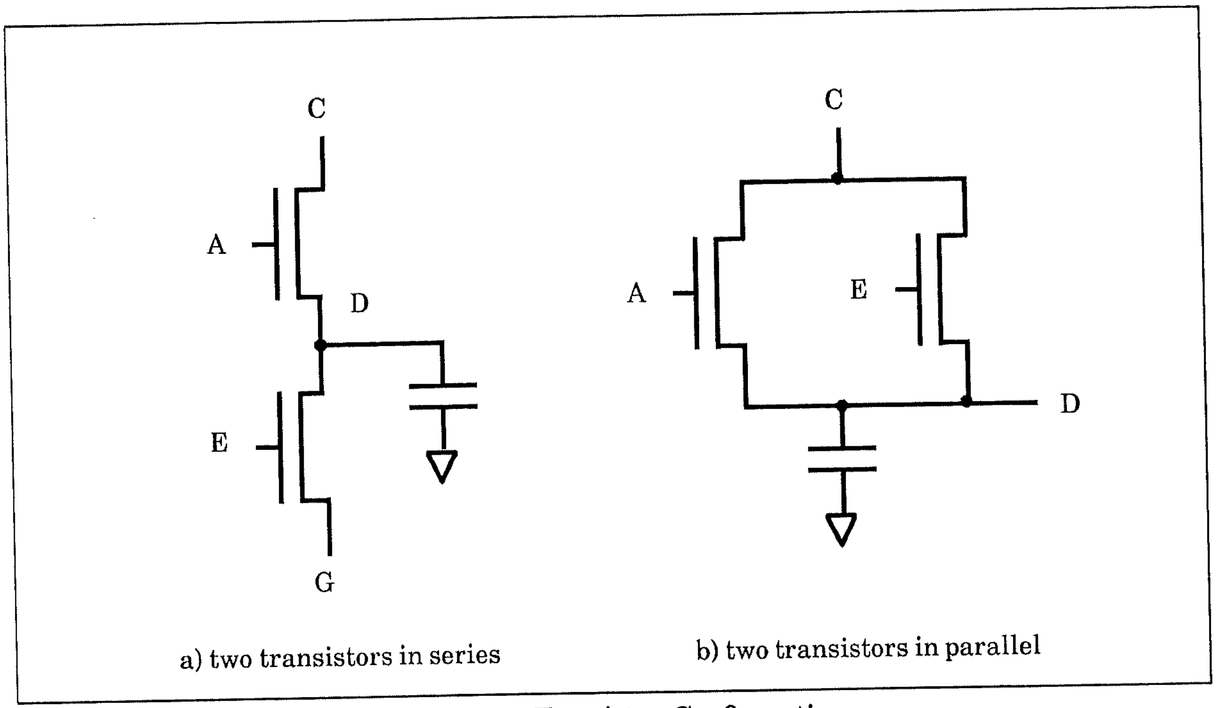


Figure 6.5: Transistor Configurations

Frequently the element is also modeled with an ideal capacitance on the output:

$$\Box(len\ 1 \supset (\mathsf{weaken}\ Y \rightarrow Y))$$

I have explicit rules for combining these formulas. These rules are based on the compositionality of the ITL delay operator, and on properties of temporal stability.

Figure 6.5 shows two transistors connected in series and two transistors connected in parallel. The rule for connecting two unclocked transistors in series is:

$$\begin{array}{l}\Box((A \wedge len\ M \supset (C \rightarrow D)) \wedge \\ \quad (E \wedge len\ N \supset (D \rightarrow G)) \wedge \\ \quad (len\ 1 \supset (\mathsf{weaken}\ D \rightarrow D))) \quad \supset \\ \Box(A \wedge E \wedge len\ (M+N) \supset (C \rightarrow G))\end{array}$$

The rule for connecting two unclocked transistors in parallel is:

$$\begin{array}{l}\Box((A \wedge len\ M \supset (C \rightarrow D)) \wedge \\ \quad (E \wedge len\ N \supset (C \rightarrow D))) \quad \supset \\ \Box(A \vee E \wedge len\ max(M, N) \supset (C \rightarrow D))\end{array}$$

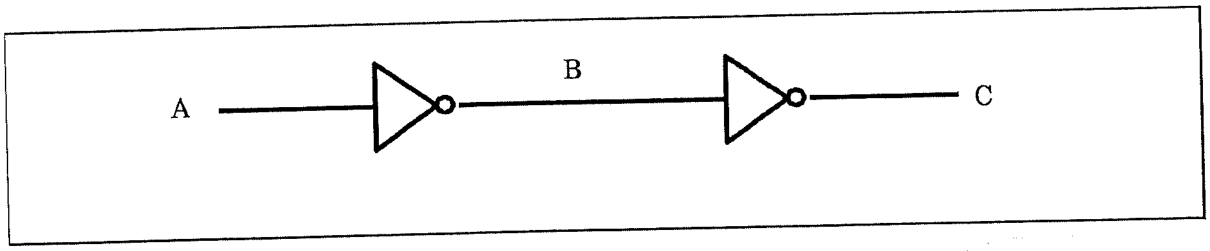


Figure 6.6: Two Inverters in Series

The rule for connecting an unclocked and a clocked logical element in series is:

$$\begin{array}{l}\Box((\mathit{len}\ M \supset (B \rightarrow C)) \wedge \\ \quad ((\Phi = \{1, \_\}) \wedge \mathit{len}\ N \supset (C \rightarrow G)) \wedge \\ \quad (\mathit{len}\ 1 \supset (\mathsf{weaken}\ C \rightarrow C))) \quad \supset \\ \Box((\Phi = \{1, \_\}) \wedge \mathit{len}\ N \supset (B \sqcup \mathsf{weaken}\ C \rightarrow G))\end{array}$$

ITL delay has the property that functional composition applies. Since Prolog does not implement higher order unification, functional composition is not automatically handled. To get around this I replace the functions in question with explicit function names, apply a rule which uses those names, and then replace the names with the original functions.

For example, suppose a circuit has two inverters in series as shown in Figure 6.6.

The behavior of such a circuit is that the output C is the input A delayed by the sum of the delays of the components. In Section 5.3.1, the behavior of the inverter is shown to be:

$$\Box((\mathit{len}\ N) \supset (\mathsf{strengthen}(\neg In)) \rightarrow Out)$$

Composing the behavior of the two inverters results in the equation:

$$\begin{array}{l}\Box((\mathit{len}\ M) \supset (\mathsf{strengthen}(\neg A)) \rightarrow B) \wedge \\ \quad ((\mathit{len}\ N) \supset (\mathsf{strengthen}(\neg B)) \rightarrow C)\end{array}$$

The rule to apply is:

$$\begin{array}{l}\Box((\mathit{len}\ M \supset (\mathit{func1}(B) \rightarrow C)) \wedge \\ \quad (\mathit{len}\ N \supset (\mathit{func2}(C) \rightarrow E))) \quad \supset \\ \Box(\mathit{len}\ (M + N) \supset (\mathit{func2}(\mathit{func1}(B)) \rightarrow E))\end{array}$$

In order to apply this rule to the behavioral equation, first I must replace strengthen$(\neg A)$ with $func1(A)$ and strengthen$(\neg B)$ with $func2(B)$, then apply the rule, and then replace $func1$ and $func2$ with strengthen $\neg$. The result is:

$$\Box(len\ (M+N) \supset ((\mathsf{strengthen} \neg\ \mathsf{strengthen}\ (\neg A)) \rightarrow C))$$

Using rules to commute the strengthen and $\neg$ operators, and the rules:
strengthen strengthen $A \Leftrightarrow$ strengthen A and $\neg\neg A = A$ results in:

$$(len\ (M+N) \supset (\mathsf{strengthen}\ A \rightarrow C))$$

This is the temporal logic description of the expected behavior for this component.

6.3 Conclusions

I have provided a general introduction to the PALM system and discussed PALM rules used to reason about circuits. In the next chapter I present several examples which illustrate how PALM is used to derive the behavior of circuits.

Chapter 7

Examples

In this chapter, I present several examples of reasoning about circuits using the methodology introduced in the previous chapters. I show how the function and timing behavior of a circuit is derived from the behavior of its components and how constraints are reasoned about.

In the first section I discuss the derivation of behavior of several fully complementary CMOS circuits. These include a shift register and a dynamic latch. Both these circuits are composed of shiftstages as well as other components.

In the second section I discuss the derivation of functional and timing behavior of a dynamic 1-bit adder. This circuit was designed using the NORA design style [GDM83], and is built out of dynamic and clocked CMOS logic blocks. I show how the derived behavior of this adder can be abstracted to a purely functional behavior, and how the abstract behavior is used in a proof of an n-bit adder.

7.1 Examples using CMOS Complementary Logic

I will show how the behavior of the CMOS shift register described in Section 5.4 and a CMOS dynamic latch can be derived from their components using PALM. A component of both these circuits is the shiftstage shown in Figure 7.2. First I discuss the derivation of behavior of the shiftstage, then I show how this behavior can be used to derive the behavior of the shift register and the dynamic latch.

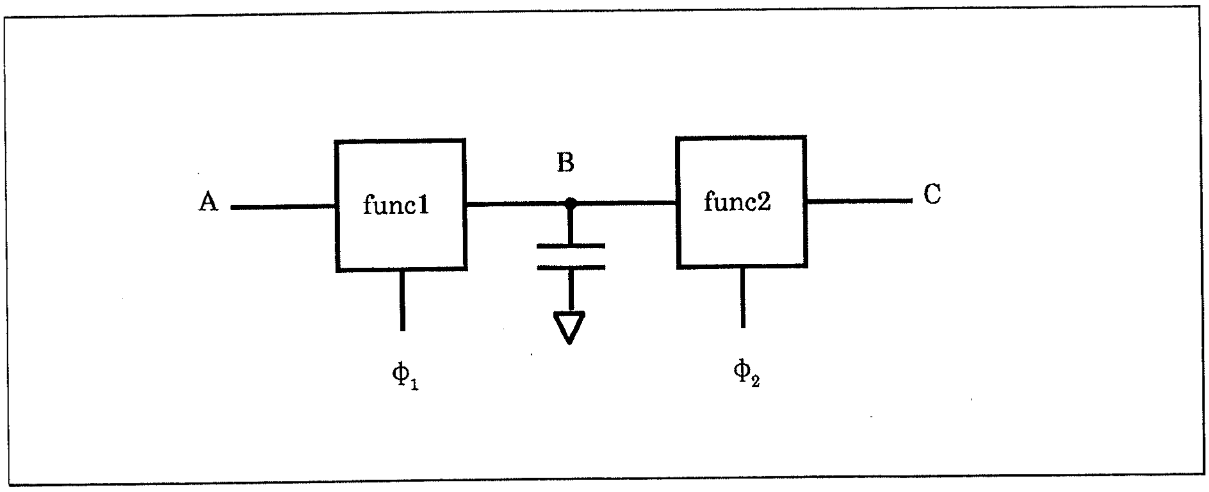


Figure 7.1: An Element which Employs 2-phase Clocking

The shift register and dynamic latch both employ a 2-phase clocking scheme. A generalized element using such a scheme is shown in Figure 7.1. The following rule is used for connecting elements which are clocked on different clock phases.

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

Note that the case $(\Phi_1 = \{1,\_\}) \wedge (\Phi_2 = \{1,\_\})$ does not arise.

This rule states that if two elements are connected in series, with the first element clocked by Φ_1 and the second element clocked by Φ_2, and the node between them is modeled with an ideal capacitance, then the behavior of the combined circuit element is the composed functional behavior of the two elements. The value of the input is latched on Φ_1. This is specified by the function $\mathsf{latched}(\Phi_1, A))$, which returns the value A had when Φ_1 was $\{1,\_\}$. The delay is the delay from Φ_2 characteristic of the second element.

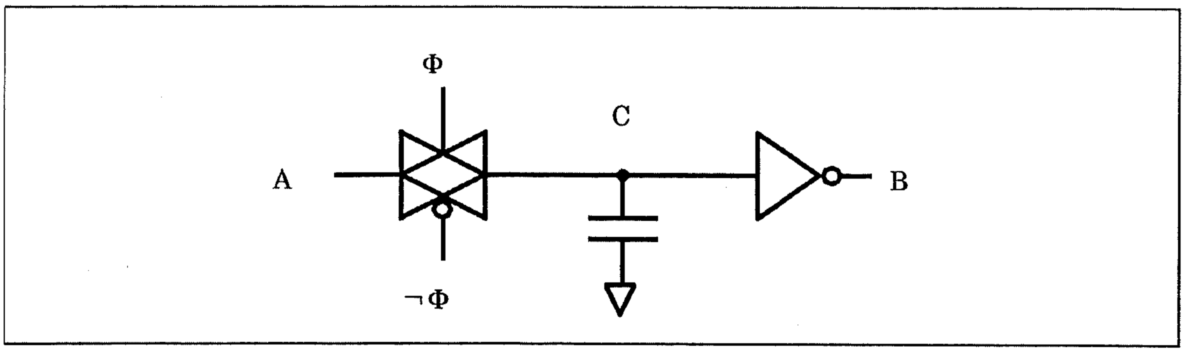


Figure 7.2: A shiftstage

7.1.1 A shiftstage

The behavior of a shiftstage, shown in Figure 7.2 is described in ITL by:

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

There is one constraint on the behavior of the shiftstage:

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

This behavior and constraints have been derived from the behavior and constraints of the components of shiftstage using the PALM system. The derivation is presented in Appendix B.4.

7.1.2 A Shift Register

A CMOS shift register, shown in Figure 7.3 is made up of two shiftstages:

$$\begin{aligned}&\mathsf{shift\text{-}reg\text{-}struct}(A, B, \Phi_1, \Phi_2, M) \quad \equiv_{\text{def}}\\ &\quad \mathsf{shiftstage}(A, c, \Phi_1, n) \wedge \mathsf{shiftstage}(c, B, \Phi_2, p)\end{aligned}$$

Here M is the time after Φ_2 becomes *true* that the output of the shift register is valid. It is straightforward to show that the behavior of the shift register, derived from the behavior of the shiftstage given above is:

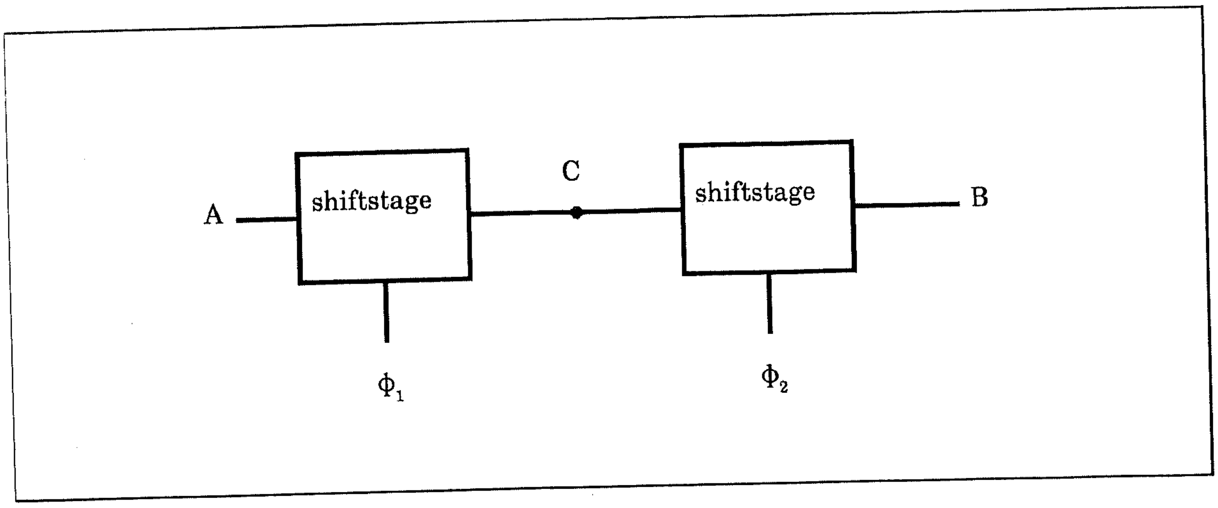


Figure 7.3: A Shift Register

$$\begin{array}{lll} (\Box(\Phi_2 = \{1,\_\}) \wedge \mathit{len}\ p & \supset & \textsf{strengthen latched}(\Phi_1, A) \rightarrow B) \wedge \\ (\Box\neg(\Phi_2 \approx \{1,\_\}) \supset \mathit{stb}\ B) \wedge & & \\ M = p & & \end{array}$$

This behavior is derived by expanding the behavior of the shiftstages, and applying several rules including the rule for elements with 2-phase clocks given above. Note that the input A is latched on clock phase Φ_1.

The constraints from the two shift stages are:

$$\begin{array}{lll} \Box((\Phi_1 \approx \{1,\_\}) & \supset & \mathit{stb}\ A) \\ \Box((\Phi_2 \approx \{1,\_\}) & \supset & \mathit{stb}\ c) \end{array}$$

The behavior of the first shift stage implies:

$$\Box((\neg\Phi_1 \approx \{1,\_\}) \quad \supset \quad \mathit{stb}\ c)$$

A property of the clocking scheme is:

$$\Box((\Phi_2 \approx \{1,\_\}) \quad \supset \quad (\neg\Phi_1 \approx \{1,\_\}))$$

The second constraint can be derived from these two implications using the rule:

$$((A \supset B) \wedge (B \supset C)) \quad \supset \quad (A \supset C)$$

Thus the second constraint is always satisfied. Therefore, the constraint for the shift register is the constraint on the behavior of the input A.

7.1.3 A Dynamic Latch

Figure 7.4 shows a CMOS dynamic latch. This circuit is a clocked version of the dynamic register discussed in section 3.3.1. The operation of this frequently used CMOS structure is described in Section 5.4 of [WE85]. The latch uses 2-phase clocking similarly to the CMOS shift register. When the latch signal L is high during Φ_1, a new value is stored in the latch. Otherwise, the previous value is saved.

The temporal logic description of the behavior of the latch is derived from the behavior of its parts. The full derivation is given in Appendix B.5. A sketch of the derivation is given below.

The structure of the latch is:

$$\begin{array}{l} \mathsf{dlatch\text{-}struct}(L, D, Q, \Phi_1, \Phi_2, R) \quad \equiv_{\mathrm{def}} \\ \qquad \mathsf{invert\text{-}mux}(L, D, Q, \Phi_1, y, m) \wedge \\ \qquad \mathsf{shiftstage}(y, Q, \Phi_2, p) \end{array}$$

Here invert-mux is a clocked multiplexer composed with a static inverter. This module is indicated in Figure 7.4. Its behavior, derived from the behaviors of its components, is:

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

To derive the behavior of the latch, we first expand the behaviors of the parts, and then apply PALM rules. Once again, the 2-phase clocking rule is used. It is straightforward to show that the behavior of the components implies the behavior of the latch, given in the following definition:

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

The constraints of the parts for the dynamic latch are:

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

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

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

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

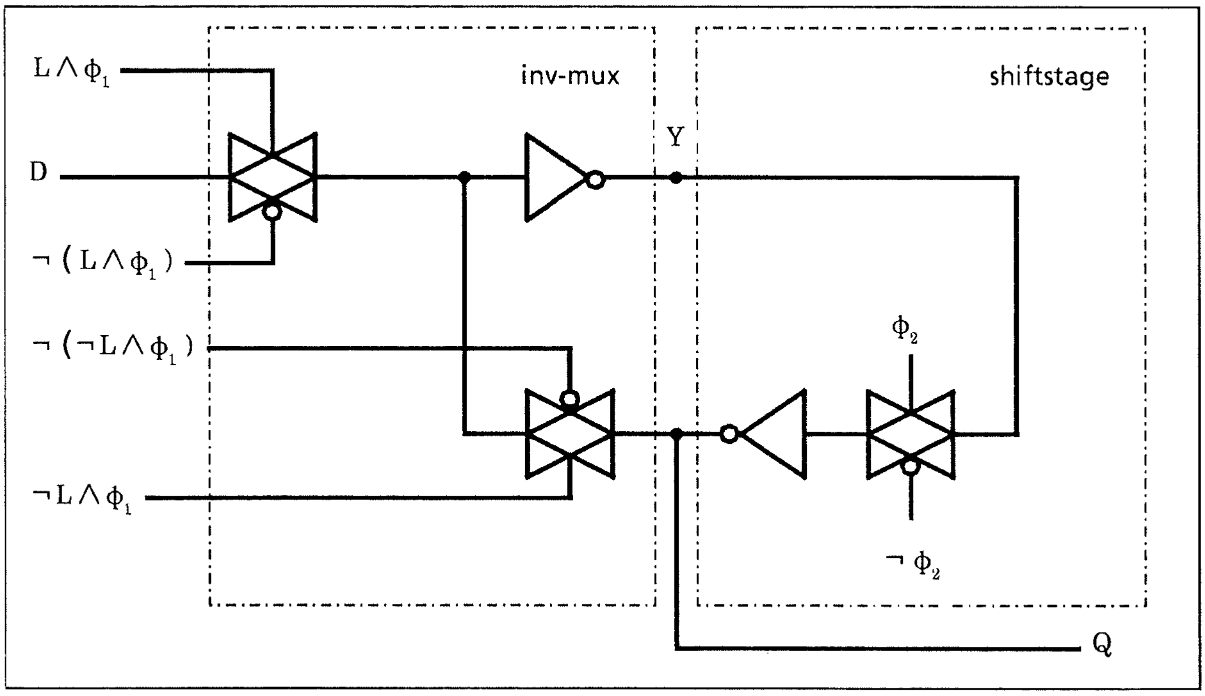


Figure 7.4: A CMOS Dynamic Latch

7.2 A Dynamic CMOS Full Adder

The examples discussed in the previous section employ a fully complementary CMOS design style. Many other logic structures, such as dynamic and clocked CMOS, are used in actual designs. These structures combine the low power advantage of complementary CMOS circuits with the area advantage of nMOS technology. For a more thorough discussion of the advantages and disadvantages of different design methodologies see Section 5.2 of [WE85].

In this section I describe a one bit adder circuit which makes use of the NORA design style [GDM83]. The adder circuit is shown in Figure 7.5; an earlier version was presented in [Mur84]. The top level Prolog specification of the adder is given below. A $\overline{}$ over a term indicates the negation of that term.

$$
\begin{aligned}
&\text{adder-struct}(A, B, C, \overline{Carry}, \overline{Sum}, \Phi_1, \Phi_2, X) :- \\
&\qquad \text{ccmos-invert}(\Phi_1, A, \overline{a}, x_1), \\
&\qquad \text{ccmos-invert}(\Phi_1, B, \overline{b}, x_2), \\
&\qquad \text{ccmos-invert}(\Phi_1, C, \overline{c}, x_3), \\
&\qquad \text{sumpart}(\overline{a}, \overline{b}, \overline{c}, e, g, \overline{\Phi_1}, x_4), \\
&\qquad \text{carrypart}(\overline{a}, \overline{b}, \overline{c}, e, \Phi_1, x_5), \\
&\qquad \text{ccmos-invert}(\Phi_2, e, \overline{Carry}, x_6), \\
&\qquad \text{ccmos-invert}(\Phi_2, g, \overline{Sum}, x_7).
\end{aligned}
$$

First I give a brief introduction to the NORA design style and describe how the adder operates. Next I discuss the results of running the direction finder on the adder. Then I describe how the function and timing of this component is reasoned about using the PALM system. I show how this behavior can be abstracted, and how the abstract behavior is used in a proof of an n-bit adder. Finally I discuss issues raised by this example.

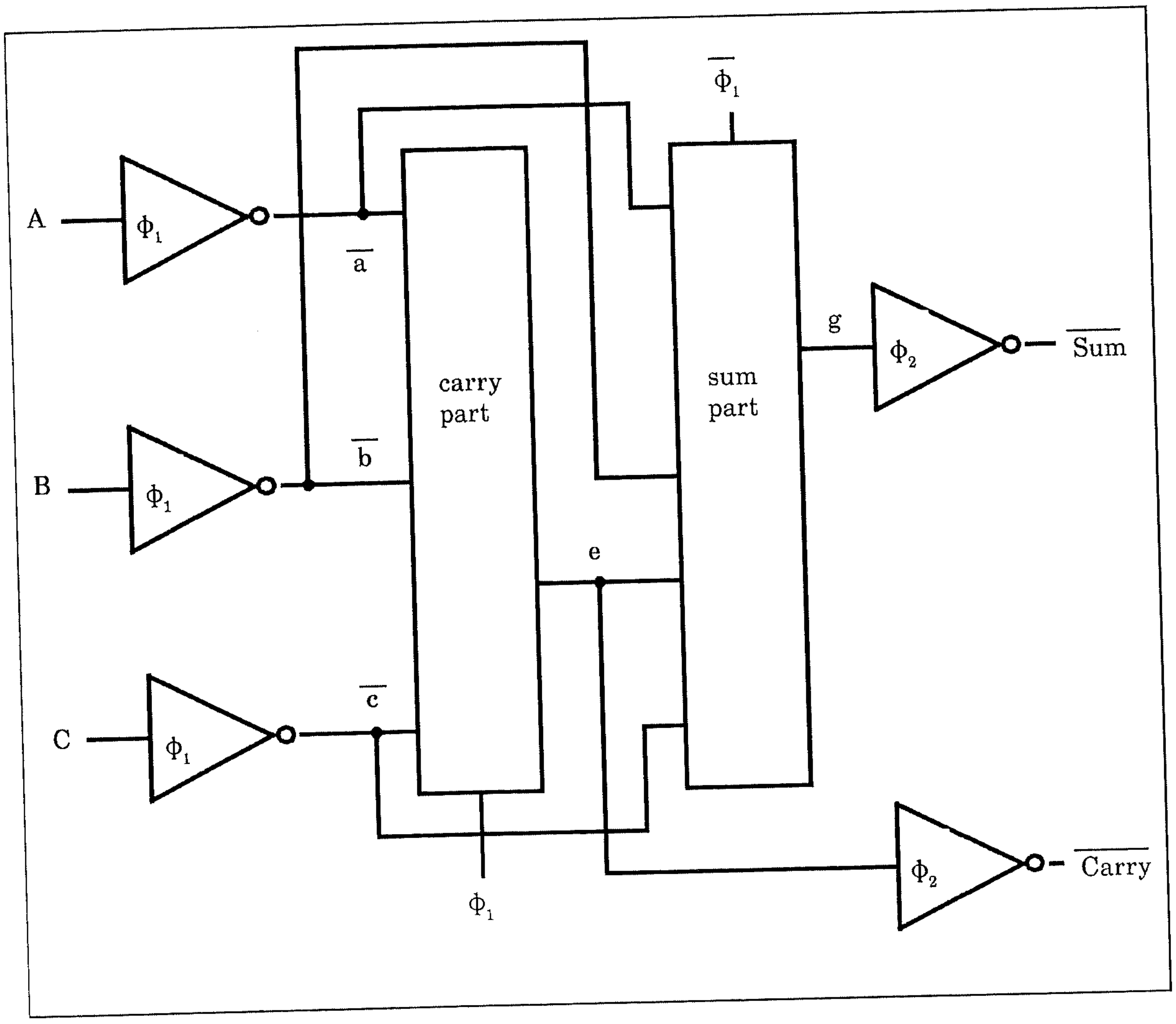


Figure 7.5: A Dynamic 1-bit Adder

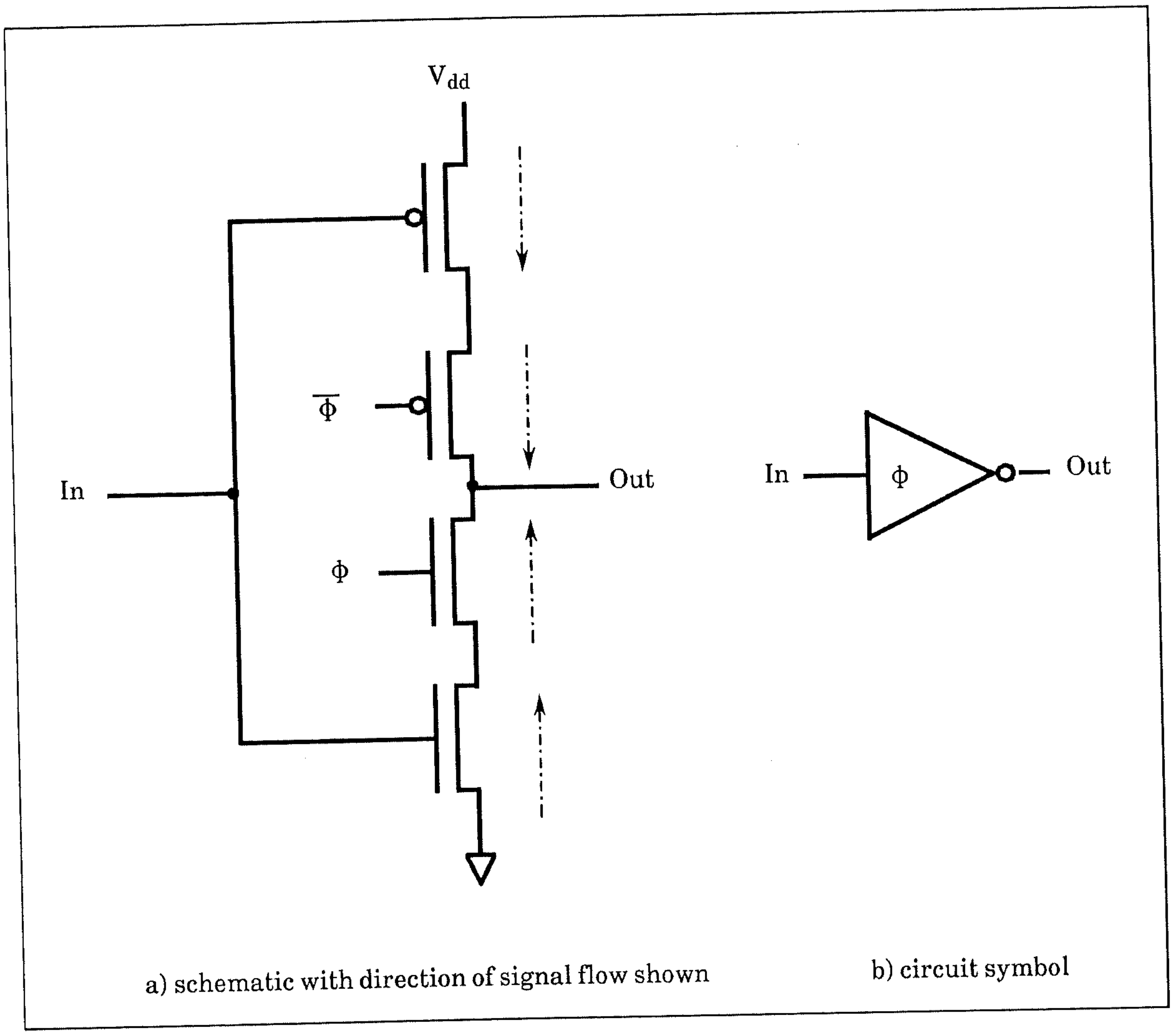


Figure 7.6: A C^2MOS Inverter

7.2.1 Operation of the Dynamic Adder

The dynamic adder shown in Figure 7.5 is built out of dynamic CMOS and C^2MOS (Clocked CMOS) structures. Dynamic and C^2MOS components use clock signals to provide separate phases of operation. The inverter, shown in Figure 7.6, is an example of a C^2MOS circuit. When Φ is high the circuit operates similarly to a static inverter. When Φ is low the output of the inverter is isolated from its inputs, and the output retains its previous value. Thus the inverter works as a combination inverter and latch; its output changes when Φ is high, and is latched when Φ is low.

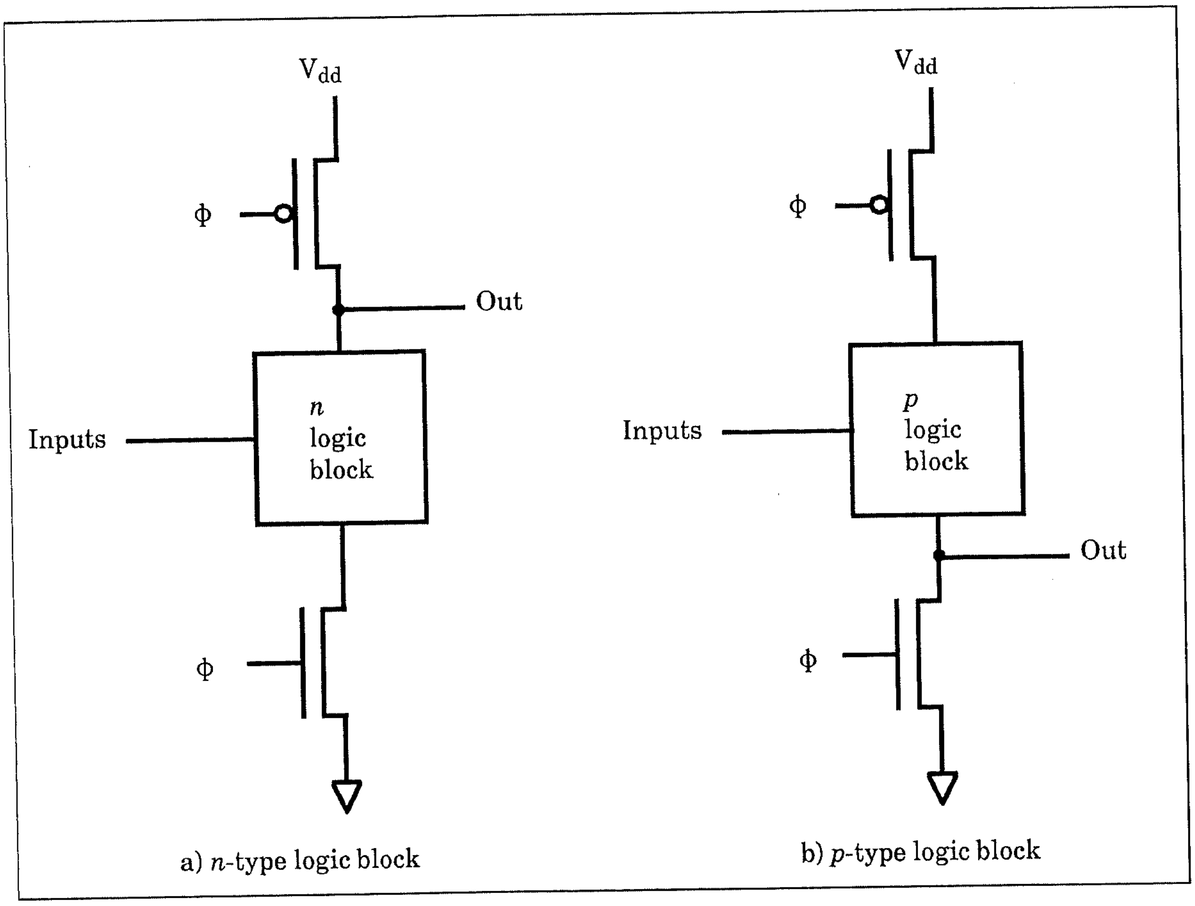
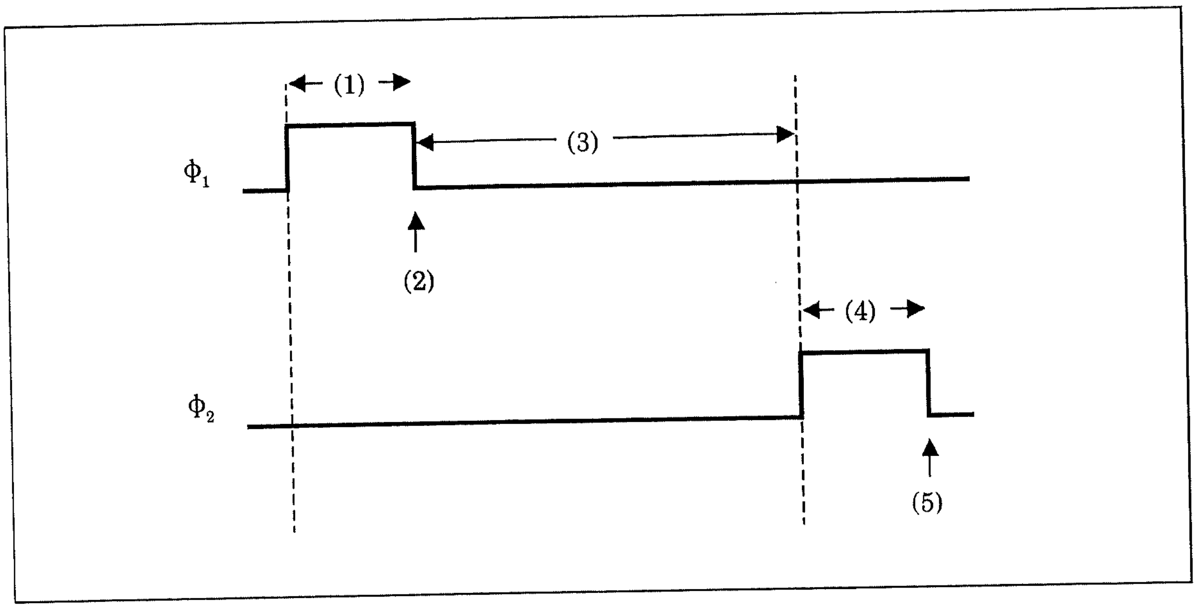


Figure 7.7: Dynamic CMOS Logic

Examples of dynamic CMOS structures are the carrypart and sumpart of the adder. General forms of n-type and p-type dynamic logic blocks are shown in Figure 7.7.

An n-type logic block consists of a logic structure built out of n-type transistors whose output node is precharged to power by a p-type transistor and conditionally discharged or evaluated by an n-type transistor connected to ground. In the p-type logic structure, an n-type transistor precharges to ground, and the p-type logic block is evaluated by a p-type transistor connected to power. Dynamic logic operates subject to the constraint that any input transition occurs at most once during the evaluation phase. For an n-type logic block, this transition must be from 0 to 1. For a p-type block, the transition must be from 1 to 0.



(1) precharge e, g; evaluate $\overline{a}$, $\overline{b}$, $\overline{c}$
(2) latch $\overline{a}$, $\overline{b}$, $\overline{c}$
(3) evaluate e, g
(4) evaluate $\overline{Sum}$, $\overline{Carry}$
(5) latch $\overline{Sum}$, $\overline{Carry}$

Figure 7.8: One Cycle of Operation of the Dynamic Adder

In NORA, logic functions are implemented using n-type and p-type dynamic CMOS and C^{2}MOS blocks. NORA provides rules for combining these blocks. The adder circuit shown in Figure 7.5 is made up of C^{2}MOS inverters, and n-type and p-type dynamic logic blocks composed according to the NORA rules. This circuit employs the 2-phase clocking scheme presented in Section 5.5. The following description of the operation of the adder is illustrated by the annotated timing diagram in Figure 7.8. The diagram shows clocks Φ_1 and Φ_2 only. The complementary clocks $\overline{\Phi_1}$ and $\overline{\Phi_2}$ have been omitted for clarity. Numbers in parentheses in this discussion correspond to labels in the figure.

The inputs A, B, and C are latched by C^{2}MOS inverters which evaluate when Φ_1 is high (1) and latch when Φ_1 is low (2). The carrypart is an n-type dynamic logic block which precharges when Φ_1 is high (1) and evaluates when Φ_1 is low (3). The sumpart is a p-type dynamic logic block which precharges and evaluates on the same phases of the clock as the carrypart. The outputs of the sumpart and carrypart are signals e and g respectively. These signals are in turn inputs to two C^{2}MOS inverters which are clocked by Φ_2. Thus the outputs, $\overline{Sum}$ and $\overline{Carry}$, are evaluated when Φ_2 is high (4), and are latched when Φ_2

is low (5). Note that the output of the carrypart, signal e, is also an input to the sumpart. NORA permits the cascading of two levels of logic which operate on the same phase of the clock provided that one block is of type n and the other is of type p. The clock period must be long enough to allow for the slowest such combination to evaluate.

7.2.2 Setting the Direction of Signal Flow

The direction of signal flow through transistors in the adder is set in the customary preprocessing step. This is done using the signal flow algorithm described in Section 3.3.1. Establishing the direction of signal flow through the transistors in the C^2MOS inverter is straightforward. The only set of directions which satisfies the Signal Law is that shown in Figure 7.6.

The signal flow algorithm has difficulty with the carrypart and sumpart components of the adder. Dynamic CMOS design styles do not lend themselves to this type of analysis. The algorithm is only able to set the direction of the precharge and evaluate transistors. The remaining transistors are labeled bidirectional and are set by hand. I use the convention that all signal flow is toward the output nodes. The resulting labels are shown in Figures 7.9 and 7.10. The signal flow analysis program was used to check that the assigned labels do not violate the Signal Law.

7.2.3 Reasoning about Function and Timing: the Components

In the next two sections I give a brief presentation of the derivation of the behavior for the dynamic adder.

Deriving the function and timing of the adder proceeds hierarchically and consists of reasoning about behavior and reasoning about constraints. In this example there are two levels of hierarchy. The bottom level consists of structure of transistors which implement the C^2MOS inverter, and the sumpart and carrypart. These components are then composed in the top level of the hierarchy to form a 1-bit dynamic adder.

First I present the bottom level derivations. I use the same transistor model to describe and reason about NORA circuits as that used for fully complementary CMOS circuits.

The constraints, however, reflect the different design methodology. The requirement that all clocks satisfy the control predicate presented in Section 5.5 remains. There are no longer constraints on individual transistors. They are replaced by constraints on the ways in which different types of logic blocks may be composed. These constraints correspond to the rules for composing NORA blocks described in [GDM83]. Dhingra [Dhi87] presents a formal proof of the rules for this design style. For the adder circuit, the only constraints which are considered here are that inputs to the n-type and p-type block are stable during the evaluate phase, and that inputs to the inverters clocked on Φ are stable on the falling edge of Φ.

The C²MOS Inverter

The implementation of the C²MOS inverter is shown in Figure 7.6. The specification for this structure is:

$$\begin{aligned}
&\mathsf{ccmos\text{-}invert\text{-}struct}(\Phi, In, Out, M) \quad \equiv_{\text{def}}\\
&\qquad \mathsf{ntrans}(In, \{0,1\}, p1, n1) \wedge \mathsf{ptrans}(In, \{1,1\}, p2, n2) \wedge\\
&\qquad \mathsf{ntrans}(\Phi, p1, Out, M) \wedge \mathsf{ptrans}(\neg(\Phi), p2, Out, M) \wedge\\
&\qquad \mathsf{cap}(p1) \wedge \mathsf{cap}(p2) \wedge \mathsf{cap}(Out)
\end{aligned}$$

The derivation of the C²MOS inverter behavior proceeds by expanding the component behavioral descriptions:

$$\begin{aligned}
&(\Box(In = \{1,\_\}) \ \wedge \ len\ n \supset (\{0,1\} \rightarrow p1)) \wedge\\
&(\Box(In = \{0,\_\}) \ \wedge \ len\ n \supset (\{1,1\} \rightarrow p2)) \wedge\\
&(\Box(\Phi = \{1,\_\}) \ \wedge \ len\ n \supset (p1 \rightarrow Out)) \wedge\\
&(\Box(\neg\Phi = \{0,\_\}) \ \wedge \ len\ n \supset (p2 \rightarrow Out)) \wedge\\
&(\Box\, len\ 1 \supset (\mathsf{weaken}\ p1 \rightarrow p1)) \wedge\\
&(\Box\, len\ 1 \supset (\mathsf{weaken}\ p2 \rightarrow p2)) \wedge\\
&(\Box\, len\ 1 \supset (\mathsf{weaken}\ Out \rightarrow Out))
\end{aligned}$$

This term is *split* to save the last conjunct on the stack. The subterm:

$$(\Box\, len\ 1 \supset (\mathsf{weaken}\ Out \rightarrow Out))$$

states that the previous value of the output is always stored. This specifies the latching behavior of the inverter when the output is isolated from the input.

The remainder of the term is manipulated to derive the behavior of the inverter when Φ is high. This is done using a combination of case analysis on In and straightforward logical manipulations. The rule for combining an unclocked and a clocked transistor in series is used:

$$\begin{aligned}
&\Box((\mathit{len}\ N \supset (A \rightarrow B)) \wedge \\
&\quad ((\Phi = \{1,\_\}) \wedge \mathit{len}\ M \supset (B \rightarrow C)) \wedge \\
&\quad (\mathit{len}\ 1 \supset (\mathsf{weaken}\ B \rightarrow B))) \quad \supset \\
&\Box((\Phi = \{1,\_\}) \wedge \mathit{len}\ M \supset \ (A \sqcup \mathsf{weaken}\ B \rightarrow C))
\end{aligned}$$

This rule states that if an unclocked and a clocked transistor are connected in series with signal flowing from the unclocked to the clocked, then the result is the same as having one transistor with the delay of the clocked transistor and the input of the unclocked. A constraint associated with this rule is that the input of the unclocked transistor is stable when the clock is active. This will be reflected in the constraint on the input of the inverter. The rule is used twice; once for the pullup portion of the inverter and once for the pulldown portion.

The result of the logical manipulations is *and*ed with the term saved on the stack. The resulting behavior for the inverter is:

$$\begin{aligned}
&\mathsf{ccmos\text{-}invert}(\Phi, In, Out, M) \quad \equiv_{\mathsf{def}} \\
&\qquad (\Box((\Phi = \{1,\_\}) \wedge \mathit{len}\ M \supset \ (\mathsf{strengthen}\ \neg In \rightarrow Out)) \wedge \\
&\qquad\quad (\mathit{len}\ 1 \supset (\mathsf{weaken}\ Out \rightarrow Out)))
\end{aligned}$$

This behavior specification states when Φ is high the circuit behaves like a static inverter, and the output always stores a *weaken*ed copy of the last driven value of the output.

The carrypart

The carrypart of the adder is shown in Figure 7.9. The specification of this component is:

$$\begin{aligned}
&\mathsf{carrypart\text{-}struct}(\overline{a},\overline{b},\overline{c},e,\Phi,X) \quad \equiv_{\mathrm{def}}\\
&\qquad \mathsf{ptrans}(\Phi,\{1,1\},d,m_1) \wedge \ \mathsf{cap}(d) \wedge\\
&\qquad \mathsf{ptrans}(\overline{a},d,g1,m_2) \wedge\\
&\qquad \mathsf{ptrans}(\overline{b},g1,e,m_3) \wedge \mathsf{cap}(g1) \wedge\\
&\qquad \mathsf{ptrans}(\overline{a},d,f,m_4) \wedge\\
&\qquad \mathsf{ptrans}(\overline{b},d,f,m_6) \wedge\\
&\qquad \mathsf{ptrans}(\overline{c},f,e,m_5) \wedge \mathsf{cap}(f) \wedge\\
&\qquad \mathsf{ntrans}(\Phi,\{0,1\},e,m_7) \wedge \mathsf{cap}(e)
\end{aligned}$$

There are three aspects of the behavior of this component which need to be demonstrated. The first is that the output is pulled down during the precharge phase. The second aspect is the delay through the carrypart during the evaluation phase. The third is that the output of the carrypart, signal e, has the value of the carry bit of the addition of the three input bits. This value corresponds to the boolean logic formula (remembering that $\overline{a}$, $\overline{b}$ and $\overline{c}$ are equivalent to $\neg a$, $\neg b$ and $\neg c$ respectively) :

$$Carry = (\neg\overline{a} \ \wedge \ \neg\overline{b}) \vee (\neg\overline{a} \ \wedge \ \neg\overline{c}) \ \vee(\neg\overline{b} \ \wedge \ \neg\overline{c})$$

We proceed by expanding the components with their temporal logic descriptions:

$$\begin{aligned}
&(\Box(\Phi=\{0,\_\}) \ \wedge \ len\, m_1 \supset (\{1,1\} \rightarrow d)) \wedge\\
&(\Box\, len\ 1 \supset (\mathsf{weaken}\ d \rightarrow d)) \wedge\\
&(\Box(\overline{a}=\{0,\_\}) \ \wedge \ len\, m_2 \supset (d \rightarrow g1)) \wedge\\
&(\Box(\overline{b}=\{0,\_\}) \ \wedge \ len\, m_3 \supset (g1 \rightarrow e)) \wedge\\
&(\Box\, len\ 1 \supset (\mathsf{weaken}\ g1 \rightarrow g1)) \wedge\\
&(\Box(\overline{a}=\{0,\_\}) \ \wedge \ len\, m_4 \supset (d \rightarrow f)) \wedge\\
&(\Box(\overline{b}=\{0,\_\}) \ \wedge \ len\, m_6 \supset (d \rightarrow f)) \wedge\\
&(\Box(\overline{c}=\{0,\_\}) \ \wedge \ len\, m_5 \supset (f \rightarrow e)) \wedge\\
&(\Box\, len\ 1 \supset (\mathsf{weaken}\ f \rightarrow f)) \wedge\\
&(\Box(\Phi=\{1,\_\}) \ \wedge \ len\, m_7 \supset (\{0,1\} \rightarrow e)) \wedge\\
&(\Box\, len\ 1 \supset (\mathsf{weaken}\ e \rightarrow e))
\end{aligned}$$

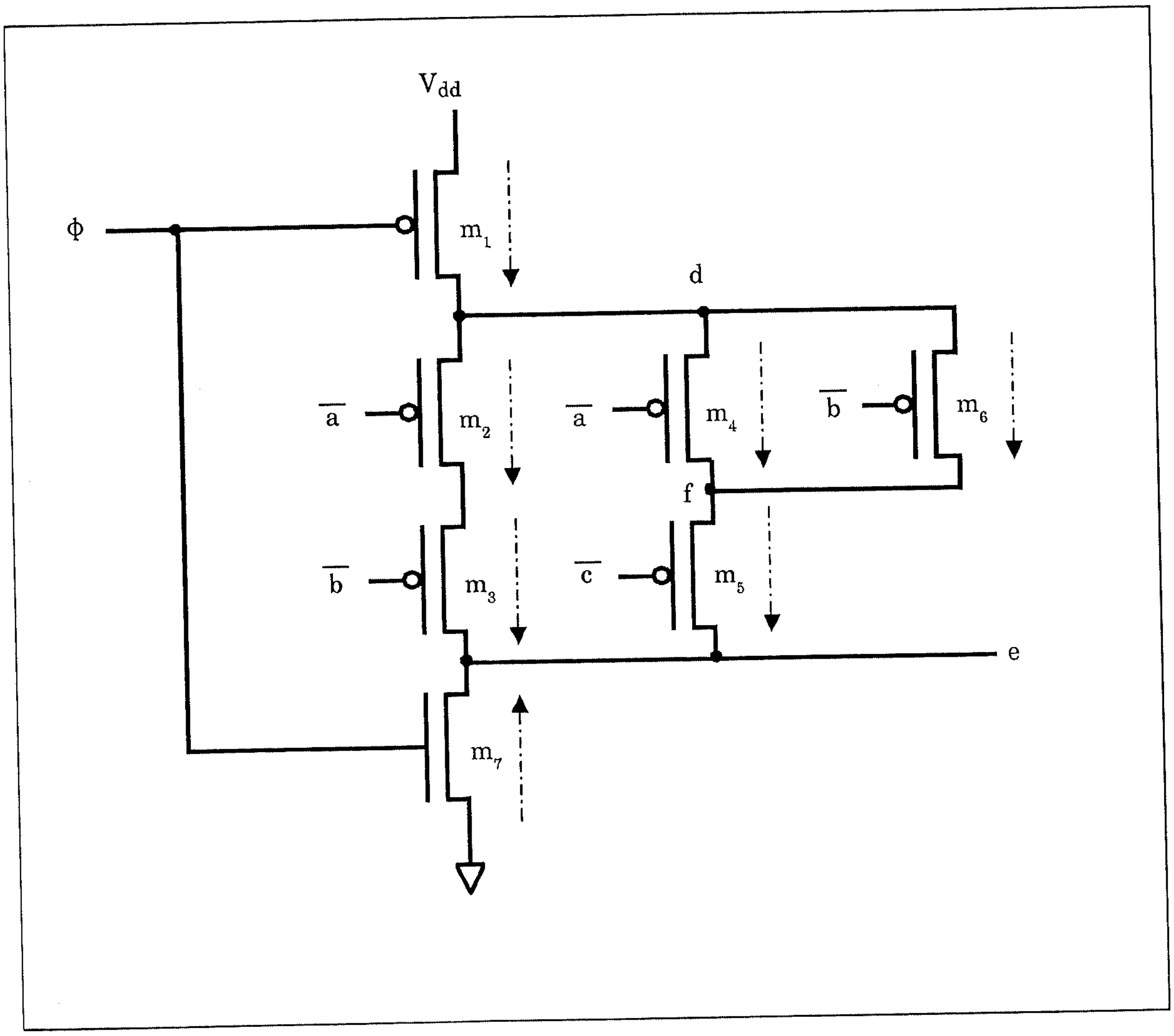


Figure 7.9: The carrypart with Direction of Signal Flow Indicated

Next we apply rules for combining transistors to simplify the behavior of the block of p-type transistors into one sub-term. This involves applying the rules for combining unclocked transistors in parallel and unclocked transistors in series discussed in Section 6.2. The simplified behavioral description is:

$$
\begin{aligned}
&\Box((\Phi = \{0,\_\}) \wedge \mathit{len}\ m_1 \supset (\{1,1\} \rightarrow d)) \wedge \\
&\quad (\mathit{len}\ 1 \supset (\mathsf{weaken}\ d \rightarrow d)) \wedge \\
&\quad ((\overline{a} = \{0,\_\}) \wedge (\overline{b} = \{0,\_\}) \vee ((\overline{a} = \{0,\_\}) \vee (\overline{b} = \{0,\_\}) \wedge (\overline{c} = \{0,\_\})) \wedge \\
&\qquad\qquad\qquad \mathit{len}\ \mathit{max}(m_2 + m_3, \mathit{max}(m_4, m_6) + m_5) \supset (d \rightarrow e)) \wedge \\
&\quad ((\Phi = \{1,\_\}) \wedge \mathit{len}\ m_7 \supset (\{0,1\} \rightarrow e)) \wedge \\
&\quad (\mathit{len}\ 1 \supset (\mathsf{weaken}\ e \rightarrow e))
\end{aligned}
$$

The derivation proceeds using case analysis on Φ. The result of deriving the behavior from carrypart-struct is the behavior specified by:

$$
\begin{aligned}
&\mathsf{carrypart}(\overline{a}, \overline{b}, \overline{c}, e, \Phi, X) \quad \equiv_{\mathsf{def}} \\
&\qquad (\Box(\Phi = \{1,1\} \supset (\mathit{len}\ m_7 \supset (\{0,1\} \rightarrow e))) \wedge \\
&\qquad\quad (\Phi = \{0,1\} \supset \ (\mathit{len}(m_1 + \mathit{max}(m_2 + m_3, \mathit{max}(m_4, m_6) + m_5)) \supset \\
&\qquad\qquad (\mathsf{strengthen}\ (\neg\overline{a} \wedge \neg\overline{b} \vee (\neg\overline{a} \wedge \neg\overline{c} \vee (\neg\overline{b} \wedge \neg\overline{c}))) \\
&\qquad\qquad\qquad \rightarrow e))) \\
&\text{where } X = (m_1 + \mathit{max}(m_2 + m_3, \mathit{max}(m_4, m_6) + m_5))
\end{aligned}
$$

This specifies the three aspects of behavior mentioned earlier. When Φ is high, signal e is pulled down to ground. When Φ is low, e has the value of the carry of the three inputs. The argument of the *len* operator is the delay of the longest path through the carrypart. The argument X of the carrypart is instantiated to this value, so the delay becomes part of the top level specification.

The sumpart

The **sumpart** of the adder is shown in Figure 7.10. The specification of this component is:

$$
\begin{array}{l}
\textsf{sumpart-struct}(\overline{a},\overline{b},\overline{c},e,g,\Phi,Y) \quad \equiv_{\mathrm{def}} \\
\quad \textsf{ntrans}(\Phi,\{0,1\},i,m_{16}) \wedge \textsf{cap}(i) \wedge \\
\quad \textsf{ntrans}(e,i,h,m_{12}) \wedge \\
\quad \textsf{ntrans}(\overline{a},h,g,m_9) \wedge \\
\quad \textsf{ntrans}(\overline{b},h,g,m_{10}) \wedge \\
\quad \textsf{ntrans}(\overline{c},h,g,m_{11}) \wedge \textsf{cap}(h) \wedge \\
\quad \textsf{ntrans}(\overline{c},i,k,m_{15}) \wedge \\
\quad \textsf{ntrans}(\overline{b},k,j,m_{14}) \wedge \textsf{cap}(k) \wedge \\
\quad \textsf{ntrans}(\overline{a},j,g,m_{13}) \wedge \textsf{cap}(j) \wedge \\
\quad \textsf{ptrans}(\Phi,\{1,1\},g,m_8) \wedge \textsf{cap}(g)
\end{array}
$$

The derivation of behavior for the **sumpart** proceeds similarly to the derivation of behavior of the **carrypart**. Note that e, the output of the **carrypart**, is an input to this component.

As with the **carrypart**, there are three aspects of the behavior of this component which need to be demonstrated. The first is that the output is pulled up during the precharge phase. The second aspect is the delay through the **sumpart** during the evaluation phase. The third is that the output of the **sumpart**, signal g, has the value of the sum bit of the addition of the three input bits. This value corresponds to the boolean logic formula:

$$
\begin{array}{rcl}
Sum & = & (\overline{a} \wedge \overline{b} \wedge \neg\overline{c}) \vee (\overline{a} \wedge \neg\overline{b} \wedge \overline{c}) \vee \\
 & & (\neg\overline{a} \wedge \overline{b} \wedge \overline{c}) \vee (\neg\overline{a} \wedge \neg\overline{b} \wedge \neg\overline{c}) \\
 & = & \neg(Carry \wedge (\overline{a} \vee \overline{b} \vee \overline{c}) \vee (\overline{a} \wedge \overline{b} \wedge \overline{c}))
\end{array}
$$

Carry is the boolean function defined above.

The derivation proceeds by expanding the components with their temporal logic descriptions. Rules for combining transistors are applied to simplify the block of n-type transistors. The resulting behavior equation is:

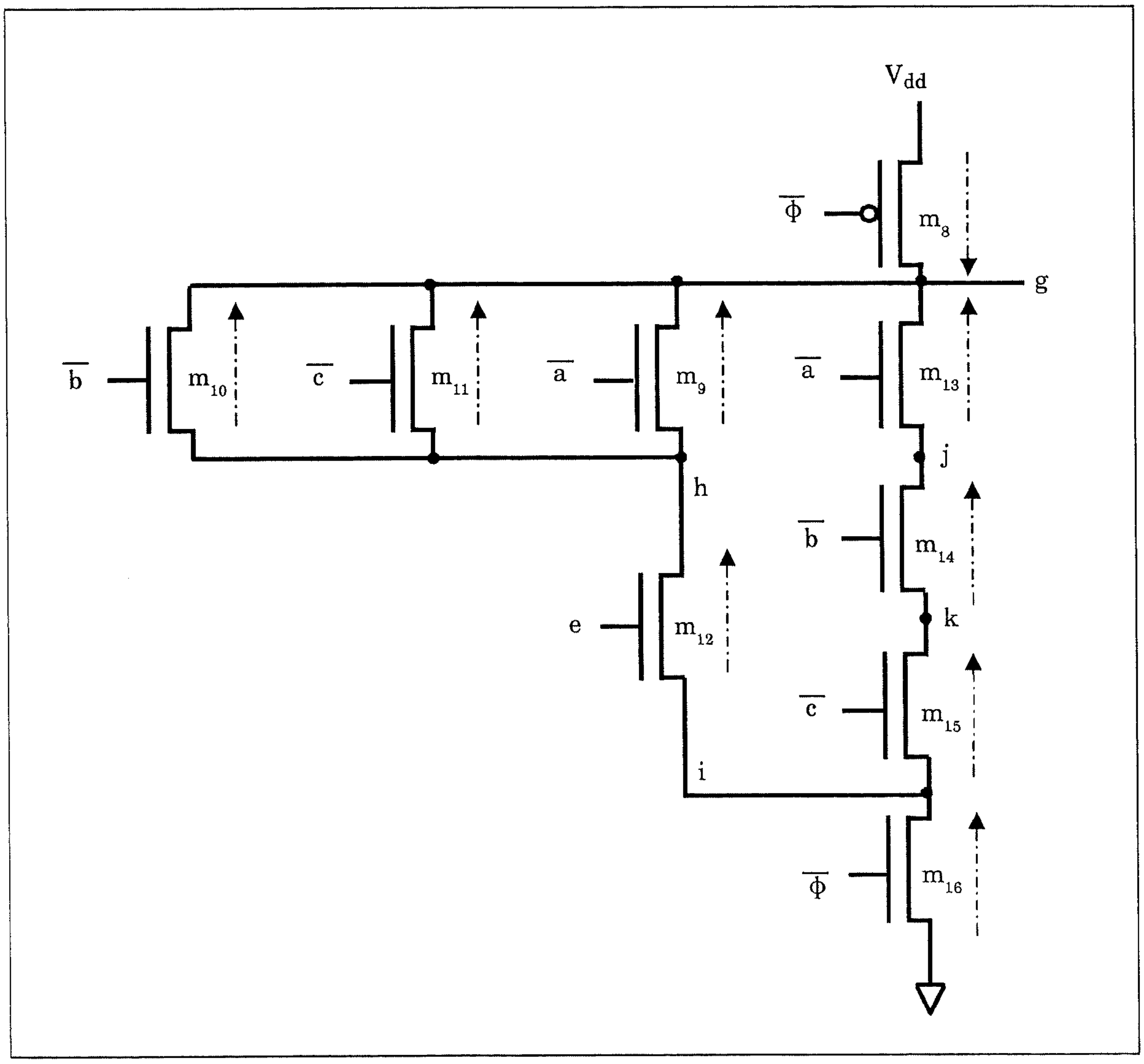


Figure 7.10: The sumpart with Direction of Signal Flow Indicated

$$
\begin{array}{l}
\Box((\Phi = \{1,\_\}) \wedge \mathit{len}\ m_{16} \supset (\{0,1\} \rightarrow i)) \wedge \\
\quad (\mathit{len}\ 1 \supset (\text{weaken } i \rightarrow i)) \wedge \\
\quad ((e = \{1,\_\}) \wedge \\
\qquad ((\overline{a} = \{1,\_\}) \vee (\overline{b} = \{1,\_\}) \vee (\overline{c} = \{1,\_\})) \vee \\
\qquad ((\overline{c} = \{1,\_\}) \wedge (\overline{b} = \{1,\_\}) \wedge (\overline{a} = \{1,\_\})) \wedge \\
\qquad\qquad \mathit{len}\ \mathit{max}(m_{12} + \mathit{max}(\mathit{max}(m_9, m_{10}), m_{11}), m_{15} + m_{14} + m_{13}) \supset \\
\qquad\qquad\quad (i \rightarrow g)) \wedge \\
\quad ((\Phi = \{0,\_\}) \wedge \mathit{len}\ m_8 \supset (\{1,1\} \rightarrow g)) \wedge \\
\quad (\mathit{len}\ 1 \supset (\text{weaken } g \rightarrow g))
\end{array}
$$

The derivation proceeds using case analysis on Φ. The result of deriving the behavior for the sumpart is the behavior specified by:

$$
\begin{array}{l}
\text{sumpart}(\overline{a}, \overline{b}, \overline{c}, e, g, \Phi, Y) \quad \equiv_{\text{def}} \\
\quad \Box(\Phi = \{0,1\} \supset (\mathit{len}\ m_8 \supset (\{1,1\} \rightarrow g))) \wedge \\
\qquad (\Phi = \{1,1\} \supset \\
\qquad\quad (\mathit{len}\ (m_{16} + \mathit{max}(m_{12} + \mathit{max}(\mathit{max}(m_9, m_{10}), m_{11}), m_{15} + m_{14} + m_{13})) \supset \\
\qquad\qquad (\text{strengthen } \neg(e \wedge (\overline{a} \vee \overline{b} \vee \overline{c}) \vee (\overline{c} \wedge \overline{b} \wedge \overline{a})) \\
\qquad\qquad\quad \rightarrow g))) \\
\text{where } Y = (m_{16} + \mathit{max}(m_{12} + \mathit{max}(\mathit{max}(m_9, m_{10}), m_{11}), m_{15} + m_{14} + m_{13}))
\end{array}
$$

This specifies the three aspects of behavior mentioned earlier. When Φ is low, signal g is pulled up to power. When Φ is high, g has the value of the sum of the three inputs. The argument of the second *len* operator is the delay of the longest path through the sumpart. The argument Y of the sumpart is instantiated to this value, so the delay becomes part of the top level specification.

7.2.4 Composing the Components

In the previous section I showed how the behavior of the components of the adder are derived from the behavior of their components. In this section I compose these components to derive the behavior of the dynamic adder. Up to this point, characteristics of the clocking scheme have not been entered into the derivation. When the components are composed, the clocking scheme is taken into account. Constraints are also examined at this level of the hierarchy. First the behavior of the adder is derived, and then constraints are discussed.

Deriving Behavior

The specification of the structure of the adder shown in Figure 7.5 is:

$$\begin{aligned}
&\mathsf{adder\text{-}struct}(A, B, C, \overline{Carry}, \overline{Sum}, \Phi_1, \Phi_2, X) \quad \equiv_{\mathrm{def}}\\
&\qquad \mathsf{ccmos\text{-}invert}(\Phi_1, A, \overline{a}, x_1) \land\\
&\qquad \mathsf{ccmos\text{-}invert}(\Phi_1, B, \overline{b}, x_2) \land\\
&\qquad \mathsf{ccmos\text{-}invert}(\Phi_1, C, \overline{c}, x_3) \land\\
&\qquad \mathsf{carrypart}(\overline{a}, \overline{b}, \overline{c}, e, \Phi_1, x_4) \land\\
&\qquad \mathsf{sumpart}(\overline{a}, \overline{b}, \overline{c}, e, g, \underline{(\neg\Phi_1)}, x_5) \land\\
&\qquad \mathsf{ccmos\text{-}invert}(\Phi_2, g, \overline{Sum}, x_6) \land\\
&\qquad \mathsf{ccmos\text{-}invert}(\Phi_2, e, \overline{Carry}, x_7)
\end{aligned}$$

This circuit demonstrates the advantage of hierarchical analysis. The behavior of the C^{2}MOS inverter component, which is used five times in this circuit, only needs to be derived once.

As usual, the derivation proceeds by replacing the components with their behavioral specifications. The specifications used are those derived in the previous section. The rule **latch**, given below, is used to derive the value of the output of the C^{2}MOS inverters used on the inputs when Φ is low. The rule states that when the clock, Φ, is low, the output of the inverter returns a weakened version of the function of its input, A, when the clock was high. This rule is applied three times, once for every inverter clocked by Φ_1:

$$\begin{aligned}
&\Box(((\Phi = \{1, \_\}) \land \mathit{len}\ N \supset (\mathit{func1}(A) \rightarrow B)) \land\\
&\quad (\mathit{len}\ 1 \supset (\mathsf{weaken}\ B \rightarrow B))) \supset\\
&\Box((\Phi = \{0, \_\}) \supset D = \mathsf{weaken}\ \mathit{func1}(\mathsf{latched}(\Phi_1, A))
\end{aligned}$$

For convenience, the term $\mathsf{latched}(\Phi_1, A)$ is abbreviated to A_1 for the remainder of this discussion. B_1 and C_1 are similar abbreviations.

Most of the derivation consists of boolean logic manipulations. The **dnf** command is used to simplify the boolean formulas for the carry and sum outputs. The rule which is used to combine elements states that, for temporal logic formulas describing delays, functional composition applies and the quantitative delays add:

$$\begin{aligned}
&\Box(((\mathit{len}\ M) \supset (\mathit{func1}(A) \rightarrow B)) \land\\
&\quad ((\mathit{len}\ N) \supset (\mathit{func2}(B) \rightarrow C))) \quad \supset\\
&\Box((\mathit{len}\ (M + N)) \supset (\mathit{func2}(\mathit{func1}(A)) \rightarrow C))
\end{aligned}$$

Reasoning proceeds by case analysis on Φ_1 and Φ_2. There are three possible states for the clocks. They correspond to the case when both clocks are low, and the two cases when one clock is high and the other is low. The case where both clocks are high simultaneously should not arise in practice and is not considered. The resulting behavior for the possible states of the clock is:

$$
\begin{array}{l}
(\Box\, \Phi_1 = \{0,1\} \wedge \Phi_2 = \{0,1\} \supset \\
\quad (\mathit{len}\, m_{carry} \supset \text{ strengthen } \mathit{func2}(A_1, B_1, C_1) \rightarrow e) \wedge \\
\quad (\mathit{len}\, m_{sum} \supset \text{ strengthen } \mathit{func1}(A_1, B_1, C_1) \rightarrow g) \wedge \\
\quad (\mathit{len}\, 1 \supset (\text{weaken } \overline{Sum} \rightarrow \overline{Sum})) \wedge \\
\quad (\mathit{len}\, 1 \supset (\text{weaken } \overline{Carry} \rightarrow \overline{Carry}))) \wedge \\
(\Box\, \Phi_1 = \{0,1\} \wedge \Phi_2 = \{1,1\} \supset \\
\quad (\mathit{len}\, x_6 \supset \text{ strengthen } \mathit{func1}(A_1, B_1, C_1) \rightarrow \overline{Sum}) \wedge \\
\quad (\mathit{len}\, 1 \supset (\text{weaken } \overline{Sum} \rightarrow \overline{Sum})) \wedge \\
\quad (\mathit{len}\, x_7 \supset (\text{strengthen} \mathit{func2}(A_1, B_1, C_1) \rightarrow \overline{Carry})) \wedge \\
\quad (\mathit{len}\, 1 \supset (\text{weaken } \overline{Carry} \rightarrow \overline{Carry}))) \wedge \\
(\Box\, \Phi_1 = \{1,1\} \wedge \Phi_2 = \{0,1\} \supset \\
\quad (\mathit{len}\, m_7 \supset (\{0,1\} \rightarrow e)) \wedge \\
\quad (\mathit{len}\, m_8 \supset (\{1,1\} \rightarrow g)) \wedge \\
\quad (\mathit{len}\, 1 \supset (\text{weaken } \overline{Sum} \rightarrow \overline{Sum})) \wedge \\
\quad (\mathit{len}\, 1 \supset (\text{weaken } \overline{Carry} \rightarrow \overline{Carry})))
\end{array}
$$

Here the functions $\mathit{func1}(A_1, B_1, C_1)$ and $\mathit{func2}(A_1, B_1, C_1)$ are abbreviations:

$$
\begin{array}{l}
\mathit{func1}(A_1, B_1, C_1) = \\
\quad (A_1 \wedge B_1 \wedge C_1 \vee (A_1 \wedge \neg B_1 \wedge \neg C_1) \\
\quad \vee(\neg A_1 \wedge B_1 \wedge \neg C_1) \vee (\neg A_1 \wedge \neg B_1 \wedge C_1))
\end{array}
$$

$$
\begin{array}{l}
\mathit{func2}(A_1, B_1, C_1) = \\
\quad \neg(A_1 \wedge B_1 \vee (A_1 \wedge C_1 \vee (B_1 \wedge C_1)))
\end{array}
$$

The abbreviations m_{sum} and m_{carry} are the delays through the sumpart and the carrypart respectively:

$$
\begin{array}{rcl}
m_{sum} & = & (m_1 + max(m_2 + m_3, max(m_4, m_6) + m_5) + \\
 & & (m_{16} + max(m_{12} + max(m_9, m_{10}, m_{11}), m_{15} + m_{14} + m_{13})))
\end{array}
$$

$$
m_{carry} = (m_1 + max(m_2 + m_3, max(m_4, m_6) + m_5))
$$

The term which specifies the capacitance on the two outputs of the adder is:

$$
\begin{array}{l}
\Box(\mathit{len}\, 1 \supset (\text{weaken } \overline{Sum} \rightarrow \overline{Sum})) \wedge \\
\quad (\mathit{len}\, 1 \supset (\text{weaken } \overline{Carry} \rightarrow \overline{Carry}))
\end{array}
$$

This appears as a subterm for every possible state of the clocks, and is simplified into a single statement. In addition, the behavioral equation is simplified by eliminating the clauses which do not apply to the ports of the adder.

The derived specification of the behavior of the adder is:

$$\begin{array}{l}
\mathsf{adder}(A, B, C, \overline{Carry}, \overline{Sum}, \Phi_1, \Phi_2, X) \quad \equiv_{\mathsf{def}} \\
\quad (\Box\, \Phi_2 = \{1,1\} \supset \\
\qquad (len\ x_7 \supset \ \mathsf{strengthen}\ func2(A_1, B_1, C_1) \rightarrow \overline{Carry}) \wedge \\
\qquad (len\ x_6 \supset \ \mathsf{strengthen}\ func1(A_1, B_1, C_1) \rightarrow \overline{Sum})) \wedge \\
\quad (\Box(len\ 1 \supset (\mathsf{weaken}\ \overline{Sum} \rightarrow \overline{Sum})) \wedge \\
\qquad (len\ 1 \supset (\mathsf{weaken}\ \overline{Carry} \rightarrow \overline{Carry}))) \wedge \\
\qquad X = max(x_6, x_7)
\end{array}$$

This states that the sum and carry outputs of the adder are calculated when Φ_2 is high, and the outputs are latched when Φ_2 is low. The outputs exhibit the required boolean behavior for the function. The argument X of the adder is instantiated to the longest delay from the rising edge of Φ_2, which in this case is $max(x_6, x_7)$.

Deriving Constraints

In addition to requiring that all clock signals obey the control predicate defined in Section 5.5 the NORA design style requires that the inputs to the dynamic logic be stable during the evaluate phase. For the adder circuit this translates to requiring that $\overline{a}$, $\overline{b}$, and $\overline{c}$ be stable when Φ_1 is low. This is a direct consequence of the behaviors of the inverters clocked on Φ_1 provided that Φ_1 is high longer than the delay through the slowest inverter. If this condition is met, these constraints are satisfied.

My transistor model also requires that the inputs to the inverters be stable when they are being evaluated. The constraints that A, B, and C be stable during Φ_1 is imposed on the environment. The constraints that e and g be stable during Φ_2 places a constraint on the duration of the clock signals. The time during which Φ_1 and Φ_2 are both low must be longer than the longest evaluate delay during $\overline{\Phi_1}$. For this circuit, that time corresponds to the worst case delay through the sumpart. This is the same as the restriction due to NORA design rules.

7.2.5 An n-bit Adder

An n-bit adder can be built by cascading n stages of 1-bit dynamic adders as shown in Figure 7.11. Static inverters are used on the outputs as specified by the NORA design style. The signals $\overline{Carry_i}$ and C_{i+1} are connected between stages through the inverters. Signals $\overline{Carry_i}$ and C_{i+1} are hidden from the remainder of the circuit, so their timing constraints must be met. The input constraint of a stage is that signal C_{i+1} is stable when Φ_1 is high. From the output behavior we can derive that $\overline{Carry_i}$ is stable when Φ_2 is low. Since Φ_2 is low when Φ_1 is high, the $\overline{Carry_i}$ signal meets the input constraint of C_{i+1}. This signal is fed through a static inverter. If the time between the falling edge of Φ_2 and the rising edge of Φ_1 is longer than the delay of the static inverter, then the constraints on the carry signals are met. This generates an additional requirement on the relative timing of the clock signals.

In [CGM86], the functional behavior of a 1-bit adder is used to derive the behavior of an n-bit adder. The results of their proof are applicable to the dynamic n-bit adder provided I can show that the functional description I derive of the dynamic 1-bit adder is equivalent to their specification of a 1-bit adder. I do this by abstracting the function from the derived behavior. Their approach can therefore be used to derive the functional behavior at higher levels of description.

The derived behavior of the 1-bit adder specifies function and timing. The function can be abstracted from this formula. Observe that the formula has the form of a clocked element with delay, modeled with capacitors on its outputs. The behavioral equation of such an element which has inputs A, B, ... and output X is of the form:

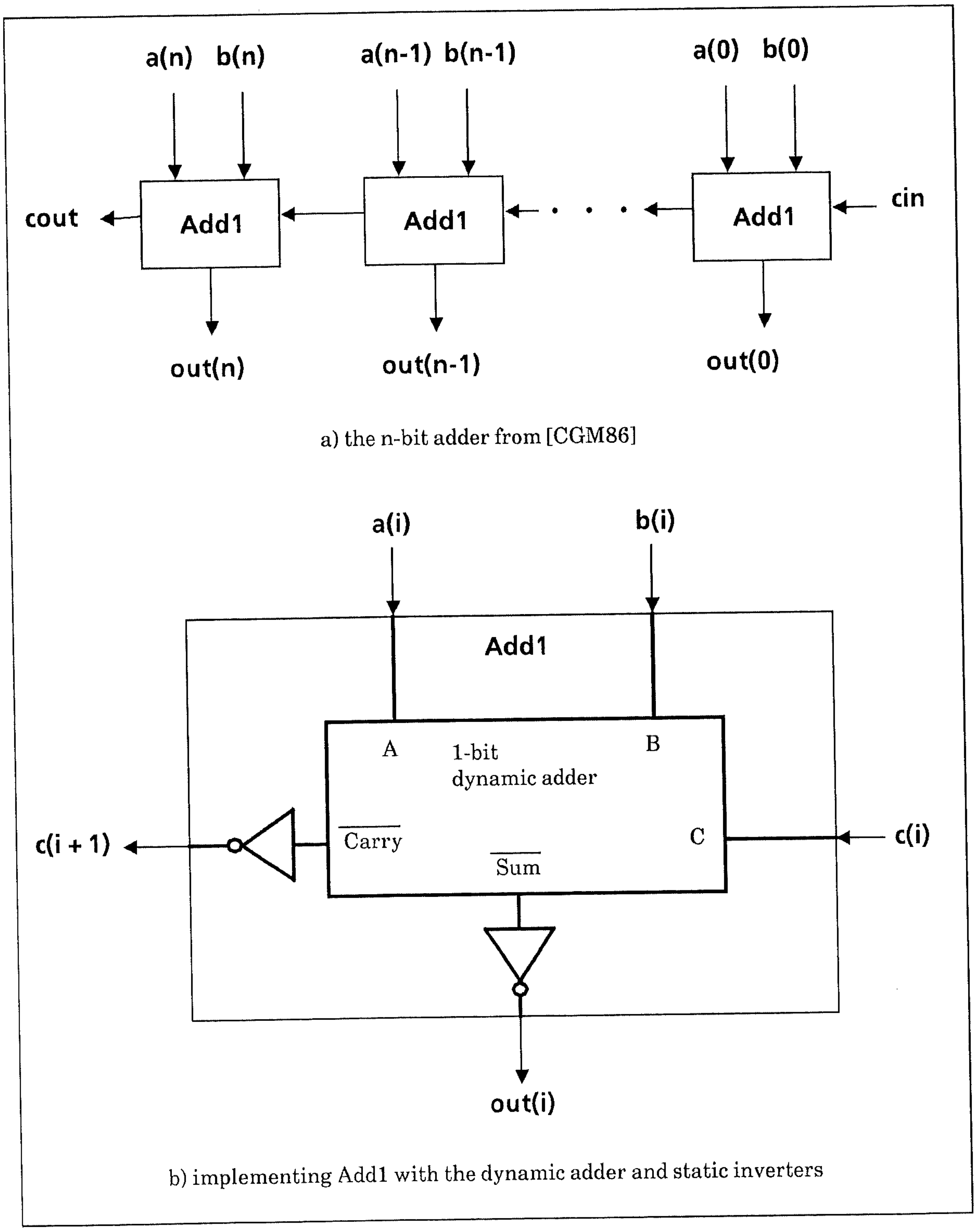
$$\begin{aligned}&\Box(\Phi \wedge len\, m \supset (f(A,B,\ldots) \rightarrow X)) \wedge\\ &\Box(len\ 1 \supset (\text{weaken } X \rightarrow X))\end{aligned}$$

The functional behavior of this element is:

$$X = f(A,B,\ldots)$$

Similarly, from the derived behavior for the adder we abstract the functional behavior:

$$\begin{aligned}&\overline{Carry} = \neg(A \wedge B \vee (A \wedge C \vee (B \wedge C))) \wedge\\ &\overline{Sum} = \neg(A \wedge B \wedge C \vee (A \wedge \neg B \wedge \neg C) \vee (\neg A \wedge B \wedge \neg C) \vee (\neg A \wedge \neg B \wedge C))\end{aligned}$$



a) the n-bit adder from [CGM86]

b) implementing Add1 with the dynamic adder and static inverters

Figure 7.11: An n-bit Dynamic Adder

By composing this with the behavior of the static inverters, we derive the same functional behavior as that used in the n-bit adder proof in [CGM86]. Therefore the results of their proof are applicable to the circuit shown in Figure 7.11.

7.2.6 Discussion

In summary, the behavior derived for a 1-bit dynamic adder is:

$$\begin{aligned}
&\mathsf{adder}(A, B, C, \overline{Carry}, \overline{Sum}, \Phi_1, \Phi_2, max(x_6, x_7)) \quad \equiv_{\mathsf{def}}\\
&\quad(\Box\,\Phi_2 = \{1,1\} \supset\\
&\qquad(\textit{len}\ x_7 \supset\ \mathsf{strengthen} func2(A_1, B_1, C_1) \rightarrow \overline{Carry}) \wedge\\
&\qquad(\textit{len}\ x_6 \supset\ \mathsf{strengthen}\ func1(A_1, B_1, C_1) \rightarrow \overline{Sum}))\wedge\\
&\quad(\Box(\textit{len}\ 1 \supset (\mathsf{weaken}\ \overline{Sum} \rightarrow \overline{Sum})) \wedge\\
&\qquad(\textit{len}\ 1 \supset (\mathsf{weaken}\ \overline{Carry} \rightarrow \overline{Carry})))
\end{aligned}$$

Where $func1(A_1, B_1, C_1)$ and $func2(A_1, B_1, C_1)$ are abbreviations:

$$\begin{aligned}
&func1(A_1, B_1, C_1) =\\
&\qquad\neg(A_1\ \wedge\ B_1\ \wedge\ C_1 \vee (A_1\ \wedge\ \neg B_1\ \wedge\ \neg C_1)\vee\\
&\qquad(\neg A_1\ \wedge\ B_1\ \wedge\ \neg C_1)\vee (\neg A_1\ \wedge\ \neg B_1\ \wedge\ C_1))
\end{aligned}$$

$$\begin{aligned}
&func2(A_1, B_1, C_1) =\\
&\qquad\neg(A_1\ \wedge\ B_1 \vee (A_1\ \wedge\ C_1 \vee(B_1\ \wedge\ C_1)))
\end{aligned}$$

From this equation we can abstract functional behavior subject to timing constraints on the availability of the inputs and outputs. The function is:

$$\begin{aligned}
&\overline{Carry} = \neg(A\ \wedge\ B \vee (A\ \wedge\ C \vee (B\ \wedge\ C))) \wedge\\
&\overline{Sum} = \neg(A \wedge B \wedge C \vee (A \wedge \neg B \wedge \neg C) \vee (\neg A \wedge B \wedge \neg C) \vee (\neg A \wedge \neg B \wedge C))
\end{aligned}$$

The constraints on the inputs are that Φ_1 and Φ_2 exhibit the characteristics of 2-phase clocks, and that A, B, and C are stable when Φ_1 is high. The circuit enforces the conditions that the outputs $\overline{Carry}$ and $\overline{Sum}$ are stable when Φ_2 is low. This abstracted behavior can then be used in proofs of circuits where the 1-bit adder is a component.

Mike Gordon has proved the behavior of this same 1-bit adder in HOL using a unidirectional transistor model [Gor87]. His proof works by exhaustive simulation on the inputs A, B, and C. The proof would not run on a DEC Microvax II with 6 megabytes of memory due to insufficient memory. It took over 6000 seconds of CPU time to execute on an

Atlas10 which runs HOL proofs approximately ten times faster than a Microvax. Seconds of cpu time on a DEC Microvax II for my derivations are given in Table 7.1. All values are approximations; the actual values were obtained using the Prolog *statistics* command. My derivation runs much faster partially because it is less rigorous than the HOL one. The improved performance also demonstrates the gains made by reasoning about a circuit rather than simulating it exhaustively.

loading	25 seconds
C^2MOS inverter	100 seconds
carrypart	150 seconds
sumpart	200 seconds
adder	450 seconds

Table 7.1: Performance Results for the One Bit Dynamic Adder

Development time for exhaustive simulation of the adder was very low. It merely required the time to code and enter this approach. Development of the approach done with PALM involving reasoning about the behavior of the circuit took considerably longer. There are clearly trade-offs between development time and run time of these two approaches. There are other trade-offs between understanding a design and simply exercising that design. One advantage in understanding a design is that errors can be localized more quickly and that small changes to the design are easily handled. Weise [Wei86] refers to this as incrementality. When I first verified the adder, I incorrectly specified the structure of the carrypart. This became apparent when trying to derive the function of this component and the error was quickly rectified. If I had derived the behavior by exhaustive simulation the error would have appeared as incorrect behavior of both the $\overline{Sum}$ and $\overline{Carry}$ outputs would have been much more difficult to pinpoint since all primitive components would have been suspect.

7.3 Conclusions

In this chapter I have presented several examples of reasoning about circuits including a fully complementary dynamic latch and a dynamic 1-bit adder. I have shown how ITL and Prolog are used to derive the behavior of a circuit from its components and to reason about constraints on the behavior of these circuits.

In the next chapter I present conclusions. I summarize the salient aspects of my approach, discuss issues and improvements, and consider how my approach could be incorporated into a computer aided design system for VLSI.

Chapter 8

Conclusions

I have presented an approach to hardware verification which uses temporal logic to reason about MOS VLSI circuits at the transistor level. Specifically, I have extended ITL to express a more realistic circuit model which includes capacitive effects of circuits. In addition, I have implemented an interactive system to aid reasoning about the circuit models. This approach combines the following features:

- Automatically deriving characteristics of a circuit from the schematic: The characteristics derived are the direction of signal flow through circuit components and the directions of ports of those components.
- Modeling low level details of MOS circuits: Signals are described as $\{value, strength\}$ pairs so capacitive effects of circuits can be expressed. Timing is treated at a detailed level by modeling transistors and combinational elements with delay.
- Symbolically reasoning about circuit behavior: Circuit specifications include timing and function. The specifications are manipulated hierarchically and incrementally. The PALM system aids in the symbolic manipulation of these behaviors.
- Reasoning about constraints: Constraints on the inputs and outputs of circuit components are explicitly stated, and are manipulated in the same way as the behavioral specifications.
- Explicitly relating timing to the clock signals: A circuit may include more than one clock signal; the relationship between clock signals is explicitly stated. Inputs and outputs are explicitly related to the different clock signals. In addition to reasoning about behavior, constraints on the different clocking periods are derived.

I have demonstrated this approach by deriving the behavior of several examples from the behavior of their components down to the transistor level. These examples included a sophisticated 1-bit adder implemented with a dynamic CMOS design style. This adder uses a 2-phase clocking scheme and exploits charge sharing. The properties derived include the functional behavior of the adder, constraints on when the inputs must be stable, the time at which the outputs are available, and constraints on the lengths of the different clock phases.

In the remainder of this chapter, I discuss some unresolved issues with this approach and some ways it can be improved. Finally, I consider how this approach could be incorporated in a computer aided design system for VLSI.

8.1 Issues and Improvements

The 'False Implies Everything' Problem

Many approaches to hardware verification require that a specification be verified by showing that it is *equivalent* to its implementation [Bar84], [Mil86a]. This requires that the details of two different levels of description be the same. It is frequently desirable to have the specification be logically *implied* by the implementation. This is the approach I use. The specification can then be more abstract or hide some of the details of the implementation. Logical implication, however, introduces a problem. It is a property of implication that if the antecedent is *false*, the entire statement is *true*. A false antecedent arises if an implementation is inconsistent. A simple example is a short circuit. Here power and ground signals are directly connected:

$$\{0,1\} \wedge \{1,1\} = \textit{false}$$

An inconsistent circuit satisfies any specification. One way of alleviating this problem is by checking that the implementation is consistent before verifying the circuit. This problem is discussed further in [CGM86].

Initialization and Termination in ITL

I have not considered the behavior of circuits at initialization or termination of intervals. Intervals in ITL may be finite or infinite. Since circuit behavior continues forever, I assume intervals are infinite. Therefore, I do not consider termination. Initialization is more difficult. I do not model the events which occur when power is first applied to a circuit. Instead I assume that the circuit has already settled and that all signals are well-behaved and obey the appropriate specifications. Frequently errors in hardware operation arise as a result of a circuit getting into an undesirable state when it is first turned on. Such errors will not be identified by this approach. Note that similar assumptions about the behavior at initialization are stated in the circuit specifications presented by Moszkowski [Mos83] and Herbert [Her86].

Improving the Transistor Model

There are several low level aspects of transistor behavior which my model does not capture. For example, my model cannot express threshold drops. Thus, the need for pMOS as well as nMOS transistors in CMOS transmission gates is not apparent in this model. In addition, I cannot detect charge sharing bugs, races or hazards. Note that I have restricted my examples to circuits in which races and hazards will not arise. These, as well as charge sharing, are important aspects of incorrect MOS circuit behavior which will arise in a more general class of MOS circuits. Consideration should go into incorporating them into the model. However, there will always be aspects of behavior which are not captured in a model. It is important that the user of a model be aware of these shortcomings. This applies to models used in simulation as well as those used in formal methods.

Improving Delay Calculations

I consider delays of circuit elements at the transistor level, but use a simple model for composing these delays. Delays of elements connected in series are added, and the maximum delay of elements connected in parallel is used. This aspect of timing modeling should be improved. Ousterhout [Ous84] presents several switch-level delay models which could be incorporated into my approach.

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

$$\textit{intrinsic delay} + R_{Load} * C_{Load}$$

Here R_{Load} is a function of the output of the component being modeled and C_{Load} is the sum of the capacitances of the devices being driven. Further improvements could be gained by taking into consideration the waveform shape of the input waveform which is dependent on the fan-out and drive capability of the preceding stage of the circuit. Currently I assume that all waveform changes are instantaneous.

Since wires add an appreciable amount of delay in MOS technologies these should be modeled, and the capacitance of the material of the wire should be considered. This has not yet been done since this information is highly dependent on the actual layout of a circuit.

8.2 Computer Aided Design with Formal Methods

I have described how to formally derive low level timing and functional behavior from transistor level descriptions of hardware components. This only addresses one aspect of the design cycle. This approach should be incorporated into a computer aided design system which encourages designers to apply formal methods from design conception to layout. A proposal for such a system is described below. Subrahmanyam [Sub86b] describes an expert system for VLSI design which incorporates formal methods as well as more conventional tools. This is an evolutionary approach. The system relates different levels of description formally, but the designer can have the impression of using tools she is familiar with. Milne [Mil86b] describes a more revolutionary approach, which does not attempt to incorporate existing tools, where a system can be described at various different levels from specification to layout and the different levels can be formally related.

8.2 COMPUTER AIDED DESIGN WITH FORMAL METHODS

A computer aided design system should consist of an interactive workstation which includes layout tools as well as formal methods for reasoning about circuits. Such a workstation should provide tools for top-down and bottom-up hardware design. In addition such a system should keep the various different levels of representation of a circuit consistent.

Top-down design should be facilitated. A specification should be simulated to ensure that it does in fact capture the desired behavior. Synthesis tools which guarantee correct behavior of implementations can also be incorporated into such a design environment.

Bottom-up design and verification will be aided by providing a design library of standard cells. New cells may be added by the user. Cells provided include logical descriptions of circuits as well as layout details. As a designer constructs a circuit out of these cells, a logical description of the circuit is built up. The designer can chose to manipulate a graphical or textual representation of the circuit.

Verification will be done at many different levels. This involves using mathematical theorem proving techniques to determine that the design correctly implements the specification. Tedious aspects of such techniques should be automated. When a designer completes the design of a component, she can verify that the implementation of that component meets its specification. When a set of components are interconnected, their specifications are composed and checked against the specification of the larger circuit. The workstation will support such verification in any order. The top-level design could be verified before the components which comprise it are designed, the components could be designed and verified first, or some combination of these strategies could be used.

This thesis described the derivation of timing and functional aspects of circuit components from their transistor level implementations. Details of delay were handled symbolically. In an integrated system, this level could first be done symbolically, and later quantitative delay parameters could be extracted from the layout.

An advantage of such a workstation is that it keeps all the different representations of a circuit consistent. This is accomplished by formally abstracting between different levels of description. When a level changes, the system should alert the user to the changes required to other levels to maintain consistency.

8.3 Final Thoughts

In this thesis, I described applying formal tools to one step in the design hierarchy: from the transistor level to the gate level. I also discussed other formal methods which address different levels of the design hierarchy. The advantages of using formal methods include the advantages gained by hierarchical and incremental analysis, and by manipulating inputs and outputs symbolically. In addition, formal methods allow different levels of description to be related formally. By incorporating many levels of description into a design system based on formal methods, errors which occur when one level is translated into another will be detected. Such errors are not detected by current design tools.

There are many advantages to be gained by incorporating formal methods into the design cycle. A design system should support many different levels of description, and provide tools for simulating design specifications at a particular level and formally relating specifications at different levels. The emergence of such systems will make formal methods more attractive for the development of real designs.

PALM Rules

In this appendix I list the rules used by the PALM system for reasoning about circuits. These rules are divided into four categories: basic logic rules, rules for signal strengths, rules for interval temporal logic, and rules for combining circuit elements. Note that the representation for *true* is $\{1, \_\}$ and the representation for *false* is $\{0, \_\}$. Thus, all rules deal with signals. The category 'rules for signal strengths' include those rules which explicitly examine the *strength* field.

Rules are referred to by name. In the PALM system, a rule is applied to a term by selecting option apply and specifying the name of the rule. Below, the rule name and the associated rule are listed. Some rules have a version in which the arguments are commuted. Such rules are listed on the same line. The name of the commutative version of the rule is the name of the original rule with the suffix -comm added.

A.1 Basic Logic Rules

Rule Name:	Rule:	Commutative Version:
not-not	$\neg\neg A = A$	
not-equals	$(\neg A = B) = \neg(A = B)$	
not-one	$\neg\{1, A\} = \{0, A\}$	
not-zero	$\neg\{0, A\} = \{1, A\}$	
and-ident	$A \wedge \{1, \_\} = A$	$\{1, \_\} \wedge A = A$
or-ident	$A \vee \{0, \_\} = A$	$\{0, \_\} \vee A = A$
and-not	$A \wedge \neg A = \{0, \_\}$	$\neg A \wedge A = \{0, \_\}$
falsity	$(\{0, \_\} = \{1, \_\}) = \{0, \_\}$	$(\{1, \_\} = \{0, \_\}) = \{0, \_\}$
and-dup	$A \wedge A = A$	
and-false	$A \wedge \{0, B\} = \{0, B\}$	$\{0, B\} \wedge A = \{0, B\}$
comm-and	$A \wedge B = B \wedge A$	
comm-or	$A \vee B = B \vee A$	
assoc-orl	$A \vee B \vee C = A \vee (B \vee C)$	
assoc-orr	$A \vee (B \vee C) = (A \vee B) \vee C$	
assoc-andl	$A \wedge B \wedge C = A \wedge (B \wedge C)$	
assoc-andr	$A \wedge (B \wedge C) = (A \wedge B) \wedge C$	
and-intror	$(A \supset B) \wedge (A \supset C) = (A \supset B \wedge C)$	
and-intror-inv	$(A \supset B \wedge C) = (A \supset B) \wedge (A \supset C)$	
and-true	$(A \wedge B = \{1, C\}) = (A = \{1, C\}) \wedge (B = \{1, C\})$	
and-eliml	$A \wedge B \supset A$	
and-elimr	$A \wedge B \supset B$	
implies-ident	$(A \supset A) = \{1, \_\}$	
or-introl	$(A \supset B) \wedge (C \supset B) = (A \vee C \supset B)$	
demorgan-not-and	$\neg(A \wedge B) = \neg A \vee \neg B$	
demorgan-not-or	$\neg(A \vee B) = \neg A \wedge \neg B$	
demorgan-and-not	$\neg A \wedge \neg B = \neg(A \vee B)$	
demorgan-or-not	$\neg A \vee \neg B = \neg(A \wedge B)$	
distrib-and-or	$A \vee C \wedge B = A \wedge B \vee (C \wedge B)$	
if-intro	$(A \supset B) \wedge (\neg A \supset C) = (\textit{if } A \textit{ then } B \textit{ else } C)$	
if-elim	$(\textit{if } A \textit{ then } B \textit{ else } C) = (A \supset B) \wedge (\neg A \supset C)$	
identity	$(A = A) = \{1, \_\}$	
implies-true	$(\{1, \_\} \supset A) = A$	
implies-false	$(\{0, \_\} \supset A) = \{1, \_\}$	
implied-true	$(A \supset \{1, \_\}) = \{1, \_\}$	
comm-equals	$(A = B) = (B = A)$	
not-equal-l	$(\neg A = B) = (A = \neg B)$	
equals-false	$(A = \{0, \_\}) = \neg A$	
equals-true	$(A = \{1, \_\}) = A$	
or-bool	$(A = \{0, \_\}) \vee (A = \{1, \_\}) = \{1, \_\}$	
import	$(A \supset (B \supset C)) = (A \wedge B \supset C)$	
export	$(A \wedge B \supset C) = (A \supset (B \supset C))$	

A.2 Signal Rules

Rule Name:	Rule:
comm-join	$A \sqcup B \Leftrightarrow B \sqcup A$
strengthen	strengthen $\{A, \_\} \Leftrightarrow \{A, 1\}$
weaken	weaken $\{A, \_\} \Leftrightarrow \{A, 0\}$
weak-strong	weaken strengthen $A \Leftrightarrow$ weaken A
strong-weak	strengthen weaken $A \Leftrightarrow$ strengthen A
weak-weak	weaken weaken $A \Leftrightarrow$ weaken A
strong-strong	strengthen strengthen $A \Leftrightarrow$ strengthen A
inv-strengthen	$\{A, 1\} \Leftrightarrow$ strengthen $\{A, \_\}$
inv-weaken	$\{A, 0\} \Leftrightarrow$ weaken $\{A, \_\}$
not-weaken	$\neg$ weaken $A \Leftrightarrow$ weaken $\neg A$
not-strengthen	$\neg$ strengthen $A \Leftrightarrow$ strengthen $\neg A$
distrib-weak-and	weaken $A \wedge$ weaken $B \Leftrightarrow$ weaken $(A \wedge B)$
distrib-strong-and	strengthen $A \wedge$ strengthen $B \Leftrightarrow$ strengthen $(A \wedge B)$
inv-distrib-weak-and	weaken $(A \wedge B) \Leftrightarrow$ weaken $A \wedge$ weaken B
inv-distrib-strong-and	strengthen $(A \wedge B) \Leftrightarrow$ strengthen $A \wedge$ strengthen B
distrib-weak-or	weaken $A \vee$ weaken $B \Leftrightarrow$ weaken $(A \vee B)$
distrib-strong-or	strengthen $A \vee$ strengthen $B \Leftrightarrow$ strengthen $(A \vee B)$
inv-distrib-weak-or	weaken $(A \vee B) \Leftrightarrow$ weaken $A \vee$ weaken B
inv-distrib-strong-or	strengthen $(A \vee B) \Leftrightarrow$ strengthen $A \vee$ strengthen B
join-strong-weak	strengthen $A \sqcup$ weaken $B \Leftrightarrow$ strengthen A

A.3 ITL Rules

Rule Name:	Rule:
and-yields-intro	$(A \rightarrow B) \wedge (C \rightarrow B) \supset (A \sqcup C \rightarrow B)$
distrib-box-and	$(\Box A) \wedge (\Box B) \Leftrightarrow (\Box A \wedge B)$
inv-distrib-box-and	$(\Box A \wedge B) \Leftrightarrow (\Box A) \wedge (\Box B)$

A.4 Circuit Rules

I assume the subformula below are embedded in a larger formula which is in the scope of a $\Box$ operator. I therefore do not include the $\Box$ operator in these formulas.

Rule Name:	Rule:
trans-unclock-phi	$(\mathit{len}\ N \supset (A \rightarrow B)) \wedge$ $((\Phi \approx \{1, \_\}) \wedge \mathit{len}\ M \supset (B \rightarrow C)) \wedge$ $(\mathit{len}\ 1 \supset (\text{weaken } B \rightarrow B))$ $\supset ((\Phi \approx \{1, \_\}) \wedge \mathit{len}\ M \supset (A \sqcup \text{weaken } B \rightarrow C))$
trans-combine	$(\mathit{len}\ M \supset (\mathit{func1}(A) \rightarrow B)) \wedge$ $(\mathit{len}\ N \supset (\mathit{func2}(B) \rightarrow C))$ $\supset (\mathit{len}\ (M + N) \supset (\mathit{func2}(\mathit{func1}(A)) \rightarrow C))$

Rule Name:	Rule:
trans-cap-combine	$(\mathit{len}\ M \supset (\mathit{func1}(A) \rightarrow B)) \wedge$ $(\mathit{len}\ N \supset (\mathit{func2}(B) \rightarrow C)) \wedge$ $(\mathit{len}\ 1 \supset (\text{weaken}\ B \rightarrow B))$ $\supset (\mathit{len}\ (M+N) \supset (\mathit{func2}(\mathit{func1}(A)) \rightarrow C))$
trans-unclock-unclock	$(G1\ \wedge\ \mathit{len}\ M \supset (A \rightarrow B)) \wedge$ $(G2\ \wedge\ \mathit{len}\ N \supset (B \rightarrow C)) \wedge$ $(\mathit{len}\ 1 \supset (\text{weaken}\ B \rightarrow B))$ $\supset\ (G1\ \wedge\ G2\ \wedge\ \mathit{len}\ (M+N) \supset (A \rightarrow C))$
trans-or-unclock-unclock	$(G1\ \wedge\ \mathit{len}\ M \supset (A \rightarrow B)) \wedge$ $(G2\ \wedge\ \mathit{len}\ N \supset (A \rightarrow B))$ $\supset\ (G1 \vee G2\ \wedge\ \mathit{len}\ \mathit{max}(M,N) \supset (A \rightarrow B))$
trans-weak-strong	$(A\ \wedge\ \mathit{len}\ B \supset (C \rightarrow D)) \wedge$ $(E\ \wedge\ \mathit{len}\ F \supset (\text{weaken}\ G \rightarrow D))$ $\supset\ (A\ \wedge\ \mathit{len}\ B \supset (C \rightarrow D))$
latch	$((\Phi \approx \{1,\_\})\ \wedge\ \mathit{len}\ N \supset (\mathit{func1}(C) \rightarrow D)) \wedge$ $(\mathit{len}\ 1 \supset (\text{weaken}\ D \rightarrow D))$ $\supset\ (\Phi \approx \{0,\_\} \supset D = \text{weaken}\ \mathit{func1}(\mathit{latched}(\Phi, C)))$
pass-strong	$(\mathit{len}\ N\ \wedge\ A \supset (\{1,1\} \rightarrow B))$ $\Leftrightarrow (\mathit{len}\ N \supset (\text{strengthen}\ A \rightarrow B))$
pass-not-strong	$(A\ \wedge\ \mathit{len}\ N \supset (\{0,1\} \rightarrow B))$ $\Leftrightarrow (\mathit{len}\ N \supset (\text{strengthen}\ \neg A \rightarrow B))$
pass-weaken	$(\mathit{len}\ 1 \supset (\text{weaken}\ A \rightarrow A)) \wedge$ $(\Phi\ \wedge\ \mathit{len}\ N \supset (A \rightarrow B))$ $\supset\ (\Phi\ \wedge\ \mathit{len}\ N \supset (\text{weaken}\ A \rightarrow B))$
pass-stb-weaken	$(\mathit{len}\ 1 \supset (\text{weaken}\ A \rightarrow A)) \wedge$ $(\mathit{len}\ N \supset (\mathit{func1}(A) \rightarrow B))$ $\supset \bigcirc^{N}\ \mathit{stb}\ B$
trans-unclock-func1	$(\mathit{len}\ M \supset (A \rightarrow B)) \wedge$ $(\mathit{len}\ N \supset (\mathit{func1}(B) \rightarrow C)) \wedge$ $(\mathit{len}\ 1 \supset (\text{weaken}\ B \rightarrow B))$ $\supset\ (\mathit{len}\ (M+N) \supset (\mathit{func1}(A) \rightarrow C))$
phi1-phi2	$((\Phi_1 \approx \{1,\_\})\ \wedge\ (\Phi_2 \approx \{0,\_\}) \supset\ (\mathit{len}\ M \supset (\mathit{func1}(A) \rightarrow B))) \wedge$ $((\Phi_1 \approx \{0,\_\})\ \wedge\ (\Phi_2 \approx \{0,\_\}) \supset \mathit{stb}\ B) \wedge$ $((\Phi_1 \approx \{0,\_\})\ \wedge\ (\Phi_2 \approx \{1,\_\}) \supset\ (\mathit{len}\ N \supset (\mathit{func2}(B) \rightarrow C)))$ $\supset\ ((\Phi_2 \approx \{1,\_\})\ \wedge\ \mathit{len}\ N \supset\ \mathit{func2}(\mathit{func1}(\mathit{latched}(\Phi_1, A))) \rightarrow C)$

Appendix B

The Dynamic Latch Derivation

In this appendix, I present the derivation of behavior of the dynamic latch discussed in Section 7.1.3. Derivation of behavior for the latch proceeds hierarchically by first deriving the behavior of its components. The latch is made up of a shiftstage and an inverting multiplexer. Components of a shiftstage are a pass transistor and an inverter. Components of an inverting multiplexer are an inverter and a clocked 2-to-1 multiplexer. I present derivations of behavior of all these components, starting with the simple components and building up to the dynamic latch. When a component's behavior is first derived, its ports are named by Prolog constants. When that behavior is used as a component of a larger circuit, the port names are generalized to Prolog variables to allow renaming.

All the derivations presented here were done using the PALM system. The text, including the English explanation of each step, was produced by PALM. The output has been edited to facilitate reading. These edits include typesetting to make formulas more readable, and combining steps to reduce the length of derivations. Nothing has been added.

Note that the behavior derived for the inverting multiplexer and for the shiftstage involves a term of the form: $(\Box\neg(\Phi = \{1,\_\}) \supset \bigcirc^{p}\, stb\ b)$. These terms are reduced to: $(\Box\neg(\Phi = \{1,\_\}) \supset stb\ b)$ when the behaviors are used in deriving the behavior of the dynamic latch. This is true provided that Φ was true long enough for b to stabilize p units before Φ became false. This puts an additional constraint on the length of the clock cycle. Such reasoning about time intervals has not yet been mechanized in the the PALM system.

B.1 The Static Inverter

The current term is:
  static-invert-struct(in, out, n)

Expand all definitions. The new term is:
  npass(in, $\{0,1\}$, out, n) $\wedge$ ppass(in, $\{1,1\}$, out, n)

Expand all definitions. The new term is:
  $(\Box(in = \{1,\_\}) \wedge len\ n \supset (\{0,1\} \rightarrow out)) \wedge$
  $(\Box(in = \{0,\_\}) \wedge len\ n \supset (\{1,1\} \rightarrow out))$

Apply rule distrib-box-and : $(\Box A) \wedge (\Box B) \Leftrightarrow (\Box A \wedge B)$
The new term is:
  $\Box((in = \{1,\_\}) \wedge len\ n \supset (\{0,1\} \rightarrow out)) \wedge$
   $((in = \{0,\_\}) \wedge len\ n \supset (\{1,1\} \rightarrow out))$

Rec-apply rule comm-and twice : $A \wedge B = B \wedge A$
Apply rule export twice : $(A \wedge B \supset C) = (A \supset (B \supset C))$
The new term is:
  $\Box(len\ n \supset (in = \{1,\_\} \supset (\{0,1\} \rightarrow out))) \wedge$
   $(len\ n \supset (in = \{0,\_\} \supset (\{1,1\} \rightarrow out)))$

Apply rule and-intror : $(A \supset B) \wedge (A \supset C) = (A \supset B \wedge C)$
The new term is:
  $\Box\, len\ n \supset (in = \{1,\_\} \supset (\{0,1\} \rightarrow out)) \wedge$
      $(in = \{0,\_\} \supset (\{1,1\} \rightarrow out))$

Start case analysis on in with values $[\{0,\_\},\{1,\_\}]$.
The current term is:
  $\Box\, in = \{0,\_\} \supset$
   $(len\ n \supset (\{0,\_\} = \{1,\_\} \supset (\{0,1\} \rightarrow out)) \wedge$
      $(\{0,\_\} = \{0,\_\} \supset (\{1,1\} \rightarrow out)))$

Apply rule falsity : $(\{0,\_\} = \{1,\_\}) = \{0,\_\}$
The new term is:
  $\Box\, in = \{0,\_\} \supset$
   $(len\ n \supset (\{0,\_\} \supset (\{0,1\} \rightarrow out)) \wedge$
      $(\{0,\_\} = \{0,\_\} \supset (\{1,1\} \rightarrow out)))$

Apply rule implies-false : $(\{0,\_\} \supset A) = \{1,\_\}$
The new term is:
  $\Box\, in = \{0,\_\} \supset$
   $(len\ n \supset \{1,\_\} \wedge (\{0,\_\} = \{0,\_\} \supset (\{1,1\} \rightarrow out)))$

Apply rule identity : $(A = A) = \{1,\_\}$
The new term is:
  $\Box\, in = \{0,\_\} \supset$
   $(len\ n \supset \{1,\_\} \wedge (\{1,\_\} \supset (\{1,1\} \rightarrow out)))$

```
Apply rule implies-true : $(\{1,\_\} \supset A) = A$
The new term is:
    $\Box in = \{0,\_\} \supset$
        $(len\ n \supset \{1,\_\} \wedge (\{1,1\} \rightarrow out))$

Apply rule and-ident-comm : $\{1,\_\} \wedge A = A$
The new term is:
    $\Box in = \{0,\_\} \supset (len\ n \supset (\{1,1\} \rightarrow out))$

Replace $\{1,1\}$ with strengthen $\neg in$ in current term.
        First show:
            $in = \{0,\_\} \supset \{1,1\} =$strengthen $\neg in$

        Replace $in$ with $\{0,\_\}$ in current term.
            First show:
                $in = \{0,\_\} \supset in = \{0,\_\}$

            Apply rule implies-ident : $(A \supset A) = \{1,\_\}$
            The new term is:
                $\{1,\_\}$
            The terms being replaced are equivalent
        The new term with replacements is:
            $\{0,\_\} = \{0,\_\} \supset \{1,1\} =$strengthen $\neg\{0,\_\}$

        Apply rule identity : $(A = A) = \{1,\_\}$
        The new term is:
            $\{1,\_\} \supset \{1,1\} =$strengthen $\neg\{0,\_\}$

        Apply rule implies-true : $(\{1,\_\} \supset A) = A$
        The new term is:
            $\{1,1\} =$strengthen $\neg\{0,\_\}$

        Apply rule not-zero : $\neg\{0,A\} = \{1,A\}$
        The new term is:
            $\{1,1\} =$strengthen $\{1,\_\}$

        Apply rule strengthen : strengthen $\{A,\_\} \nLeftrightarrow \{A,1\}$
        The new term is:
            $\{1,1\} = \{1,1\}$
        Apply rule identity : $(A = A) = \{1,\_\}$
        The new term is:
            $\{1,\_\}$
        The terms being replaced are equivalent
The new term with replacements is:
    $\Box in = \{0,\_\} \supset (len\ n \supset (\text{strengthen } \neg in \rightarrow out))$

Do next case of case analysis.
The current term is:
    $\Box in = \{1,\_\} \supset$
        $(len\ n \supset (\{1,\_\} = \{1,\_\} \supset (\{0,1\} \rightarrow out)) \wedge$
                $(\{1,\_\} = \{0,\_\} \supset (\{1,1\} \rightarrow out)))$
```

Apply rule falsity-comm : $(\{1,\_\} = \{0,\_\}) = \{0,\_\}$
The new term is:
$\Box in = \{1,\_\} \supset$
$(len\ n \supset (\{1,\_\} = \{1,\_\} \supset (\{0,1\} \rightarrow out)) \wedge$
$(\{0,\_\} \supset (\{1,1\} \rightarrow out)))$

Apply rule implies-false : $(\{0,\_\} \supset A) = \{1,\_\}$
The new term is:
$\Box in = \{1,\_\} \supset$
$(len\ n \supset (\{1,\_\} = \{1,\_\} \supset (\{0,1\} \rightarrow out)) \wedge \{1,\_\})$

Apply rule identity : $(A = A) = \{1,\_\}$
The new term is:
$\Box in = \{1,\_\} \supset$
$(len\ n \supset (\{1,\_\} \supset (\{0,1\} \rightarrow out)) \wedge \{1,\_\})$

Apply rule implies-true : $(\{1,\_\} \supset A) = A$
The new term is:
$\Box in = \{1,\_\} \supset$
$(len\ n \supset (\{0,1\} \rightarrow out) \wedge \{1,\_\})$

Apply rule and-ident : $A \wedge \{1,\_\} = A$
The new term is:
$\Box in = \{1,\_\} \supset (len\ n \supset (\{0,1\} \rightarrow out))$

Replace $\{0,1\}$ with strengthen $\neg in$ in current term.
First show:
$in = \{1,_\} \supset \{0,1\} =$strengthen $\neg in$

Replace in with $\{1,\_\}$ in current term.
First show:
$in = \{1,\_\} \supset in = \{1,\_\}$

Apply rule implies-ident : $(A \supset A) = \{1,\_\}$
The new term is:
$\{1,\_\}$
The terms being replaced are equivalent.
The new term with replacements is:
$\{1,_\} = \{1,_\} \supset \{0,1\} =$strengthen $\neg\{1,\_\}$

Apply rule identity : $(A = A) = \{1,\_\}$
The new term is:
$\{1,_\} \supset \{0,1\} =$strengthen $\neg\{1,\_\}$

Apply rule implies-true : $(\{1,\_\} \supset A) = A$
$\{0,1\} =$strengthen $\neg\{1,\_\}$

Apply rule not-one : $\neg\{1, A\} = \{0, A\}$
The new term is:
$\{0,1\} =$strengthen $\{0,\_\}$

Apply rule strengthen : strengthen $\{A,\_\} \nLeftrightarrow \{A,1\}$
The new term is:
$\{0,1\} = \{0,1\}$

Apply rule identity : $(A = A) = \{1, \_\}$
The new term is:
$\{1, \_\}$
The terms being replaced are equivalent
The new term with replacements is:
$\Box in = \{1, \_\} \supset (len\ n \supset (\text{strengthen } \neg in \rightarrow out))$

End case analysis.
The result of case analysis is:
$(\Box in = \{0, \_\} \supset (len\ n \supset (\text{strengthen } \neg in \rightarrow out))) \wedge$
$(\Box in = \{1, \_\} \supset (len\ n \supset (\text{strengthen } \neg in \rightarrow out)))$

Apply rule distrib-box-and : $(\Box A) \wedge (\Box B) \Leftrightarrow (\Box A \wedge B)$
The new term is:
$\Box(in = \{0, \_\} \supset (len\ n \supset (\text{strengthen } \neg in \rightarrow out))) \wedge$
$(in = \{1, \_\} \supset (len\ n \supset (\text{strengthen } \neg in \rightarrow out)))$

Apply rule or-introl : $(A \supset B) \wedge (C \supset B) = (A \vee C \supset B)$
The new term is:
$\Box(in = \{0, \_\}) \vee (in = \{1, \_\}) \supset$
$(len\ n \supset (\text{strengthen } \neg in \rightarrow out))$

Apply rule or-bool : $(A = \{0, \_\}) \vee (A = \{1, \_\}) = \{1, \_\}$
The new term is:
$\Box\{1, \_\} \supset (len\ n \supset (\text{strengthen } \neg in \rightarrow out))$

Apply rule implies-true : $(\{1, \_\} \supset A) = A$
The new term is:
$\Box\, len\ n \supset (\text{strengthen } \neg in \rightarrow out)$

B.2 The Clocked 2-to-1 Multiplexer

The current term is:
two-one-mux-struct(g, a, b, x, Φ, m)

Expand all definitions.
The new term is:
npass$(g \wedge \Phi, a, x, m) \wedge$ cap$(x) \wedge$ npass$(\neg g \wedge \Phi, b, x, m) \wedge$ cap(x)

Expand all definitions.
The new term is:
$(\Box(g \wedge \Phi = \{1, \_\}) \wedge len\ m \supset (a \rightarrow x)) \wedge$
$(\Box\, len\ 1 \supset (\text{weaken } x \rightarrow x)) \wedge$
$(\Box(\neg g \wedge \Phi = \{1, \_\}) \wedge len\ m \supset (b \rightarrow x)) \wedge$
$(\Box\, len\ 1 \supset (\text{weaken } x \rightarrow x))$

Rec-apply rule comm-and : $A \wedge B = B \wedge A$
Apply rule assoc-andr : $A \wedge (B \wedge C) = (A \wedge B) \wedge C$
Apply rule distrib-box-and : $(\Box A) \wedge (\Box B) \Leftrightarrow (\Box A \wedge B)$
The new term is:

$(\Box((\neg g \wedge \Phi = \{1,\_\}) \wedge \mathit{len}\, m \supset (b \to x)) \wedge$
$\quad((g \wedge \Phi = \{1,\_\}) \wedge \mathit{len}\, m \supset (a \to x))) \wedge$
$(\Box\, \mathit{len}\ 1 \supset (\text{weaken } x \to x)) \wedge$
$(\Box\, \mathit{len}\ 1 \supset (\text{weaken } x \to x))$

Apply rule assoc-andl : $A \wedge B \wedge C = A \wedge (B \wedge C)$
Apply rule distrib-box-and : $(\Box A) \wedge (\Box B) \Leftrightarrow (\Box A \wedge B)$
Apply rule comm-and : $A \wedge B = B \wedge A$
The new term is:

$(\Box(\mathit{len}\ 1 \supset (\text{weaken } x \to x)) \wedge$
$\quad(\mathit{len}\ 1 \supset (\text{weaken } x \to x))) \wedge$
$(\Box((\neg g \wedge \Phi = \{1,\_\}) \wedge \mathit{len}\, m \supset (b \to x)) \wedge$
$\quad((g \wedge \Phi = \{1,\_\}) \wedge \mathit{len}\, m \supset (a \to x)))$

Split current term.
The new term is:

$\Box(\mathit{len}\ 1 \supset (\text{weaken } x \to x)) \wedge$
$\quad(\mathit{len}\ 1 \supset (\text{weaken } x \to x))$

The term saved on the stack is:

$\Box((\neg g \wedge \Phi = \{1,\_\}) \wedge \mathit{len}\, m \supset (b \to x)) \wedge$
$\quad((g \wedge \Phi = \{1,\_\}) \wedge \mathit{len}\, m \supset (a \to x))$

Apply rule and-dup : $A \wedge A = A$
The new term is:

$\Box\, \mathit{len}\ 1 \supset (\text{weaken } x \to x)$

Swap current term with top of stack.
The new term is:

$\Box((\neg g \wedge \Phi = \{1,\_\}) \wedge \mathit{len}\, m \supset (b \to x)) \wedge$
$\quad((g \wedge \Phi = \{1,\_\}) \wedge \mathit{len}\, m \supset (a \to x))$

Apply rule and-true twice: $(A \wedge B = \{1, C\}) = (A = \{1, C\}) \wedge (B = \{1, C\})$
Apply rule assoc-andl : $A \wedge B \wedge C = A \wedge (B \wedge C)$
The new term is:

$\Box((\neg g = \{1,\_\}) \wedge ((\Phi = \{1,\_\}) \wedge \mathit{len}\, m) \supset (b \to x)) \wedge$
$\quad((g = \{1,\_\}) \wedge (\Phi = \{1,\_\}) \wedge \mathit{len}\, m \supset (a \to x))$

Apply rule assoc-andl : $A \wedge B \wedge C = A \wedge (B \wedge C)$
Rec-apply rule comm-and : $A \wedge B = B \wedge A$
Apply rule export twice : $(A \wedge B \supset C) = (A \supset (B \supset C))$
The new term is:

$\Box((\Phi = \{1,\_\}) \wedge \mathit{len}\, m \supset (\neg g = \{1,\_\} \supset (b \to x))) \wedge$
$\quad((\Phi = \{1,\_\}) \wedge \mathit{len}\, m \supset (g = \{1,\_\} \supset (a \to x)))$

Apply rule and-intror : $(A \supset B) \wedge (A \supset C) = (A \supset B \wedge C)$
The new term is:

$\Box(\Phi = \{1,\_\}) \wedge \mathit{len}\, m \supset$
$\qquad(\neg g = \{1,\_\} \supset (b \to x)) \wedge$
$\qquad(g = \{1,\_\} \supset (a \to x))$

Apply rule not-equals : $(\neg A = B) = \neg(A = B)$
The new term is:
$\Box(\Phi = \{1,\_\}) \wedge \mathit{len}\ m \supset$
$\quad (\neg(g = \{1,\_\}) \supset (b \rightarrow x)) \wedge$
$\quad (g = \{1,\_\} \supset (a \rightarrow x))$

Rec-apply rule comm-and : $A \wedge B = B \wedge A$
Apply rule if-intro : $(A \supset B) \wedge (\neg A \supset C) = (\mathit{if}\ A\ \mathit{then}\ B\ \mathit{else}\ C)$
The new term is:
$\Box(\Phi = \{1,\_\}) \wedge \mathit{len}\ m \supset$
$\quad (\mathit{if}\ g = \{1,\_\}\ \mathit{then}\ (a \rightarrow x)\ \mathit{else}\ (b \rightarrow x))$

And current term with top of stack.
The new term is:
$(\Box(\Phi = \{1,\_\}) \wedge \mathit{len}\ m \supset$
$\quad (\mathit{if}\ g = \{1,\_\}\ \mathit{then}\ (a \rightarrow x)\ \mathit{else}\ (b \rightarrow x))) \wedge$
$(\Box\,\mathit{len}\ 1 \supset (\text{weaken}\ x \rightarrow x))$

B.3 The Inverting Multiplexer

The current term is:
inverting-mux-struct(a, b, c, Φ, d, p)

Expand all definitions.
The new term is:
two-one-mux$(a, b, c, x, \Phi, m) \wedge$ static-invert(x, d, n)

Expand all definitions.
The new term is:
$(\Box(\Phi = \{1,\_\}) \wedge \mathit{len}\ m \supset$
$\quad (\mathit{if}\ a = \{1,\_\}\ \mathit{then}\ (b \rightarrow x)\ \mathit{else}\ (c \rightarrow x))) \wedge$
$(\Box\,\mathit{len}\ 1 \supset (\text{weaken}\ x \rightarrow x)) \wedge$
$(\Box\,\mathit{len}\ n \supset (\text{strengthen}\ \neg x \rightarrow d))$

Apply rule distrib-box-and 2 times : $(\Box A) \wedge (\Box B) \Leftrightarrow (\Box A \wedge B)$
The new term is:
$\Box((\Phi = \{1,\_\}) \wedge \mathit{len}\ m \supset$
$\quad (\mathit{if}\ a = \{1,\_\}\ \mathit{then}\ (b \rightarrow x)\ \mathit{else}\ (c \rightarrow x))) \wedge$
$\quad (\mathit{len}\ 1 \supset (\text{weaken}\ x \rightarrow x)) \wedge$
$\quad (\mathit{len}\ n \supset (\text{strengthen}\ \neg x \rightarrow d))$

Apply rule if-elim : $(\mathit{if}\ A\ \mathit{then}\ B\ \mathit{else}\ C) = (A \supset B) \wedge (\neg A \supset C)$
The new term is:
$\Box((\Phi = \{1,\_\}) \wedge \mathit{len}\ m \supset$
$\quad (a = \{1,\_\} \supset (b \rightarrow x)) \wedge$
$\quad (\neg(a = \{1,\_\}) \supset (c \rightarrow x))) \wedge$
$\quad (\mathit{len}\ 1 \supset (\text{weaken}\ x \rightarrow x)) \wedge$
$\quad (\mathit{len}\ n \supset (\text{strengthen}\ \neg x \rightarrow d))$

Start case analysis on Φ with values $[\{1,\_\},\{0,\_\}]$.
The current term is:

$$\begin{array}{l}\Box\,\Phi = \{1,\_\} \supset \\ \quad ((\{1,\_\} = \{1,\_\}) \wedge \mathit{len}\ m \supset \\ \qquad\qquad (a = \{1,\_\} \supset (b \to x)) \wedge \\ \qquad\qquad (\neg(a = \{1,\_\}) \supset (c \to x))) \wedge \\ \quad (\mathit{len}\ 1 \supset (\text{weaken } x \to x)) \wedge \\ \quad (\mathit{len}\ n \supset (\text{strengthen } \neg x \to d))\end{array}$$

Apply rule identity : $(A = A) = \{1,\_\}$
Apply rule and-ident-comm : $\{1,\_\} \wedge A = A$
The new term is:

$$\begin{array}{l}\Box\,\Phi = \{1,\_\} \supset \\ \quad (\mathit{len}\ m \supset (a = \{1,\_\} \supset (b \to x)) \wedge \\ \qquad\qquad (\neg(a = \{1,\_\}) \supset (c \to x))) \wedge \\ \quad (\mathit{len}\ 1 \supset (\text{weaken } x \to x)) \wedge \\ \quad (\mathit{len}\ n \supset (\text{strengthen } \neg x \to d))\end{array}$$

Start case analysis on a with values $[\{1,\_\},\{0,\_\}]$.
The current term is:

$$\begin{array}{l}\Box\,a = \{1,\_\} \supset (\Phi = \{1,\_\} \supset \\ \quad (\mathit{len}\ m \supset (\{1,\_\} = \{1,\_\} \supset (b \to x)) \wedge \\ \qquad\qquad (\neg(\{1,\_\} = \{1,\_\}) \supset (c \to x))) \wedge \\ \quad (\mathit{len}\ 1 \supset (\text{weaken } x \to x)) \wedge \\ \quad (\mathit{len}\ n \supset (\text{strengthen } \neg x \to d)))\end{array}$$

Apply rule identity 2 times : $(A = A) = \{1,\_\}$
The new term is:

$$\begin{array}{l}\Box\,a = \{1,\_\} \supset (\Phi = \{1,\_\} \supset \\ \quad (\mathit{len}\ m \supset (\{1,\_\} \supset (b \to x)) \wedge \\ \qquad\qquad (\neg\{1,\_\} \supset (c \to x))) \wedge \\ \quad (\mathit{len}\ 1 \supset (\text{weaken } x \to x)) \wedge \\ \quad (\mathit{len}\ n \supset (\text{strengthen } \neg x \to d)))\end{array}$$

Apply rule not-one : $\neg\{1, A\} = \{0, A\}$
Apply rule implies-true : $(\{1,\_\} \supset A) = A$
The new term is:

$$\begin{array}{l}\Box\,a = \{1,\_\} \supset (\Phi = \{1,\_\} \supset \\ \quad (\mathit{len}\ m \supset (b \to x) \wedge \\ \qquad\qquad (\{0,\_\} \supset (c \to x))) \wedge \\ \quad (\mathit{len}\ 1 \supset (\text{weaken } x \to x)) \wedge \\ \quad (\mathit{len}\ n \supset (\text{strengthen } \neg x \to d)))\end{array}$$

Apply rule implies-false : $(\{0,\_\} \supset A) = \{1,\_\}$
Apply rule and-ident : $A \wedge \{1,\_\} = A$
Apply rule assoc-andl : $A \wedge B \wedge C = A \wedge (B \wedge C)$
The new term is:

$$\begin{array}{l}\Box\,a = \{1,\_\} \supset (\Phi = \{1,\_\} \supset \\ \quad (\mathit{len}\ m \supset (b \to x)) \wedge \\ \quad ((\mathit{len}\ 1 \supset (\text{weaken } x \to x)) \wedge \\ \quad (\mathit{len}\ n \supset (\text{strengthen } \neg x \to d))))\end{array}$$

Replace strengthen $\neg X$ with $func1(X)$ in current term.
The new term with replacements is:

$$\begin{aligned}&\Box a = \{1,\_\} \supset (\Phi = \{1,\_\} \supset \\ &\quad (len\ m \supset (b \rightarrow x)) \wedge \\ &\quad ((len\ 1 \supset (\text{weaken } x \rightarrow x)) \wedge \\ &\quad (len\ n \supset (func1(x) \rightarrow d))))\end{aligned}$$

Rec-apply rule comm-and : $A \wedge B = B \wedge A$
Apply rule assoc-andr : $A \wedge (B \wedge C) = (A \wedge B) \wedge C$
Apply rule trans-unclock-func1 :

$$\begin{aligned}&(len\ M \supset (A \rightarrow B)) \wedge \\ &(len\ N \supset (func1(B) \rightarrow C)) \wedge \\ &(len\ 1 \supset (\text{weaken } B \rightarrow B)) \\ &\quad \supset (len\ (M+N) \supset (func1(A) \rightarrow C))\end{aligned}$$

The new term is:

$$\begin{aligned}&\Box a = \{1,\_\} \supset (\Phi = \{1,\_\} \supset \\ &\quad (len\ (m+n) \supset (func1(b) \rightarrow d)))\end{aligned}$$

Replace $func1(Y)$ with strengthen $\neg Y$ in current term.
The new term with replacements is:

$$\begin{aligned}&\Box a = \{1,\_\} \supset (\Phi = \{1,\_\} \supset \\ &\quad (len\ (m+n) \supset (\text{strengthen } \neg b \rightarrow d)))\end{aligned}$$

Do next case of case analysis.
The current term is:

$$\begin{aligned}&\Box a = \{0,\_\} \supset (\Phi = \{1,\_\} \supset \\ &\quad (len\ m \supset ((\{0,\_\} = \{1,\_\} \supset (b \rightarrow x)) \wedge \\ &\quad\quad (\neg(\{0,\_\} = \{1,\_\}) \supset (c \rightarrow x))) \wedge \\ &\quad (len\ 1 \supset (\text{weaken } x \rightarrow x)) \wedge \\ &\quad (len\ n \supset (\text{strengthen } \neg x \rightarrow d)))\end{aligned}$$

Apply rule falsity 2 times : $(\{0,\_\} = \{1,\_\}) = \{0,\_\}$
Apply rule not-zero : $\neg\{0, A\} = \{1, A\}$
The new term is:

$$\begin{aligned}&\Box a = \{0,\_\} \supset (\Phi = \{1,\_\} \supset \\ &\quad (len\ m \supset (\{0,\_\} \supset (b \rightarrow x)) \wedge \\ &\quad\quad (\{1,\_\} \supset (c \rightarrow x))) \wedge \\ &\quad (len\ 1 \supset (\text{weaken } x \rightarrow x)) \wedge \\ &\quad (len\ n \supset (\text{strengthen } \neg x \rightarrow d)))\end{aligned}$$

Apply rule implies-false : $(\{0,\_\} \supset A) = \{1,\_\}$
Apply rule implies-true : $(\{1,\_\} \supset A) = A$
Apply rule and-ident-comm : $\{1,\_\} \wedge A = A$
Apply rule assoc-andl : $A \wedge B \wedge C = A \wedge (B \wedge C)$
The new term is:

$$\begin{aligned}&\Box a = \{0,\_\} \supset (\Phi = \{1,\_\} \supset \\ &\quad (len\ m \supset (c \rightarrow x)) \wedge \\ &\quad ((len\ 1 \supset (\text{weaken } x \rightarrow x)) \wedge \\ &\quad (len\ n \supset (\text{strengthen } \neg x \rightarrow d))))\end{aligned}$$

Replace strengthen $\neg Z$ with $func1(Z)$ in current term.
The new term with replacements is:

$$\begin{aligned}&\Box\, a = \{0,\_\} \supset (\Phi = \{1,\_\} \supset \\ &\quad (len\ m \supset (c \rightarrow x)) \wedge \\ &\quad ((len\ 1 \supset (\text{weaken } x \rightarrow x)) \wedge \\ &\quad (len\ n \supset (func1(x) \rightarrow d))))\end{aligned}$$

Rec-apply rule comm-and : $A \wedge B = B \wedge A$
Apply rule assoc-andr : $A \wedge (B \wedge C) = (A \wedge B) \wedge C$
Apply rule trans-unclock-func1 :

$$\begin{aligned}&(len\ M \supset (A \rightarrow B)) \wedge \\ &(len\ N \supset (func1(B) \rightarrow C)) \wedge \\ &(len\ 1 \supset (\text{weaken } B \rightarrow B)) \\ &\quad \supset (len\ (M+N) \supset (func1(A) \rightarrow C))\end{aligned}$$

The new term is:

$$\begin{aligned}&\Box\, a = \{0,\_\} \supset (\Phi = \{1,\_\} \supset \\ &\quad (len\ (m+n) \supset (func1(c) \rightarrow d)))\end{aligned}$$

Replace $func1(W)$ with strengthen $\neg W$ in current term.
The new term with replacements is:

$$\begin{aligned}&\Box\, a = \{0,\_\} \supset (\Phi = \{1,\_\} \supset \\ &\quad (len\ (m+n) \supset (\text{strengthen } \neg c \rightarrow d)))\end{aligned}$$

End case analysis.
The result of case analysis is:

$$\begin{aligned}&(\Box\, a = \{1,\_\} \supset (\Phi = \{1,\_\} \supset \\ &\qquad (len\ (m+n) \supset (\text{strengthen } \neg b \rightarrow d)))) \wedge \\ &(\Box\, a = \{0,\_\} \supset (\Phi = \{1,\_\} \supset \\ &\qquad (len\ (m+n) \supset (\text{strengthen } \neg c \rightarrow d))))\end{aligned}$$

Apply rule distrib-box-and : $(\Box A) \wedge (\Box B) \Leftrightarrow (\Box A \wedge B)$
The new term is:

$$\begin{aligned}&\Box(a = \{1,\_\} \supset (\Phi = \{1,\_\} \supset \\ &\qquad (len\ (m+n) \supset (\text{strengthen } \neg b \rightarrow d)))) \wedge \\ &\quad (a = \{0,\_\} \supset (\Phi = \{1,\_\} \supset \\ &\qquad (len\ (m+n) \supset (\text{strengthen } \neg c \rightarrow d))))\end{aligned}$$

Apply rule import 4 times : $(A \supset (B \supset C)) = (A \wedge B \supset C)$
The new term is:

$$\begin{aligned}&\Box((a = \{1,\_\}) \wedge (\Phi = \{1,\_\}) \wedge len\ (m+n) \supset \\ &\qquad (\text{strengthen } \neg b \rightarrow d)) \wedge \\ &\quad ((a = \{0,\_\}) \wedge (\Phi = \{1,\_\}) \wedge len\ (m+n) \supset \\ &\qquad (\text{strengthen } \neg c \rightarrow d))\end{aligned}$$

Apply rule assoc-andl 2 times : $A \wedge B \wedge C = A \wedge (B \wedge C)$
Rec-apply rule comm-and : $A \wedge B = B \wedge A$
Apply rule export : $(A \wedge B \supset C) = (A \supset (B \supset C))$
The new term is:

$$\begin{aligned}&\Box((\Phi = \{1,\_\}) \wedge len\ (m+n) \supset \\ &\qquad (a = \{1,\_\} \supset (\text{strengthen } \neg b \rightarrow d))) \wedge \\ &\quad ((\Phi = \{1,\_\}) \wedge len\ (m+n) \supset \\ &\qquad (a = \{0,\_\} \supset (\text{strengthen } \neg c \rightarrow d)))\end{aligned}$$

Apply rule and-intror : $(A \supset B) \wedge (A \supset C) = (A \supset B \wedge C)$
The new term is:
$\square(\Phi = \{1,\_\}) \wedge len\ (m+n) \supset$
$(a = \{1,\_\} \supset (\text{strengthen } \neg b \rightarrow d)) \wedge$
$(a = \{0,\_\} \supset (\text{strengthen } \neg c \rightarrow d))$

Replace $a = \{0,\_\}$ with $\neg(a = \{1,\_\})$ in current term.
First show:
$\Phi = \{1,\_\} \supset (a = \{0,\_\}) = \neg(a = \{1,\_\})$

Apply rule equals-true : $(A = \{1,\_\}) = A$
The new term is:
$\Phi \supset (a = \{0,\_\}) = \neg(a = \{1,\_\})$

Apply rule equals-true : $(A = \{1,\_\}) = A$
The new term is:
$\Phi \supset (a = \{0,\_\}) = \neg a$

Apply rule equals-false : $(A = \{0,\_\}) = \neg A$
The new term is:
$\Phi \supset \neg a = \neg a$

Apply rule identity : $(A = A) = \{1,\_\}$
The new term is:
$\Phi \supset \{1,\_\}$

Apply rule implied-true : $(A \supset \{1,\_\}) = \{1,\_\}$
The new term is:
$\{1,\_\}$
The terms being replaced are equivalent
The new term with replacements is:
$\square(\Phi = \{1,\_\}) \wedge len\ (m+n) \supset$
$(a = \{1,\_\} \supset (\text{strengthen } \neg b \rightarrow d)) \wedge$
$(\neg(a = \{1,\_\}) \supset (\text{strengthen } \neg c \rightarrow d))$

Apply rule if-intro : $(A \supset B) \wedge (\neg A \supset C) = (\textit{if } A \textit{ then } B \textit{ else } C)$
The new term is:
$\square(\Phi = \{1,\_\}) \wedge len\ (m+n) \supset$
$(\textit{if } a = \{1,\_\} \textit{ then } (\text{strengthen } \neg b \rightarrow d)$
$\textit{else } (\text{strengthen } \neg c \rightarrow d))$

Do next case of case analysis.
The current term is:
$\square\, \Phi = \{0,\_\} \supset$
$((\{0,\_\} = \{1,\_\}) \wedge len\ m \supset$
$(a = \{1,\_\} \supset (b \rightarrow x)) \wedge$
$(\neg(a = \{1,\_\}) \supset (c \rightarrow x))) \wedge$
$(len\ 1 \supset (\text{weaken } x \rightarrow x)) \wedge$
$(len\ n \supset (\text{strengthen } \neg x \rightarrow d))$

Apply rule falsity : $(\{0,\_\} = \{1,\_\}) = \{0,\_\}$
Apply rule and-false-comm : $\{0, B\} \wedge A = \{0, B\}$
The new term is:

$$\begin{array}{l} \Box\,\Phi = \{0,\_\} \supset \\ \quad (\{0,\_\} \supset \\ \qquad (a = \{1,\_\} \supset (b \rightarrow x)) \wedge \\ \qquad (\neg(a = \{1,\_\}) \supset (c \rightarrow x))) \wedge \\ \quad (len\ 1 \supset (\text{weaken } x \rightarrow x)) \wedge \\ \quad (len\ n \supset (\text{strengthen } \neg x \rightarrow d)) \end{array}$$

Apply rule implies-false : $(\{0,\_\} \supset A) = \{1,\_\}$
Apply rule and-ident-comm : $\{1,\_\} \wedge A = A$
The new term is:

$$\begin{array}{l} \Box\,\Phi = \{0,\_\} \supset \\ \quad (len\ 1 \supset (\text{weaken } x \rightarrow x)) \wedge \\ \quad (len\ n \supset (\text{strengthen } \neg x \rightarrow d)) \end{array}$$

Replace strengthen $\neg V$ with $func1(V)$ in current term.
The new term with replacements is:

$$\begin{array}{l} \Box\,\Phi = \{0,\_\} \supset \\ \quad (len\ 1 \supset (\text{weaken } x \rightarrow x)) \wedge \\ \quad (len\ n \supset (func1(x) \rightarrow d)) \end{array}$$

Apply rule pass-stb-weaken :

$$\begin{array}{l} (len\ \ 1 \supset (\text{weaken } A \rightarrow A)) \wedge \\ (len\ \ N \supset (func1(A) \rightarrow B)) \\ \quad \supset \bigcirc^N stb\ B \end{array}$$

The new term is:

$$\Box\,\Phi = \{0,\_\} \supset \bigcirc^n stb\ d$$

End case analysis.
The result of case analysis is:

$$\begin{array}{l} (\Box(\Phi = \{1,\_\}) \wedge len\ (m+n) \supset \\ \qquad (if\ a = \{1,\_\}\ then\ (\text{strengthen } \neg b \rightarrow d) \\ \qquad\qquad\qquad\qquad else\ (\text{strengthen } \neg c \rightarrow d))) \wedge \\ (\Box\,\Phi = \{0,\_\} \supset \bigcirc^n stb\ d) \end{array}$$

Apply rule distrib-box-and : $(\Box\, A) \wedge (\Box\, B) \Leftrightarrow (\Box\, A \wedge B)$
The new term is:

$$\begin{array}{l} \Box((\Phi = \{1,\_\}) \wedge len\ (m+n) \supset \\ \qquad (if\ a = \{1,\_\}\ then\ (\text{strengthen } \neg b \rightarrow d) \\ \qquad\qquad\qquad\qquad else\ (\text{strengthen } \neg c \rightarrow d))) \wedge \\ \qquad (\Phi = \{0,\_\} \supset \bigcirc^n stb\ d)) \end{array}$$

B.4 The Shiftstage

The current term is:

shift-stage-struct(a, b, Φ, m)

Expand all definitions.
The new term is:
  npass(Φ, a, c, n) $\wedge$ cap(c) $\wedge$ static-invert(c, b, p)

Expand all definitions.
The new term is:
  $(\Box(\Phi = \{1,\_\}) \wedge \mathit{len}\ n \supset (a \rightarrow c)) \wedge$
  $(\Box\, \mathit{len}\ 1 \supset (\text{weaken } c \rightarrow c)) \wedge$
  $(\Box\, \mathit{len}\ p \supset (\text{strengthen } \neg c \rightarrow b))$

Apply rule distrib-box-and 2 times : $(\Box A) \wedge (\Box B) \Leftrightarrow (\Box A \wedge B)$
The new term is:
  $\Box((\Phi = \{1,\_\}) \wedge \mathit{len}\ n \supset (a \rightarrow c)) \wedge$
    $(\mathit{len}\ 1 \supset (\text{weaken } c \rightarrow c)) \wedge$
    $(\mathit{len}\ p \supset (\text{strengthen } \neg c \rightarrow b))$

Start case analysis on Φ with values $[\{1,\_\}, \{0,\_\}]$.
The current term is:
  $\Box\, \Phi = \{1,\_\} \supset$
    $((\{1,\_\} = \{1,\_\}) \wedge \mathit{len}\ n \supset (a \rightarrow c)) \wedge$
    $(\mathit{len}\ 1 \supset (\text{weaken } c \rightarrow c)) \wedge$
    $(\mathit{len}\ p \supset (\text{strengthen } \neg c \rightarrow b))$

Apply rule identity : $(A = A) = \{1,\_\}$
Apply rule and-ident-comm : $\{1,\_\} \wedge A = A$
The new term is:
  $\Box\, \Phi = \{1,\_\} \supset$
    $(\mathit{len}\ n \supset (a \rightarrow c)) \wedge$
    $(\mathit{len}\ 1 \supset (\text{weaken } c \rightarrow c)) \wedge$
    $(\mathit{len}\ p \supset (\text{strengthen } \neg c \rightarrow b))$

Apply rule assoc-andl : $A \wedge B \wedge C = A \wedge (B \wedge C)$
Rec-apply rule comm-and : $A \wedge B = B \wedge A$
Apply rule assoc-andr : $A \wedge (B \wedge C) = (A \wedge B) \wedge C$
The new term is:
  $\Box\, \Phi = \{1,\_\} \supset$
    $(\mathit{len}\ n \supset (a \rightarrow c)) \wedge$
    $(\mathit{len}\ p \supset (\text{strengthen } \neg c \rightarrow b)) \wedge$
    $(\mathit{len}\ 1 \supset (\text{weaken } c \rightarrow c))$

Replace a with $\mathit{func1}(a)$ in current term.
Replace strengthen $\neg X2$ with $\mathit{func2}(X2)$ in current term.
The new term with replacements is:
  $\Box\, \Phi = \{1,\_\} \supset$
    $(\mathit{len}\ n \supset (\mathit{func1}(a) \rightarrow c)) \wedge$
    $(\mathit{len}\ p \supset (\mathit{func2}(c) \rightarrow b)) \wedge$
    $(\mathit{len}\ 1 \supset (\text{weaken } c \rightarrow c))$

Apply rule trans-cap-combine :
  $(\mathit{len}\ M \supset (\mathit{func1}(A) \rightarrow B)) \wedge$
  $(\mathit{len}\ N \supset (\mathit{func2}(B) \rightarrow C)) \wedge$
  $(\mathit{len}\ 1 \supset (\text{weaken } B \rightarrow B))$
    $\supset (\mathit{len}\ (M + N) \supset (\mathit{func2}(\mathit{func1}(A)) \rightarrow C))$

The new term is:

$$\Box\,\Phi = \{1,\_\} \supset \\ (len\ (n+p) \supset (func2(func1(a)) \rightarrow b))$$

Replace $func2(X3)$ with strengthen $\neg X3$ in current term.
Replace $func1(a)$ with a in current term.
The new term with replacements is:

$$\Box\,\Phi = \{1,\_\} \supset\ (len\ (n+p) \supset (\text{strengthen } \neg a \rightarrow b))$$

Do next case of case analysis.
The current term is:

$$\Box\,\Phi = \{0,\_\} \supset \\ ((\{0,\_\} = \{1,\_\})\ \wedge\ len\ n \supset (a \rightarrow c)) \wedge \\ (len\ 1 \supset (\text{weaken } c \rightarrow c)) \wedge \\ (len\ p \supset (\text{strengthen } \neg c \rightarrow b))$$

Apply rule falsity : $(\{0,\_\} = \{1,\_\}) = \{0,\_\}$
Apply rule and-false-comm : $\{0,B\}\ \wedge\ A = \{0,B\}$
The new term is:

$$\Box\,\Phi = \{0,\_\} \supset \\ (\{0,\_\} \supset (a \rightarrow c)) \wedge \\ (len\ 1 \supset (\text{weaken } c \rightarrow c)) \wedge \\ (len\ p \supset (\text{strengthen } \neg c \rightarrow b))$$

Apply rule implies-false : $(\{0,\_\} \supset A) = \{1,\_\}$
Apply rule and-ident-comm : $\{1,\_\}\ \wedge\ A = A$
The new term is:

$$\Box\,\Phi = \{0,\_\} \supset \\ (len\ 1 \supset (\text{weaken } c \rightarrow c)) \wedge \\ (len\ p \supset (\text{strengthen } \neg c \rightarrow b))$$

Replace strengthen $\neg c$ with $func1(c)$ in current term.
The new term with replacements is:

$$\Box\,\Phi = \{0,\_\} \supset \\ (len\ 1 \supset (\text{weaken } c \rightarrow c)) \wedge \\ (len\ p \supset (func1(c) \rightarrow b))$$

Apply rule pass-stb-weaken :

$$(len\ \ 1 \supset (\text{weaken } A \rightarrow A)) \wedge \\ (len\ \ N \supset (func1(A) \rightarrow B)) \\ \supset \bigcirc^{N}\, stb\ B$$

The new term is:

$$\Box\,\Phi = \{0,\_\} \supset\ \bigcirc^{p}\, stb\ b$$

End case analysis.
The result of case analysis is:

$$(\Box\,\Phi = \{1,\_\} \supset (len\ (n+p) \supset\ (\text{strengthen } \neg a \rightarrow b))) \wedge \\ (\Box\,\Phi = \{0,\_\} \supset\ \bigcirc^{p}\, stb\ b)$$

Replace $\Phi = \{0,\_\}$ with $\neg(\Phi = \{1,\_\})$ in current term.
First show:

$$(\Phi = \{0,\_\}) = \neg(\Phi = \{1,\_\})$$

Apply rule equals-true : $(A = \{1, \_\}) = A$
The new term is:
$(\Phi = \{0, \_\}) = \neg\Phi$

Apply rule equals-false : $(A = \{0, \_\}) = \neg A$
The new term is:
$\neg\Phi = \neg\Phi$

Apply rule identity : $(A = A) = \{1, \_\}$
The new term is:
$\{1, \_\}$
The terms being replaced are equivalent

The new term with replacements is:
$(\Box\, \Phi = \{1, \_\} \supset (\mathit{len}\ (n+p) \supset \text{(strengthen } \neg a \longrightarrow b))) \wedge$
$(\Box\, \neg(\Phi = \{1, \_\}) \supset \bigcirc^{p}\, \mathit{stb}\ b)$

B.5 The Dynamic Latch

The current term is:
dlatch-struct$(l, d, q, \Phi_1, \Phi_2, r)$

Expand all definitions.
The new term is:
invert-mux$(l, d, q, \Phi_1, x, m) \wedge$ shift-stage(x, q, Φ_2, p)

Expand all definitions.
The new term is:
$(\Box((\Phi_1 = \{1, \_\}) \ \wedge\ \mathit{len}\ m \supset$
$(\mathit{if}\ l = \{1, \_\}\ \mathit{then}\ \text{(strengthen } \neg d \longrightarrow x)$
$\mathit{else}\ \text{(strengthen } \neg q \longrightarrow x))) \wedge$
$(\Phi_1 = \{0, \_\} \supset \mathit{stb}\ x)) \wedge$
$((\Box\, \Phi_2 = \{1, \_\} \supset$
$(\mathit{len}\ p \supset \text{(strengthen } \neg x \longrightarrow q))) \wedge$
$(\Box\, \neg(\Phi_2 = \{1, \_\}) \supset \mathit{stb}\ q))$

Split current term.
The new term is:
$(\Box((\Phi_1 = \{1, \_\}) \ \wedge\ \mathit{len}\ m \supset$
$(\mathit{if}\ l = \{1, \_\}\ \mathit{then}\ \text{(strengthen } \neg d \longrightarrow x)$
$\mathit{else}\ \text{(strengthen } \neg q \longrightarrow x))) \wedge$
$(\Phi_1 = \{0, \_\} \supset \mathit{stb}\ x)) \wedge$
$(\Box\, \Phi_2 = \{1, \_\} \supset (\mathit{len}\ p \supset \text{(strengthen } \neg x \longrightarrow q)))$
The term saved on the stack is:
$\Box\, \neg(\Phi_2 = \{1, \_\}) \supset \mathit{stb}\ q$

Apply rule distrib-box-and : $(\Box A) \wedge (\Box B) \Leftrightarrow (\Box A \wedge B)$
The new term is:

$$\begin{array}{l} \Box((\Phi_1 = \{1,\_\}) \wedge len\ m \supset \\ \quad (if\ l = \{1,\_\}\ then\ (\text{strengthen}\ \neg d \rightarrow x) \\ \qquad else\ (\text{strengthen}\ \neg q \rightarrow x))) \wedge \\ \quad (\Phi_1 = \{0,\_\} \supset stb\ x) \wedge \\ \quad (\Phi_2 = \{1,\_\} \supset (len\ p \supset (\text{strengthen}\ \neg x \rightarrow q))) \end{array}$$

Start case analysis on Φ_2 with values $[\{0,\_\},\{1,\_\}]$
The current term is:

$$\begin{array}{l} \Box\,\Phi_2 = \{0,\_\} \supset \\ \quad ((\Phi_1 = \{1,\_\}) \wedge len\ m \supset \\ \qquad (if\ l = \{1,\_\}\ then\ (\text{strengthen}\ \neg d \rightarrow x) \\ \qquad\quad else\ (\text{strengthen}\ \neg q \rightarrow x))) \wedge \\ \quad (\Phi_1 = \{0,\_\} \supset stb\ x) \wedge \\ \quad (\{0,\_\} = \{1,\_\} \supset \\ \qquad (len\ p \supset (\text{strengthen}\ \neg x \rightarrow q))) \end{array}$$

Apply rule falsity : $(\{0,\_\} = \{1,\_\}) = \{0,\_\}$
Apply rule implies-false : $(\{0,\_\} \supset A) = \{1,\_\}$
Apply rule and-ident : $A \wedge \{1,\_\} = A$
The new term is:

$$\begin{array}{l} \Box\,\Phi_2 = \{0,\_\} \supset \\ \quad ((\Phi_1 = \{1,\_\}) \wedge len\ m \supset \\ \qquad (if\ l = \{1,\_\}\ then\ (\text{strengthen}\ \neg d \rightarrow x) \\ \qquad\quad else\ (\text{strengthen}\ \neg q \rightarrow x))) \wedge \\ \quad (\Phi_1 = \{0,\_\} \supset stb\ x) \end{array}$$

Start case analysis on Φ_1 with values $[\{0,\_\},\{1,\_\}]$
The current term is:

$$\begin{array}{l} \Box\,\Phi_1 = \{0,\_\} \supset (\Phi_2 = \{0,\_\} \supset \\ \quad ((\{0,\_\} = \{1,\_\}) \wedge len\ m \supset \\ \qquad (if\ l = \{1,\_\}\ then\ (\text{strengthen}\ \neg d \rightarrow x) \\ \qquad\quad else\ (\text{strengthen}\ \neg q \rightarrow x))) \wedge \\ \quad (\{0,\_\} = \{0,\_\} \supset stb\ x)) \end{array}$$

Apply rule falsity : $(\{0,\_\} = \{1,\_\}) = \{0,\_\}$
Apply rule and-false-comm : $\{0,B\} \wedge A = \{0,B\}$
The new term is:

$$\begin{array}{l} \Box\,\Phi_1 = \{0,\_\} \supset (\Phi_2 = \{0,\_\} \supset \\ \quad (\{0,\_\} \supset \\ \qquad (if\ l = \{1,\_\}\ then\ (\text{strengthen}\ \neg d \rightarrow x) \\ \qquad\quad else\ (\text{strengthen}\ \neg q \rightarrow x))) \wedge \\ \quad (\{0,\_\} = \{0,\_\} \supset stb\ x)) \end{array}$$

Apply rule implies-false : $(\{0,\_\} \supset A) = \{1,\_\}$
Apply rule and-ident-comm : $\{1,\_\} \wedge A = A$
The new term is:

$$\begin{array}{l} \Box\,\Phi_1 = \{0,\_\} \supset (\Phi_2 = \{0,\_\} \supset \\ \quad (\{0,\_\} = \{0,\_\} \supset stb\ x)) \end{array}$$

Apply rule identity : $(A = A) = \{1, \_\}$
Apply rule implies-true : $(\{1, \_\} \supset A) = A$
The new term is:

$$\Box\, \Phi_1 = \{0, \_\} \supset (\Phi_2 = \{0, \_\} \supset stb\ x)$$

Do next case of case analysis.
The current term is:

$$\begin{aligned}&\Box\, \Phi_1 = \{1, \_\} \supset (\Phi_2 = \{0, \_\} \supset \\ &\quad ((\{1, \_\} = \{1, \_\}) \wedge len\ m \supset \\ &\qquad (if\ l = \{1, \_\}\ then\ (\text{strengthen } \neg d \rightarrow x) \\ &\qquad\qquad else\ (\text{strengthen } \neg q \rightarrow x))) \wedge \\ &\quad (\{1, \_\} = \{0, \_\} \supset stb\ x))\end{aligned}$$

Apply rule identity : $(A = A) = \{1, \_\}$
Apply rule and-ident-comm : $\{1, \_\} \wedge A = A$
The new term is:

$$\begin{aligned}&\Box\, \Phi_1 = \{1, \_\} \supset (\Phi_2 = \{0, \_\} \supset \\ &\quad (len\ m \supset (if\ l = \{1, \_\}\ then\ (\text{strengthen } \neg d \rightarrow x) \\ &\qquad\qquad else(\text{strengthen } \neg q \rightarrow x))) \wedge \\ &\quad (\{1, \_\} = \{0, \_\} \supset stb\ x))\end{aligned}$$

Apply rule falsity-comm : $(\{1, \_\} = \{0, \_\}) = \{0, \_\}$
Apply rule implies-false : $(\{0, \_\} \supset A) = \{1, \_\}$
Apply rule and-ident : $\{1, \_\} \wedge A = A$
The new term is:

$$\begin{aligned}&\Box\, \Phi_1 = \{1, \_\} \supset (\Phi_2 = \{0, \_\} \supset \\ &\quad (len\ m \supset (if\ l = \{1, \_\}\ then\ (\text{strengthen } \neg d \rightarrow x) \\ &\qquad\qquad else\ (\text{strengthen } \neg q \rightarrow x))))\end{aligned}$$

End case analysis.
The result of case analysis is:

$$\begin{aligned}&(\Box\, \Phi_1 = \{0, \_\} \supset (\Phi_2 = \{0, \_\} \supset stb\ x)) \wedge \\ &(\Box\, \Phi_1 = \{1, \_\} \supset (\Phi_2 = \{0, \_\} \supset \\ &\quad (len\ m \supset (if\ l = \{1, \_\}\ then\ (\text{strengthen } \neg d \rightarrow x) \\ &\qquad\qquad else\ (\text{strengthen } \neg q \rightarrow x)))))\end{aligned}$$

Apply rule distrib-box-and : $(\Box\, A) \wedge (\Box\, B) \Leftrightarrow (\Box\, A \wedge B)$
The new term is:

$$\begin{aligned}&\Box(\Phi_1 = \{0, \_\} \supset (\Phi_2 = \{0, \_\} \supset stb\ x)) \wedge \\ &\quad (\Phi_1 = \{1, \_\} \supset (\Phi_2 = \{0, \_\} \supset \\ &\qquad (len\ m \supset (if\ l = \{1, \_\}\ then\ (\text{strengthen } \neg d \rightarrow x) \\ &\qquad\qquad else\ (\text{strengthen } \neg q \rightarrow x)))))\end{aligned}$$

Do next case of case analysis.
The current term is:

$$\begin{aligned}&\Box\, \Phi_2 = \{1, \_\} \supset \\ &\quad ((\Phi_1 = \{1, \_\}) \wedge len\ m \supset \\ &\qquad (if\ l = \{1, \_\}\ then\ (\text{strengthen } \neg d \rightarrow x) \\ &\qquad\qquad else\ (\text{strengthen } \neg q \rightarrow x))) \wedge \\ &\quad (\Phi_1 = \{0, \_\} \supset stb\ x) \wedge \\ &\quad (\{1, \_\} = \{1, \_\} \supset (len\ p \supset (\text{strengthen } \neg x \rightarrow q)))\end{aligned}$$

Apply rule identity : $(A = A) = \{1, \_\}$
Apply rule implies-true : $(\{1, \_\} \supset A) = A$
The new term is:

$$\begin{aligned}
&\Box\, \Phi_2 = \{1, \_\} \supset \\
&\quad ((\Phi_1 = \{1, \_\}) \wedge \mathit{len}\; m \supset \\
&\qquad (\mathit{if}\; l = \{1, \_\}\; \mathit{then}\; (\text{strengthen } \neg d \rightarrow x) \\
&\qquad\qquad \mathit{else}\; (\text{strengthen } \neg q \rightarrow x))) \wedge \\
&\quad (\Phi_1 = \{0, \_\} \supset \mathit{stb}\; x) \wedge \\
&\quad (\mathit{len}\; p \supset (\text{strengthen } \neg x \rightarrow q))
\end{aligned}$$

Start case analysis on Φ_1 with values $[\{0, \_\}]$
The current term is:

$$\begin{aligned}
&\Box\, \Phi_1 = \{0, \_\} \supset (\Phi_2 = \{1, \_\} \supset \\
&\quad ((\{0, \_\} = \{1, \_\}) \wedge \mathit{len}\; m \supset \\
&\qquad (\mathit{if}\; l = \{1, \_\}\; \mathit{then}\; (\text{strengthen } \neg d \rightarrow x) \\
&\qquad\qquad \mathit{else}(\text{strengthen } \neg q \rightarrow x))) \wedge \\
&\quad (\{0, \_\} = \{0, \_\} \supset \mathit{stb}\; x) \wedge \\
&\quad (\mathit{len}\; p \supset (\text{strengthen } \neg x \rightarrow q)))
\end{aligned}$$

Apply rule falsity : $(\{0, \_\} = \{1, \_\}) = \{0, \_\}$
Apply rule and-false-comm : $\{0, B\} \wedge A = \{0, B\}$
The new term is:

$$\begin{aligned}
&\Box\, \Phi_1 = \{0, \_\} \supset (\Phi_2 = \{1, \_\} \supset \\
&\quad (\{0, \_\} \supset \\
&\qquad (\mathit{if}\; l = \{1, \_\}\; \mathit{then}\; (\text{strengthen } \neg d \rightarrow x) \\
&\qquad\qquad \mathit{else}\; (\text{strengthen } \neg q \rightarrow x))) \wedge \\
&\qquad (\{0, \_\} = \{0, \_\} \supset \mathit{stb}\; x) \wedge \\
&\qquad (\mathit{len}\; p \supset (\text{strengthen } \neg x \rightarrow q)))
\end{aligned}$$

Apply rule implies-false : $(\{0, \_\} \supset A) = \{1, \_\}$
Apply rule and-ident-comm : $\{1, \_\} \wedge A = A$
The new term is:

$$\begin{aligned}
&\Box\, \Phi_1 = \{0, \_\} \supset (\Phi_2 = \{1, \_\} \supset \\
&\quad (\{0, \_\} = \{0, \_\} \supset \mathit{stb}\; x) \wedge \\
&\quad (\mathit{len}\; p \supset (\text{strengthen } \neg x \rightarrow q)))
\end{aligned}$$

Apply rule identity : $(A = A) = \{1, \_\}$
Apply rule implies-true : $(\{1, \_\} \supset A) = A$
The new term is:

$$\begin{aligned}
&\Box\, \Phi_1 = \{0, \_\} \supset (\Phi_2 = \{1, \_\} \supset \\
&\qquad \mathit{stb}\; x \wedge (\mathit{len}\; p \supset (\text{strengthen } \neg x \rightarrow q)))
\end{aligned}$$

Apply rule and-elimr : $A \wedge B \supset B$
The new term is:

$$\begin{aligned}
&\Box\, \Phi_1 = \{0, \_\} \supset (\Phi_2 = \{1, \_\} \supset \\
&\quad (\mathit{len}\; p \supset (\text{strengthen } \neg x \rightarrow q)))
\end{aligned}$$

End case analysis.
The result of case analysis is:

$$\begin{aligned}
&\Box\, \Phi_1 = \{0, \_\} \supset (\Phi_2 = \{1, \_\} \supset \\
&\quad (\mathit{len}\; p \supset (\text{strengthen } \neg x \rightarrow q)))
\end{aligned}$$

End case analysis.
The result of case analysis is:

$$\begin{array}{l}
(\Box(\Phi_1 = \{0,\_\} \supset (\Phi_2 = \{0,\_\} \supset \mathit{stb}\ x)) \wedge \\
\quad (\Phi_1 = \{1,\_\} \supset (\Phi_2 = \{0,\_\} \supset \\
\qquad (\mathit{len}\ m \supset (\mathit{if} l = \{1,\_\}\ \mathit{then}\ (\text{strengthen } \neg d \rightarrow x) \\
\qquad\qquad\qquad \mathit{else}\ (\text{strengthen } \neg q \rightarrow x)))))) \wedge \\
(\Box\, \Phi_1 = \{0,\_\} \supset (\Phi_2 = \{1,\_\} \supset \\
\qquad (\mathit{len}\ p \supset (\text{strengthen } \neg x \rightarrow q))))
\end{array}$$

Apply rule distrib-box-and : $(\Box A) \wedge (\Box B) \Leftrightarrow (\Box A \wedge B)$
The new term is:

$$\begin{array}{l}
\Box(\Phi_1 = \{0,\_\} \supset (\Phi_2 = \{0,\_\} \supset \mathit{stb}\ x)) \wedge \\
\quad (\Phi_1 = \{1,\_\} \supset (\Phi_2 = \{0,\_\} \supset \\
\qquad (\mathit{len}\ m \supset (\mathit{if}\ l = \{1,\_\}\ \mathit{then}\ (\text{strengthen } \neg d \rightarrow x) \\
\qquad\qquad\qquad \mathit{else}\ (\text{strengthen } \neg q \rightarrow x))))) \wedge \\
\quad (\Phi_1 = \{0,\_\} \supset (\Phi_2 = \{1,\_\} \supset \\
\qquad (\mathit{len}\ p \supset (\text{strengthen } \neg x \rightarrow q))))
\end{array}$$

Apply rule import 3 times : $(A \supset (B \supset C)) = (A \wedge B \supset C)$
The new term is:

$$\begin{array}{l}
\Box((\Phi_1 = \{0,\_\}) \wedge (\Phi_2 = \{0,\_\}) \supset \mathit{stb}\ x) \wedge \\
\quad ((\Phi_1 = \{1,\_\}) \wedge (\Phi_2 = \{0,\_\}) \supset \\
\qquad (\mathit{len}\ m \supset (\mathit{if}\ l = \{1,\_\}\ \mathit{then}\ (\text{strengthen } \neg d \rightarrow x) \\
\qquad\qquad\qquad \mathit{else}\ (\text{strengthen } \neg q \rightarrow x)))) \wedge \\
\quad ((\Phi_1 = \{0,\_\}) \wedge (\Phi_2 = \{1,\_\}) \supset \\
\qquad (\mathit{len}\ p \supset (\text{strengthen } \neg x \rightarrow q)))
\end{array}$$

Apply rule if-elim : $(\mathit{if}\ A\ \mathit{then}\ B\ \mathit{else}\ C) = (A \supset B) \wedge (\neg A \supset C)$
The new term is:

$$\begin{array}{l}
\Box((\Phi_1 = \{0,\_\}) \wedge (\Phi_2 = \{0,\_\}) \supset \mathit{stb}\ x) \wedge \\
\quad ((\Phi_1 = \{1,\_\}) \wedge (\Phi_2 = \{0,\_\}) \supset \\
\qquad (\mathit{len}\ m \supset (l = \{1,\_\} \supset \\
\qquad\qquad (\text{strengthen } \neg d \rightarrow x)) \wedge \\
\qquad (\neg(l = \{1,\_\}) \supset (\text{strengthen } \neg q \rightarrow x)))) \wedge \\
\quad ((\Phi_1 = \{0,\_\}) \wedge (\Phi_2 = \{1,\_\}) \supset \\
\qquad (\mathit{len}\ p \supset (\text{strengthen } \neg x \rightarrow q)))
\end{array}$$

Start case analysis on l with values $[\{0,\_\}, \{1,\_\}]$
The current term is:

$$\begin{array}{l}
\Box\, l = \{0,\_\} \supset \\
\qquad ((\Phi_1 = \{0,\_\}) \wedge (\Phi_2 = \{0,\_\}) \supset \mathit{stb}\ x) \wedge \\
\qquad ((\Phi_1 = \{1,\_\}) \wedge (\Phi_2 = \{0,\_\}) \supset \\
\qquad\quad (\mathit{len}\ m \supset (\{0,\_\} = \{1,\_\} \supset (\text{strengthen } \neg d \rightarrow x)) \wedge \\
\qquad\qquad\qquad (\neg(\{0,\_\} = \{1,\_\}) \supset (\text{strengthen } \neg q \rightarrow x)))) \wedge \\
\qquad ((\Phi_1 = \{0,\_\}) \wedge (\Phi_2 = \{1,\_\}) \supset \\
\qquad\quad (\mathit{len}\ p \supset (\text{strengthen } \neg x \rightarrow q)))
\end{array}$$

Apply rule falsity 2 times : $(\{0,\_\} = \{1,\_\}) = \{0,\_\}$
Apply rule implies-false : $(\{0,\_\} \supset A) = \{1,\_\}$
The new term is:

$$\begin{array}{l} \Box\, l = \{0,\_\} \supset \\ \quad ((\Phi_1 = \{0,\_\}) \wedge (\Phi_2 = \{0,\_\}) \supset stb\ x) \wedge \\ \quad ((\Phi_1 = \{1,\_\}) \wedge (\Phi_2 = \{0,\_\}) \supset \\ \qquad (len\ m \supset \{1,\_\} \wedge \\ \qquad\quad (\neg\{0,\_\} \supset (\text{strengthen } \neg q \rightarrow x)))) \wedge \\ \quad ((\Phi_1 = \{0,\_\}) \wedge (\Phi_2 = \{1,\_\}) \supset \\ \qquad (len\ p \supset (\text{strengthen } \neg x \rightarrow q))) \end{array}$$

Apply rule not-zero : $\neg\{0, A\} = \{1, A\}$
The new term is:

$$\begin{array}{l} \Box\, l = \{0,\_\} \supset \\ \quad ((\Phi_1 = \{0,\_\}) \wedge (\Phi_2 = \{0,\_\}) \supset stb\ x) \wedge \\ \quad ((\Phi_1 = \{1,\_\}) \wedge (\Phi_2 = \{0,\_\}) \supset \\ \qquad (len\ m \supset \{1,\_\} \wedge \\ \qquad\quad (\{1,\_\} \supset (\text{strengthen } \neg q \rightarrow x)))) \wedge \\ \quad ((\Phi_1 = \{0,\_\}) \wedge (\Phi_2 = \{1,\_\}) \supset \\ \qquad (len\ p \supset (\text{strengthen } \neg x \rightarrow q))) \end{array}$$

Apply rule implies-true : $(\{1,\_\} \supset A) = A$
Apply rule and-ident-comm : $\{1,\_\} \wedge A = A$
Rec-apply rule comm-and : $A \wedge B = B \wedge A$
The new term is:

$$\begin{array}{l} \Box\, l = \{0,\_\} \supset \\ \quad ((\Phi_1 = \{1,\_\}) \wedge (\Phi_2 = \{0,\_\}) \supset \\ \qquad (len\ m \supset (\text{strengthen } \neg q \rightarrow x))) \wedge \\ \quad ((\Phi_1 = \{0,\_\}) \wedge (\Phi_2 = \{0,\_\}) \supset stb\ x) \wedge \\ \quad ((\Phi_1 = \{0,\_\}) \wedge (\Phi_2 = \{1,\_\}) \supset \\ \qquad (len\ p \supset (\text{strengthen } \neg x \rightarrow q))) \end{array}$$

Replace strengthen $\neg q$ with $func1(q)$ in current term.
Replace strengthen $\neg x$ with $func2(x)$ in current term.
The new term with replacements is:

$$\begin{array}{l} \Box\, l = \{0,\_\} \supset \\ \quad ((\Phi_1 = \{1,\_\}) \wedge (\Phi_2 = \{0,\_\}) \supset \\ \qquad (len\ m \supset (func1(q) \rightarrow x))) \wedge \\ \quad ((\Phi_1 = \{0,\_\}) \wedge (\Phi_2 = \{0,\_\}) \supset stb\ x) \wedge \\ \quad ((\Phi_1 = \{0,\_\}) \wedge (\Phi_2 = \{1,\_\}) \supset \\ \qquad (len\ p \supset (func2(x) \rightarrow q))) \end{array}$$

Apply rule phi1-phi2 :

$$\begin{array}{l} ((\Phi_1 \approx \{1,\_\}) \wedge (\Phi_2 \approx \{0,\_\}) \supset (len\ \ M \supset (func1(A) \rightarrow B))) \wedge \\ ((\Phi_1 \approx \{0,\_\}) \wedge (\Phi_2 \approx \{0,\_\}) \supset stb\ B) \wedge \\ ((\Phi_1 \approx \{0,\_\}) \wedge (\Phi_2 \approx \{1,\_\}) \supset (len\ \ N \supset (func2(B) \rightarrow C))) \\ \quad \supset ((\Phi_2 \approx \{1,\_\}) \wedge len\ \ N \supset func2(func1(latched(\Phi_1, A))) \rightarrow C) \end{array}$$

The new term is:

$$\begin{array}{l} \Box\, l = \{0,\_\} \supset \\ \quad ((\Phi_2 = \{1,\_\}) \wedge len\ p \supset \\ \qquad func2(func1(latched(\Phi_1, q))) \rightarrow q) \end{array}$$

Replace $func2(X1)$ with strengthen $\neg X1$ in current term.
Replace $func1(X2)$ with strengthen$\neg X2$ in current term.
The new term with replacements is:

$$\begin{array}{l}\Box\, l = \{0,\_\} \supset \\ \qquad ((\Phi_2 = \{1,\_\}) \ \wedge \ len\ p \supset \\ \qquad\qquad \text{strengthen } \neg\text{strengthen } \neg latched(\Phi_1, q) \rightarrow q)\end{array}$$

Apply rule not-strengthen : $\neg$ strengthen A $\Leftrightarrow$ strengthen $\neg A$
Apply rule strong-strong : strengthen strengthen A $\Leftrightarrow$ strengthen A
Apply rule not-not : $\neg\neg A = A$
The new term is:

$$\begin{array}{l}\Box\, l = \{0,\_\} \supset \\ \qquad ((\Phi_2 = \{1,\_\}) \ \wedge \ len\ p \supset \ \text{strengthen } latched(\Phi_1, q) \rightarrow q)\end{array}$$

Do next case of case analysis.
The current term is:

$$\begin{array}{l}\Box\, l = \{1,\_\} \supset \\ \qquad ((\Phi_1 = \{0,\_\}) \ \wedge \ (\Phi_2 = \{0,\_\}) \supset stb\ x) \ \wedge \\ \qquad ((\Phi_1 = \{1,\_\}) \ \wedge \ (\Phi_2 = \{0,\_\}) \supset \\ \qquad\qquad (len\ m \supset (\{1,\_\} = \{1,\_\} \supset \ (\text{strengthen } \neg d \rightarrow x)) \ \wedge \\ \qquad\qquad\qquad (\neg(\{1,\_\} = \{1,\_\}) \supset \ (\text{strengthen } \neg q \rightarrow x)))) \ \wedge \\ \qquad ((\Phi_1 = \{0,\_\}) \ \wedge \ (\Phi_2 = \{1,\_\}) \supset \ (len\ p \supset (\text{strengthen } \neg x \rightarrow q)))\end{array}$$

Apply rule identity 2 times : $(A = A) = \{1,\_\}$
Apply rule implies-true : $(\{1,\_\} \supset A) = A$
The new term is:

$$\begin{array}{l}\Box\, l = \{1,\_\} \supset \\ \qquad ((\Phi_1 = \{0,\_\}) \ \wedge \ (\Phi_2 = \{0,\_\}) \supset stb\ x) \ \wedge \\ \qquad ((\Phi_1 = \{1,\_\}) \ \wedge \ (\Phi_2 = \{0,\_\}) \supset \\ \qquad\qquad (len\ m \supset (\text{strengthen } \neg d \rightarrow x) \ \wedge \\ \qquad\qquad\qquad (\neg\{1,\_\} \supset (\text{strengthen } \neg q \rightarrow x)))) \ \wedge \\ \qquad ((\Phi_1 = \{0,\_\}) \ \wedge \ (\Phi_2 = \{1,\_\}) \supset \\ \qquad\qquad (len\ p \supset (\text{strengthen } \neg x \rightarrow q)))\end{array}$$

Apply rule not-one : $\neg\{1, A\} = \{0, A\}$
Apply rule implies-false : $(\{0,\_\} \supset A) = \{1,\_\}$
Apply rule and-ident : $A \ \wedge \ \{1,\_\} = A$
Rec-apply rule comm-and : $A \ \wedge \ B = B \ \wedge \ A$
The new term is:

$$\begin{array}{l}\Box\, l = \{1,\_\} \supset \\ \qquad ((\Phi_1 = \{1,\_\}) \ \wedge \ (\Phi_2 = \{0,\_\}) \supset \\ \qquad\qquad (len\ m \supset (\text{strengthen } \neg d \rightarrow x))) \ \wedge \\ \qquad ((\Phi_1 = \{0,\_\}) \ \wedge \ (\Phi_2 = \{0,\_\}) \supset stb\ x) \ \wedge \\ \qquad ((\Phi_1 = \{0,\_\}) \ \wedge \ (\Phi_2 = \{1,\_\}) \supset \\ \qquad\qquad (len\ p \supset (\text{strengthen } \neg x \rightarrow q)))\end{array}$$

Replace strengthen$\neg d$ with $func1(d)$ in current term.
Replace strengthen $\neg x$ with $func2(x)$ in current term.
The new term with replacements is:

$$\begin{aligned}&\Box\, l = \{1,\_\} \supset\\ &\quad((\Phi_1 = \{1,\_\}) \wedge (\Phi_2 = \{0,\_\}) \supset\\ &\qquad(len\ m \supset (func1(d) \rightarrow x))) \wedge\\ &\quad((\Phi_1 = \{0,\_\}) \wedge (\Phi_2 = \{0,\_\}) \supset stb\ x) \wedge\\ &\quad((\Phi_1 = \{0,\_\}) \wedge (\Phi_2 = \{1,\_\}) \supset\\ &\qquad(len\ p \supset (func2(x) \rightarrow q)))\end{aligned}$$

Apply rule phi1-phi2 :

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

The new term is:

$$\begin{aligned}&\Box\, l = \{1,\_\} \supset\\ &\quad((\Phi_2 = \{1,\_\}) \wedge len\ p \supset\\ &\qquad func2(func1(latched(\Phi_1, d))) \rightarrow q)\end{aligned}$$

Replace $func2(X3)$ with strengthen $\neg X3$ in current term.
Replace $func1(X4)$ with strengthen$\neg X4$ in current term.
The new term with replacements is:

$$\begin{aligned}&\Box\, l = \{1,\_\} \supset\\ &\quad((\Phi_2 = \{1,\_\}) \wedge len\ p \supset\\ &\qquad \text{strengthen } \neg\text{strengthen } \neg latched(\Phi_1, d) \rightarrow q)\end{aligned}$$

Apply rule not-strengthen : $\neg$ strengthen A $\Leftrightarrow$ strengthen $\neg A$
Apply rule strong-strong : strengthen strengthen A $\Leftrightarrow$ strengthen A
Apply rule not-not : $\neg\neg A = A$
The new term is:

$$\begin{aligned}&\Box\, l = \{1,\_\} \supset\\ &\quad((\Phi_2 = \{1,\_\}) \wedge len\ p \supset \text{strengthen } latched(\Phi_1, d) \rightarrow q)\end{aligned}$$

End case analysis.
The result of case analysis is:

$$\begin{aligned}&(\Box\, l = \{0,\_\} \supset\\ &\quad((\Phi_2 = \{1,\_\}) \wedge len\ p \supset\\ &\qquad \text{strengthen } latched(\Phi_1, q) \rightarrow q)) \wedge\\ &(\Box\, l = \{1,\_\} \supset\\ &\quad((\Phi_2 = \{1,\_\}) \wedge len\ p \supset\\ &\qquad \text{strengthen } latched(\Phi_1, d) \rightarrow q))\end{aligned}$$

Apply rule distrib-box-and : $(\Box A) \wedge (\Box B) \Leftrightarrow (\Box A \wedge B)$
Apply rule comm-and : $A \wedge B = B \wedge A$
The new term is:

$$\begin{aligned}&\Box(l = \{1,\_\} \supset\\ &\quad((\Phi_2 = \{1,\_\}) \wedge len\ p \supset\\ &\qquad \text{strengthen } latched(\Phi_1, d) \rightarrow q)) \wedge\\ &\ (l = \{0,\_\} \supset\\ &\quad((\Phi_2 = \{1,\_\}) \wedge len\ p \supset\\ &\qquad \text{strengthen } latched(\Phi_1, q) \rightarrow q))\end{aligned}$$

Replace $l = \{0,\_\}$ with $\neg(l = \{1,\_\})$ in current term.

First show:

$(l = \{0,\_\}) = \neg(l = \{1,\_\})$

Apply rule equals-true : $(A = \{1,\_\}) = A$
The new term is:

$(l = \{0,\_\}) = \neg l$

Apply rule equals-false : $(A = \{0,\_\}) = \neg A$
The new term is:

$\neg l = \neg l$

Apply rule identity : $(A = A) = \{1,\_\}$
The new term is:

$\{1,\_\}$

The terms being replaced are equivalent
The new term with replacements is:

$$\begin{aligned}&\Box(l = \{1,\_\} \supset\\ &\qquad((\Phi_2 = \{1,\_\}) \wedge \mathit{len}\ p \supset\\ &\qquad\quad \text{strengthen } \mathit{latched}(\Phi_1, d) \rightarrow q)) \wedge\\ &\quad(\neg(l = \{1,\_\}) \supset\\ &\qquad((\Phi_2 = \{1,\_\}) \wedge \mathit{len}\ p \supset\\ &\qquad\quad \text{strengthen } \mathit{latched}(\Phi_1, q) \rightarrow q))\end{aligned}$$

Apply rule if-intro : $(A \supset B) \wedge (\neg A \supset C) = (\mathit{if}\ A\ \mathit{then}\ B\ \mathit{else}\ C)$
The new term is:

$$\begin{aligned}&\Box\, \mathit{if}\ l = \{1,\_\}\\ &\qquad \mathit{then}((\Phi_2 = \{1,\_\}) \wedge \mathit{len}\ p \supset\\ &\qquad\quad \text{strengthen } \mathit{latched}(\Phi_1, d) \rightarrow q)\\ &\qquad \mathit{else}((\Phi_2 = \{1,\_\}) \wedge \mathit{len}\ p \supset\\ &\qquad\quad \text{strengthen } \mathit{latched}(\Phi_1, q) \rightarrow q)\end{aligned}$$

and current term with top of stack.
The new term is:

$$\begin{aligned}&\Box\, \mathit{if}\ l = \{1,\_\}\\ &\qquad \mathit{then}\ ((\Phi_2 = \{1,\_\}) \wedge \mathit{len}\ p \supset\\ &\qquad\quad \text{strengthen } \mathit{latched}(\Phi_1, d) \rightarrow q)\\ &\qquad \mathit{else}\ ((\Phi_2 = \{1,\_\}) \wedge \mathit{len}\ p \supset\\ &\qquad\quad \text{strengthen } \mathit{latched}(\Phi_1, q) \rightarrow q) \wedge\\ &(\Box \neg(\Phi_2 = \{1,\_\}) \supset \mathit{stb}\ q)\end{aligned}$$

Bibliography

[AB*80] V. D. Agrawal, A. K. Bose, et al. A mixed-mode simulator. In *17th ACM/IEEE Design Automation Conference Proceedings*, pages 618–625, 1980.

[Bar84] H. G. Barrow. Proving the correctness of digital hardware designs. *VLSI Design*, 64–77, July 1984.

[Bat83] J. W. Batten. *Prolog: Its Potential for Hardware Description and Verification.* Technical Report, U.M.I.S.T., September 1983.

[BCDM86] M. C. Browne, E. M. Clarke, D.L. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, C-35(12):1035–1043, December 1986.

[BF76] M. A. Breuer and A. D. Friedman. *Diagnosis & Reliable Design of Digital Systems.* Computer Science Press, Inc., 1976.

[Boc82] G. V. Bochmann. Hardware specification with temporal logic: an example. *IEEE Transactions on Computers*, C-32(3):223–231, March 1982.

[Bry81] R. E. Bryant. *A Switch-Level Simulation Model For Integrated Logic Circuits.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1981.

[Bry85] R. E. Bryant. Symbolic verification of MOS circuits. In H. Fuchs, editor, *Chapel Hill Conference on Very Large Scale Integration*, pages 419–438, Computer Science Press, 1985.

[CGM86] A. J. Camillieri, M. J. C. Gordon, and T. F. Melham. Hardware verification using higher-order logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, North Holland, September 1986.

[Chi86] Shiu-Kai Chin. *Synthesis of Digital Designs by Equivalence Transformations.* PhD thesis, Department of Electrical Engineering, Syracuse University, January 1986.

[CL*84] C. F. Chen, C-Y Lo, et al. The second generation Motis mixed-mode simulator. In *21st ACM/IEEE Design Automation Conference Proceedings*, pages 10–17, 1984.

[CL86] W. F. Clocksin and M. E. Leeser. Automatic determination of signal flow through MOS transistor networks. *Integration*, 4:53–63, 1986.

[Clo87] W. F. Clocksin. Logic programming and digital circuit analysis. *Journal of Logic Programming*, 4:59–82, 1987.

[CM84] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, second edition, 1984.

[Coh87] A. Cohn. *A Proof of Correctness of the Viper Microprocessor: The First Level.* Technical Report 104, University of Cambridge Computer Laboratory, January 1987.

[DC86] D. L. Dill and E. M. Clarke. Automatic verification of asynchronous circuits using temporal logic. *IEE Proceedings E: Computers and Digital Techniques*, 133(5):276–282, September 1986.

[Dhi87] I. S. Dhingra. *Formal Validation of An Integrated Circuit Design Style.* Technical Report 115, University of Cambridge Computer Laboratory, August 1987.

[DPS*87] A. Despain, Y. Patt, V. Srini, et al. Aquarius. *Computer Architecture News*, 15(1):22–34, 1987.

[FKTM86] M. Fujita, S. Kono, H. Tanaka, and T. Moto-oka. Aid to hierarchical and structured logic design using temporal logic and Prolog. *IEE Proceedings E: Computers and Digital Techniques*, 133(5):283–294, September 1986.

[FST84] A. Fusaoka, H. Seki, and K. Takahashi. Description and reasoning of VLSI circuit in temporal logic. *New Generation Computing*, 2:79–90, 1984.

[FTM83] M. Fujita, H. Tanaka, and T. Moto-oka. Verification with Prolog and temporal logic. In T. Uehara and M. Barbacci, editors, *Computer Hardware Description Languages and Their Applications*, pages 103–114, North Holland, 1983.

[GDM83] N. F. Goncalves and H. J. De Man. NORA: a racefree dynamic CMOS technique for pipelined logic structures. *IEEE Journal Of Solid-State Circuits*, SC-18(3):261–266, June 1983.

[GH87] M. J. C. Gordon and R. W. S. Hale. Embedding Modal Logics in Higher Order Logic. 1987. Personal Communication.

[Gor85a] M. J. C. Gordon. *Hardware Verification by Formal Proof.* Technical Report 74, University of Cambridge Computer Laboratory, August 1985.

[Gor85b] M. J. C. Gordon. *HOL: A Machine Oriented Formulation of Higher Order Logic.* Technical Report 68, University of Cambridge Computer Laboratory, July 1985.

[Gor87] M. J. C. Gordon. 1987. Personal Communication.

[Gul85] E. Gullichsen. Heuristic circuit simulation using Prolog. *Integration*, 3:283–318, 1985.

[Gup86] R. Gupta. Test-pattern generation for VLSI circuits in a Prolog environment. In *Third International Conference on Logic Programming*, pages 528–535, 1986.

[HD86a] F. K. Hanna and N. Daeche. Specification and verification of digital systems using higher-order predicate logic. *IEE Proceedings E: Computers and Digital Techniques*, 133(5):242–254, September 1986.

[HD86b] F. K. Hanna and N. Daeche. Specification and verification using higher-order logic: a case study. In G. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 179–213, North-Holland, 1986.

[Her86] J. M. J. Herbert. *Application of Formal Methods to Digital System Design.* PhD thesis, Computer Laboratory, University of Cambridge, 1986.

[Hor83] P. W. Horstmann. Expert systems and logic programming for CAD. *VLSI Design*, 37–46, November 1983.

[Hun86] W. A. Hunt, Jr. *FM8501: A Verified Microprocessor.* PhD thesis, Institute for Computing Science, The University of Texas at Austin, 1986.

[Jou83] N. P. Jouppi. TV: an nMOS timing analyzer. In *Proceedings of the 3rd Caltech VLSI Conference*, pages 71–76, 1983.

[Jou87] N. P. Jouppi. Timing analysis and performance improvement of MOS VLSI designs. *IEEE Transactions on Computer-Aided Design*, CAD-6(4):650–665, July 1987.

[Joy87] J. J. Joyce. *Hardware Verification of VLSI Regular Structures.* Technical Report 109, University of Cambridge Computer Laboratory, July 1987.

[LM86] T-M Lin and C. A. Mead. A hierarchical timing simulation model. *IEEE Transactions on Computer-Aided Design*, CAD-5(1):188–197, January 1986.

[Mak77] A. K. Makworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[Mil86a] G. J. Milne. Behavioural description and VLSI verification. *IEE Proceedings E: Computers and Digital Techniques*, 133(3):127–137, May 1986.

[Mil86b] R. Milne. Design transformation and chip planning. In G. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 23–43, North-Holland, 1986.

[Mos83] B. C. Moszkowski. *Reasoning about Digital Circuits.* PhD thesis, Department of Computer Science, Stanford University, July 1983.

[Mos86] B. C. Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, 1986.

[MP83] Z. Manna and A. Pnueli. *Verification of concurrent Programs: A Temporal Proof System.* Technical Report STAN-CS-83-967, Department of Computer Science, Stanford University, June 1983.

[Mur84] A. F. Murray. CMOS design technique to eliminate the stuck-open fault problem of testability. *Electronics Letters*, 20(19):758–760, September 1984.

[Ous83] J. K. Ousterhout. Crystal: a timing analyzer for nMOS VLSI circuits. In *Proceedings of the 3rd Caltech VLSI Conference*, pages 57–69, 1983.

[Ous84] J. K. Ousterhout. Switch-level delay models for digital MOS VLSI. In *21st ACM/IEEE Design Automation Conference*, pages 542–548, 1984.

[PD86] J. D. Pincus and A. M. Despain. Delay reduction using simulated annealing. In *23rd ACM/IEEE Design Automation Conference*, pages 690–695, 1986.

[SA84] D. Svanæs and E. J. Aas. Test generation through logic programming. *Integration*, 2:49–67, 1984.

[Sha87] M. P. Shanahan. *Exploiting Dependencies in Search and Inference Mechanisms.* PhD thesis, Computer Laboratory, Cambridge University, 1987.

[She83] M. Sheeran. *μFP, An Algebraic VLSI Design Language.* PhD thesis, University of Oxford, 1983.

[She86] M. Sheeran. Design and verification of regular synchronous circuits. *IEE Proceedings E: Computers and Digital Techniques*, 133(5):295–304, September 1986.

[Sub86a] P. A. Subrahmanyam. The algebraic basis of an expert system for VLSI design. In G. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 59–81, North-Holland, 1986.

[Sub86b] P. A. Subrahmanyam. Synapse: an expert system for VLSI design. *Computer*, 78–89, July 1986.

[Suz85] N. Suzuki. Concurrent Prolog as an efficient VLSI design language. *Computer*, 33–40, February 1985.

[WE85] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective.* Addison-Wesley Publishing Company, 1985.

[Wei86] D. W. Weise. *Formal Multilevel Hierarchical Verification of Synchronous MOS VLSI Circuits.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, August 1986.

[Win87a] G. Winskel. *A Compositional Model of MOS Circuits.* Technical Report 105, University of Cambridge Computer Laboratory, April 1987.

[Win87b] G. Winskel. *Relating Two Models of Hardware.* Technical Report 110, University of Cambridge Computer Laboratory, July 1987.

[Wol82] P. L. Wolper. *Synthesis of Communicating Processes from Temporal Logic Specifications.* PhD thesis, Department of Computer Science, Stanford University, 1982.