A modeling technique for specification and simulation of digital systems

Luc Morin

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

for the degree of Doctor of Philosophy at

Concordia University

Montréal, Québec, Canada

March 1993

ISBN 0-315-90925-0

ABSTRACT

A modeling technique for specification and simulation of digital systems

Luc Morin, Ph. D.

Concordia University, 1993

In this thesis, a new logic modeling technique based on a mathematical abstraction of analog circuits into logic models is proposed. Many approaches have been used for the modeling of digital systems. They range from pure logic to real time models. Mixed circuit-logic models with switches and strength factors are the most popular in commercial simulators. All of these approaches are meant to provide simple and unambiguous descriptions of the logic devices. However, except in their specification of logic functions and simple timing specifications, they differ in other aspects such as the definition of an event, the number of logic levels, the strength factors, the modeling of wire capacitance and the processing of timing constraints and timing violations. The lack of a consensus is the source of many compatibility problems in modern integrated CAD systems where designers and customers need to share information processed by computers. The objective of this work is to lay down the foundations necessary for such a consensus.

The originality of the proposed approach lies in the mathematical formulation of the abstraction mechanism. Precise hypotheses are used to define the logic devices. Then strict construction rules are used to derive their logic models. Three aspects of the logic models are separately considered: the transmission of information over wires using logic events, the propagation delay and the transformation of continuous time to discrete time.

A new logic event definition called master-slave events and a new delay model preserving continuity in the modeled logic signal and causality (cause-effect relationship) in the scheduling of the logic events are proposed. Finally, a new graph based algorithm called a Continuous Time Automaton (CTA) is described to model logic devices and integrate the input timing constraints with the logic function. A CTA is a finite state machine that processes the sequence of input events and transforms the continuous change of state (continuous time) into a timed state sequence (discrete time).

A prototype simulator was developed and CTAs for clocks, pattern generators, gates, combinational circuits, flip-flops, tri-state devices and synchronous sequential circuits are proposed. Without optimization, the performance of the prototype simulator was comparable to many logic simulators for memory requirements, speed and timing accuracy. This modeling technique provides a more accurate logic signal representation, a better model for the physical phenomenon associated with propagation delay and a systematic method to specify and process input timing constraints. It is also easier to use than other logic modeling techniques and it closely matches the engineering approach. On the negative side, the modeling technique is limited to unidirectional signal. However it still permits the use of tri-state busses. Because it accepts and generates analog signals, the proposed model easily interfaces with other circuit models in mixed mode simulation.

This work should be useful to someone working on the specification, design, simulation, formal verification and testing of digital systems. For instance, the new modeling technique could be used for the development of primitives in VHDL.

Preface

This work was carried out over a period of five years. The ideas presented here have evolved all along from simple hints to, hopefully, an acceptable and applicable theory. The initial work was carried on metastability and was followed by a study of the logic models used in hardware description languages and simulators. In both cases, I was left with an odd feeling that something was incorrect. There were numerous approaches for the modeling of digital systems, but none seem to be able to justify its superiority on solid grounds. This was confirmed in a paper by M. R. Lightner where he stated that a formalism was yet to be developed for digital elements and their interconnections.

Trying to develop a new formalism for logic model proved to be a difficult task, chiefly because well established concepts had to be questioned and often rejected. The idea of using a finite state machine came early on, but it took quite some time to apply it to a simple buffer, to convince myself of the value of the approach and to refine the concept. These ideas further evolved and became the logic model consisting of classified interface ports, a timing specification, a continuous time automaton (CTA) and a continuity-preserving delay model. Finally, the master slave event concept was introduced to preserve the timing relationships, thus the causality, on the signal and at the input ports. In order to use and exercise the models in some sort of design environment, a hardware description language (HDIL), a data structure (HDIDS) and a logic simulator have been developed.

The core of this thesis consists of three chapters. The formal framework and the definitions are presented in chapter 2. The transmission of information over wires is studied in chapter 3, then the models for logic signals and for propagation delays are described. Finally, the concept of CTA is presented in chapter 4.

I would like to thank Dr. H. F. Li for his support and valuable criticism. This work was carried out at the computer science department of Concordia University, and

I would like to thank all the people who helped me in my work. This work was made possible by the financial support of Université du Québec à Chicoutimi through a three years scholarship. Finally, I would like to thank Michèle and Jacynthe for their personnal support.

Table of content

Chapter 3 Time model: logic signal and propagation delay

Chapter 4 State model: Continuous Time Automaton (CTA)

Chapter 5 Implementation, results and comparative analysis

Abbreviations

ASC	Asynchronous Sequential Circuit
BDT	Binary Decision Theorem
CIF	Caltech Intermediate Form
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
CTA	Continuous Time Automaton
DDMS	Design Data Management System
DRC	Design Rule Checking
DSP	Digital Signal Processing
ERC	Electrical Rule Checking
HDIDS	Hardware Description and Integration Data Structure
HDIL	Hardware Description and Integration Language
HDL	Hardware Description Language
MITL	Metric Interval Temporal Logic
PLA	Programmable Logic Array
RAM	Random Access Memory
ROM	Read Only Memory
RTL	Register Transfer Logic
SSC	Synchronous Sequential Circuit
TA	Timed Automaton
TL	Temporal Logic
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration

Glossary

1.0 Basic definitions

Function	Unique mapping of a set called the domain onto another set called the range.
Continuous	Function mapping real onto real.
Discrete	Function mapping natural onto real.
Quantized	Function mapping real onto natural.
Digital	Function mapping natural onto natural.
Sampling	Transformation of the domain set from real to natural.
Quantization	Transformation of the range set from real to natural.
Signal	Abstraction used to represent the information contained in a physical variable.
Analog	An analog signal is a continuous function of time. All signals are analog.
Point	Element of an analog signal.
Discrete	A signal is discrete if it is completely described by a discrete function and an interpolation function.
Sample	Element of a discrete signal.
Digital	A signal is digital if it is completely described by a digital function and an interpolation function. The digital function can map to a set like {0, 1} not necessarily related to voltage thresholds.
Event	Element of a digital function. An event describes the assertion of a proposition, for example "crossing a threshold, $f(t)=V_{th}$".
Interpolation function	Function that describes how the analog signal is reconstructed from the samples or the events.
Synchronous signal	Two digital signals are synchronized if any pair of events taken from these signals are synchronized. Two events are synchronized if the relation between their occurrence is explicitly known.

2.0 Digital systems

Physical information	Information related to the physical aspects, such as: layout, mask, electrical parameters.
Layout	Geometric description of the integrated circuit produced by the IC designer and used for mask generation.
Mask	Geometric description of the integrated circuit used for the fabrication.
Wafer	Unit for the production of ICs. There are many ICs fabricated on one wafer.
Chip	Portion of the wafer cut and installed in a package.
Package	Mechanical support for the chip. The package with the chip forms the integrated circuit.
Pad	Metal area on the chip for bonding the external wire.
IOcircuit	Refers to the interface circuit for the pad (protection circuit for inputs and buffers for outputs).
Frame	Refers to the element of the layout delimiting the boundaries of the chip.
Cell	Refers to a pre-designed function ready to be used within a chip.
Electrical properties of the material	Refers to the surface resistivity or the normalized capacitance for the various materials or elements fabricated on a chip.
Structural information	Information about the organization of the digital circuit, such as:
Circuit	This is the basic abstraction in digital system, the physical world is discretized into elements and wires which form the circuit.
Element	The element is either a part, like a 68000, or another circuit allowing for hierarchical description.
Wire	This is the abstraction of the conductor interconnecting the elements in a circuit.
Port	This is the abstraction of the point of connection on the element. An element may have many ports.
Technology independence	A piece of information defined such that technology does not affect its definition. Physical information is technology dependent and structural and behavioral

information are technology independent. For example, the electrical model of a gate is technology dependent since it would be different for CMOS and TTL gates. On the other hand, the logic model is technology independent since it would be the same for both technology.

3.0 Design process

Design — This is the first phase in problem solving, the specification is studied and a knowledge based model is constructed. When there is enough detail in the model, we assume a solution has been found. This part of the problem solving process is rarely automated. It consists in searching information in the design data base either manually or with the help of design assistant programs.

- Specification — The initial description of the desired digital system or component.
- Solution — This is the result of the design phase. A solution is a knowledge based model, it is often imprecise even contains contradiction which will be resolved with design verification cycles.
- Technology based model — Class of information, such as the extracted model, normally processed with deterministic algorithm.
- Knowledge based model — Class of information normally processed with non-deterministic algorithm, such as the information during the early phase of the design.
- Synthesis — Automated process that takes a specification and produces a solution or directly a circuit.

Implementation — This is the second phase of the problem solving process. Often automated, the implementation takes a solution and produces either a model or the actual hardware.

- Model — This is the result of the implementation. The verification consists in the simulation or the analysis of the model.
- Actual hardware — This is the result of the implementation. The verification consists in testing the actual hardware.
- Custom design — Technique of implementation where most of the cells are designed individually and specifically for the digital system.

Standard cell — Technique of implementation where most of the individual cells are common to many projects and are usually stored in a library. The implementation consists in placing the cells and connecting them.

Gate array — Technique of implementation where the cells are already placed. The implementation consists in connecting them.

Placement — Process, now automated, to find an optimum place for each cell. The main criteria is the minimization of wire length. There are many algorithms, usually beginning with constructive placement followed by iterative placement.

- Constructive placement — Placement technique where the area is initially filled from one end to the other.
- Iterative placement — Placement technique using pair-wise interchange to optimize the initial placement.

Routing — Process of connecting the cells according to the netlist. The routing consists in finding a channel(s) for each wire and then assigning a track to the wire in the channel.

- Route — Sequence of channels used by a wire.
- Channel — Region of the layout reserved for wires.
- Track — Each channel is divided in tracks.
- Global routing — Process of finding a route for a wire.
- Channel routing — Process of assigning a track to a wire.

Silicon compilation — Automated process to create the layout based on a user description of the desired function. The compiler assembles the layout primitives or library cells to build the desired circuit.

- Structural compilation — Compilation based on the structural information, usually the schematic diagram.
- Behavioral compilation — Compilation based on the behavioral information.

<u>Verification</u> — This is the third phase of the problem solving process. Once a model or the actual hardware is available, it must be verified. Verification consists in simulation, analysis

	or formal verification on the model or test on the hardware.
Simulation	Process of exercising the model under certain operating conditions. No comparison is made to the specification.
Analysis	The model is analyzed and specific aspects of the specification are compared to the value obtained with the model. Timing analysis is a popular analysis technique, where propagation delays are computed and compared with the specified ones.
Formal verification	The model is formally compared with the specification. All aspects of the specification are verified.
Test	Process of exercising the actual hardware.
Result	Describe the results obtained from the verification phase.
Evaluation	This is the fourth phase of the problem solving process, where the results are analyzed and compared with the specification and design decision or modification are made.
Design/evaluation cycle	Refers to the repetition of the design process. Many cycles are usually required to obtain a system meeting completely the specification.
Hierarchical design	The design proceeds in step-wise refinement, from system level down to circuit level or layout. At each level, the design/evaluation cycles are repeated as required.

4.0 Design environment

Design environment	Computer based environment used by the digital system designer. A design environment consists of a set of tools and a design data base.
Design data base	A generic term that refers to the information available to the designer, including books, papers and colleagues.
Design assistant	Since complete automation of the early phases of the design is impossible, specific programs, called design assistant, can be used to help the designer search the design data base and guide him in his decisions.
Tool	A program specialized in solving a specific task, like a PLA compiler or a placement and routing program. If the tool is used in the design phase it is called a design assistant.

Integration	The tools in the design environment need to share information. A well integrated design environment allows the tools to share their information without human intervention.
External integration	Refers to the sharing of the information through files.
Internal integration	Refers to the sharing of the information through active memory.

5.0 Model and simulation (behavioral information)

Model	Mathematical representation of a system intended to reproduce its response to input stimuli. There are many levels of models: circuit, switch, timing, logic, register based.
Electrical model	Model describing the elements by equations relating voltages and currents at the ports.
Timing model	Simplified electrical model allowing fast computation of the timing. Switch models with capacitances are often used.
Logic model	Model consisting of interface ports, a state, a state transition function, an output function and a timing specification.
Port class	For the purpose of modeling, each port is classified as either input, output, asynchronous input, synchronous input, bidirectional, high voltage input, clock ...
State (Q)	Whatever is required to allow the prediction of future events without the knowledge of the past inputs.
Logic state	Abstract state required to model the logic behavior (flip-flop, registers ...).
Time related state variables	Set of state variables required to process accurately the timing specification of input and output events, often used to store the occurrence of past events.
Continuous change of state	Refers to a system whose model is characterized by an infinite number of changes of state in a finite interval of time.
Discrete change of state or timed state sequence	Refers to a system whose model is characterized by a finite number of changes of state in a finite interval of time.
State transition function (δ)	Describes the change of state based on the current state and the current inputs.

Output function (λ)	Describes the output to be produced based on the current state and the current inputs.
Time invariant load	Characteristic of a load stating that it does not change during the course of a simulation.
Event set	A set of propositions describing properties of a signal that occur at specific instant. Logic events are often propositions describing the crossing of thresholds.
Threshold based events	Describes an event set based on fixed threshold crossing.
Signal based events	Describes an event set based on certain properties of the signal such as $\frac{dx}{dt} = 0$.
2-valued logic	Algebra based on the Boolean values 0 and 1, often augmented by U (undefined) and Z (high impedance) for simulation purpose.
Multiple-valued logic	Algebra based on more than two values. Values 0, X and 1, augmented by U (undefined) and Z (high impedance) are often used.
Model extraction	Process of computing the model based on layout information.
Hardware linking	Process of computing the logic model parameters such as the propagation delay based on the interconnection network characteristics such the wire capacitance. Linking rules (fan-in and fan-out) are also verified during hardware linking.
Discrete event simulation	Simulation based on the discretization of time and on the causality of real systems.
Mixed-mode simulation	Simulation using mixed models, like circuit and logic.
Timing specification	Consist of:
Timing constraints	Set of separation times between input slave events that must be meet to insure proper operation of the logic device.
Propagation delays	Set of separation times between input slave events and output master events that models the effect of RC circuits.
Timing violation	Pair of events whose separation does not meet a timing constraint.
Continuous Time Automaton	State machine used to transform continuous change of state into timed state sequence.

Constrained state transition	State transition in a CTA that requires a finite amount of simulation time.
Unconstrained state transition	State transition in a CTA that might require zero simulation time.
Constrained cycle	Cycle in a CTA made of at least one constrained state transition.
Unconstrained cycle	Cycle in a CTA made only of unconstrained state transition.
Constrained signal	Logic signal made of a finite number of event in any interval of time.
Unconstrained signal	Logic signal made of possibly an infinite number of events in a given interval of time.
Timed sequence of events	Sequence of events describing a constrained signal.

Chapter 1

Introduction

Since the early days of digital systems, designers have been modeling logic devices for various purposes. There was the simple approach using Boolean equations, then the engineering approach leading to data books and the more recent approach using logic models specified with hardware description languages like VHDL. Even if logic devices are simple to understand and use, modeling them is not so obvious because of:

- the complex relation between the logic function and the timing specification,
- the continuous nature of the analog circuits from which logic devices are built
- and indirectly, the need of unambiguous models for computer analysis and simulation.

The design of very high speed integrated circuits requiring highly optimized logic devices and the absence of simple but accurate models have placed a continuous demand for research in this area. The objective in this chapter is to describe the problem of constructing logic models for analog devices and to outline the proposed logic modeling technique.

1.1 Problem overview

The object of this thesis is to study the abstraction mechanism involved in modeling an analog circuit at logic level. In particular, we wish to accurately model glitches and fine timing details as required by modern technology. An abstraction mechanism is comprised of :

- a set of axioms and hypotheses to define a level of abstraction (analog),
- a set of axioms and hypotheses to define another level of abstraction (logic),

- and a set of rules for the construction of a mathematical representation at one level based on the other level.

The general problem is the lack of a consensus on the axioms, the hypotheses and the construction rules for logic models leading to numerous and often incompatible logic modeling techniques [28]. For example, the following problems have been reported as sources of incompatibilities with VHDL models [32]:

- Logic level definition
- Signal strength definition
- Logic event definition
- Interpretation of timing specification
- Handling of unknown operating condition

In order to improve the logic model, each object used will be mathematically defined and each step of the process will be mathematically formulated. The process of finding a logic model for an analog circuit will be divided into three steps.

- Step 1: define the analog circuit properties that makes it a logic device.
- Step 2: define the characteristics of a logic model for that logic device.
- Step 3: construct the logic model.

The construction of a logic model is further divided into three parts:

- the transmission of information on wires,
- the propagation delays and
- the state.

In this chapter, current modeling techniques will be reviewed to illustrate the importance of following the above steps. Then, the definitions for logic devices (step 1) and logic models (step 2) will be given, followed by an outline of the proposed modeling technique (step 3).

1.2 Basic definitions

It is necessary to begin our discussion about logic modeling with intuitive but carefully formulated definitions for the objects that will be used.

- Wire: A wire is a physical element carrying the continuous voltages and currents between the ports of analog devices or logic devices.
- Analog device: An analog device is characterized by a set of analog ports and continuous functions. An analog port transfers the voltage and the current from the inside to the outside of the device or vice versa. The device processes the continuous voltages and currents at the ports according to some continuous behavior specification.
- Analog circuit: An analog circuit is a network of analog devices connected with wires. The relation between the voltages and the currents at the ports of the analog devices and in the analog circuit are governed by Kirchhoff laws.
- Analog signal: An analog signal is a mathematical representation of the information carried by a wire. It consists of a continuous function of time modeling the voltage or the current.
- Analog model: An analog model is a mathematical representation of the behavior of an analog device. It consists of set of analog ports, a set of continuous state variables and a set of continuous equations describing how the state, the voltages and the currents are related.
- Logic device: A logic device is characterized by a set of logic ports and a logic function. A logic port is an analog port with the following restrictions: a logic input port is sensitive to threshold crossing and a logic output port produces a voltage compatible with the input ports. The device processes the continuous voltages and currents at the ports according to some logic behavior specification.

- Logic circuit: A logic circuit is a network of logic devices connected with wires. The relation between the voltages and the currents at the ports of the logic devices and in the logic circuit are governed by Kirchhoff laws.
- Event set: An event set is a set of labelled predicates describing an instantaneous property on an analog signal. For example, the binary event set is often defined as:

  $\{ \text{ label } 0 : \ (v(t) = V_{th} \text{ and } \frac{d\,v(t)}{dt} < 0), \ \text{ label } 1 : (v(t) = V_{th} \text{ and } \frac{d\,v(t)}{dt} > 0) \}$
- Logic event: A logic event is a tuple consisting of a label describing the event and the time it happens.
- Logic signal: A logic signal is a time ordered sequence of logic events.
- Logic model: A logic model is a mathematical representation of the behavior of a logic device. It consists of set of logic ports, a set of discrete state variables and a set of discrete and logic equations describing how the state, the input events and the output events are related.
- Continuous time or continuous change of state: This is the characteristic of a model whose state is a continuous function of time. This is also referred to as dense time.
- Discrete time or discrete change of state: This is a characteristic of a model whose state is a discrete function of time. In discrete time, there is a finite number of state changes in any interval of time.
- Timed state sequence: A timed state sequence is a sequence of state changes in discrete time.
- Timing constraints: The set of timing constraints is a set of real numbers used to define an acceptable sequence of logic events that can drive a given logic device (Ex: set-up time, hold time). Event separations are compared with timing constraints to detect timing violations.

• Propagation delays: The set of propagation delays is a set of real numbers used to model the delay between an input event and an output event in a logic device. The propagation delays are used in simulation by a propagation delay model.

1.3 Design process

The objective of modeling in this thesis is to support the specification, the analysis and specially the simulation of digital systems at logic level. Since these constitute the major components in digital system design, it is therefore appropriate to spend some time to review the design process. As illustrated in Fig. 1.1, the design process is decomposed into two parts, design and verification. Design consists of transforming a specification into a circuit and verification insures that the proposed

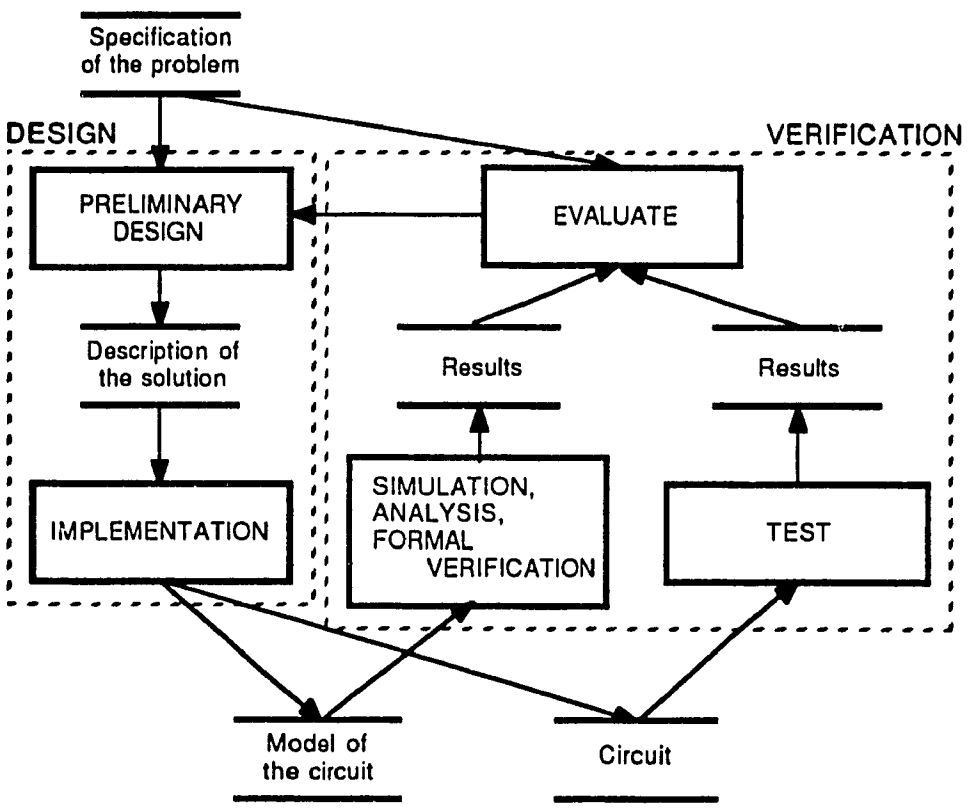


Fig. 1.1 Digital system design process

circuit meets the specification. Design/verification cycles are repeated as many times as required until a correct circuit is obtained. Design generally begins with a preliminary design followed by an implementation. In the preliminary design phase, the specification is analyzed, a number of solutions are studied and one of them is selected. In the implementation phase, the selected solution is realized. The design checking then consists of an analysis, a formal verification or a simulation of the model. Verification terminates with an evaluation phase, where the results are analyzed and the design is modified as required. Design/verification cycles are carried out at different levels. From a general specification a system architecture can be derived, from a system architecture a register transfer model can be designed, and with more design cycles, detailed circuits are obtained. In this thesis, we are interested in the design and verification at logic level.

Until 1980, separate computer programs were used for solving specific problems in the design process [1]. For example, different tools were used for layout editing, physical rule checking, compaction, circuit extraction and simulation. Since then, design environments have greatly evolved and usually provide an integrated set of tools to support the designer in most phases of the design process [2]. The design of digital systems at logic level, consists in using manually designed elements or compiled modules (PLA, ROM, RAM, DSP, ...) and drawing the schematic diagram of the circuit. This is normally followed by automatic placement and routing, model extraction and simulation. In certain cases, the design might be reduced to writing a high level description and compiling it directly into a chip [3].

1.4 Simulation and verification

Even though the trend in CAD systems is to automate synthesis [4], the need for effective verification tools is still strong. The tools are classified according to the completeness of checking of a design against its specification:

- Formal verification tools: The extracted or computed model of the composite system is compared with the requirement specification. Both the model and the specification are described using a special description language and the model is proved to satisfy the specification mathematically. This is quite complex and is practicable for smaller systems.
- Analysis tools: The model is partially compared with certain requirements in the specification. For example, timing analysis is often used to compare the computed timing parameters of the model with the specified timing.
- Simulation tools: The model is exercised for a restricted set of inputs and the simulation results are compared against the specification. The simulator does not make any comparison.

Each of these tools relies on a good model, one that faithfully captures the essential properties of the real system. Simulation tools are still the most popular. They are classified according to the level of abstraction: device, circuit or logic.

Device simulation

Device modeling uses discrete elements to model the physical structure (two or three dimensions) of a device and the simulation consists of predicting the current and voltage distributions in the device [5]. This type of simulation is useful to determine current concentration, to assert the circuit parameter values or to check the precision of a circuit model.

Circuit simulation

Circuit models assume that the 3-dimensional continuous physical world is discretized into elements and wires. The elements are modeled by a set of equations describing the relationships among the voltages and currents of the ports. A simulation of these elements involves in solving the set of network equations based on Kirchoff laws [6, 7]. Third generation circuit simulators have improved performance

by making use of space and time sparseness [8] and special circuit structure [9-11]. By using tabular models with adjustable precision, the simulation of a circuit can be further speeded up [12].

The large digital systems to simulate and the quest for speed have forced the designers to use simplified models. Since only timing specification need be determined, the simplified models were optimized to predict the propagation delays (see Fig 1.2). Two approaches have been used: macro-models and switches. A switch is a simplified model for a transistor and a switch based simulation often called timing simulation consists in predicting the waveshape of signals by assuming that the voltage at each port is the result of a network of resistors, capacitors and switches. In the case of MOS circuits, the switch model is easily extracted from the layout. Its interpretation is simple and the simulation results are reasonably accurate. This probably accounts for its popularity [13-20]. The switch-level modeling technique has also been applied to model bipolar logic circuits [21-23].

Macro-models are used to model a group of transistors performing a logic function. The electrical behavior of the output stage is modeled in detail using controlled sources, resistors and capacitors [24-26]. The objective is to be able to connect logic devices to analog circuit, perform a simulation and improve the accuracy compared with a simple logic model with only a propagation delay. As shown in Fig. 1.2, macro-models are also used in hybrid circuit-logic simulators [36-38]. The macro model of a logic device is similar to an operational amplifier which is modeled using Thevenin equivalent and transfer function instead of transistors. Switch and macro based simulations are useful to simulate circuit whose complexity would make an analog simulation too costly.

As shown in Fig. 1.2, hybrid models use macro-models, switch networks or strength factors or a combination of these. Hybrid models combine the speed of logic simulation for the internal operation of the devices with the accuracy of analog

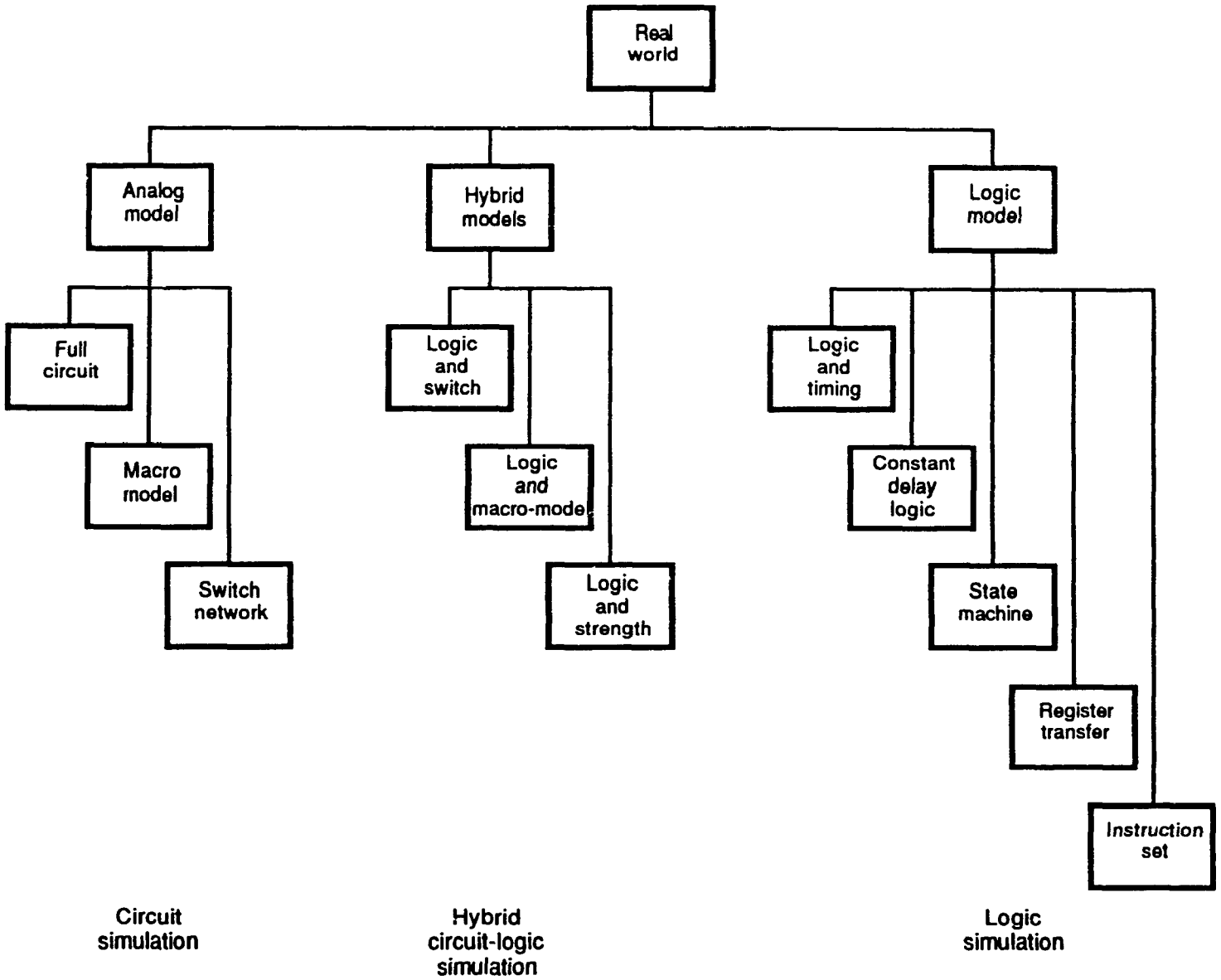


Fig. 1.2 Classification of models

simulation for the interconnection network and the special devices. The major problem with hybrid simulation is that network equations based on Kirchoff laws need to be solved, but again this is a trade off between accuracy and speed.

Logic simulation

The next level up in the abstraction is the logic level. *At analog level, the device state is a continuous function of time and at logic level, the device state is a discrete function of time.* Basically, continuous time is discretized. The definition for logic models will be given in the next section, but we can safely say that a logic model consists of a logic function, a state and a timing specification. Since the subject of this thesis is logic simulation, we will review all of the related topics at the proper time.

1.5 The problem of modeling analog devices with logic models

The construction of logic models begins with analog circuits. It is therefore essential to understand the behavior of analog devices and systems. The usual model taken from classical system theory will be used. The system S in Fig. 1.3 represents a logic device. For simplicity, imagine it is a logic buffer. It is driven by an input signal $u_p(t)$ and produces an output signal $v_p(t)$. The input and output signals are parameterized using the continuous parameter p.

Assumptions:
1. S is a continuous, causal and time invariant system.
2. $u_p(t)$ is a function dependent on the continuous parameters p and t (time). Varying p generates a family of parameterized input signals.
3. $v_p(t)$ is the output of S for a given $u_p(t)$.

More formally, the system has a state $s \in \Sigma$ and is modeled by a continuous function φ:

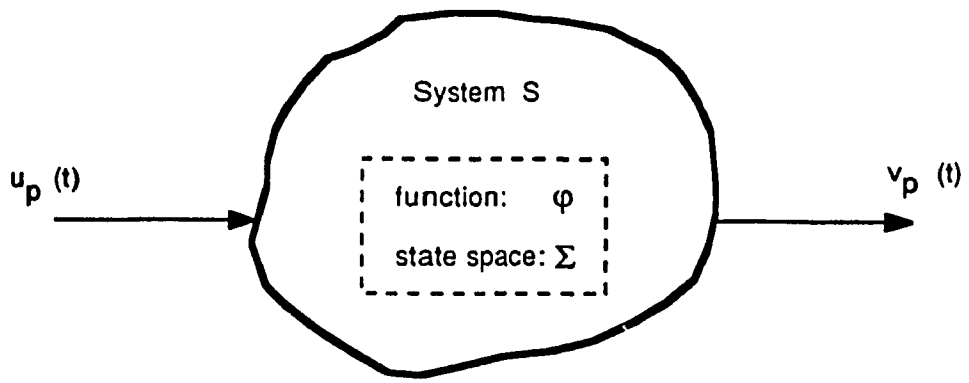


Fig. 1.3 Logic device

$\Sigma : \{ s \mid s \text{ is a state of } S \}$ $\Sigma \equiv R \times R \times R \ldots \times R \equiv R^m$

Σ is the set of all state variables.

$U_p = \{ u_p(t) \mid u_p(t) \text{ is a parameterized input function} \}$

Each element of U_p is a function of the parameter p and the time t:

$$u_p(t): R \times R \rightarrow R$$

$\varphi : \Sigma \times U_p \times R \rightarrow R$

The function φ maps a state, $s \in \Sigma$, at time t=0, an input function $u_p(t) \in U_p$ on the interval [0, t], and the time, $t \in R$, into the value of $v_p(t)$ at time t.

$$v_p(t) = \varphi\ (s, u_p, t)$$

The devices used in digital systems are continuous, causal and time invariant. While constructing a logic model, it will be required to decide whether or not these

properties should be preserved. In the remaining of this section, these properties will be defined and studied with respect to logic devices and logic simulation.

1.5.1 Time invariance and logic simulation

A system is time invariant if, given the initial state at t_0 and the input signals beginning at t_0, the resulting changes of state and the output signals after t_0 will always be the same for any t_0. More formally, since $v_p(t) = \varphi\ (s, u_p, t)$ is the response of the system S on the interval [0, t], then if the initial state at time T is also s and if the input signal is simply time shifted $u_p(t-T)$, then $v_p(t-T)$ will be the response on the interval [T, t+T].

Time invariance is easily preserved in a logic simulator. Given any initial time, a logic simulator normally produces the same results for the same initial state and the same input signals.

1.5.2 Causality and logic simulation

A system is causal if any changes at the inputs always precede its effect on the state and the outputs. More formally, $v_p(t')$ may depend on $u_p(t)\ \forall t < t'$ but does not depend on $u_p(t)\ \forall t \geq t'$. Causality is the essential justification for modeling propagation delay.

Whether it is analog or logic, a simulation always consists of predicting the future given the state of the model at current time and the inputs. Analog simulation is based on the sample-and-hold principle, the state and the outputs at the next time step are all evaluated simultaneously. The time step is very small and the prediction is based on the linearization of the set of differential equations that describe the circuit. Causality is always preserved since all voltages and all currents and therefore all changes of state are computed for the same time interval.

Logic simulation is event-driven and the simulation algorithms are based on the cause-effect principle: event X causes event Y. The new state and the outputs

are evaluated sequentially, not simultaneously. *Thus causality is crucial and establishing causality in the logic model and handling timing relationship and causal events efficiently and accurately are the key to satisfactory logic simulation.* Since the event occurrences depend on the event definition, the selection of an event set will be critical to the success/effectiveness in logic simulation. Otherwise, management of time advancement poses an inefficiency problem in simulation.

Preserving causality in the logic model simply requires a strictly positive propagation delay model. With negative propagation delay, an input event X arriving at time t_X would produce an output event Y at time $t_Y < t_X$, meaning that the output signal for $t > t_Y$, depends on an input signal at time $t_X > t$. According to the definition, such a model does not preserve causality.

1.5.3 Continuity and combinational devices

A system is continuous if a small change at the inputs produce a small change at the outputs. Mathematically, given a function f(a) and a limit $a=\lim a_p$ where a_p describes any convergent sequence of values for a, f(a) is continuous at a if $\lim f(a_p)= f(a)$ [27]. It is quite simple to verify if continuity is preserved in a logic model. Specifically, the model of the system in Fig. 1.3 is continuous in p, if for a small change in the parameter p, there is a small change in the output.

For example, assume the system S is a buffer and its model is a constant propagation delay with a single parameter (t_p). Suppose now that a small pulse of width p is applied as the input $u_p(t)$ and that transitions times are zero. As shown in Fig. 1.4, this delay model preserves continuity since for any small changes in the parameter p (pulse width) there is a small change in the output. In section 1.6, signal representations that do not preserve continuity will be examined.

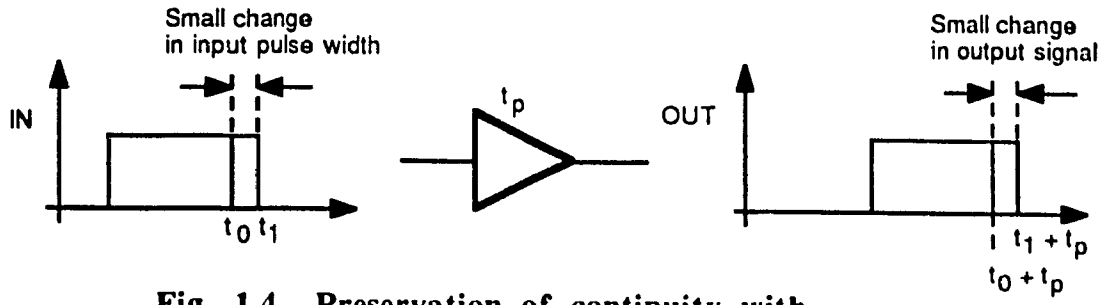


Fig. 1.4 Preservation of continuity with a constant propagation delay model

1.5.4 Continuity and sequential devices

Many logic devices are modeled using logic states (flip-flop, register, random access memory). The binary state of each logic variables depends on some past input condition. For example, at the rising edge of the clock of a D flip-flop, the state variable and the output take the value of the input (0 or 1). Because of the continuous nature of the devices, all intermediate values are possible, perfect binary decision is not possible in continuous systems. The BDT (Binary Decision Theorem) is presented in Appendix A. This is also called the synchronization problem or the metastability problem.

The problem is solved by either allowing sufficient decision time so the probability of having an intermediate value is near zero or else by considering as unacceptable the input conditions that produce intermediate values. This has led to timing constraints like set-up and hold times. Either of these approaches should be used when modeling logic devices with logic state.

1.5.5 Continuity and changes of state

In a continuous system, the state variables are continuous and they change continuously. When modeling a logic device, it is not desirable to keep track of continuously changing state variables. A logic model is characterized by discrete

events and changes of state. Discrete changes of state are often refered to as a timed state sequence. *In a timed state sequence, there is a finite number of changes of state in any time interval whereas in continuous change of state there is theoretically an infinite number of changes of state in any interval of time.*

Modeling the continuous change of state with a timed state sequence effectively transforms continuous time into discrete time. This is a major issue in the modeling of logic devices and the solution is not simple. In simulation, a logic device can effectively be driven by arbitrary input signals that can change an infinite (very large) number of times in any interval. For example, a counter limited to 10 Mhz might be driven by a 100 Mhz clock in a given circuit. This causes a continuous change of state in the modeled device. *The problem is to process the theoretically continuous stream of input events and generate a timed state sequence.*

This summarizes the key points that must be considered when constructing the logic models for analog circuits.

1.6 Problems with current modeling techniques

In this section, the current modeling techniques will be reviewed to illustrate various problems. When possible, a probable cause related to some theoretical aspect of modeling causal, continuous and time invariant systems will be suggested. Problems are grouped into three areas:

- Signal representation problems
- Propagation delay model problems
- Changes of state problems

All of the problems described here are caused by the desire to preserve accuracy in modeling continuous signals and fine timing details using logic simulation. The complexity and the performance of modern digital designs require such accuracy and therefore these problems are quite real.

1.6.1 Signal representation problems

The objective in modeling the signal on a wire is to accurately describe the transmission of the information from one device to another. At analog level, this information is modeled with the node voltages and the branch currents. At logic level, the information is modeled with logic events describing logic signals. Various models for signal representation will be next examined.

The simplest model consist in a list of logic events describing when the signal changes to zero or one based on the crossing of a fixed common threshold. This model is inadequate for modern designs where wiring capacitances have a non negligible impact on the speed performance. In MOS circuits, the transition times are significant and can not be neglected. As illustrated in Fig. 1.5, such an event definition introduces important timing errors. These errors may cause incorrect changes in the modeled logic state of the receiving circuits and may require negative propagation delays.

Threshold based logic events do not model the transfer of information accurately because the analog signal is not properly described. Using multiple thresholds is not sufficient either to properly model the analog signal. As shown in

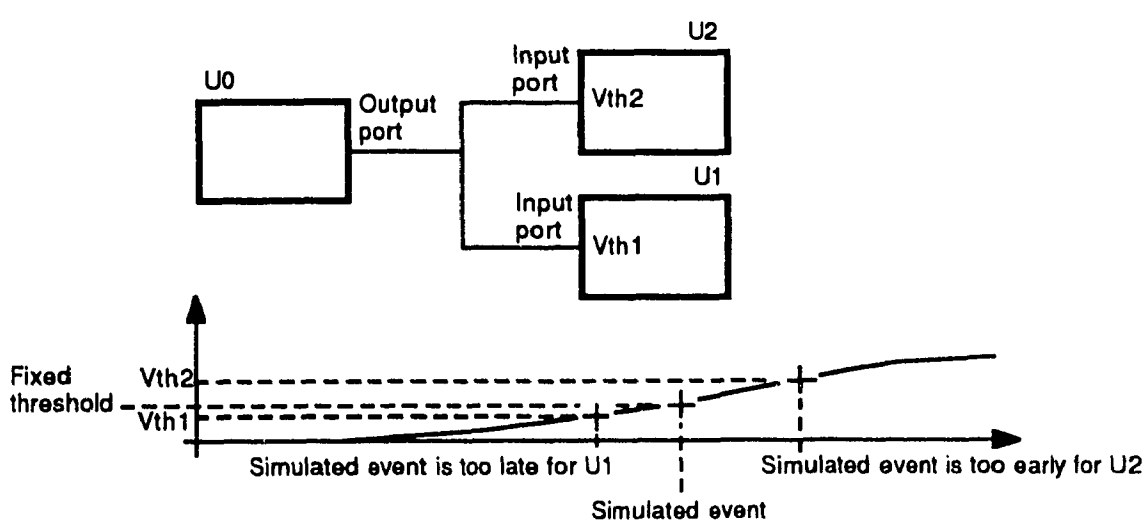


Fig. 1.5 Timing errors with a common fixed threshold for input and output events

Fig. 1.6a, glitches in the X range or in the 0 range are not described by the sequence of threshold based events `X1X0`. Fictitious events as in Fig. 1.6b have been proposed [30] to simulate the crossing of the threshold for events within either ranges. The processing of such events is difficult since the chronological order is not maintainted. For example, the sequence of events for the signal in Fig. 1.6b is `XXX1X00X` and should have been `XXX1X0X0` [30]. Minima and maxima have also been combined with threshold based events [30] as in Fig. 1.6c. Mixing minima and maxima with threshold based events in such manner is not necessary. In fact minima and maxima are the basis of the new event set proposed in chapter 3.

Each of the problems described above seems to be associated with a poor representation of the signal or some missing information about the signal. To solve

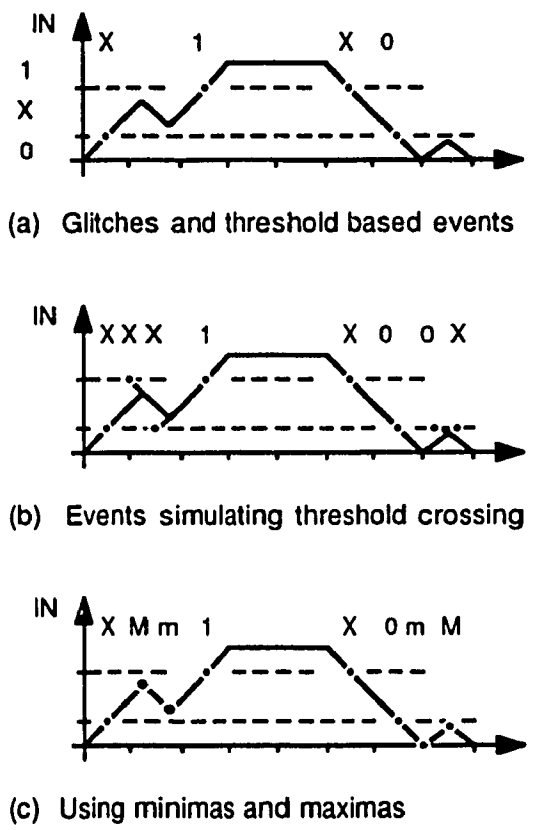


Fig. 1.6 Modeling analog signal with threshold based events

the problem we propose to use the analog signal in a detailed form to transmit the information between logic devices. In chapter 3, a logic event set will be defined to describe analog signals.

1.6.2 Propagation delay model problems

Propagation delays are used in logic simulators to model the passage of time. They are interpreted as the delay between the input events and the resulting output events. When using the constant delay model, the occurrence of an output event is computed by adding a value (t_p) to the occurrence of the input event. Unfortunately, real devices rarely behave in a simple fashion. The propagation delay for low to high transitions are different from that for high to low transitions; the propagation delays depend on things such as which input is causing the event, if the receiver device ignores glitches and so on. Therefore, more elaborate delay models have been used and they include inertial delay [29-31, 47-50], transport delay [31, 48, 49] and spike delay [29] models. These models modify the output signal by replacing or removing events to model a specific behavior or to insure logical consistency in the produced signal. For example, a 1 event following another 1 event will be removed. The popular inertial delay model will be used to illustrate this.

Inertial delays are often used to model the effect of either a large capacitive load or a large internal capacitor that filters out narrow pulses. As illustrated in Fig. 1.7a, pulse 'a' does not produce an output event since its width is smaller than the propagation delay. The problem with inertial delay is related to the transmission of information. Suppose that inertial delay is used to model a large load capacitance. Then the actual output signal might resemble the signal shown in Fig. 1.7c. The pulse a was removed but it could actually be seen by a receiving device with a lower threshold. Such problems have been described in section 1.6.1. In this case, the receiving circuit and not the transmitting circuit should decide on the removal of the

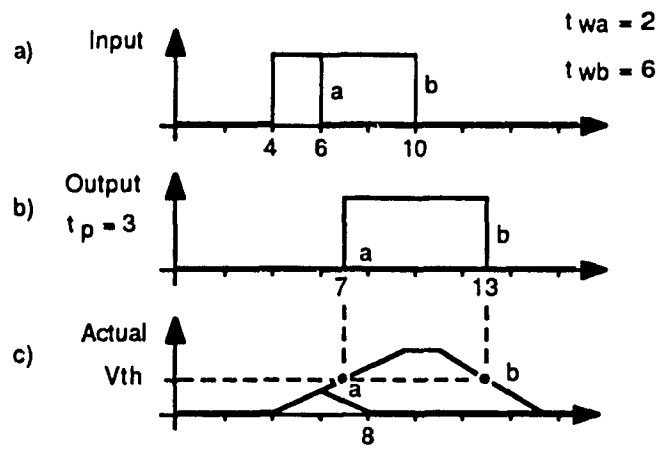


Fig. 1.7 Inertial delay model

event. This is clearly a failure to model the transmission of information between devices.

The fundamental problem is that these delay models are combining three aspects of the behavior that should be separately modeled: transmission of information (signal model), propagation delay and logic state. The modeling of a large wire capacitance is associated with the signal while an internal capacitance would be part of the device logic model. The internal capacitance might produce a pure propagation delay or a logic state. In this thesis we will demonstrate that modeling these aspects separately will improve the model for the transmission of information between logic devices and will result in simpler or at least more consistent logic models.

Another aspect that will be considered is the preservation of continuity. Even if it is not obvious how this will improve logic models, we believe that since all logic devices are continuous, it is wise to preserve this property. Unfortunately, inertial propagation delay models do not preserve continuity. This is demonstrated in Fig. 1.8 where a small change in the input pulse width produces a large change in the output waveform. In fact this is not possible in the real world, continuity commands that as

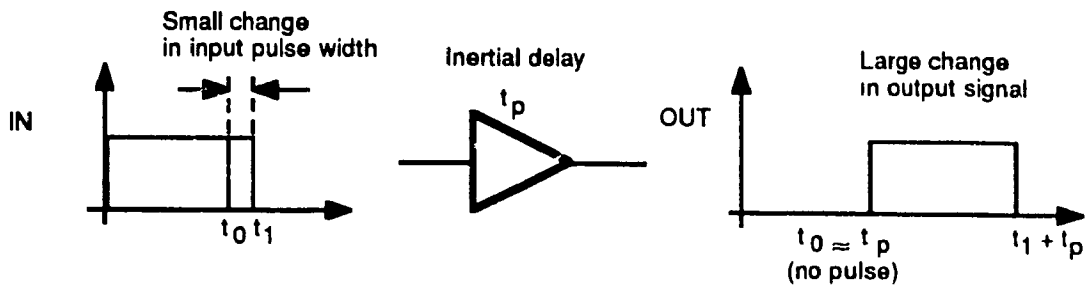


Fig. 1.8 Absence of continuity in the inertial delay model

$t_1 \rightarrow t_0$, the difference between the output waveforms gets smaller. This model clearly fails to reproduce the correct behavior.

1.6.3 Changes of state problems

As discussed in section 1.5.5, a logic model must perform the transformation of continuous changes of state into a timed state sequence. This is accomplished by adding timing constraints, for example, by limiting the clock period of a counter or the width of a reset pulse. These constraints add to the constraints associated with binary decision described in section 1.5.4. In this section, we will analyze how the following modeling techniques have succeeded at doing the continuous time to discrete time transformation:

- device timing constraints
- signal timing constraints
- real time formalism

1.6.3.1 Device timing constraints

The most popular logic modeling technique uses timing constraints such as set-up and hold times to define the acceptable sequences of input events. A typical simulator will compare the simulated event separation with the requirement and output a message in case of violation [29, 31, 48]. The transformation of continuous change of state into a timed state sequence is often not performed. For example, in

many logic models, the state of a flip-flop does not change to undefined in case of a timing violation. In the modified inertial delay model [29], some sort of transformation to a timed sequence of events is performed by replacing many close events by an undefined value.

Use of device timing constraints is basically correct for transforming continuous time into discrete time. The remaining problem is the absence of a systematic mechanism to construct a model that processes timing constraints and at the same time performs the transformation of continuous change of state into a timed state sequence. Since violation of timing constraints affects the state of the device they must be verified as part of the device model and not as an external check on the logic signal.

1.6.3.2 Signal timing constraints

In their quest for better logic models, designers have also used multiple thresholds to identify glitches directly on the signal. The signal voltage range is then divided into three (0, 1, X). It was demonstrated in section 1.6.1 that multiple thresholds events do not preserve continuity and do not easily model continuous signals. In addition, events defined using multiple thresholds require a more complex delay model and a more complex logic function and are therefore difficult to process.

For example, multiple thresholds lead to multiple-valued logic [42]. Table 1.1 shows the AND function using 2-valued logic while table 1.2 shows the same function using 6-valued logic. Each entry in table 1.2 is a sequence of three consecutive events based on two threshold dividing the signal range into three values (0, 1 and X).

AND	0	1
0	0	0
1	0	1

Table 1.1 AND function using 2-valued logic

AND	(0,0,0)	(1,1,1)	(0,X,1)	(1,X,0)	(0,X,0)	(1,X,1)
(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)
(1,1,1)	(0,0,0)	(1,1,1)	(0,X,1)	(1,X,0)	(0,X,0)	(1,X,1)
(0,X,1)	(0,0,0)	(0,X,1)	(0,X,1)	(0,X,0)	(0,X,0)	(0,X,1)
(1,X,0)	(0,0,0)	(1,X,0)	(0,X,0)	(1,X,0)	(0,X,0)	(1,X,0)
(0,X,0)	(0,0,0)	(0,X,0)	(0,X,0)	(0,X,0)	(0,X,0)	(0,X,0)
(1,X,1)	(0,0,0)	(1,X,1)	(0,X,1)	(1,X,0)	(0,X,0)	(1,X,1)

Table 1.2 AND function using 6-valued logic

Multiple-valued logic can be done for simple gates but then predicting the behavior of complex devices becomes quite complicated and no simple answer has been provided yet. Multiple-valued logic is the result of multiple-thresholds and significantly complicates the processing of events. It will be shown that 2-valued logic is sufficient even to process glitches.

Using multiple thresholds is probably the worst approach; the transformation of the continuous change of state into a timed state sequence is performed on the signal instead of the device state and the logic function gets very complex. Even with single threshold, use of signal timing constraints to identify glitches on the signal is not necessary.

1.6.3.3 Real time systems

Computer scientists have studied the modeling of real time systems which should include logic systems and have attacked the problem in a more formal way. The basic approach consists of including time into first order logic with the objective of proving timing properties such as the safety of a control system. In temporal logic (TL) [53], the operator "eventually ($\Diamond$)" has been added to propositional logic to define real time properties. Time constrained version of temporal operator have been used to more accurately describe the system's behavior. For example, in metric interval temporal logic (MITL) [54], the operator $\Diamond_{[2,4]}$ would indicate eventually

between 2 and 4 unit of time. Another approach called timed automata (TA) [55, 56] uses artificial variables called clocks to time the change of state. Whereas all of these techniques aimed at formal proofs, TA is also well suited for simulation, since a mechanism is implicitly used to increment time.

There are two aspects to discrete time abstraction: (1) is the model capable of processing continuous time? and (2) does it perform the transformation into a timed state sequence? TA is capable of processing continuous time and can perform the transformation if the designer does construct a correct automaton. Unfortunately, TA are general purpose and are difficult to use. For example, TA does not distinguish timing constraints from propagation delays. We beleive they should be separately modeled and processed.

As such, there is no particular problem with TA. They are simply incomplete to deal with the whole modeling problem addressed here because the relation between TA and continuous systems is not clearly formulated. A modified TA will be proposed in chapter 6. The new TA, called Continuous Time Automaton (CTA), is specially designed to process timing constraints and transform continuous time into discrete time.

1.7 Proposed logic modeling technique

Basically, the discrete nature of logic signals and devices does not blend easily with the continuous analog world they have to capture. As mentioned by M. R. Lightner [33], important theoretical work is still required to formalize the hypotheses and axioms for the abstraction of analog circuits to logic models. Even if logic simulation with timing specification has been an active area of research [34-39], very little work has been done to define a better formalism [30, 40]. Most researchers are concerned with simulator efficiency (timing accuracy, speed and memory requirements). Our objective is to define such a formalism while maintaining timing

accuracy and simulation efficiency. Such a formalism will improve the accuracy in the signal representation and provide a more realistic delay model and a systematic technique to process timing constraints.

1.7.1 Step 1: What is a logic device?

In analog circuit simulation, the elements are described by a set of equations relating the voltages and the currents at the interface ports and the simulation consists of solving the network equations based on Kirchhoff laws. The abstraction of an analog circuit to a logic circuit consists of grouping the analog devices and partitioning the circuit into logic devices and wires. Even though Kirchhoff laws could be used to model part of the logic circuit as in hybrid simulation, it was decided not to use them in the logic model. Each group of analog devices must meet three hypotheses to be considered a logic device. Meeting these hypotheses also insures that Kirchhoff laws are not required. Therefore, a logic device is an analog circuit designed to meet certain criteria. The first hypothesis specifies the type of input port:

<u>Input port hypothesis</u>

Input ports are threshold sensitive

or in general sensitive to certain events.

This corresponds to the usual assumption made by digital system designers. A logic device will react only when a threshold is crossed. The effect might be a change of state or a change on the output signals. The second hypothesis specifies the type of output port:

<u>Output signal hypothesis</u>

Output ports generate signals that do not normally
linger around the threshold voltages of the input ports
or in general that produce events compatibles with the input ports

The thresholds are simply crossed, otherwise, the signal is not considered an acceptable logic signal. Even if this might seem fuzzy for now, this hypothesis enforces the device hypothesis by further limiting the class of analog circuits that are considered logic devices. In general, these two hypotheses indicate that, for logic circuits to operate as specified, the output signals must be compatible with the thresholds of the input ports connected to it. In simpler term, since the logic devices are sensitive to threshold, the output signals must neatly traverse the thresholds. The third hypothesis specifies how output ports are connected to input ports:

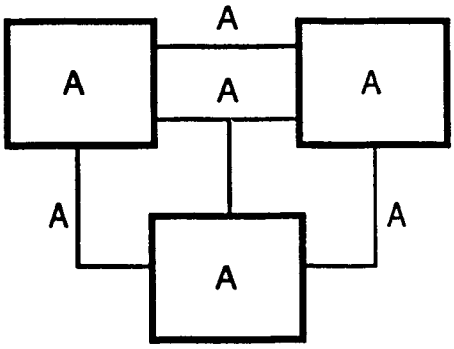
<u>Network hypothesis</u>

A wire always consists of a unique output port
driving one or more input ports forming a time invariant load.
The output port might change with time.

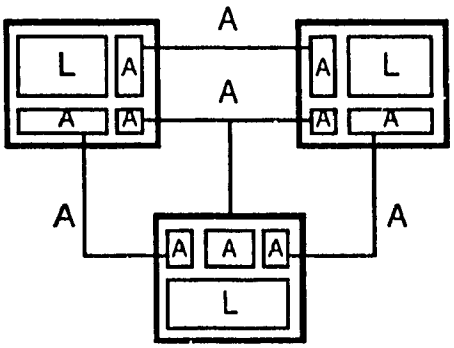
This is the basic hypothesis that allows a designer to model a device without using Kirchhoff laws. This is restrictive, but it actually corresponds to the engineering practice. For example, devices specified in data books [43, 44] including standard cells for integrated circuit design [45] are all described assuming the network hypothesis. According to this hypothesis tristate devices are logic devices, but open collector devices and switches are not. Bus resolution function [illegible]ted by VHDL [31] would be limited to tristate busses and signals. Obviously, our [illegible]ective

is not to decide what can or can not be simulated. We merely describe what is theoretically considered a logic circuit.

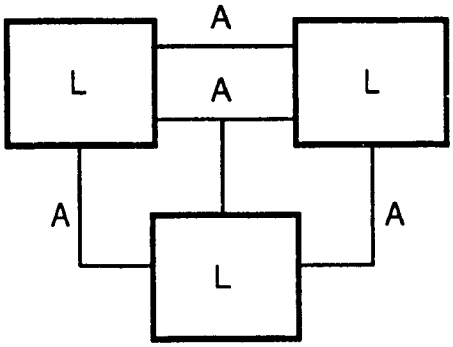
These hypotheses make no reference to the signal representation. In fact, logic devices also process analog signals. As shown in Fig. 1.9, an analog simulator uses differential equations for the devices to compute the analog signals. A hybrid simulator uses logic models for the internal operation of the device and differential equations for the interface part of the devices to compute the analog signals. A logic simulator uses a logic model for the devices and also computes the analog signal. In section 1.6.1, we have demonstrated that trying to define logic signals without



(a) Analog devices and analog signals
(analog simulation)



(b) Hybrid devices and analog signals
(hybrid simulation)



(c) Logic devices and analog signals
(logic simulation)

Fig. 1.9 Classification of simulators

reference to analog signals leads to problems in the simulation. *Allowing logic models to process analog signals is a key point in our approach and it will lead to logic events describing analog signals.*

1.7.2 Step 2: What is a logic model?

The next step is to define a logic model for the logic device defined in step 1. The difference between an analog model and a logic model lies in the representation and the processing of the time and of the state changes. In an analog model, the time and the state changes are continuous whereas in logic models, they are assumed discrete.

During an analog simulation, multiple input events or waveforms are processed simultaneously, the time progresses continuously and the output events or waveforms are generated simultaneously. During a logic simulation, multiple input events are processed sequentially. State changes and output events are computed and predicted separately for each input event. Intuitively, a logic simulator processes the input signals as follow:

- The input events are derived from the analog input signals using the input event set definition.
- For each input event, the simulator waits some time corresponding to the input delay.
- Then the simulator processes each input event and produces a change of state if required. In general, a single input event may produce a sequence of state changes.
- Based on the input event and the new state, output events are generated after some time corresponding to the output delay.

In the general case, input and output delays are arbitrary, therefore the chronological order of the predicted state changes and output events might not be

preserved and might become difficult to process. For example, assume a device X with a state S and two input ports A and B. If the input delay on port A is smaller than the input delay on port B, then the change of state S_A might occur before the change of state S_B even if the event on A occurs after that on B. With complex devices, not preserving the chronological order in the changes of state might be difficult to model. *To simplify the construction of logic models, the analog circuits will be partitioned such that the input and output delays are grouped and associated with each output.* As shown in Fig. 1.10, the input delays are reduced to zero and the delays are inserted at the output.

The conceptual model of Fig. 1.10 is based on two rules. The first rule indicates that the input delay is zero:

> State model rule
>
> In a logic model the change of state is simultaneous with the input event causing it.

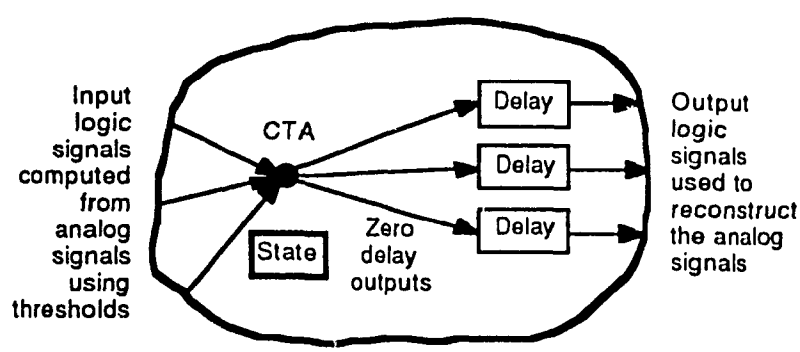


Fig. 1.10 Conceptual model of a logic device

This rule insures that the chronological order of the changes of state is preserved with respect to the input sequence of events. This rule also indicates that delays must be associated with outputs. The second rule concerns the delay model:

> Delay model rule
>
> The propagation delay model for each output must preserve continuity

This is required to prevent problems such as those described in section 1.6.2. According to these rules, a logic circuit consists of logic devices connected as in Fig. 1.11. This conceptual model for logic circuits suggests how simulation will proceed. First, input events are computed using the analog signal and the input threshold. Second, the logic model is used and a change of state is computed. Then, if required, a continuity preserving delay model is used to compute the output events.

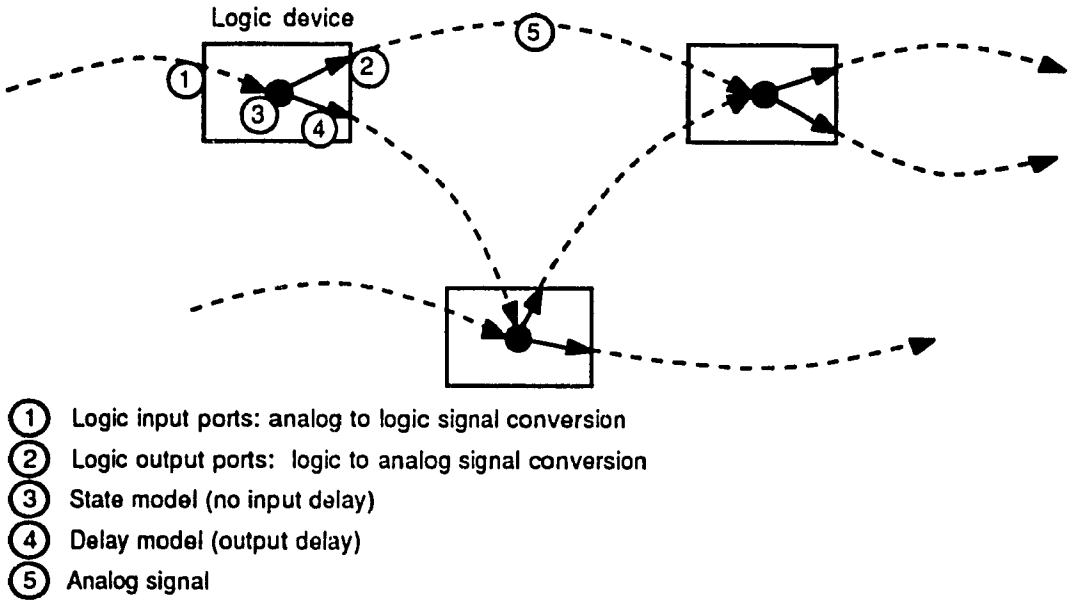


Fig. 1.11 Partitioning an analog circuit into logic devices

The output events which are defined differently from the input events, are then used to reconstruct the analog signal.

1.7.3 Outline of step 3: Construction of logic models

This third step is the core of this thesis. It is assumed that the analog circuit is partitioned into logic devices according to the conceptual model of Fig. 1.11. In step 3, a logic model is constructed for each logic device. The logic model construction process will match the conceptual model by clearly separating the modeling of the transmission of information, the propagation delays and the state changes.

1.8 Summary

In this chapter, the problem of modeling analog circuits with logic model was described. Basically, a better mathematical formalism for the description of the objects and the logic modeling process is required. In particular, modeling the waveform of the signal seems essential and preserving continuity, causality and time invariance should be considered.

Abstraction hypotheses for logic devices and construction rules for logic models have been formulated in section 1.7.1 and 1.7.2 . The remaining problem is to follow a mathematically formulated process and construct logic models according to the conceptual model of Fig. 1.11. This conceptual model divides the logic model into three parts, each with a specific modeling objective:

- A logic signal model to accurately model the transmission of information between logic devices.
- A propagation delay model to accurately model the delay between changes of state and output events.
- A state model to accurately model the changes of state and the transformation of continuous changes of state into a timed state sequence.

These correspond to the research objectives of the work described in this thesis. Chapter 2 describes the formal framework that supports logic modeling. The framework is a mathematical definition of the objects that will be used: logic events, logic signals, logic devices and logic simulator. It is based on simple mathematical premises and on the classical system theory. Modeling of transmission of information and the delay model are studied in chapter 3 and the state model in chapter 4. Chapter 5 includes a summary of the results and a comparison with other modeling techniques.

This thesis should be useful to those interested in the specification, design, verification, simulation, testing and formal verification of digital systems.

Chapter 2

Formal framework

The first step to a better formalism is to examine the mathematical premises and define the objects we want to use: logic events, logic signals, logic devices and logic simulators. In this chapter, the mathematics of functions and basic information theory principles are reviewed and applied to define logic signals and logic events. The definitions emphasize the importance of relating logic signals to analog signals. Then, the discrete event system model developed by B. P. Zeigler [48] is used and adapted to define logic devices and logic simulators. This formalism is based on the classical system theory where inputs, outputs, states, state transition functions and output functions are the constituents of the model.

2.1 Mathematical premises

The mathematical concept of *functions* is reviewed. To make things simple, it is assumed that functions can map to or from the set of Natural (N) or the set of Real ($\Re$) numbers. Fig. 2.1 shows the four possible function types. In signal processing theory, distinction between the different types of functions lies in the continuous ($\Re$) or discrete (N) nature of the domain and the range. The range can be either *continuous* or *discrete* (quantized) and the domain (time) can be *continuous* or *discrete* leading to the following functions:

Continuous function: $\{ f : \Re \rightarrow \Re \}$

Discrete function: $\{ f : N \rightarrow \Re \}$

Quantized function: $\{ f : \Re \rightarrow N \}$

Logic function: $\{ f : N \rightarrow N \}$

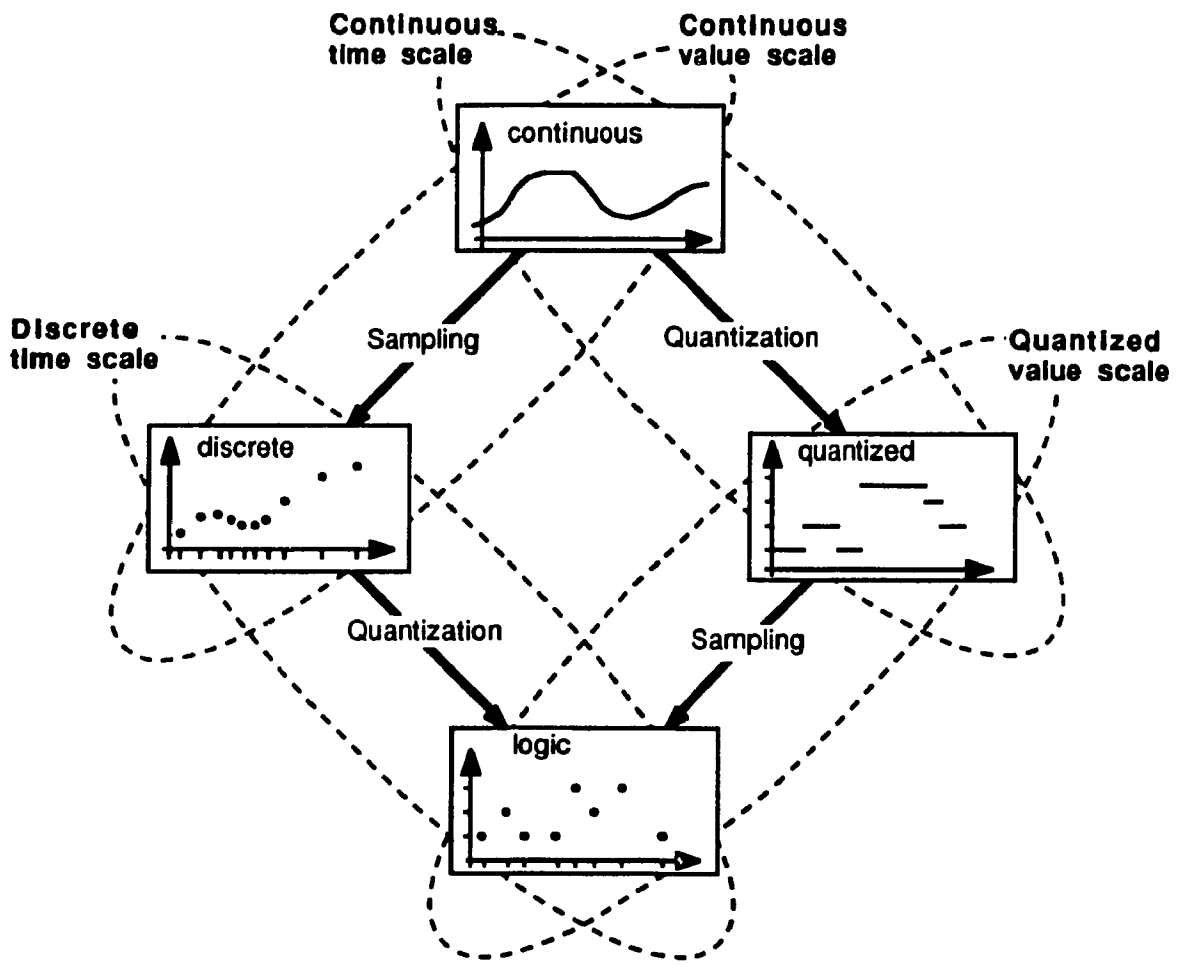


Fig. 2.1 Classification of mathematical functions

As shown in Fig. 2.1, there are also two fundamental operations that can be used to transform these functions:

Sampling: transformation of the domain set from real to natural

Quantization: transformation of the range set from real to natural

2.2 Signals and information

In signal processing theory, we are concerned with the information carried by a signal. A signal is a function of time representing a physical variable, like a voltage, and the information it carries is defined as the part of the signal that cannot be

inferred. Without going into lengthy mathematical discussion about the measure of information carried by different signals, we can define the three types of signals generally encountered. Since all signals are continuous functions of time, we must define them by giving the properties that differentiate them.

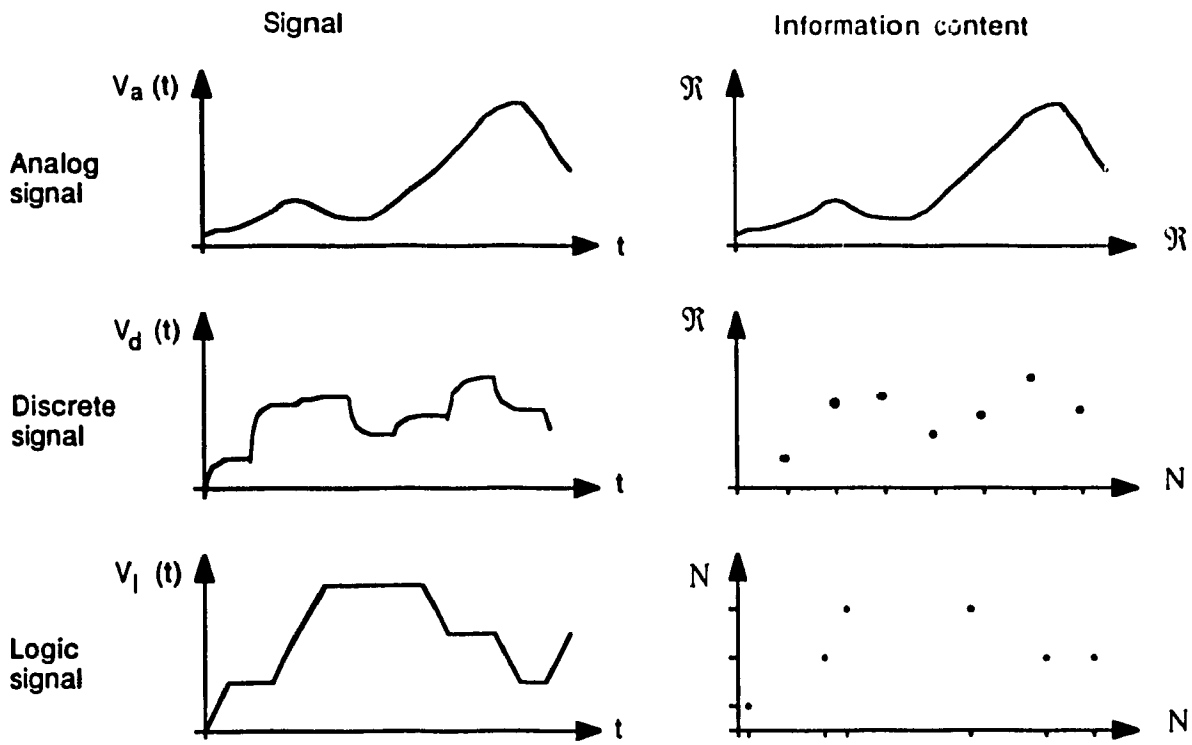


Fig. 2.2 Classification of signals

The three types of signal shown in Fig. 2.2 are:

Analog signal: An analog signal is a continuous function of time, $f(t) : \Re \rightarrow \Re$. All signals are analog. The information is carried by *points (t, f(t))*.

Discrete signal: A signal is considered discrete if it can be approximated by a discrete function, $s_k : N \rightarrow \Re$, and an interpolation function. The interpolation function is used to reconstruct the analog signal between the samples. For example, an exponential function is used for interpolation in

Fig. 2.2. The information is carried by *samples* (n, s_n), where n is used as a time index, instead of the real time value.

Logic signal: A signal is considered logic if it can be approximated by a logic function, $e_k : N \rightarrow N$, and a known interpolation function. For example, line segments are used for interpolation in Fig. 2.2. The information is carried by *events* (n, e_n), where n is used as a time index, and e_n is an event. Notice that the definition of logic signals will be slightly modified to be used in simulation, see section 2.5. Once it is established that the signal is a logic signal, only events need to be processed, the interpolation function is only required to reconstruct the analog signal.

2.3 Signal transformation

As mentioned, the definition given for discrete and logic signals should be interpreted as properties that an analog signal must have in order to be "modeled" as a discrete or logic signal. According to the definition, the output signal $v_o(t)$ in Fig. 2.3 is a logic signal since it can be approximated by a sequence of events and an interpolation function. In this case, the interpolation function will depend on the output characteristics of the buffer. The buffer is therefore a device that transforms an analog signal into a logic signal.

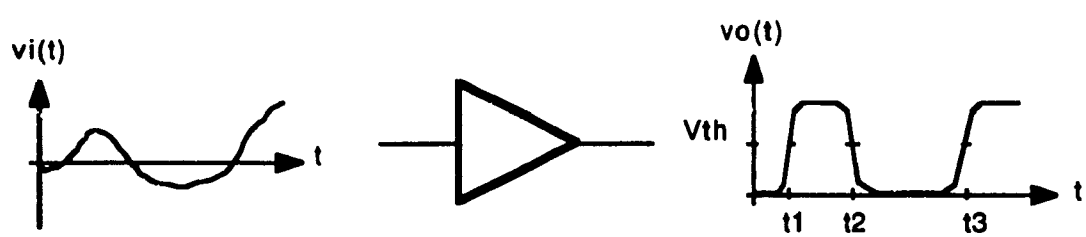


Fig. 2.3 Buffer

Similarly, the output signal $v_0(t)$ in Fig. 2.4 is a discrete signal since it can be approximated by a sequence of samples and an interpolation function, here an exponential function. Even if a clock is required to control the sampling instants, it does not determine the nature of the output signal. The nature of a signal depends on its information content: a list of samples for discrete signals and a list of events for logic signals.

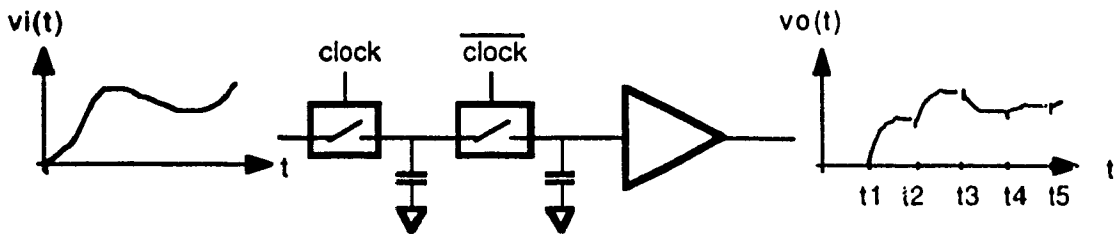


Fig. 2.4 Sample and hold

Signal transformation will not be discussed further, it is more related to information theory than logic simulation.

2.4 Logic event

A logic event is a tuple (n, e_n) where $n \in N$ is the time index corresponding to $t_n \in \Re$ and $e_n \in E$ is the event. The event set E which is a subset of N. For a single threshold based logic signal, the event set E for binary signals is:

$E = \{0, 1\}$ where $1 : [(v(t)=V_{Th}) \text{ and } (\dot{v}(t)>0)]$

$0 : [(v(t)=V_{Th}) \text{ and } (\dot{v}(t)<0)]$

and $\dot{v}(t)$ is the first derivate of $v(t)$.

The 1 event occurs when the signal crosses the threshold while rising and the 0 event occurs when it crosses the threshold while falling. For two-threshold logic signals, the event set would be:

$E = \{0X, X1, 1X, X0\}$ where $X1 : [(v(t)=V_{ThH}) \text{ and } (\dot{v}(t)>0)]$

$$1X : [(v(t)=V_{ThH}) \text{ and } (\dot{v}(t)<0)]$$

$$0X : [(v(t)=V_{ThL}) \text{ and } (\dot{v}(t)>0)]$$

$$X0 : [(v(t)=V_{ThL}) \text{ and } (\dot{v}(t)<0)]$$

The above event sets illustrate events based on threshold crossing. Notice that an event is not necessarily the crossing of a threshold. In general, an event is the assertion of a predicate. For example, the following predicates define the events for minima and maxima:

$$m : [(\dot{v}(t)=0) \text{ and } (\ddot{v}(t)>0)]$$

$$M : [(\dot{v}(t)=0) \text{ and } (\ddot{v}(t)<0)]$$

2.5 Logic signal

A logic signal maps a natural number, the time index, to an event. It follows that many logic signals may have the same index at different times, for example n=1 might correspond to t=35nsec for a signal and to t=10nsec for another. A global time index could be used, but using a real time scale ($\Re$) is preferred. This will allow analog, discrete and logic signals to coexist in a simulator. The new definitions based on a real time scale are:

Analog signal: $f(t) : \Re \rightarrow \Re$ maps time ($t \in \Re$) to point ($f \in \Re$)

Discrete signal: $s(t): \Re \rightarrow \Re^+$ maps time ($t \in \Re$) to sample or nil ($s \in \Re^+$)

$$\Re^+ = \Re \cup \Lambda$$

Logic signal: $e(t) : \Re \rightarrow E^+$ maps time ($t \in \Re$) to event or nil ($e \in E^+$)

$$E^+ = E \cup \Lambda$$

All signals map the same continuous time scale. This modified definition for signals has been used by B. P. Zeigler for discrete systems [40]. Analog signals map into a continuous variable, while discrete signals map into a sample or nil (Λ) if no sample exists. Similarly, logic signals map into an event when there is one or else to nil (Λ). The interpolation function has been removed from the signal definitions, it

does not carry any information and is normally a known parameter in the simulation. In the case of discrete and logic signals the analog signal could be reconstructed if an interpolation function is used instead of nil (Λ). If the event set is $E = \{0, 1\}$ then the augmented event set for simulation is $E^+ = \{0, 1, \Lambda\}$ and a binary logic signal would map the time to 0, 1 or Λ.

2.6 Logic devices and digital systems

In chapter 1, we presented three hypotheses that define the logic devices class. This class of devices process the information carried by logic signals as defined in section 2.5. In this section, a more formal definition for logic devices and digital systems is given. Digital systems are instances of a larger class of systems: discrete event systems. In the literature, a discrete event system refers to a system which is driven by events. It should not be confused with discrete systems which are characterized by discrete signals. Discrete event systems have been studied for a long time, and there are many references on the subject of discrete event system simulation. Axiomatic definitions for discrete event system modeling and simulation have been developed. B. P. Zeigler [48] proposes such a formalism where the models are constructed by defining levels of abstraction based on the amount of details in the models. This technique is directly applicable to digital systems and the resulting formalism for digital systems will be described in this section. The application is straightforward and merely consists in using names and abbreviations corresponding to the digital systems nomenclature. The five levels of abstraction are:

- Level 0. Observation Frame
- Level 1. Input/Output Relation Observation
- Level 2. Input/Output Function Observation
- Level 3. Logic Device
- Level 4. Digital System

- Level 5. Hierarchical Digital System

For simplicity, the logic elements are assumed to have one input and one output. Generalization to multiple inputs and outputs is straightforward.

Input/Output Observation Frame (Levels 0, 1 and 2)

For simplicity, levels 0, 1 and 2 have been combined. Assuming both input and output use the same event set E^+ as described in section 2.5, the observation frame is:

$O = < T, E^+ >$ where: $T = \Re$ is the time

$E^+ = \{0, 1, \Lambda\}$ is the event set

The input and output logic signals are defined as functions of time T as:

$\omega : T \rightarrow E^+$ Input logic signal $\omega(t)$

$\rho : T \rightarrow E^+$ Output logic signal $\rho(t)$

The set of all input and output logic signals are denoted Ω and R, therefore $\omega(t) \in \Omega$ and $\rho(t) \in R$.

Logic Device (Level 3)

A logic device processes the logic signal defined in the observation frame and is defined as:

$LD = < T, E^+, Q, \delta, \lambda >$

where: $<T, E^+>$ is an observation frame

Q is the state

δ is the state transition function

λ is the output function

The above abstraction corresponds to the model used in classical system theory. A system is completely described by an initial state, a state transition function and an output function. The state Q is not necessarily a logic state, like the state of a flip-flop, it includes any information required to uniquely predict the future

behavior for any input logic signal. Given an initial time $t_i \in T$ and a final time $t_f \in T$, the state transition function maps the state $q_i \in Q$ at t_i and the input logic signal $\omega \in \Omega$ in the interval $<t_i, t_f>$ into a new state $q_f = \delta(q_i, \omega) \in Q$ at t_f.

$$\delta : Q \times \Omega \rightarrow Q$$

The output function maps the state $q_i \in Q$ at t_i and the input logic signal $\omega \in \Omega$ in the interval $<t_i, t_f>$ into an output logic signal $\rho = \lambda(q_i, \omega) \in R$ on the same interval.

$$\lambda : Q \times \Omega \rightarrow (Y,T)$$

Digital Systems (Level 4)

Based on Zeigler's approach, each component is influenced by others and the information about who influences who is contained in the influencer set. The influencer set is an important aspect of event driven simulation. During simulation this set is used to schedule the components to be simulated when an event is produced. In Zeigler's nomenclature, a typical network is described as:

$$N = < D, \{S_d\}, \{I_d\}, \{Z_d\} >$$

Where D is the designation set containing the names of all components. For each component $d \in D$, there is a system model (S_d), a set of components influencing the component d (I_d)and an interface map (Z_d) describing how the output from the components in I_d affect the component d. Basically, there is a set of components $< D, \{S_d\} >$ and a coupling scheme $< \{I_d\}, \{Z_d\} >$. The coupling scheme as defined by Zeigler does not transpose easily to digital systems, so we propose a slightly different definition for digital systems where the coupling scheme is based on ports and wires. A *digital system* is a set of *logic devices*, a set of *ports* and a set of *wires* carrying *logic signals*, as depicted in Fig. 2.5.

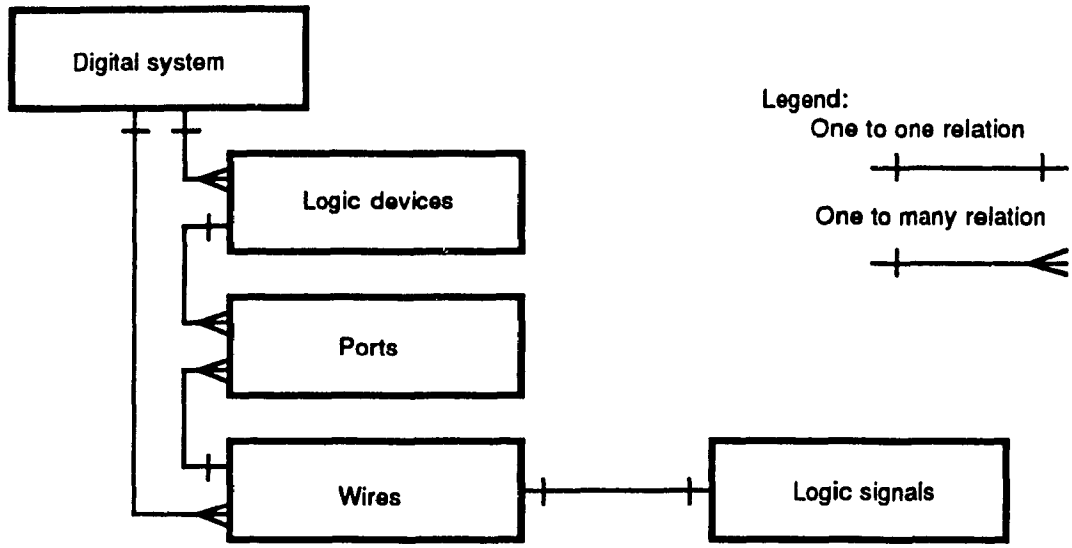


Fig. 2.5 Digital system basic data model

Devices have ports which are connected with wires to form a network, called the digital system or circuit, as in Fig. 2.6. More formally, a digital system (DS) is a network of logic devices (LD):

$DS = < D \{LD_d\} \{P_d\} \{W\} >$

D is the designation set containing the name of all logic devices. For each device named d there is a logic device model LD (level 4) and $\{LD_d\}$ is the set of all logic devices, one for each device name in D.

$LD_d = < T, E^+, Q, \delta, \lambda >$ for device d

P_d is the set of port names associated to device named d. $\{P_d\}$ is the set of all P_d. $\{W\}$ is the set of wires as usually defined in digital systems.

$W_i \subset \bigcup_{d \in D} d \times P_d$ A wire W_i is a subset of ports.

$\bigcap_{\text{all } i} W_i = \text{nil}$ A port appears only once in any wire.

For example, the digital system of Fig. 2.6 is composed of three devices and two wires and is described as:

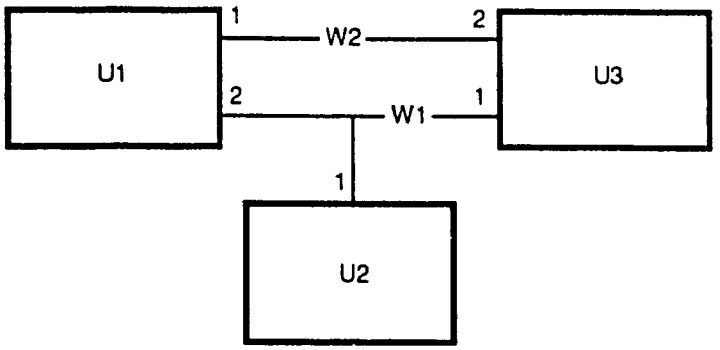


Fig. 2.6 Typical digital system

$DS = \langle D \{LD_d\} \{P_d\} \{W\} \rangle$ where:

$D = \{U1, U2, U3\}$

$\{LD_d\} = \{LD_{U1}, LD_{U2}, LD_{U3}\}$

$\{N_d\} = \{P_{U1}, P_{U2}, P_{U3}\}$

$P_{U1} = \{1, 2\}$

$P_{U2} = \{1\}$

$P_{U3} = \{1, 2\}$

$\{W\} = \{W_1, W_2\}$

$W_1 = \{U1\#2, U2\#1, U3\#1\}$

$W_2 = \{U1\#1, U3\#2\}$

Hierarchical Digital Systems (Level 5)

Although it is not critical for simulation, the formalism for hierarchical construct is important for digital systems. A hierarchical digital system is:

$HDS = \langle TR \{DS\} \{LD\} \rangle$

Where: TR is a tree

{DS} is a set of digital systems, the interior ports of TR

{LD} is a set of logic devices, the leaf ports of TR

2.7 Logic Simulator

2.7.1 Purpose of simulation

Without reviewing all theories about simulation, it is interesting to question ourselves about its purpose in general and the usefulness of logic simulation in particular. As discussed in [40], a model is supposed to help answer certain questions about an object, see Fig. 2.7. For digital systems, the questioner is the digital system designer, the object is a specific digital system and the model is the abstracted object, the model of the digital system. The simulation is the action of exercising a model. The usual questions asked by the designers are:

Does it meet the specification?

What is its behavior?

What is the critical path?

What is the worst case timing?

Does it work in the worst case condition?

Are the glitches harmful?

The usefulness of a model abstraction is formulated as follows:

Abs is a useful abstraction for answering the question *Q?* about the object *Obj* if:

(i) *Q?* is applicable to *Abs*

The question is meaningful when applied to the model. For example, questioning about the area of a circuit is not applicable to a logic model.

(ii) *Abs* is valid for *Q?* with respect to *Obj*

The answer obtained using the model is applicable to the object itself, i.e. Abs is a valid model of Obj.

(iii) space/time (*Q?*, *Abs*, *Procedure*) < space/time (*Q?*, *Obj*, *Procedure*)

It is cheaper to use the model than to use the object itself, which may not even be available.

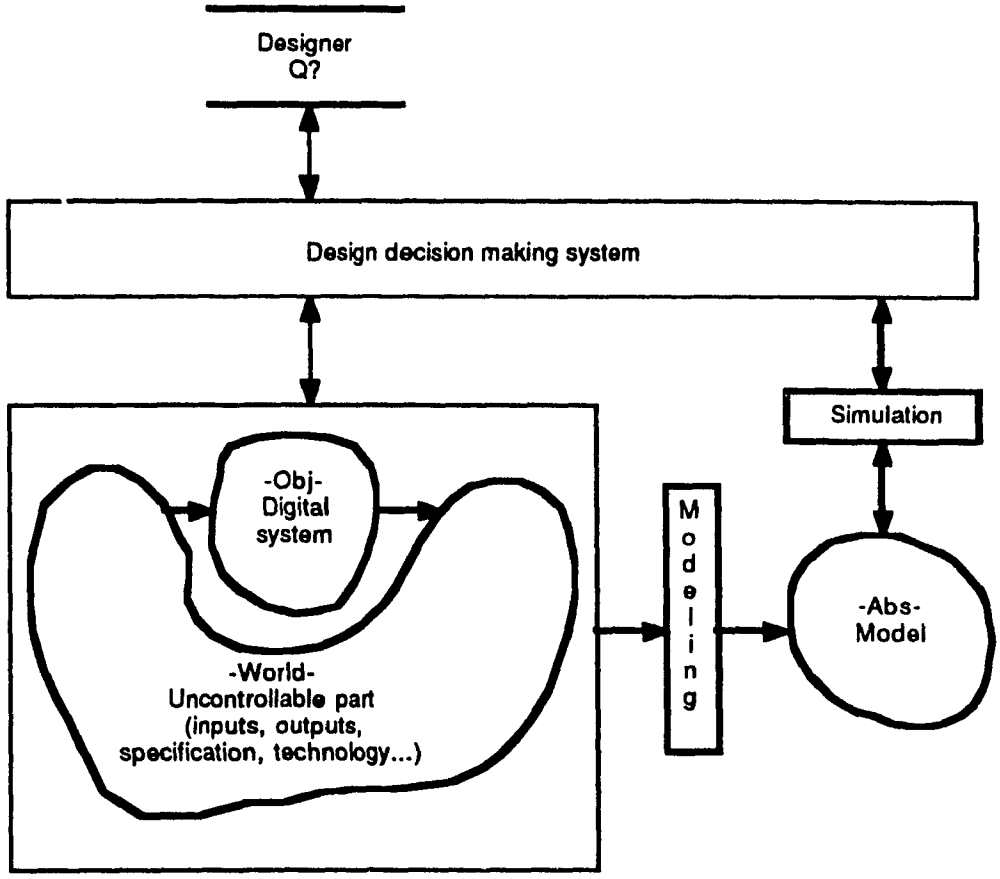


Fig. 2.7 Simulation environment

In general, multiple objectives imply multiple models. For example, digital systems can be modelled for area, power dissipation or functionality. The logic model described here is a model for answering questions about functionality. Modeling and simulation are part of the general decision making process, Fig. 2.7, where the designer uses the model (Abs) to help him take design decisions about the object of the design (Obj).

2.7.2 Definition of a logic simulator

In Zeigler's nomenclature, a logic simulator is a discrete event system (DEVS). A DEVS is a machine M_{DS} defined as follows:

$M_{DS} = < DS, S, t_a >$

Where: DS is the digital system.

S is a scheduler.

t_a is the time advance function.

Actually a discrete event simulator is a simple machine. The time advance function t_a selects the next event to be simulated and determines the wire or the signal on which it occurs. Then, the scheduler S calls all logic devices connected to the wire. The algorithm for M_{DS} is:

```
algorithm SimulateDS(tmax)
   repeat
      ta(CurentTime,CurrentEvent,CurrentWire);
            {Increases the CurrentTime, returns
               the next event to be simulated
                 and the wire on which it occurs}
      S(CurrentWire);
            {calls each LDd for which
              ∃i such that a port d#i ∈ CurrentWire}
   until time>tmax
end SimulateDS
```

2.8 Conclusion

In this chapter, the objects of logic simulation have been formally defined: logic events, logic signals, logic devices and logic simulation algorithms. The formal framework presented in this chapter describes the link between information theory, logic model and discrete time formalism. All of these aspects have been discussed in the past, but to our knowledge they have never been integrated in a single formalism. Even if most of the concepts discussed here are quite simple and generally well known, having them formally defined should prove to be very useful for the construction of logic models.

Chapter 3

Time model: logic signal and propagation delay

According to the conceptual model described in chapter 1, the construction of logic models breaks down into signal modeling, propagation delay modeling and state modeling. This chapter discusses the modeling of signals and propagation delays while state modeling is the subject of chapter 5. The objective in modeling the signals and the propagation delays is to accurately predict the occurrences of the logic events. This chapter therefore studies the handling of time. The logic signals model the *transmission time* between devices and the propagation delays model the *processing time* within the devices.

3.1 Logic event set

As discussed in chapter 1, it seems that unless the analog signal is used, the transmission of information between logic devices is not accurately modeled. The logic event set must therefore contain the information to reconstruct the analog signal and also the occurrences of the threshold crossings for all the associated input ports. To do so, separate event definitions are required for input and output ports. Output events describe the analog signal while input events are associated with the crossing of the input thresholds. Independent of variability of thresholds, both input and output event sets have the same universal logic abstraction:

$$E = \{0, 1, U, Z\}$$

The set includes two basic events (0 and 1), an undefined event (U) and a high impedance event (Z). Even if the event labels might look similar to others in use, they are defined rather differently.

3.1.1 Master output events

The output event set definition is based on piecewise linearization shown in Fig. 3.1. The dotted signal v(t) is approximated by the signal $v_a(t)$.

Basic events (0 and 1)

The dotted signal v(t) in Fig. 3.1a is the analog signal we wish to model. The event set is designed to minimize the error when reconstructing the analog signal using the logic events, even in the range bounded by the two thresholds V_{thL} and V_{thH}. In order to obtain an accurate model of the analog signal and prevent problems described in section 1.6.1, each signal will be characterized by transition times (t_{TLH} and t_{THL}) and logic levels (V_L and V_H). The transition times are defined between V_L and V_H. These parameters should be computed to reduce the error in the line segment approximation of the analog signal near threshold crossing. Normally, the logic levels are V_L=0v and V_H=5v and are not related to the device thresholds.

The events are defined as the corners of the signal $v_a(t)$. More formally, if V_L^+ is slightly above V_L and V_H^- is slightly below V_H, the master output events are defined as follows:

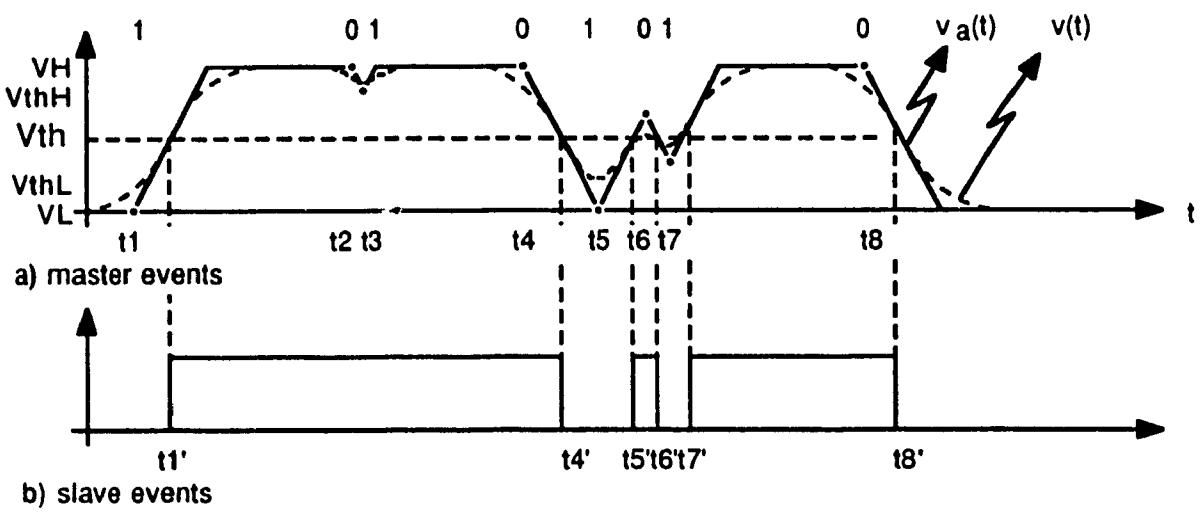


Fig. 3.1. Basic events (0 and 1)

1 event : $[(v_a(t)=V_L^+)$ and $\dot{v}_a(t)>0)]$

or $[(V_L^+<v_a(t)<V_H^-)$ and $(\dot{v}_a(t)=0)$ and $(\ddot{v}_a(t)>0)]$

0 event : $[(v(t)=V_H^-)$ and $\dot{v}_a(t)<0)]$

or $[(V_L^+<v_a(t)<V_H^-)$ and $(\dot{v}_a(t)=0)$ and $(\ddot{v}_a(t)<0)]$

According to the above predicates, an event occurs if the signal leaves the lower level (t_1, t_5) or the upper level (t_2, t_4, t_8) or when a maximum or a minimum is reached (t_3, t_6, t_7). In short, points on the signal $v_a(t)$ where $\frac{dv_a}{dt} = 0$ are crucial. The logic signal definition of section 2.5 stipulates that a logic signal consists of a sequence of events and the logic signal in Fig. 3.1a is composed of the following sequence of events:

$(1, t_1)$ $(0, t_2)$ $(1, t_3)$ $(0, t_4)$ $(1, t_5)$ $(0, t_6)$ $(1, t_7)$ $(0, t_8)$

If we wish to reconstruct the approximated signal $v_a(t)$ we need the interpolation function. Because of the network hypothesis about time invariant loads, the interpolation function can be computed most of the time before logic simulation once the load is known. As discussed in section 1. ' is a severe limitation but it seems necessary to prevent the explicit use of Kirchhoff laws. In this case we assume that the interpolation function consists of one or two line segments characterized by V_L, V_H, t_{TLH} and t_{THL}. The transition times are defined between V_L and V_H. For close events (t_2-t_3, t_4-t_5, t_5-t_6, t_6-t_7), the interpolation is a straight line with a slope corresponding to the transition time. If the events are far apart (t_1-t_2, t_3-t_4, t_7-t_8, t_8-...), the interpolation between two events uses two line segments. The first line segment joins the first event to a logic level (V_H or V_L) with a slope corresponding to t_{TLH} or t_{THL}. Then the second line segment is horizontal and joins this point to the second event.

The line segment approximation and therefore the interpolation function used in the analog signal reconstruction correspond to the approximation generally used by designers [43-45]. The use of simple transition times and voltage levels instead of a more complex approximation is necessary to keep the definition as simple as possible and to help standardize the event definition. Master events are designed to describe the analog signal, therefore there is no problem in describing the signal of Fig. 1.6. The sequence of events would be described using 2-valued logic as follow:

(1, 0) (0, 1.2) (1, 1.7) (0, 5) (7, 1) (0, 7.5)

Undefined events (U)

Even if the basic events are sufficient to model a clean switching signal, the construction of a complete logic model will require undefined events in order to handle an analog signal of arbitrary shape. An undefined event indicates the beginning of an envelope during which the signal can take any value as shown in Fig. 3.2a.

An undefined event is followed by either a 0 or a 1 event, indicating the end of the envelope. Since the boundaries of the envelopes are basic events, the predicates for defining the undefined events are the same.

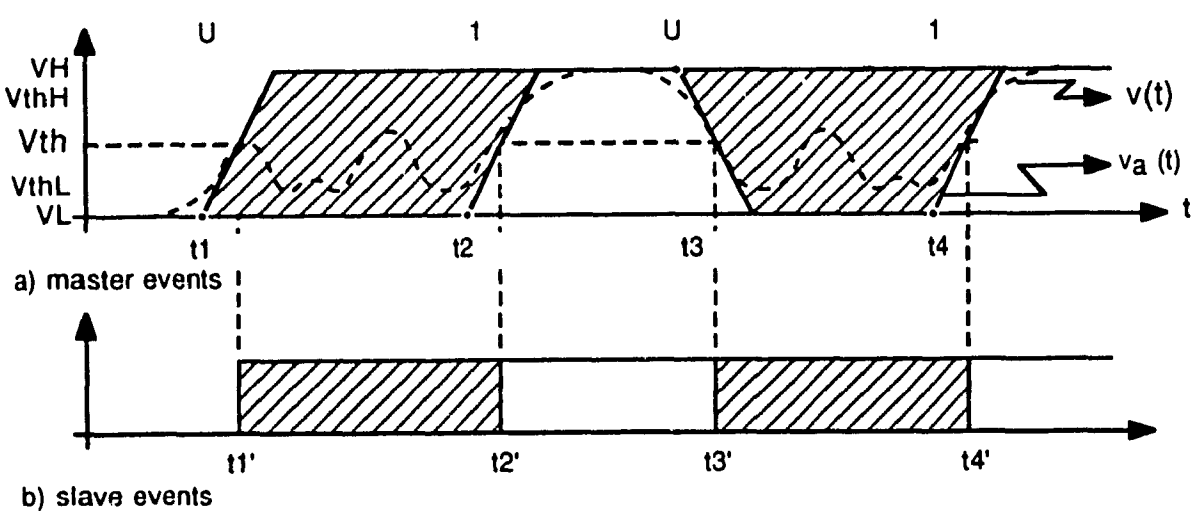


Fig. 3.2 Undefined events (U)

High impedance events (Z)

The high impedance state allows more than one output to drive a line at different time. This is frequently used to time-multiplex many signals on a single wire or a group of wires called a bus. During simulation we must verify that only one device drives the line at any time. A special event is required for this purpose. An open event (Z) occurs when an output starts to behave like an open circuit. The exact time is usually independent of the characteristic of the wire and depends on the driver. This is why most manufacturers[51-53] define the occurrence of the open event as the time at which the signal changes by 10%, when pulled high or low. This translates into the following predicate:

Z event: [v(t)=0.5] {In a pull-up test circuit}

or [v(t)=4.5] {In a pull-down test circuit}

We propose a more general definition similar to the definition used for basic events:

Z event: $[(v_a(t)=V_L^+)$ and $(\dot{v}_a(t)>0)]$ {In a pull-up test circuit}

or $[(v_a(t)=V_H^-)$ and $(\dot{v}_a(t)<0)]$ {In a pull-down test circuit}

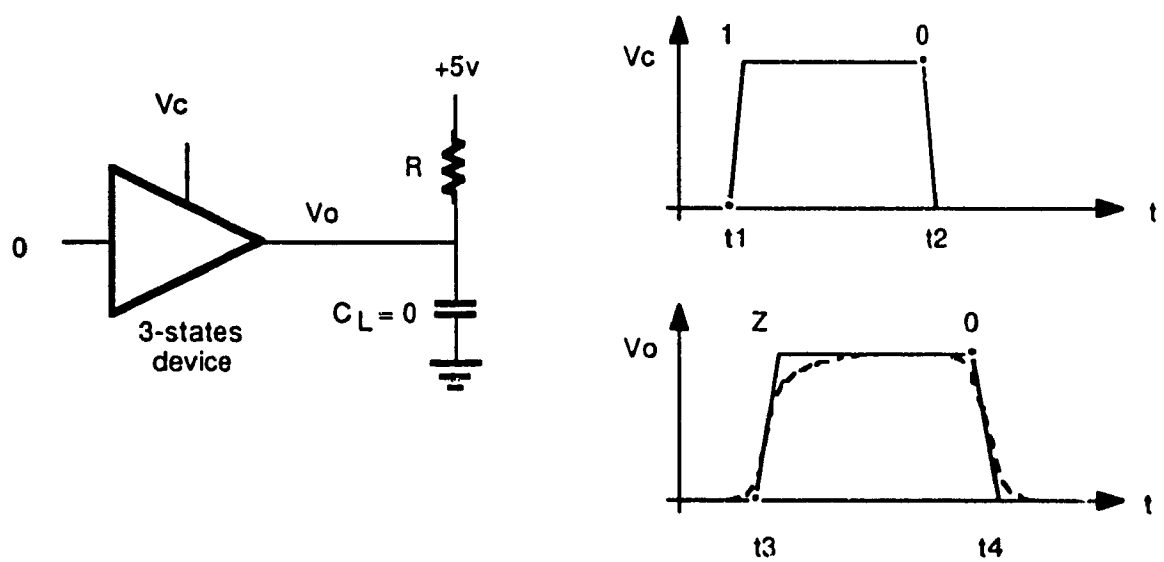


Fig. 3.3 High impedance events (Z)

The test circuit in Fig. 3.3 is similar to the circuits used by most manufacturers and will not be discussed further. We simply suggest that the corner of the line segment approximation of the signal should be used instead of the voltage thresholds of 0.5v. and 4.5v. The corner is technology independent and indicates more precisely when the output changes to high impedance. The difference between these two definitions is illustrated in Fig. 3.4.

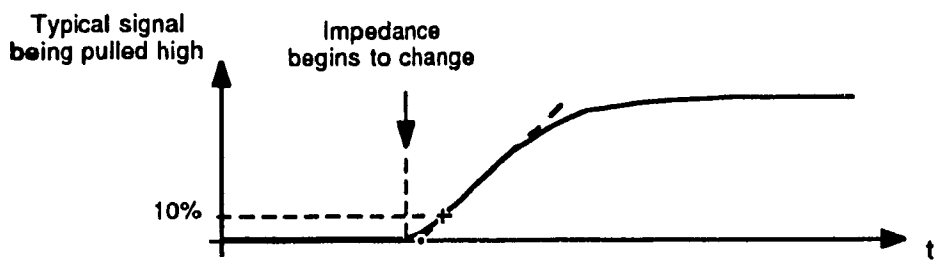


Fig. 3.4 Comparing manufacturer specification (+) with the proposed event definition (•) for for high impedance event (Z)

Fig. 3.5a illustrates a signal driven by three different devices. The sequence of master events is: $(1, t_1)$ $(0, t_2)$ (Z, t_3) $(0, t_4)$ (U, t_5) (Z, t_6) $(1, t_7)$ $(0, t_8)$

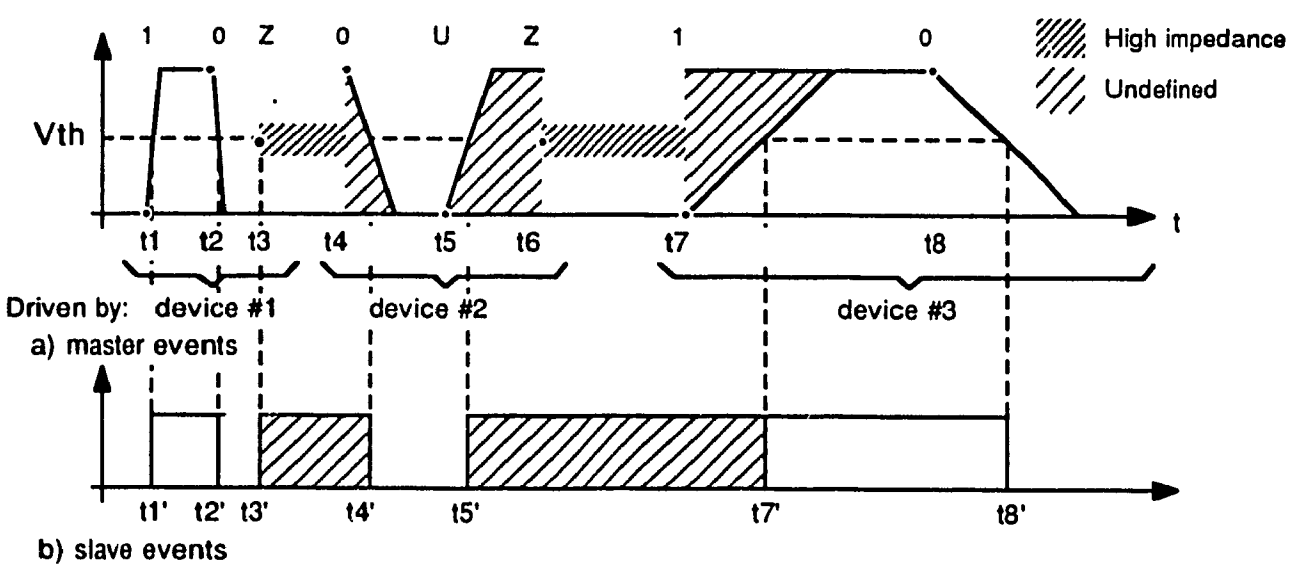


Fig. 3.5 Timed-multiplexed signal

During a simulation, driving conflict would be detected by directly comparing the occurrences of events, for example t_3 with t_4 and t_6 with t_7. Detecting bus conflict using signal based events should be no more difficult than using threshold based events.

3.1.2 Slave input events

Logic devices are threshold sensitive by hypothesis and by design and the input slave events are used to accurately model the event arrivals at each input port. Both input and output events are logically identical, their logical values in a Boolean equation are the same.

Fig. 3.6 illustrates how master events are associated with output ports and slave events with input ports. During simulation, the device U0 produces the master

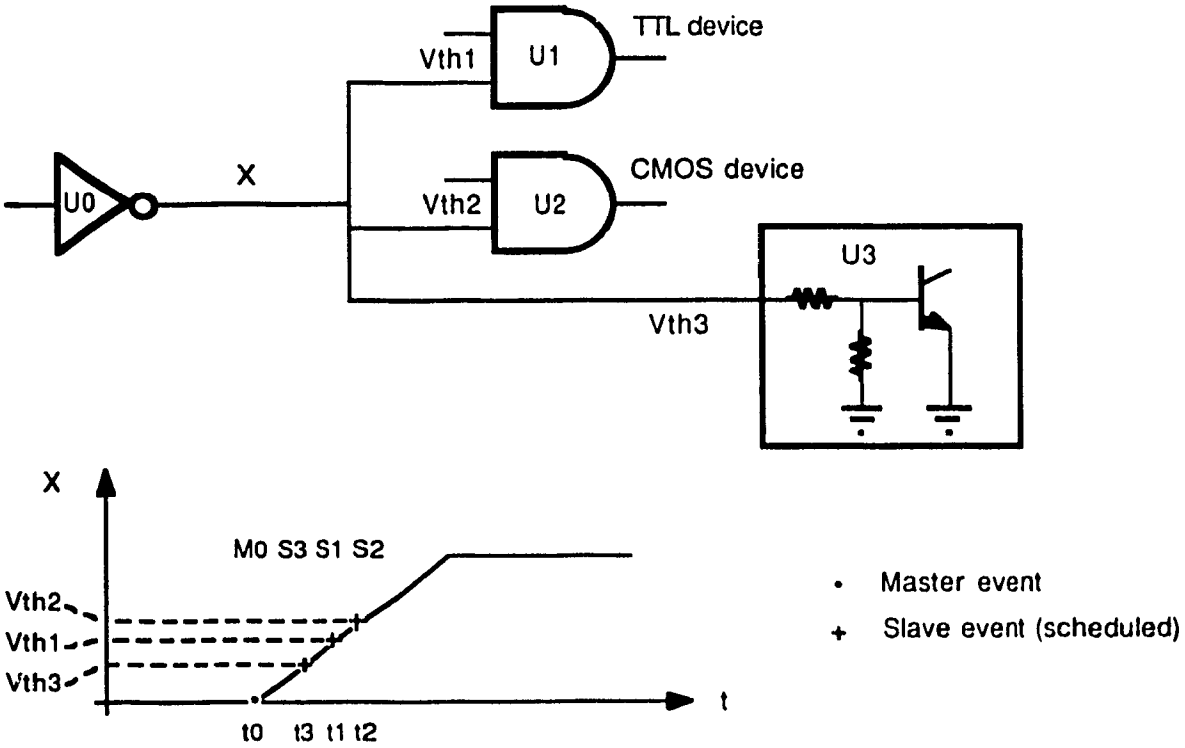


Fig. 3.6 Master and slave events

event M0 and the slaves events S1, S2 and S3 are computed using the logic voltages and the transition times of the signal and the input thresholds. The master events are used by the simulator to compute the slave events and the simulator schedules the simulation of each receiving device based on the slave events. Each device will be simulated at the correct time, U3 first, then U1 and U2. Therefore, the problems illustrated in Fig. 1.5 are automatically taken care of. The transmission of information problem associated with inertial delay models is also teken care of. The smaller pulse in Fig. 1.7 (a) will be seen by any devices with a small threshold and will not be seen by the devices with a large threshold; therefore information is properly transmitted from the output port to the input ports.

The master-slave event concept might be interpreted as a mere extension of threshold based events using transition times and we agree that some simulators have obtained similar results. We assert that it is more than an extension, the transmission of information between devices must be modeled separately from the logic device and the use of analog signal is essential to model fine timing details even at logic level. The definitions for the slave events are:

Basic events (0 and 1)

The basic slave events are the classic textbook events defined as the crossing of a threshold. More formally, they are defined as follow:

1 event : $[(v(t)=V_{Th})$ and $(\dot{v}(t)>0)]$

0 event : $[(v(t)=V_{Th})$ and $(\dot{v}(t)<0)]$

where: $\dot{v}(t)$ is the first derivate of $v(t)$.

The signal in Fig. 3.1b shows the sequence of slave events corresponding to the top signal.

Undefined events (U)

The undefined slave events (U) are computed like the basic slave events since the undefined event envelope boundaries are basic events. The signal in Fig. 3.2b shows the sequence of slave events corresponding to the top signal.

High impedance events (Z)

High impedance slave events (Z) are logically identical to undefined events and are converted at the input port into undefined events. There is no high impedance slave event. This is reasonable since the signal can take any value as soon as it becomes high impedance. The occurrence of the slave undefined event is equal to the corresponding master high impedance event. The signal in Fig. 3.5b shows the sequence of slave events corresponding to the top signal. The sequence of slave events is: $(1, t_1')$ $(0, t_2')$ (U, t_3) $(0, t_4')$ (U, t_5') $(1, t_7')$ $(0, t_8')$. The undefined event at t_6 was removed since it follows an undefined event.

3.2 Delay model

The sole purpose of a delay model is to delay a sequence of logic events in a manner that corresponds to the actual behavior of the logic device. The delay model is a one input and one output model. According to the conceptual model of section 1.7.2, the analog circuit is partitioned such that the propagation delays are associated with the output ports. Changes of state are simultaneous with the input events and the delay model describes the delay between any change of state and a given output port.

Many types of propagation delay models are being used: constant delay (t_p), inertial delay [29-31, 47-50], transport delay [31, 48, 49] and spike delay [29]. Even if these models work in practice, in a sense they do not model propagation delay since some events are removed and not delayed. In addition to the propagation delay model, transmission time and part of the logic state and the logic function are combined into these adhoc models.

The question is therefore: what are the characteristics of an adequate propagation delay model? In simple term, the propagation delay model should add a delay to each event of the input logic signal without removing any and without changing the order. In the proposed model, the removal of events or their replacement by undefined events is considered a logic function and is not part of the delay model. Theoretically, this means that the input sequence of events must be continuously mapped into an output sequence of events. Said otherwise, continuity must be preserved in the delay model.

It was demonstrated in section 1.5.3 that the constant delay model (t_P) preserves continuity and is therefore an adequate delay model. The problem begins when $t_{PLH} \neq t_{PHL}$. A popular solution is to use an inertial delay model, but it was also demonstrated in section 1.6.2 that the inertial delay model and other similar models do not preserve continuity. We shall now describe a propagation delay model where $t_{PLH} \neq t_{PHL}$ which preserve continuity.

The proposed delay model is illustrated in Fig. 3.7. It uses two parameters, t_{PLH} and t_{PHL} which are the usual propagation delays for separate logic events. The actual propagation delay used during simulation will be computed using t_{PLH} and t_{PHL} corrected according to Fig. 3.7 if the events are close. If events are far apart, the user's specified values t_{PLH} and t_{PHL} are used. If the separation with the preceding event is smaller than t_B (Equ. 3.1), a correction is computed as follow. The largest propagation delay is not corrected (Equ. 3.2) and the smallest propagation delay is augmented according to Equ. 3.3. Notice that t_{PLH} and t_{PHL} are reversed if t_{PHL} is larger.

$$t_B = (t_{PLH} - t_{PHL}) / K \qquad \text{Equ. 3.1}$$

$$t_{PLHs} = t_{PLH} \qquad \text{Equ. 3.2}$$

$$t_{PHLs} = t_{PLH} - K\,(\text{CurrentTime} - \text{LastEventTime}) \qquad \text{Equ. 3.3}$$

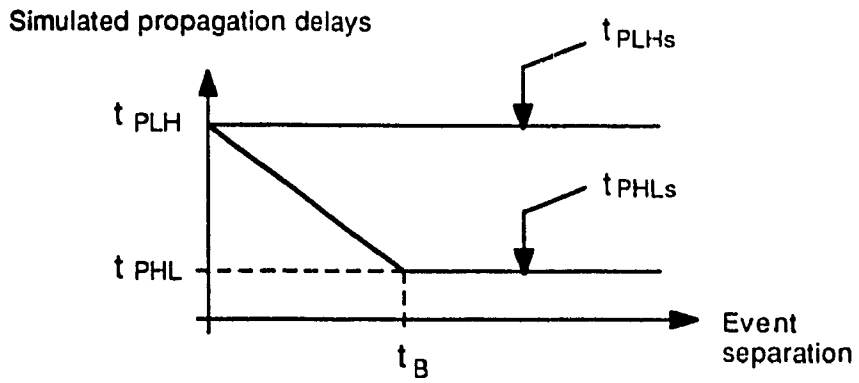


Fig. 3.7 Continuity-preserving delay model

To maintain the chronological order, K must be smaller than 1. Fig. 3.8 shows an example of a buffer using this delay model with K=0.5. Table 3.1 summarizes the computations. Notice that no events are removed and that the chronological order is preserved. This example should be sufficient to demonstrate that the proposed delay model is a continuous mapping of the input sequence of events and therefore is an adequate propagation delay model. Other models could be developed to more closely match the actual behavior of logic devices, but this was not investigated. The model proposed here should be adequate to guide in the development of other models.

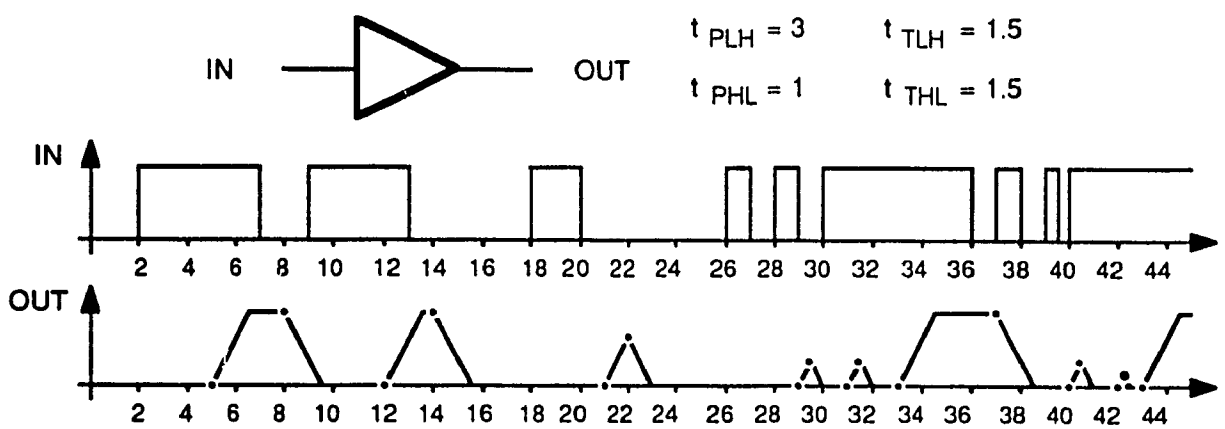


Fig. 3.8 Example of a continuity-preserving delay model

Input event		Separation	Computed	Computed	Output event
value	time	time	t_{PLH}	t_{PHL}	time
1	2	-	3		5
0	7	5		1	8
1	9	2	3		12
0	13	4		1	14
1	18	5	3		21
0	20	2		2	22
1	26	6	3		29
0	27	1		2.5	29.5
1	28	1	3		31
0	29	1		2.5	31.5
1	30	1	3		33
0	36	6		1	37
1	37	1	3		40
0	38	1		2.5	40.5
1	39	1	3		42
0	39.5	0.5		2.75	42.25
1	40	0.5	3		43

Table 3.1 Propagation delay computations

As described in section 3.1, a logic signal might also include undefined events (U) and high impedance events (Z). Undefined events use basic events (0 and 1) to define envelopes, therefore the same delay model can be used. Fig. 3.9 shows a propagation delay model for an output signal that includes high impedance events. As for the delay model in Fig. 3.7, if events are separate, the user's defined value are

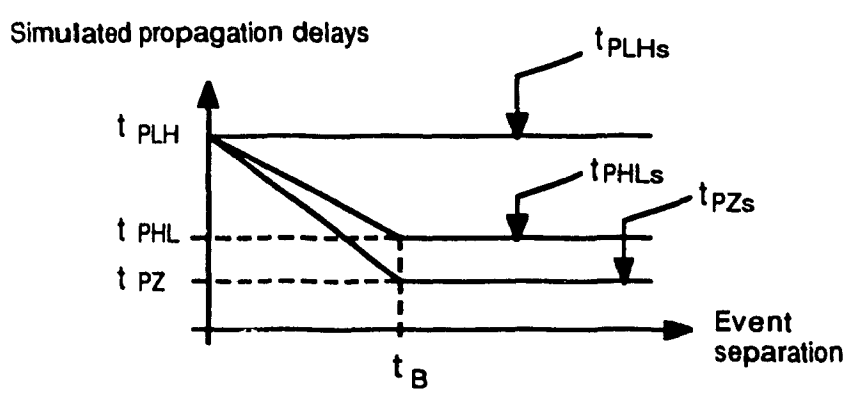


Fig. 3.9 Continuity-preserving delay model

used (t_{PLH}, t_{PHL} and t_{PZ}) and as the separation between events gets smaller, a correction similar to Eqn. 3.3 is computed.

The remaining question is: how do we model the physical phenomenon associated with inertial delay? This phenomenon is modeled either using a logic state and a logic function if it is due to an internal capacitance or with the master-slave event mechanism if it is due to external wire capacitance.

3.3 Simulation algorithms

The logic models discussed here will be primarily used within a discrete event simulator. The formalism for discrete event simulation was described in section 2.7. In this section, the simulation algorithms will be reviewed in light of the new master slave event concept. As discussed earlier, event driven simulation and algorithms for it are based on causality, and preserving timing relationships is one of our main objectives. In an event driven simulator, events are separated into *past events* and *predicted events*. The simulation begins by selecting the next event from the predicted events and simulating all devices that receive this event. The simulation of a device consists in computing the new component state and the predicted output events based on the current state and the current event, as illustrated in Fig. 3.10.

The current event is then moved in the past event list and the simulation continues with the next event. If causality is preserved, the current event will only affect predicted events. Otherwise past events might be changed which would require backtracking the simulation in time up to the last unmodified event. Backtracking is expensive since it requires restoring the state of all components, and it is best to avoid it.

Data structures like the event list and algorithms used in the simulator are well known [55] and will not be described. In the prototype simulator, a simple time ordered list of events was used. More efficient algorithms could be used, but our

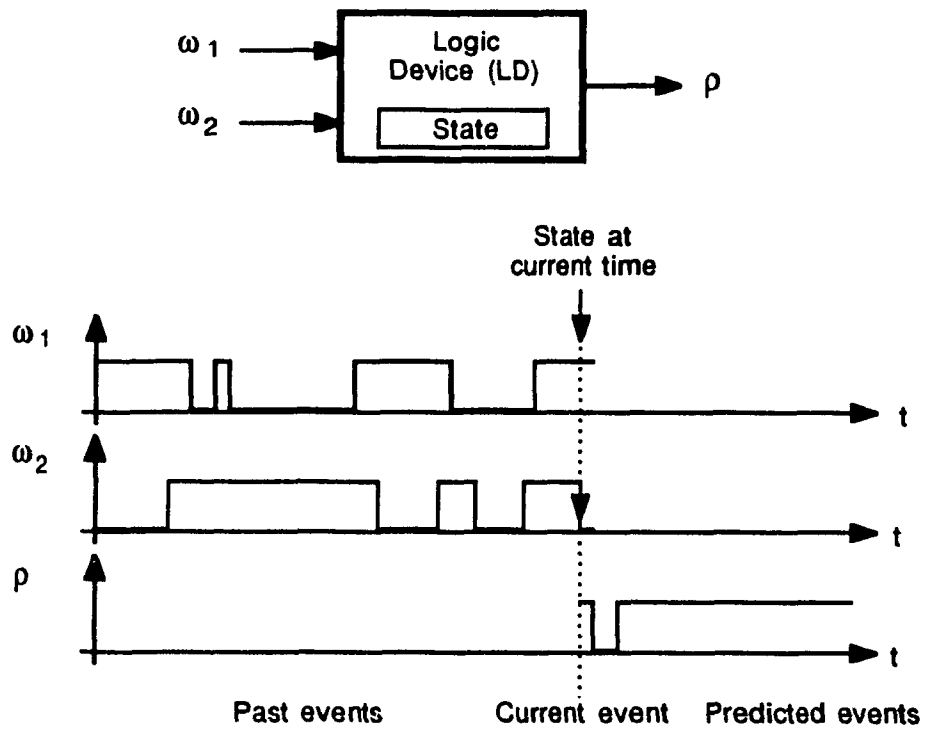


Fig. 3.10 Typical input and output signals

objective was to demonstrate that the proposed model is working, not to improve simulation efficiency. Therefore, only the changes in the simulation algorithms due to the master slave event concept will be presented. The most significant change makes the simulator event list a list of slave events. The discrete event simulator processes the event list as follows:

```
algorithm simulate(SimulationTime)
  begin
    while CurrentTime<SimulationTime
      CurrentEvent=NextEvent(SimulatorEventList);
      Set(CurrentTime);
         {Simulate the device associated
               with the current slave event}
      case DeviceType of
        ...
        ...
        gate:  (The device is a gate)
          begin
            Depending on the CurrentEvent, the port on which
              it occurs, the separation time with previous
```

```
                events and the device state do
                  begin
                    PerformChangeOfState;
                    GenerateOutputEvents;
                  end
            end
          ...
          ...
      end (while)
    end (simulate)
```

The procedure `GenerateOutputEvents` will produce the output events if necessary and will call the following procedure to insert the master event into the desired signal, compute the slave events and insert them into the simulator event list:

```
algorithm Insert(MasterEvent,OutputSignal)
  begin
    Insert the MasterEvent in the list of master events of
      the OutputSignal.
    for all input ports connected to the OutputSignal do
      begin
        Compute(SlaveEvent);
        Insert(SlaveEvent,SimulatorEventList);
      end
  end (Insert)
```

The procedure `Insert(SlaveEvent,SimulatorEventList)` is used to insert the new event at the appropriate place in the list so the function `NextEvent(SimulatorEventList)` can easily obtain the next event to simulate. *Except for the use of slave events, these algorithms are similar to any event driven simulator algorithms and will not be discussed further.*

3.4 Timing specification using master slave events

Even though logic device modeling is the subject of the next chapter, it is appropriate now to discuss the timing specification of simple devices to illustrate how the concept of master slave event can be applied to commercial devices. The specification of a logic device consists of input ports, output ports, states, state transition function, output function and timing specification. The timing specification includes timing constraints and propagation delays. In this section, we will give

examples of timing constraints and propagation delays based on the proposed event set. Discrete CMOS gates and flip-flops from Motorola [44] will be used because their published specifications are reasonably complete. This should demonstrate that the master-slave event concept is practical and in effect leads to simpler device specifications.

3.4.1 Hardware linker and Electrical Rule Checking (ERC)

In this thesis, we have made a fundamental assumption about time invariant loads. As it will be shown with examples in this section, propagation delays are not load independent but once the load is known, they become time invariant. The process of computing the timing specifications for a given load is called *hardware linking*. Hardware linking is a simple technique used by designers and well documented in data books [43-45]. This process is straightforward and the logic model examples given in the following sections will illustrated it. While linking logic elements, such constraints as current drive capability and limits on transition times can also be verified. Hardware linking is also refered to as Electrical Rule Checking (ERC).

3.4.2 Timing specification of a gate

The timing specification of the CMOS 2-input nand gate (MC14011B) consists of propagation delays and transition times. The following specifications and typical values have been extracted from the data sheet. C_L is the load capacitance.

$t_{TLHm} = t_{THLm} = (1.35 \text{ nsec/pF})\ C_L + 33 \text{ nsec}$

Typ: 100 nsec

$t_{PLHm} = t_{PHLm} = (0.90 \text{ nsec/pF})\ C_L + 80 \text{ nsec}$

Typ: 125 nsec

Motorola specifies timing constraints and propagation delays between 50% points and transition times between 10% and 90% points. Assuming that thresholds are effectively at 50%, simple arithmetic can be used to translate the manufacturer's

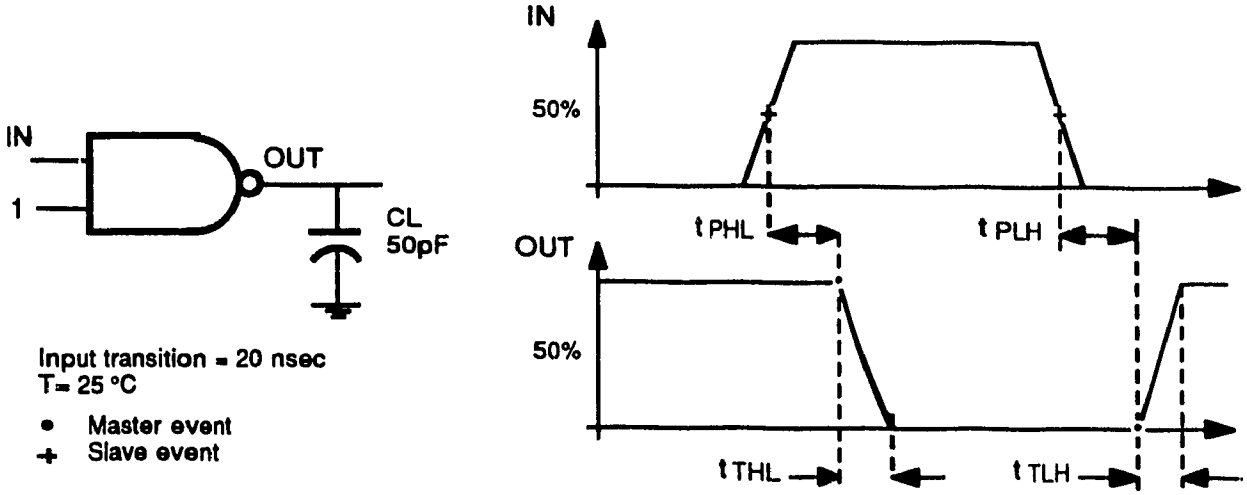


Fig. 3.11 Timing specification of the MC14011B

specifications into signal based event specifications. The resulting definitions and test conditions are given in Fig. 3.11 and the new timing specifications are:

$$t_{TLH} = t_{THL} = \frac{t_{TLHm}}{0.8} = (1.69 \text{ nsec/pF})\, C_L + 41 \text{ nsec}$$

Typ: 125 nsec

$$t_{PLH} = t_{PHL} = t_{PLHm} - \frac{1}{2}\,\frac{t_{TLHm}}{0.8} = (0.05 \text{ nsec/pF})\, C_L + 47 \text{ nsec}$$

Typ: 50 nsec

Because the propagation delay for signal based events is measured at the beginning of the transition time instead of at 50%, the dependence on load capacitance is almost reduced to zero (0.05 nsec/pF). Using master-slave events is therefore an improvement.

3.4.3 Timing specification of a flip-flop

The timing specification of the CMOS D flip-flop (MC14013B) consists of propagation delays, transition times, setup time, hold time, clock high time and clock low time. The following specifications and typical values have been extracted from the data sheet:

$t_{TLHm} = (3.0 \text{ nsec/pF})\, C_L + 30 \text{ nsec}$

Typ: 180 nsec

t_{THLm} = (1.5 nsec/pF) C_L + 25 nsec

Typ: 100 nsec

$t_{PLHm} = t_{PHLm}$ = (1.7 nsec/pF) C_L + 90 nsec

Typ: 175 nsec

t_{SUm} = 20 nsec

t_{HDm} = 20 nsec

t_{Hm} = 125 nsec

t_{Lm} = 125 nsec

The resulting definitions and test conditions are given in Fig. 3.12 and the new timing specifications are:

$$t_{TLH} = \frac{t_{TLHm}}{0.8} = (3.75 \text{ nsec/pF})\ C_L + 37.5 \text{ nsec}$$

Typ: 180 nsec

$$t_{THL} = \frac{t_{THLm}}{0.8} = (1.88 \text{ nsec/pF})\ C_L + 31.3 \text{ nsec}$$

Typ: 100 nsec

$$t_{PLH} = t_{PHL} = t_{PLHm} - \frac{1}{2}\frac{1}{0.8}\frac{t_{TLHm}+t_{THLm}}{2} = (0.3 \text{ nsec/pF})\ C_L + 73 \text{ nsec}$$

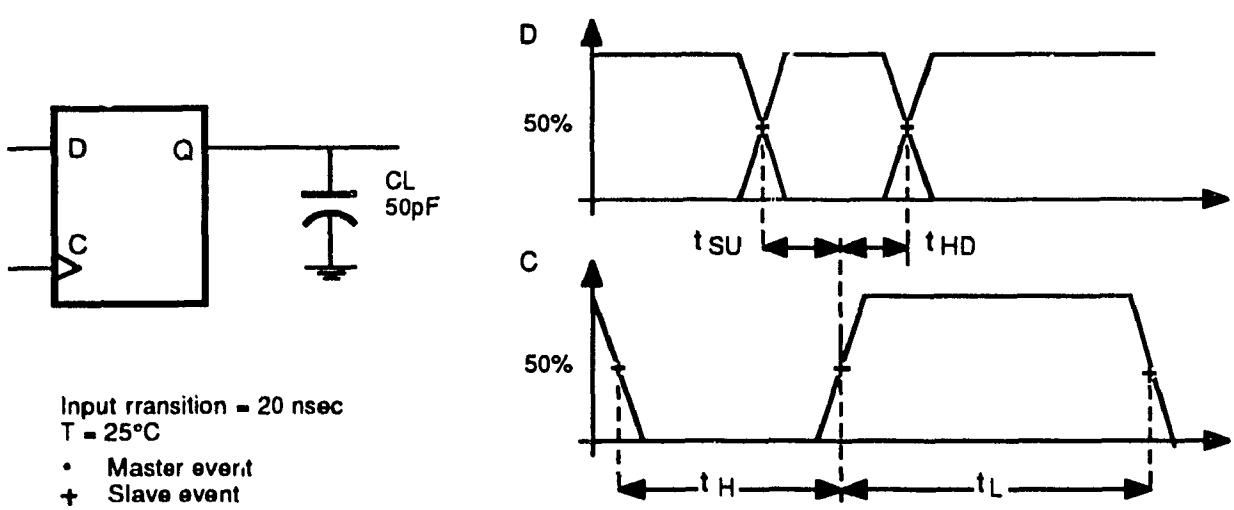


Fig. 3.12 Timing specification of the MC14013B

Typ: 175 nsec

$t_{SU} = t_{SUm} = 20$ nsec

$t_{HD} = t_{HDm} = 20$ nsec

$t_H = t_{Hm} = 125$ nsec

$t_L = t_{Lm} = 125$ nsec

Again, the propagation delay dependence of the load is greatly reduced. The timing constraints t_{SU}, t_{HD}, t_H and t_L are defined between input slave events as required for signal based events.

3.4.4 Timing specification of a 3-state buffer

The timing specification of the CMOS 3-state buffer (MC14503B) consists of a propagation delay and a transition time for the data input and propagation delays for the control input. The following specifications and typical values have been extracted from the data sheet:

$t_{TLHm} = t_{THLm} = (0.5 \text{ nsec/pF})\ C_L + 20$ nsec

Typ: 45 nsec

$t_{PLHm} = t_{PHLm} = (0.3 \text{ nsec/pF})\ C_L + 50$ nsec

Typ: 75 nsec

$t_{PHZm} = 75$ nsec

$t_{PLZm} = 80$ nsec

$t_{PZHm} = t_{PZLm} = 45$ nsec

The resulting definitions and test conditions are given on Fig. 3.13 and the new timing specifications are:

$$t_{TLH} = t_{THL} = \frac{t_{TLHm}}{0.8} = (0.63 \text{ nsec/pF})\ C_L + 25 \text{ nsec}$$

Typ: 56 nsec

$$t_{PLH} = t_{PHL} = t_{PLHm} - \frac{1}{2}\frac{t_{TLHm}}{0.8} = (-0.01 \text{ nsec/pF})\ C_L + 38 \text{ nsec}$$

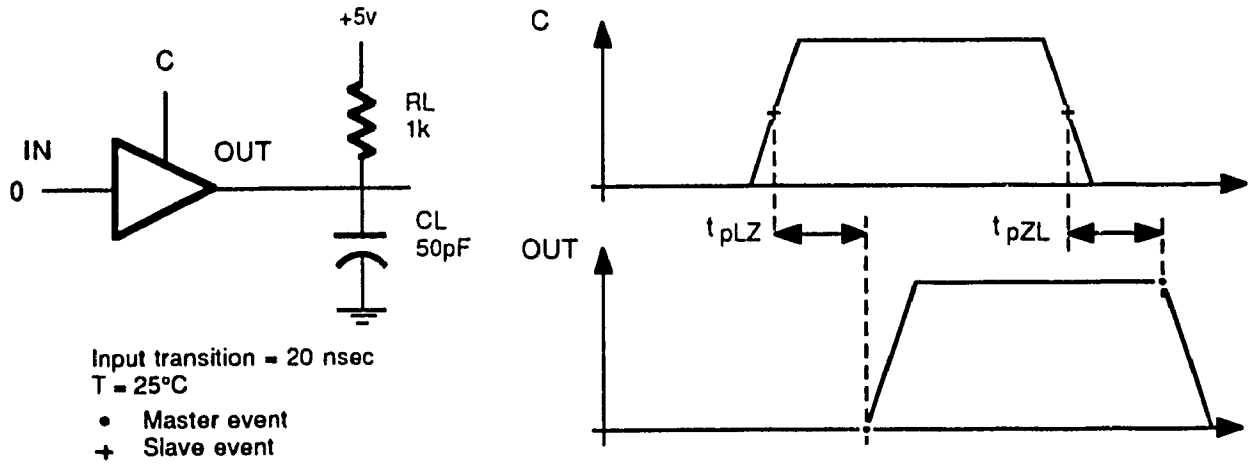


Fig. 3.13 Timing specification of the MC14503B

Typ: 38 nsec

$$t_{PHZ} = t_{PHZm} - \frac{0.1}{0.8} t_{rLHm} \approx 68 \text{ nsec}$$

$$t_{PLZ} = t_{PLZm} - \frac{0.1}{0.8} t_{THLm} \approx 73 \text{ nsec}$$

$$t_{PZH} = t_{PZL} = t_{PZHm} = 45 \text{ nsec}$$

Here, the propagation delay is virtually independent of the load. The definitions for high impedance events are slightly different with the proposed definition, and 1/8 of the rise time should be removed to compensate for the time it takes to reach 0.5v or 4.5v.

3.5 Conclusion

In this chapter, a time model was developed for logic devices. The basis of the model is the new master-slave event definition. This event definition provides a simple and natural means of describing the logic signals since it is related to the analog signal. For instance, the event definition is easily applied to commercial devices. One key aspect is to separate the input events from the output events leading to two time models. The master-slave event concept models the

logic devices and a new propagation delay model is proposed to describe the processing time within the device. This new propagation delay model preserves continuity and does not remove any events nor changes the sequence of events being delayed. We claim that the sole purpose of a propagation delay model should be to delay events. In this sense, inertial delay models and similar models are not true propagation delay models.

This carefully formulated time model should be useful to anyone working on the specification, design and verification of digital systems. The 30 year old fashion of specifying logic events using threshold crossing is not adequate for modern CAD systems and the proposed event definition could replace it. Although the capability to reconstruct analog signals and the use of individual thresholds are more complicated to process, they provide a way to neatly interconnect any logic families with analog devices. This is an important advantage in modern CAD systems and in a rapidly changing technology. If widely accepted and used, the proposed event definition would improve the compatibility of the specification of logic devices and systems between the designers and the various simulators and CAD tools.

Chapter 4

State model: Continuous Time Automaton

As mentioned by M. R. Lightner [33] some time ago, important theoretical work was still required to formalize the hypothesis and axioms of the circuit to logic abstraction. About logic signals he says: "*The transition from continuous signals to discrete signals* is obviously of key importance in producing a viable model. It is surprising that such a small amount of work has been done in examining this transition." About digital elements he continues: "Although many people have constructed discrete models, there is very little work that has been done in providing *a formal framework in which to construct discrete models.*" And finally about interconnection models, he states: "To the best of the author's knowledge, no one has carried out *a systematic study of alternate impedance assumptions* to develop a theoretical understanding of the problem. ... In summary, for the simplest case the interconnection of discrete elements is well understood. However, for the more complex cases that arise from modern technology careful theoretical development of solutions has not taken place and reasonable, but ad hoc, approaches are presently being used."

We believe the approach presented in the previous chapters provides an answer to these remarks. The proposed formal framework clearly describes the abstraction of analog circuits into logic devices and logic models, the relation between analog signals and logic signals and the impedance assumptions that characterize the interconnection of logic devices. The next step and the object of this chapter is to study the abstraction of continuous change of state into a timed state sequence.

4.1 Introduction

The construction rules leading to the conceptual model of Fig. 1.10 allow us to separate the modeling of time from the processing of the timing constraints and the logic function. In chapter 3, we have established how time is modeled, the master-slave events concept models the transmission time between devices while the propagation delay models the processing time. *The remaining problem is to determine how the logic function and the timing constraints should be used to compute the next state and the output events.*

The premise: zero processing time

Since the logic model must meet the state model rule, simultaneous input events generate simultaneous changes of state and also simultaneous changes on the zero delay outputs. The zero delay outputs are the modeled output before the propagation delay is applied (see Fig. 1.10). Therefore the state model we are looking for is punctual in time and does not take any simulation time. This is represented by a dot (•) in Fig. 1.10.

The problem: modeling the device state

In order to predict accurately the behavior of a device, each device model in the simulator must remember everything relevant that has happened. In logic system this means remembering the effect of past events. In Fig 4.1, remembering whatever significant has happened for the signal A is quite easy as nothing happened. For signals B and C, it would be advisable to have a few state variables in the device logic model to remember the necessary past. But for signal D, it might be impossible or impractical to remember the necessary information. This is the problem associated with continuous change of state.

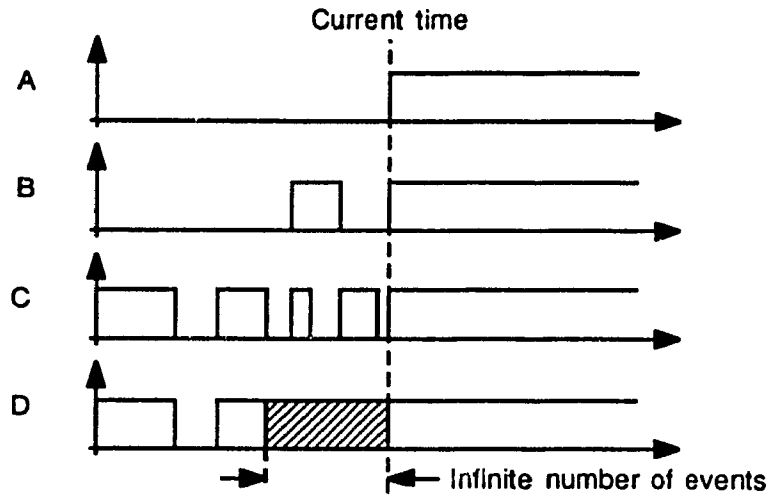


Fig. 4.1 Examples of input signals in event driven simulation

An infinite number of input events in a small interval of time could drive the device into an infinite number of changes of state. Even if an analog model with a theoretically infinite number of changes of state could be used, a logic device as a discrete device is more efficiently modeled using a timed state sequence.* The problem is to relate the timed state sequence with the continuous sequence of input events.

The answer is simple and has been used by designers for decades: add timing constraints to restrict the number of input sequences of events to those that the device has been designed to process. For all other subsequences, the state and the outputs of the device are simply assumed to be undefined. *Basically, the set of all sequences of input events on a finite interval, is divided into a subset of sequences having a finite number of events in that interval leading to normal operation and a subset of sequences that might have an infinite number of events leading to abnormal operation. Associating all abnormal sequences with an undefined state is the key to*

---

* Refer to the definitions given in section 1.2.

generating a timed state sequence. The infinite number of events is defined with respect with the highest speed of operation of the device. For example, a sequence of input events changing at 100Mhz and driving a device designed to operate at 10Mhz is considered an infinite number of events on a finite interval. During the simulation time interval, the finite number of events will drive the device state through a finite number of state changes, called a timed state sequence, and the infinite number of events will also drive the device into a finite number of state changes since the device state will remain undefined. Therefore a finite number of states is sufficient and a finite state machine can be used to transform continuous change of state into a timed state sequence.

Timing violation , undefined state and undefined events

Input sequences of events are classified as acceptable or unacceptable depending on whether they satisfy timing constraints or not. A timing violation or a safety violation is defined as a timing condition on the sequence of input events that leads to undefined or unsafe behavior, for example not meeting set-up time. *Since the undefined state is required to obtain a timed state sequence, it is therefore an essential part of all logic models and processing of undefined events is mandatory.*

Objective

In summary, the objective is to develop a systematic mechanism:

- to process the continuous sequence of logic input events,
- to transform the continuous change of state into a timed state sequence,
- to integrate the processing of timing constraints with the logic function,
- to process undefined states and undefined events and
- to produce zero delay output events.

This mechanism is called a Continuous Time Automaton (CTA).

4.2 Current logic modeling techniques

Before we begin our discussion about the construction of logic models, we will examine two approaches closest to CTA: the logic simulator SAMSON [30] and real time system model based on timed automata [55]. In chapter 1, we have reviewed some of the problems associated with current modeling techniques. We shall now study the basic principles causing these problems. These two representative approaches will be analyzed with respect to the following objectives:

- modeling the transmission of information between devices,
- modeling the processing time in the devices (propagation delay) and
- modeling the transformation of continuous change of state into a timed state sequence.

Logic simulator, SAMSON

SAMSON is one of a few logic simulators that was developed with some formalism. It uses both timing constraints on logic devices (like set-up time and hold time) and timing constraints on the logic signals (multiple thresholds describing glitches and hazards) to perform the discretization of time. Logic model construction is usually a top-down process in which fine timing details are added to the logic function. In SAMSON, a bottom-up approach where precise abstraction steps are applied to simplify the circuit model and to construct the logic model is used. Unfortunately, this bottom-up approach was not sufficient and resulted in event definitions, logic models and simulation algorithms similar to all other logic simulators [33]. To correct this problem ad hoc techniques such as inertial delay have been used. This has led to peculiar models such as the model of a gate in Fig. 4.2 [30].

This model does not follow a state space approach, the state is not well identified and the state transition and the output functions are not clearly defined. In

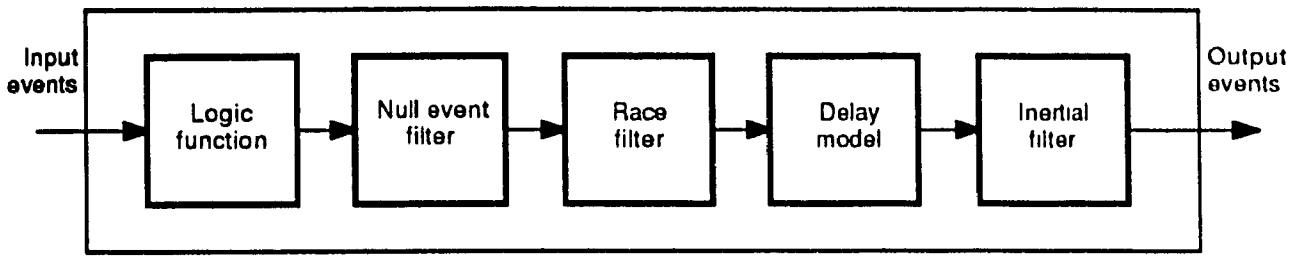


Fig. 4.2 Classical logic model for a gate

addition, the transition of continuous change of state into a timed state sequence is not performed since there is no undefined state. This model mixes the logic function, the delay model and the transmission of information. We believe a state space model where each aspect is separately modeled should be used.

Real time systems and Timed Automaton (TA)

A timed automaton is a state machine whose change of state is also controlled by time. This is accomplished using artificial signals called clocks which are automatically and simultaneously increased to simulate the passage of time. These clocks are reset and tested as required in the automaton to implement timing constraints and propagation delays. For example, on exiting a given state, the clock x is set to zero and in some other state the automaton waits until x is 20. TA are used to prove timing properties such as the safety of a control system and also to model logic devices.

Timed automata are designed to parse "timed languages". A timed language is a sequence of events where each event consists of a label taken from an alphabet and an associated real time value. A TA has an initial state and an accept state indicating that a particular instance of the timed language is accepted. More details on TA can be found in [55]. We will now describe how TA can be used in simulation.

The TA of a buffer is shown in Fig. 4.3. This TA was modified for simulation purpose. Initial and accept states have been removed since a simulation can start and end in any state. The TA for the buffer operates as follow. Suppose the initial state is S1. The buffer remains in state S1 until A is set, at which time the clock X is reset. While in state S2, the clock increases automatically and when it equals 3, B is set and the state changes to S3. The TA state changes from S3 to S4 then to S1 in a similar fashion if the input returns to zero. This TA models a buffer with t_{PLH}=3 and t_{PHL}=1.

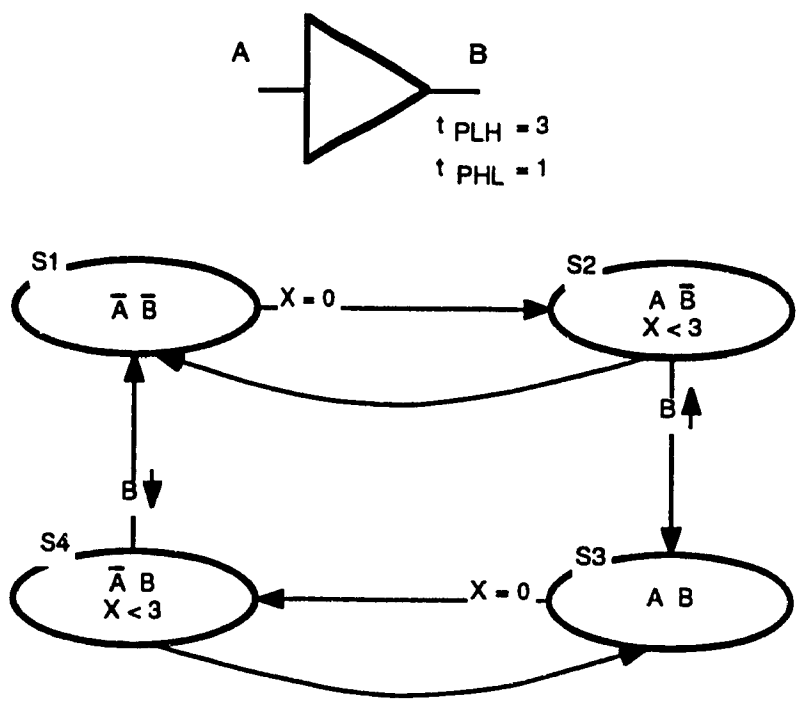


Fig. 4.3 Timed Automaton for a buffer

A TA is a general purpose tool and the user is left with the problem of designing the automaton using clocks to detect timing violation such as not meeting a set-up time and to model the propagation delays. The conceptual model of a TA is shown in Fig. 4.4. This model combines the logic function and the logic state with the processing of timing and propagation delays. This makes TA difficult to use if fine timing details need be modeled. Separating the modeling of time (propagation delay

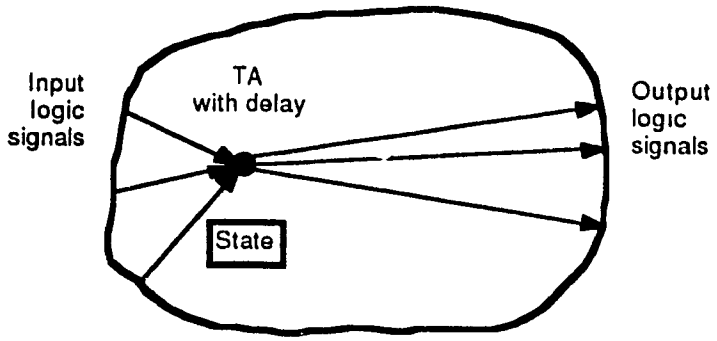


Fig. 4.4 Conceptual model of a Timed Automaton

and transmission time) from the behavior as per the conceptual model of Fig. 1.10 should make the construction of CTA for logic devices simpler. TAs are good at parsing timed languages, but are difficult to use to model logic devices because it assumes that all variables have been synchronized with a hypothetical high speed clock. Problems associated with continuity like simultaneity of events, metastability or coherence in the individual bits of a state variable* are not explicitly considered. Glitches and hazards associated with a continuous sequence of input events are not considered either.

For example, the TA for the buffer in Fig. 4.3 does not model undefined events, close events or timing violation. It only models the propagation delay and it is already complex for a simple buffer. A TA does not distinguish normal sequences from abnormal sequences, the user is left with the complicate task of doing everything simultaneously: process the continuous sequence of input events including undefined events, process the timing constraints using undefined states including error recovery, produce a timed state sequence, integrate the logic function and model the propagation

---

* See appendix A.

delay. Even if it might be possible to construct a TA for a buffer, we believe that a simpler mechanism will be more appropriate. The CTA is a simplified TA with strict construction rules.

4.3 Overview of a CTA

A CTA is an event driven state machine. Each event arriving at any input port of a given logic device is processed by the CTA. In a TA, the machine remains in the same state until the corresponding Boolean expression becomes false. The machine then changes to the state with a true Boolean expression. A change in a Boolean expression is initiated by an event and in CTA, the event terminology is preferred.

A typical CTA is shown in Fig. 4.5. The change of state is initiated by any event arriving at one of the device input ports. Then a Boolean expression (A, B, C or D) is used to decide on the next state and on the action to take (W, X, Y, Z).

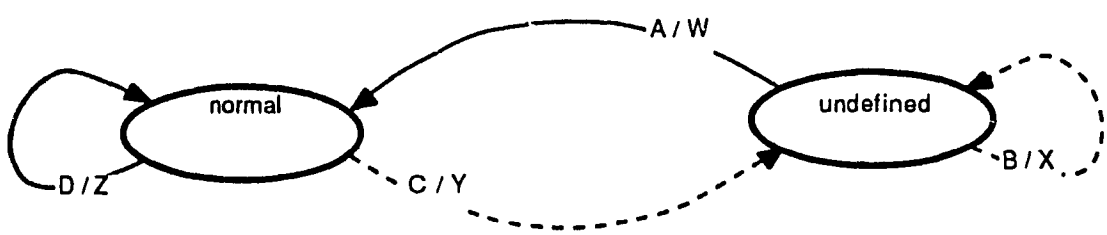


Fig. 4.5 Typical state transitions

A Continuous Time Automaton (CTA) differs from a TA in many respects:

- it does not process propagation delay, it only processes timing constraints,
- except for a wait statement, it does not use explicit clocks,
- it must systematically process undefined state and events,
- it must include an error recovery mechanism, an error state is the result of a timing violation.

- it does not include the logic function except as an argument and
- it must transform the continuous change of state into a timed state sequence.

A CTA is designed to work at a lower level than a TA.

4.4 Definitions

Fig. 4.6 is an expanded view of the conceptual model of Fig. 1.10. A CTA based model consists of input ports, output ports, a state Q, a logic function (state transition function and output function), device model arguments and simulator arguments. Here are some definitions associated with logic models:

- Port class: Ports are classified according to their function in the finite state machine. For example, a flip-flop has input ports, output ports and a clock. Events arriving on ports of the same class are identically processed by the CTA. Timing constraints and propagation delays are specified between classes of ports.
- State: The state includes the CTA state, the logic state, the predicted events and the time related state variables. The logic state is user defined and captures steady state effects. The time related state variables are built in the simulator to capture the recent past and include the occurrence of the last event for each class of events. Notice that the state variables are not necessarily discrete values (N); for example the occurrences of last events for each class of ports are continuous state variables ($\Re$).
- State transition and output functions: The state transition and output functions include a user defined logic function and the CTA itself. The state machine takes the timing constraints and the user defined logic function to decide on the next state. Other time related state variables, like the occurrence of the last event in each class, might also be updated.

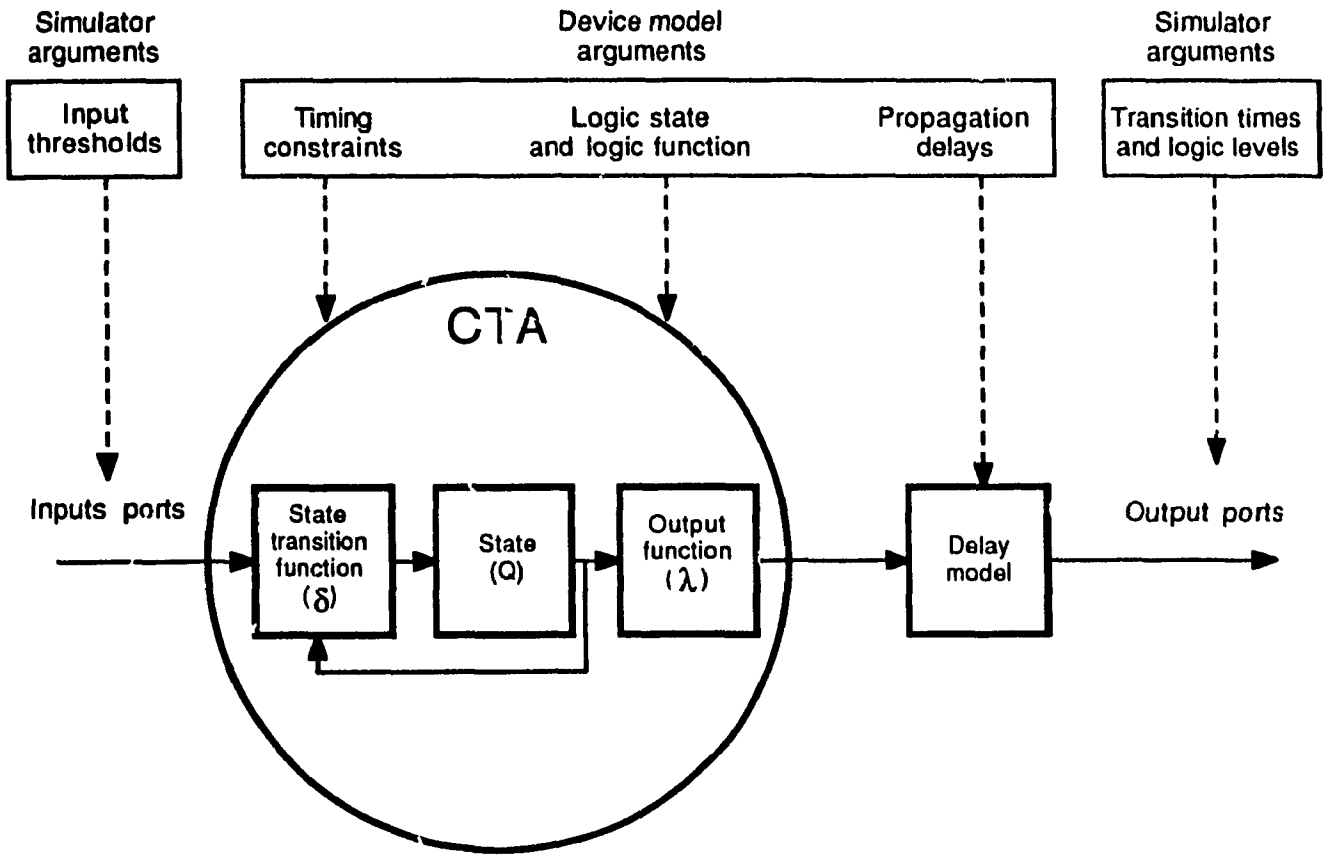


Fig. 4.6 CTA based logic model implementation

- Delay model: The delay model takes the propagation delay arguments and computes the occurrences of the output events. Because the change of state is a timed state sequence, the delay model is based on a finite set of arguments.
- Device arguments: The device model uses the timing constraints, the propagation delays, the logic states and the logic function which includes the state transition function and the output function as arguments. Timing constraints are associated with the state transition function and are specified between input slave events. The propagation delays are

associated with the delay model and are specified from input slave events to output master events.

- Simulator arguments: The logic simulator that manages the events takes the input thresholds, the output transition times and the output logic levels as arguments to model the transmission of information between logic devices. Simulation can be speeded up by using default values and tolerances can be used on each parameters for worst case simulation. The definition of these parameters is based on the line segment approximation of the analog signal shown in Fig. 3.1 and examples are given in section 3.4.

As shown in Fig. 4.5, a CTA is a directed graph made of states and state transitions. The terms associated with CTAs are defined in this section.

- CTA state: This is the basic state of a logic device or a discrete device. The CTA state represents the current condition as required to model the behavior. Each CTA state is given a specific name.
- CTA state transition: These are the arcs describing the possible changes of state. There are two types of state transitions. The first one is initiated by input events and the second by wait events. In the first type, the boolean expressions in Fig. 4.5 (A, B, C and D) have been divided in two expressions. The result of evaluating these expressions are anded to control the state transition. In the fist type of state transition, a three part label is attached to each arc (X:Y/Z). The first part (X) is the Boolean expression controlling the state transition. It might use the user defined logic function and is a function of:

  - the class of the port carrying the current event,
  - the current values at the input ports and
  - the current logic state.

  The second part (Y) is a Boolean expression describing a test on a timing

constraint and also controlling the state transition. For example, t_{SU} would indicate that the set-up timing constraint is met and $\overline{t_{SU}}$ would indicate a timing violation. The third part (Z) describes the function to perform with this state transition. It might describe a change of logic state or an output event.

- Wait state transition: This is the special state transition that controls the state duration from which the transition originates. Wait 10 would mean that the machine would enter the state and remain in this state for at most 10 units of time. There is at most one wait state transition for each state.
- State transition time: It corresponds to the time at which the state transition occurs. Since the change of state is simultaneous with the input event or the wait event, then the state transition time will be equal to the occurrence of the input event causing the change of state.
- State duration: The state duration is the time during which the machine is in a given CTA state. It is the difference between the state transition times of the last incoming state transition and of the next outgoing state transition.
- Constrained state transition: Given a current state resulting from an incoming state transition at t_1. A constrained state transition at t_2 is a state transition that causes the state duration $(t_2 - t_1)$ to be lower bounded (not zero).
- Unconstrained state transition: Given a current state resulting from an incoming state transition at t_1. A constrained state transition at t_2 is a state transition that might cause a zero state duration $(t_2 - t_1 = 0)$.
- Cycle: A cycle is any close path in the CTA. A cycle is described with a sequence of state transitions starting in a given state and ending in the same state.

- Constrained cycle: In a constrained cycle, the sequence of state transitions contains at least one constrained state transition.
- Unconstrained cycle: In an unconstrained cycle, the sequence of state transitions consists of unconstrained state transitions only.
- Constrained sequence of events or timed sequence of events: A constrained sequence of events is a sequence of events having a finite number of events in a finite interval of time. Finite should be interpreted with respect with the operation of the circuit being modeled, as a sequence of events leading to normal operation.
- Unconstrained sequence of events or continuous sequence of events: An unconstrained sequence of events is a sequence of events having an infinite number of events in a finite interval of time. Infinite should be interpreted with respect with the operation the circuit being modeled, as a large number of events leading to abnormal operation.

4.5 Examples of CTAs

In this section, the operation of CTA based models and most features of the CTA will be illustrated with examples. The next section will describe how to construct a complete CTA.

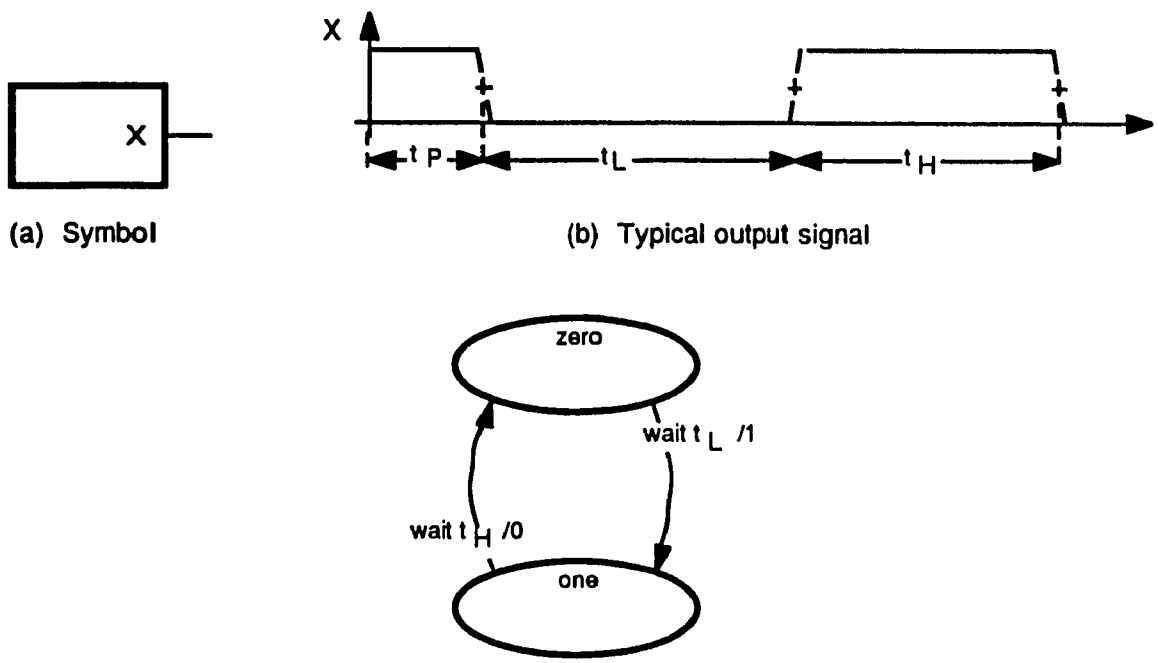
4.5.1 The wait state transition: a clock

This first example will illustrate the basic operation and the use of a wait state transition. As shown in Fig. 4.7a, a clock is a simple device with one output. The simulator uses a specification for each part. The language HDIL (Hardware Description and Integration Language) described in Appendix B is used and the specification of a typical clock would look like:

```
circuit CLK1Mh
   interface                (Note this is a comment)
      X:  bit (output  0.5n  0.5n  0   5)
```

```
                    (tTLH, tTHL, VL, VH)
            (delay 1n);(tp)
   implementation  clock (CTA to be used)
      (timing 0.5u 0.5u)    (CTA timing constraints)
            (tL   tH)       (Clock low and high time)
end CLK1Mh
```

The language provides the support to enter the required arguments used by the simulator and the CTA itself. The first part (**interface**) lists the interface ports, the simulator arguments and the delay model with its arguments. There is one output (X) of type bit. The corresponding output waveform is controlled by t_{TLH}, t_{THL}, V_L and V_H. The delay to be used between the internal change of state and the output is also specified. Even if there are no inputs, the conceptual model requires a delay for each output The second part (**implementation**) indicates the CTA to be used (gate) and its arguments. Two timing arguments are used to control the clock frequency (t_L and t_H). A typical clock signal is shown in Fig. 4.7b.



(a) Symbol

(b) Typical output signal

(c) Continuous Time Automaton

Fig. 4.7 Model of a clock

The CTA of Fig. 4.7c models the behavior of a clock. It consists of two states named zero and one and two state transitions. Each state corresponds to the state of the output signal. The CTA starts in a state determined during initialization. The label associated with the state transitions indicates how much time will be spent in each state (wait t_L or wait t_H) and the new value of the output. In this example, the machine leaves the state zero after t_L and the state one after t_H. The output is set to 0 (.../0) when leaving state one and to 1 (.../1) when leaving state zero. This machine obviously produces a timed state sequence since there is a finite number of changes of state for any interval of time. There is only a constrained cycle in this CTA.

The CTA based model is designed to be easily used in an event driven logic simulator. In such an environment, the device model or the procedure that implements it is called for each new input event or when the wait time has elapsed. This diagram is easily translated into the following procedure:

```
algorithm SimulateClock
   begin       {This procedure can only be called
                     to process a wait event
                     since there are no inputs}
      case CTAstate of
         zero: CTAstate=one;
               insert(1,output);
         one:  CTAstate=zero;
               insert(0,output);
   end.
```

The procedure `insert(value,signal)` inserts a master event of `value` on the `signal` using the propagation delay model and computes the slave events.

4.5.2 The logic function as an argument: a pattern generator (1)

The following example is an extension of the clock CTA. It illustrates how a logic function is used as an argument in a CTA. For simplicity, the pattern generator shown in Fig. 4.8a has only one output. The specification of a typical pattern generator looks like:

(a) Symbol (b) Continuous Time Automaton

Fig. 4.8 Model of a pattern generator

```
circuit PatGen
   interface
      X:  bit (output  0.5n  0.5n  0   5)
                       (tTLH, tTHL, VL, VH)
              (delay 1n);(tp)
   implementation  Pattern  (CTA to be used)
      (timing 5u)     (Timing constraints)
             (tp)     (Period)
      (function
         (out X (not X)))
end PatGen
```

Notice how the logic function is specified separately from the timing specification. When using CTA, there is no need to integrate the timing specification into the logic function: these are two separate arguments. This pattern generator simply produces a clock of 100 kHz.

As shown in Fig. 4.8b, the corresponding CTA is very simple and has only one state and one state transition. On each state transition, the function is used to compute the new logic state and the new output. This single output pattern generator can be easily expanded to multiple outputs by adding output ports and by modifying the function.

4.5.3 The state variables: a pattern generator (2)

This example illustrates the use of a simple state variable. When evaluating the logic function, the current signal values are used. Therefore, in the pattern generator circuit, if the propagation delay is larger than the period, the current value of

the signal X might remain 1 for more than one period. A state variable can be used to solve the problem. The specification of such a pattern generator looks like:

```
circuit PatGen2
   interface
      X:  bit (output  0.5n  0.5n  0   5)
                      {tTLH, tTHL, VL, VH}
              (delay 20n);{tp}
   variable
      A:  bit (IFV 0);  {IFV: Initial Forced Value}
   implementation  Pattern  {CTA to be used}
      (timing 5n)     {Timing constraints}
             (tp)     {Period}
      (function       {Logic function}
         (set A (not A))
         (out X A))
end PatGen2
```

Because of the conceptual model assumed, there is no delay between the changes of CTA state and the change in the logic state variable. The variable changes as soon as the set function is executed. The operation of the CTA is the same as for the pattern generator described in section 4.5.2. The variable A will change on each state transition and the output X will equal the variable A delayed by 20n.

4.5.4 The timing constraints and the undefined states: a counter

The preceding examples do not have inputs, therefore no input timing constraints are required. This example illustrates the use of timing constraints and the need of a undefined states. The CTA for the counter is similar to the CTA of a flip-flop. For simplicity, the processing of undefined input events is omitted and will be described in section 4.6. The specification of the counter shown in Fig. 4.9a is:

```
circuit CounterDiv3
   interface
      C: bit   (clock   2.5);   {The signal C is a bit of class
                        VTh      clock.  It therefore has an
                                 input threshold.}
      Q: word2 (output  0.5n  0.5n  0   5)
                       {tTLH, tTHL, VL, VH}
               (delay 1n);{tp}
   variable
      A: word2 (IFV 0); {IFV: Initial Forced Value}
   implementation  SSSC  {CTA to be used:
                          Simple Synchronous Sequential Circuit
                          without processing of undefined
```

```
                                    clock events}
        (timing 5n 5n)     {Timing constraints}
                {tL tH}
        (function       {Logic function}
           (case A
              0  (set A 1)
              1  (set A 2)
              2  (set A 0)
           )
           (out Q A)
        )
end CounterDiv3
```

This is a two bit divide-by-three counter. There are two outputs (Q.0 and Q.1), a clock (C) and a two bit state variable (A). According to the definition of port class in section 4.4, there are two classes of ports: clock and output. The timing

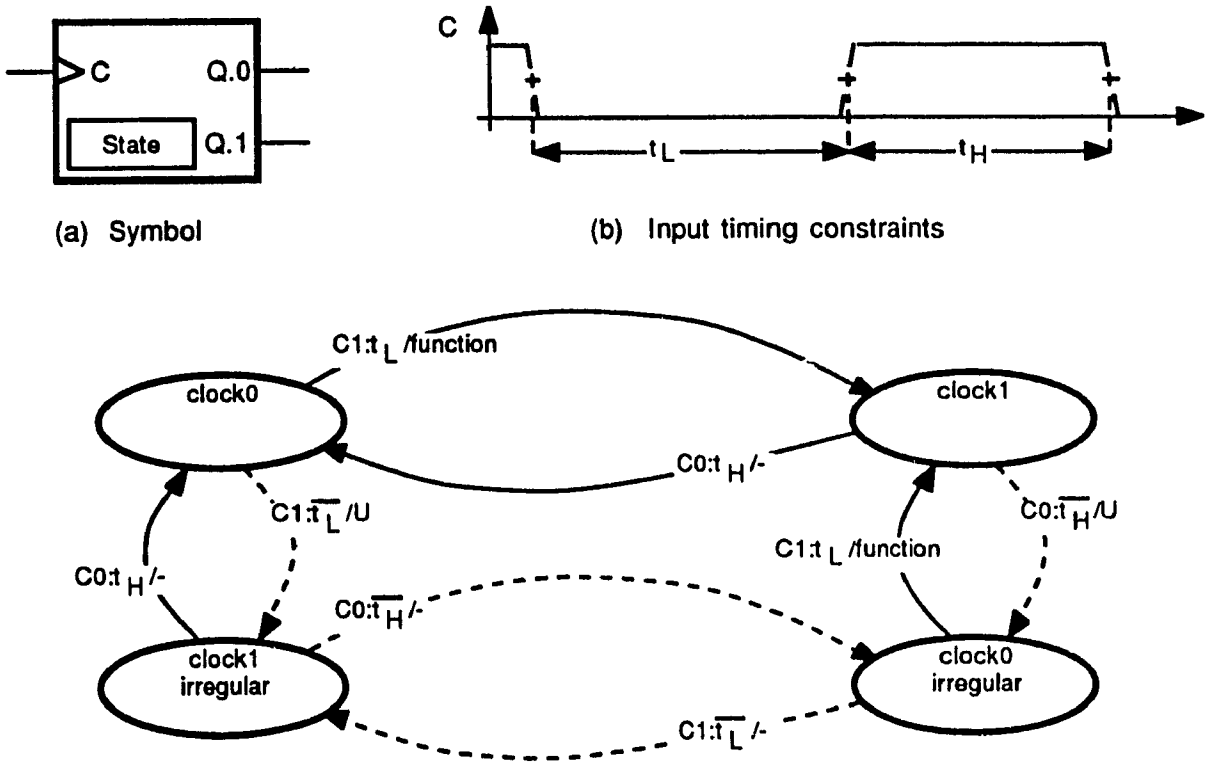


Fig. 4.9 Model of a counter

constraints t_L and t_H are shown in Fig. 4.9b and limit the operating frequency to 100MHz.

The CTA of Fig. 4.9c consists of four states. Two states (clock0 and clock1) are used for normal operation. The state names simply indicate the level of the clock signal. A clock signal meeting the timing constraints will make the CTA cycle between states clock0 and clock1. When in state clock0, if the clock rises and the timing constraints t_L is met, the logic function is executed (C1:t_L/function). Then, if the clock falls and the timing constraint t_H is met, the CTA state changes to clock0 and nothing else happens (C0:tH/-). A "-" indicates no change. This is a constrained cycle since it is made of two constrained state transitions. A constrained state transition is indicated by a solid arc. If a timing constraint is not met, the CTA state changes to one of the undefined state clock0irregular or clock1irregular (C0: $\overline{t_H}$ /U or C1: $\overline{t_L}$ /U). When in these states, the logic state variables and the outputs are all made undefined (.../U). If the clock signal continues to be too fast, the machine cycles between the two undefined states (C0: $\overline{t_H}$ /- and C0: $\overline{t_H}$ /-). These are two unconstrained state transitions indicated by a dotted arrow. When cycling between the two undefined states, the logic state does not change (.../-). *Even if this cycle is unconstrained, because the logic state does not change, the continuous change of state is transformed into a timed state sequence.* The only way to return to normal operation is to meet the timing constraints (C0:t_H/- or C1:t_L/function).

4.5.5 Undefined input logic events: a gate

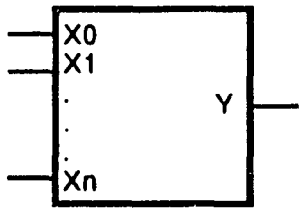
In this section, the CTA for a gate is described. This complete example will be used to illustrate how the CTA processes sequences of input events including undefined events. A gate is a combinational circuit with one or more inputs and one output (see Fig. 4.10a). A gate behaves properly, meaning that no hazard or glitch is produced on the output for separate input events. According to the conceptual model

of Fig. 1.10, the gate combines the input events into one sequence of output events. In the case of a gate, the logic function argument is used to combine the events and then the resulting signal drives the CTA. The specification of a typical two inputs nand gate looks like:

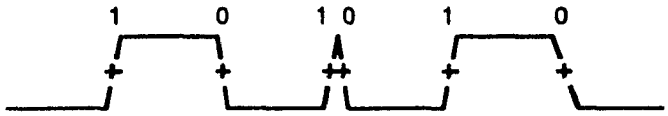
```
circuit nand2
   interface
      A B:bit (input  +3,0);
                      (VTh)
      X:  bit (output +0.5n +0.5n +0  +5)
                      (tTLH, tTHL,  VL, VH)
              (delay2 +2n   +3n);(tPLH, tPHL)
   implementation gate  (CTA to be used)
      (timing +0.1n)       (Input timing constraints)
              (ts)
      (function (out X (nand A B))) (Logic function)
end nand2
```

For a gate, there are two classes of ports: input and output. Each input port has a threshold (V_T) and each output port has two delay model arguments (t_{PLH} and t_{PHL}) and four waveform related arguments (t_{TLH}, t_{THL}, V_L and V_H). The delay model (delay2) is a continuity-preserving delay model as described in section 3.2. The CTA gate requires two arguments: an input timing constraint, the minimum separation time (t_S), and a logic function.

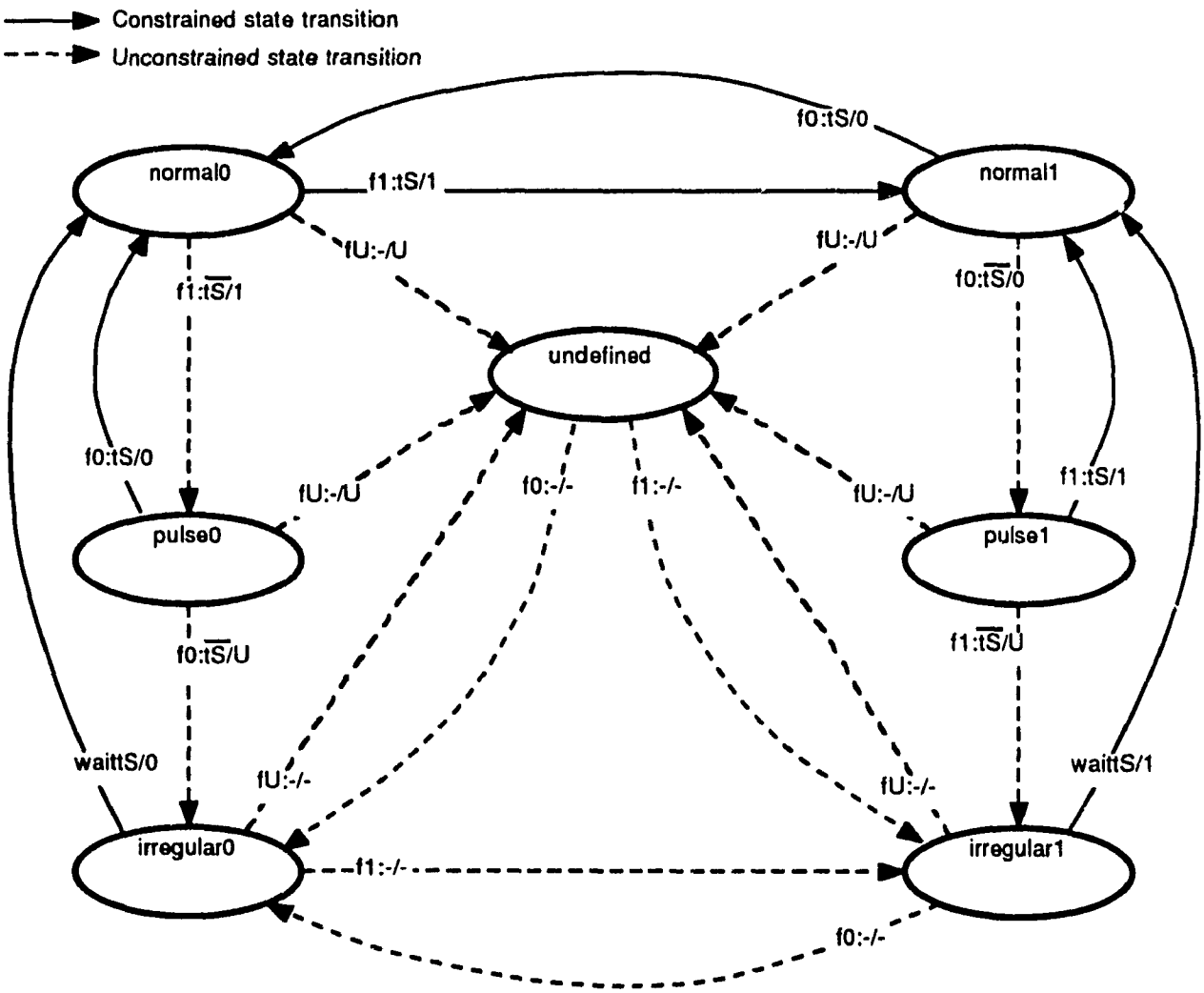
What distinguishes the CTAs is the set of timing constraints. As discussed previously, all logic models, even a simple gate, must have timing constraints to perform the transformation of continuous time into discrete time. The input timing constraint for a gate (t_S) describes the minimal separation between the input events that insures correct operation of the gate. A gate is not designed to process long sequences of very close events. Our intuitive model of its behavior excludes such sequences. The separation time can be set to some small value, like a third of the input transition times. Another aspect that distinguishes the CTA of a gate from another CTA is how the logic function argument is used. For example, in a gate the function is used before the change of state to determine if one is required and in a



(a) Symbol



(b) Typical sequence of input events



(c) Continuous Time Automaton

Fig. 4.10 Model of a gate

counter the function is used on each positive edge of the clock. A CTA therefore characterizes a class of behavior based on a set of timing constraints and a logic function.

In the proposed model, the CTA has been designed to accept separate events or pair of close events. Sequences with more than two close events will cause a timing violation and generate an undefined output. An input sequence of events that would go unchanged through the CTA is shown in Fig. 4.10b. Notice that this sequence is the result of evaluating the function on input events. It drives the CTA which modifies the sequence if necessary to obtain a timed state sequence. The CTA of Fig. 4.10c models the behavior of a gate. The CTA starts in a state determined during initialization. The format of the label associated with each arc is X:TEST/ACTION. In the case of a gate, X indicates the result of evaluating the function for each input events. For example, f0: ... indicates that the function evaluates to 0.

The CTA consists of 7 states. Two states (normal0 and normal1) are used for normal operation. A sequence of separate 0 and 1 is acceptable and will make the CTA cycle between normal0 and normal1 states. For example, when in state normal0 (i.e. function = 0 or f0: ...), if the input event makes the function changes to 1 (i.e. function = 1 or f1: ...) and the separation time is met, the output becomes 1 ($f1:t_S/1$). This is a constrained cycle. The proposed gate CTA also accepts isolated pulses. For example, when in state normal0 if the timing constraint is not met, then the CTA state changes to pulse0 indicating that a short pulse has been received ($f1: \overline{t_S}/1$). Cycling through normal0 and pulse0 is an acceptable sequence consisting of separate pulses and it is a constrained cycle. On the other hand, if from the state pulse0 a second close event is received, then the state changes to irregular0 and the output is made undefined ($f0: \overline{t_S}/U$). If the input events continue to be close, the CTA will cycle through states irregular0 and irregular1, without changing the output

(f0:-/-, f1:-/-). The only way to end a cycle where the output remains undefined is to allow the gate to settle (wait t_S). The cycle between irregular0 and irregular1 is an unconstrained cycle but since the output does not change, the result is still a timed state sequence.

The CTA must also process undefined input events. In the case of a gate, if the function evaluates to undefined the state becomes undefined as well as the output (fU:-/U). Exit from the undefined state goes through irregular states to insure a timed state sequence. This condensed diagram can be easily translated into the SimulateGate procedure as follow:

```
algorithm SimulateGate
   begin
      if not a wait event
         begin                          {Evaluate logic function}
            NewValue=function(inputs);
            if NewValue<>LastValue then
                     {Perform a change of state only if
               begin      the output changes, else do nothing.}
                  case CTAstate of
                     normal0: case NewValue of
                        1: if separate
                              then CTAstate=normal1;
                                   insert(1,output);
                              else CTAstate=pulse0;
                                   insert(1,output);
                                      {Accept two close events}
                        U: CTAstate=undefined;
                           Insert(U,output);
                     normal1:    .....
                     pulse0: .....
                     pulse1: .....
                     irregular0: .....
                        InsertWaitEvent(tS);
                     irregular1: .....
                        InsertWaitEvent(tS);
                     undefined:  .....
         end
      else   {it is a wait event}
         case CTAstate of
            irregular0: CTAstate=normal0;
                        insert(0,output);
            irregular1: CTAstate=normal1;
                        insert(1,output);
         end
      LastValue = NewValue;
   end.
```

The function `separate` verifies that the separation time between input events is larger than the specification t_S. The simulator keeps track of the time of the last event for the class of input ports and the separation time is measured between the current time and this value. The procedure `insert(value,signal)` inserts a master event of `value` on the `signal` using the propagation delay model and computes the slave events. The procedure `InsertWaitEvent(time)` inserts a wait event at `CurrentTime + time`.

4.6 Construction of a CTA for a flip-flop

The first step to obtain a logic model is to understand the fine details of operation of the device. One could start by classifying the interface ports, identifying the logic states and defining the logic function. Then the behavior of the device must be examined to determine exactly how the device will react to all sequences of input events and timing constraints should be defined between classes of ports.

Once the behavior of the device is well understood, we can start the construction of the state machine. The process is decomposed into three phases: normal operation, timing violations and undefined inputs. The process will be illustrated with the construction of the CTA for a flip-flop. A D type flip-flop as in Fig. 4.11a is used. The following HDIL listing describes the D flip-flop:

```
circuit FF7474
   interface
      D :bit (input   2);
                     {VT}
      C :bit (clock   2);
                     {VT}
      Q :bit (output 0.5n  0.5n 0   5)
                    {tTLH, tTHL, VL, VH}
             (delay2 2n    3n);
                    {tPLH, tPHL}
   implementation DFlipFlop    (CTA to be used)
      (timing 5n 5n  0.5n  0n)
             {tL tH  tSU   tHD}
      (function (out Q D))
end FF7474
```

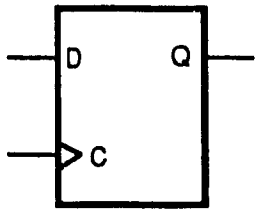
The D flip-flop has three ports: an input D, an output Q and a clock C. As for the gate, input ports have threshold arguments and output ports have waveform related arguments and propagation delay model arguments. As illustrated in Fig. 4.11b, the flip-flop is characterized by four timing constraints: t_L and t_H specify the minimum low time and high time of the clock and t_{SU} and t_{HD} specify the set-up time and the hold time for the input D with respect to the clock. The logic function reduces to Q=D. We are now ready to construct the CTA for the D flip-flop.

Normal operation

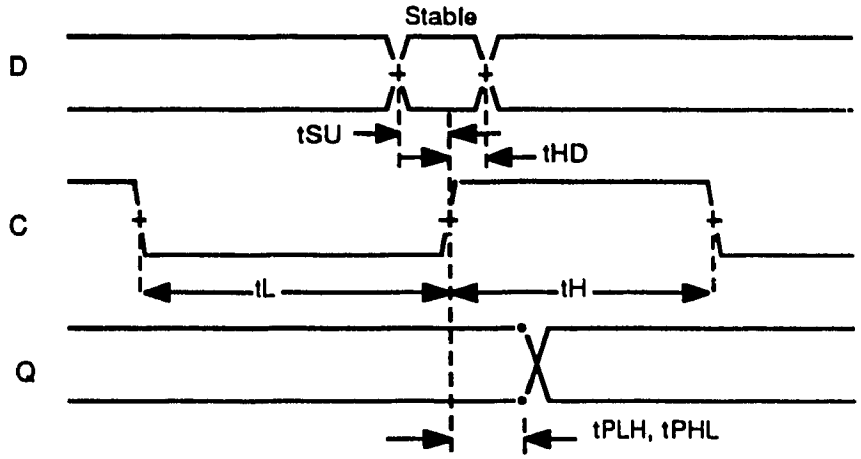
The first step consists in creating a state machine for acceptable sequences. The part of the CTA that process acceptable sequences for a flip-flop is illustrated in Fig. 4.11c. There are three states corresponding to the clock signal: clock0, clock1a and clock1b. Let's suppose the clock of the flip-flop is 0 and therefore the CTA state is clock0. Events arriving on the D input do not affect the flip-flop (D-:-/-). If the clock signal changes to 1 and the set-up and the low times are met, the state changes to clock1a and the value of D is transferred to the output Q (C1:$t_{SU}t_L$/D). Now the clock is 1 and no changes on the input D is allowed for the hold time period (wait t_{HD}). Once the hold time has elapsed, the state changes to clock1b, where the clock is still one but now the D input can change (D-:-/-). The cycle ends if the clock changes back to 0 and if the high time is met, the output does not change (C0:t_H/-). This completes the description of normal cycles for a flip-flop. Each of the cycles in Fig. 4.11c are either constrained cycles or else they do not affect the state. Therefore this part of the CTA produces a timed state sequence.

Timing violation

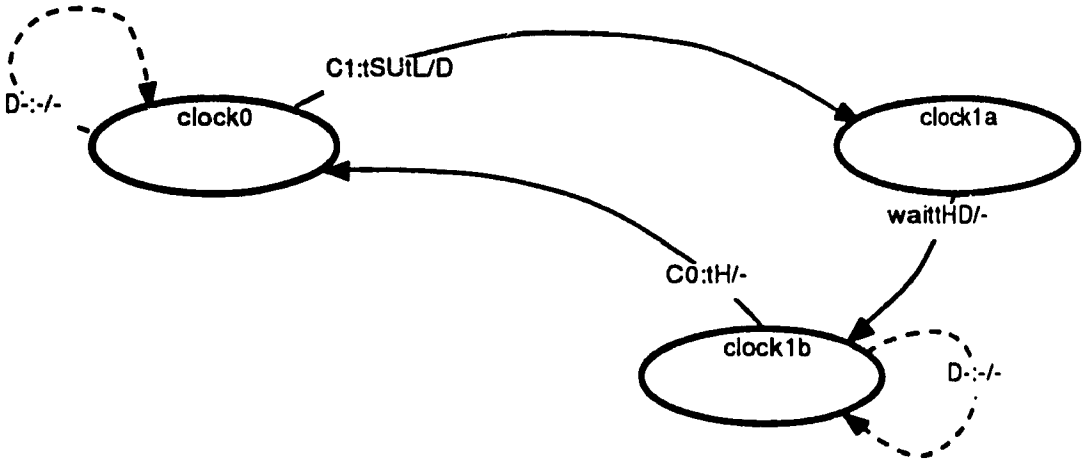
The next step is to decide what to do with timing violations. The processing of timing violations for the flip-flop is shown in Fig. 4.11d. For clarity, all arcs associated with normal operation have been removed. There are four cases to



(a) Symbol

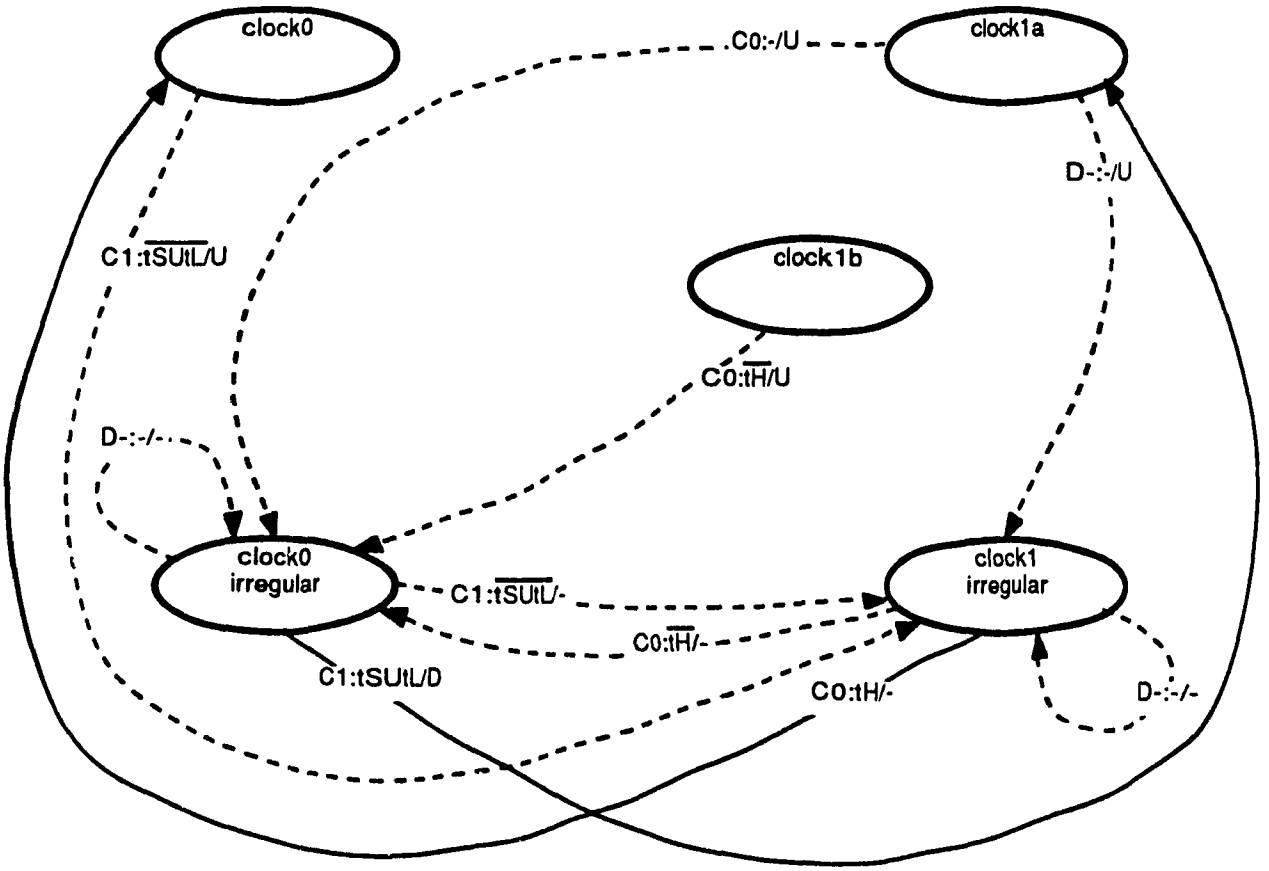


(b) Timing constraints and propagation delays for a flip-flop

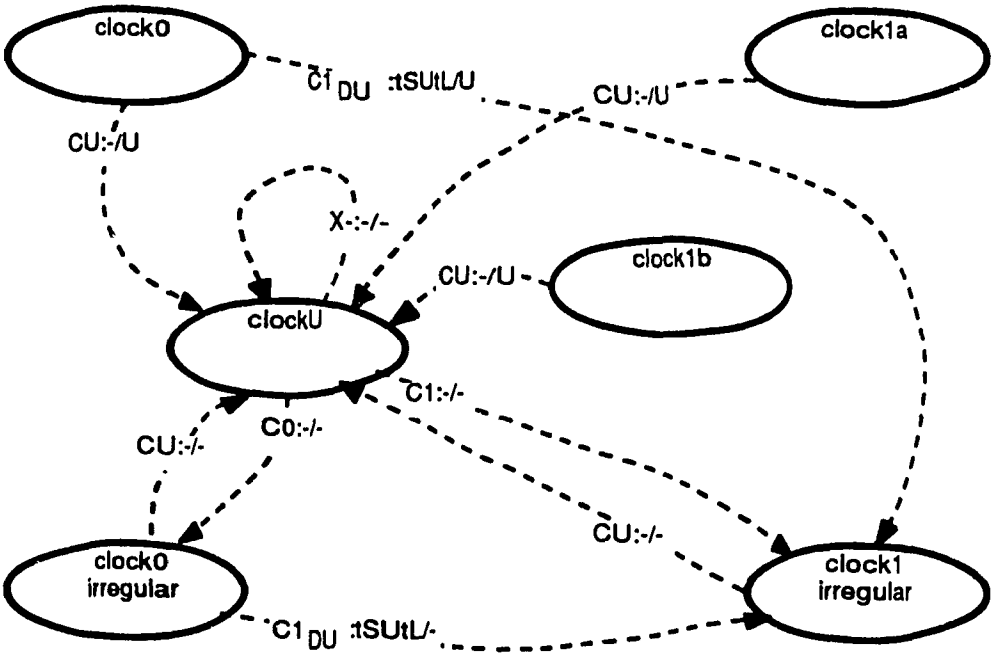


(c) CTA of a flip-flop: normal operation

Fig. 4.11 Model of a flip-flop



(d) CTA of a flip-flop: timing violations



(e) CTA of a flip-flop: undefined inputs

Fig. 4.11 Model of a flip-flop (continued)

consider: set-up time or low time not met when in clock0 (C1: $\overline{t_{SU}t_L}$ /U), clock or data changing when in clock1a violating hold time (C0:-/U and D-:-/U) and high time not met when in clock1b (C0: $\overline{t_H}$ /U). Two new states are used (clock0irregular and clock1irregular) and in each case the output is made undefined (.../U). The CTA will keep cycling through these two states until the clock has settled (C0:t_H/- or C1:$t_{SU}t_L$/D). Most of these state transitions are unconstrained but since the logic state and the output remain undefined, the machine produces a timed state sequence.

Undefined inputs

The last step is to process undefined inputs. Undefined D input is easily processed since D never affects the state unless the clock is rising and the set-up and low times (t_{SU}, t_L) are met. As shown in Fig. 4.11e, two arcs from clock0 and clock0irregular have been added to process this case ($C1_{DU}$:$t_{SU}t_L$/U and $C1_{DU}$:$t_{SU}t_L$/-). If the clock becomes undefined, the state changes to clockU and the CTA will exit from this state only if the clock changes to 0 for at least t_L or to 1 for at least t_H. This is accomplished by going through the irregular states.

The complete CTA for a flip-flop integrating Fig. 4.11c, d and e is given in Appendix D. As for the gate, the translation of this diagram into a procedure is straightforward. Notice that since synchronous sequential circuits (SSC) have a similar behavior, their CTA will be almost identical. Furthermore, this CTA can be used to model the behavior of complex devices such as microprocessors. We are currently working on the model of the 68000 microprocessor.

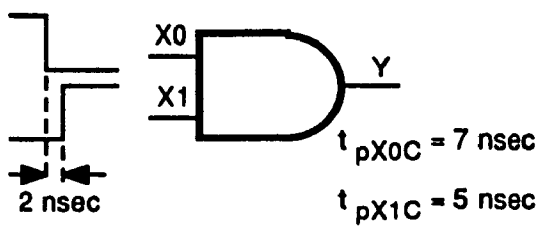
4.7 Circuit partitioning and event combination

The conceptual model of Fig. 1.10 and 1.11 assumes that input events are simultaneously combined, that the change of state is simultaneous to the input events and that zero delay output events are generated by the CTA. The requirement about simultaneous event arrival leading to simultaneous state changes suggests that a

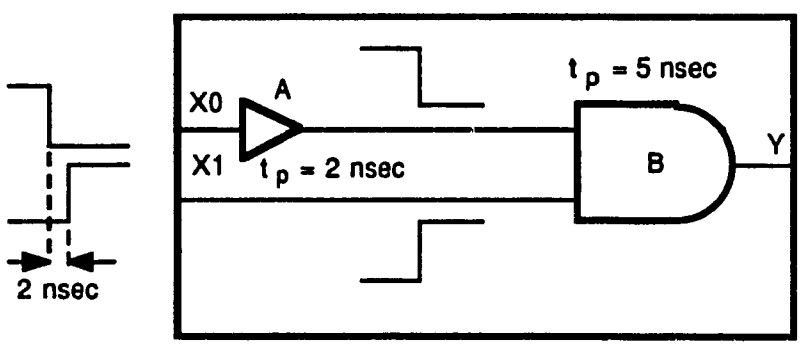
logic model acts as an event combiner and that the circuit should be partitioned accordingly. This brings back a well known problem about distinct propagation delays on inputs, as in the gate in Fig. 4.12a. This section describe the problem and how to model the event combination. Event combination with CTA will be discussed for gates, flip-flops, combinational circuits, 3-state devices and synchronous sequential circuits with and without asynchronous inputs.

4.7.1 Gates

When the state model rule (see section 1.7.2) is verified, the input ports are said to be "time aligned" with respect to their effect on the device state. For example in the case of gates, all input ports are assumed to be time aligned and the input events are combined using the logic function. It is therefore assumed that the gate is



(a) Non simultaneous arrival of events having effect at the same time



(b) Simultaneous arrival of events having effect at the same time

Fig. 4.12 Modeling multiple input gate with different propagation delays

a perfect combiner. The multiple emitter transistor of a TTL gate or the common signal of a MOS gate performs the event combination. In real gates, event combination is not perfect, very close events might not be combined as expected. The separation time (t_S) used in CTA automaton of the gate in Fig. 4.10, indicates the limit to the capability of a gate to combine close events.

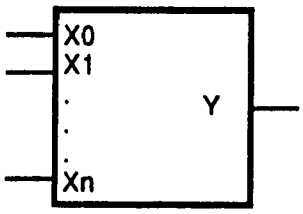
To illustrate the time alignment problem, let's use a 2-input AND gate with different propagation delays for the two inputs as shown in Fig. 4.12a. In this case, the inputs do not have simultaneous effect on the state and the device can not be modeled using a single logic model. The problem is easily resolved by using two logic models. As shown in Fig. 4.12b, the logic model A with a propagation delay of 2nsec is used to time align the signals X0 and X1 to insure that the logic model B meets the state model rule.

4.7.2 Flip-flops and SSC

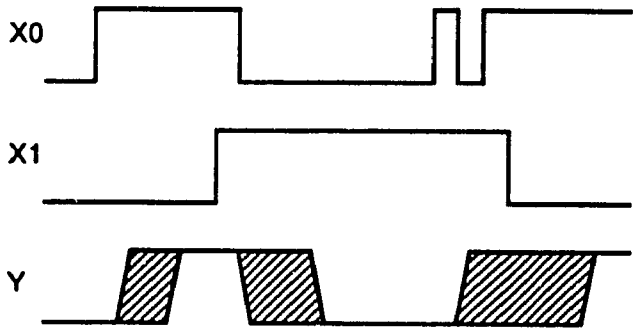
In the case of a flip-flop, the input D and the clock C must be combined. Events on D and C are assumed to be time aligned and have simultaneous effects on the state. The limit to the capability of a flip-flop to combine these events is modeled by the set-up and the hold-times. Violation of these constraints leads to an undefined state. The same technique is applied to SSC.

4.7.3 Combinational circuits (ROM)

In a general combinational circuit such as in Fig. 4.13a, the propagation delay might be different for each input. In addition, a single input transition does not necessarily imply a single output transition and the so called static and dynamic glitches are produced. Because of their internal organization, ROMs are particularly affected. ROMs are often modeled using an access time that defines how long the outputs are unstable following any change on the input address lines. It is assumed that all inputs are time aligned and an undefined period or envelope is used to model



(a) Symbol



(b) Typical signals

Fig. 4.13 Event combination in combinational circuits (ROM)

the variation in propagation delays. The undefined time is also used by the CTA to discriminate acceptable sequences of events. This is illustrated in Fig. 4.13b where each input event produces an undefined event and the combination of close events simply extends the envelope. The width of this envelope (t_U) is a measure of the capability of a combinational circuit to combine events. The CTA for combinational circuits is given in Appendix D.

4.7.4 Tri-states devices

Tri-states devices are often used in logic circuits. They consist of some logic function with an enable line that controls the output high impedance state (see Fig. 4.14a). In answering the question how the enable input (E) is combined with the other inputs we observe that the propagation delays are often different. The proposed model is shown on Fig. 4.14b and consists of three devices: one that combines the

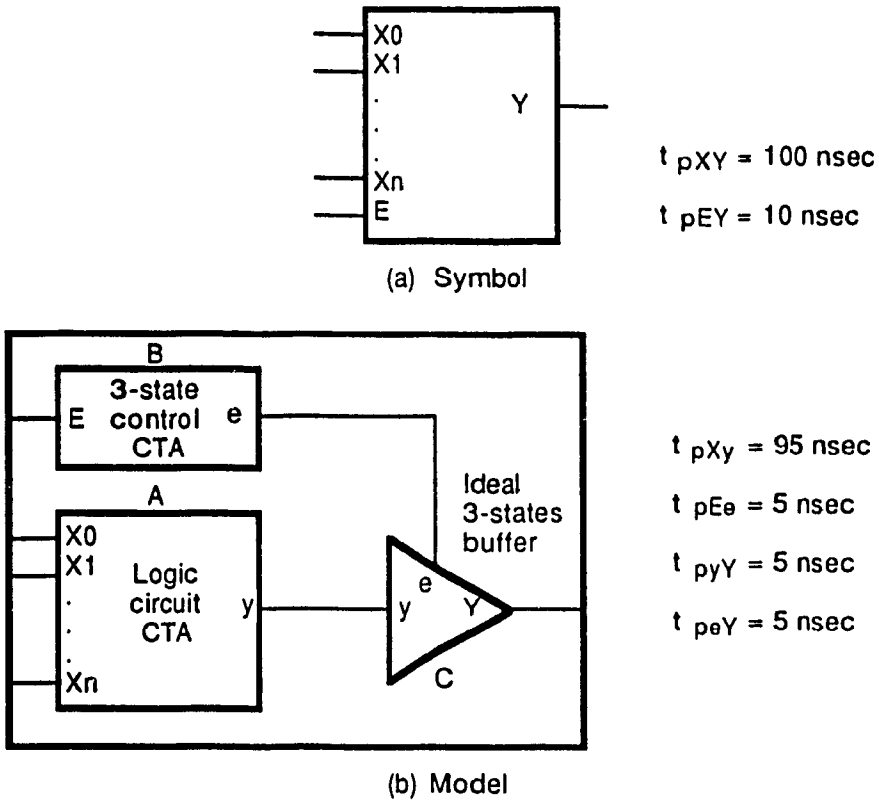


Fig. 4.14 Partitionning tri-states devices

logic inputs (A), one that process the enable signal E (B) and an ideal 3-state buffer that combines the outputs e and y (C). Notice how this corresponds to the designer's intuitive view of a tri-state devices and how each device meets the state model rule. The CTA for the 3-state control circuit (B) is given in Appendix D.

4.7.5 Flip-flops and SSC with asynchronous inputs

The circuit to be modeled is a flip-flop with an asynchronous reset as pictured in Fig. 4.15a. The question is: How are the clock (C) and the reset (R) combined? First, if the propagation delays from clock to output and reset to output are different, the model must be broken-up as in Fig. 4.15b. The models for each devices including their CTAs are given in Appendix D. The device A combines the D input with the C input and the device B combines the reset signal with the output of the device A (cq)

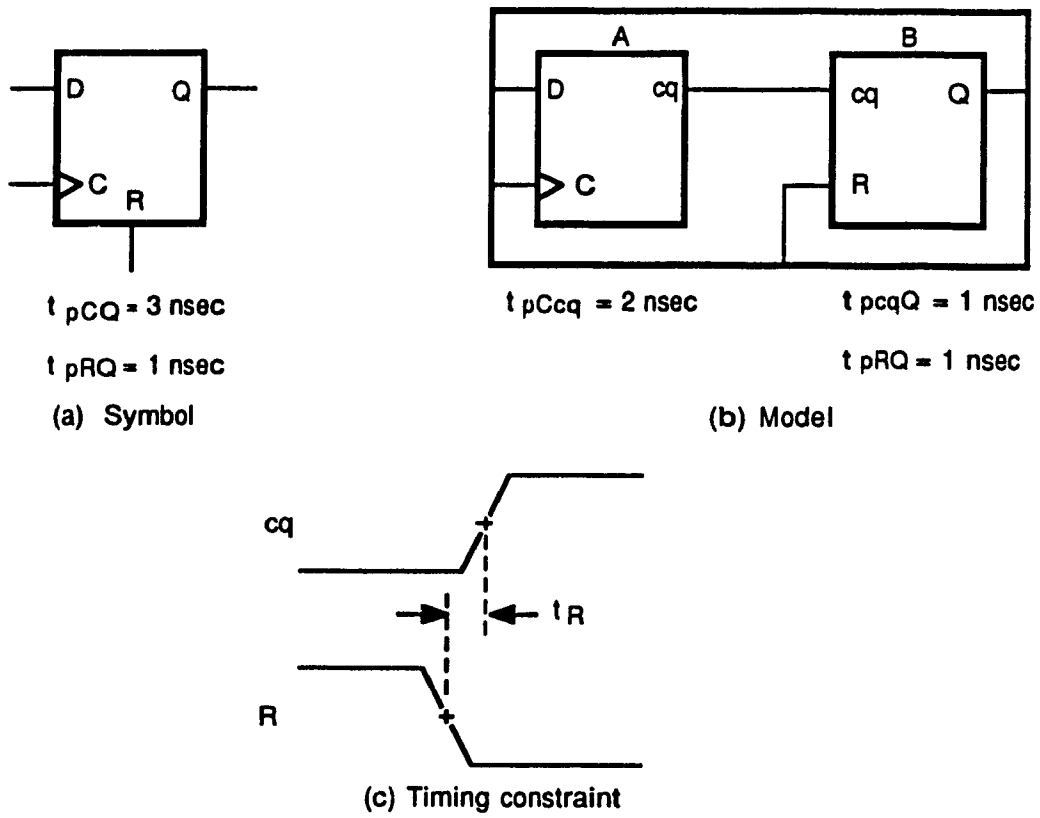


Fig. 4.15 Event combination for asynchronous input

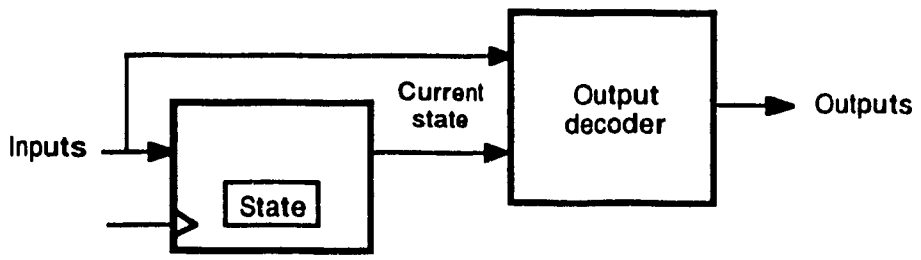
which is time aligned with the reset (R) to insure that device B meets the state model hypothesis.

Next, we need to determine how cq and R are combined. It is assumed that the reset overrides the clock. There is a conflict if the reset is removed and if the input cq simultaneously forces a 1. To resolve this conflict, a timing constraint similar to hold time must be added: the reset line must not move to zero while the clock is rising and the data is one. This timing constraint between cq and R is called the release time (t_R) and is shown in Fig. 4.15c. The CTA for the device B is given in Appendix D.

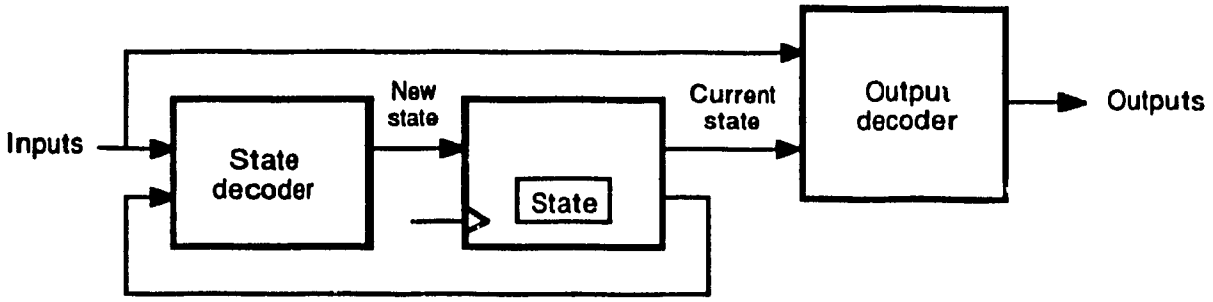
4.7.6 More about SSC

In the next example, we will study how the inputs are combined with the clock in a SSC. In a simple flip-flop and in a simple SSC (Moore type), the outputs only depend on the change of state which in turn is controlled by the clock. Combination of the clock and the inputs has been discussed in section 4.6. In the case of a more complex SSC (Mealy type), the state variables must also be combined with the inputs. Since the propagation delays from input to output and from clock to output are probably different, the circuit must be separated into two parts (see Fig. 4.16a). The first part can be modeled as in section 4.6 and the second part as a gate or a combinational circuit. Each circuit meets the state model rule and combines the input events accordingly.

In a typical SSC, the propagation delay of the input decoder is such that the clock signal arrives at the flip-flops approximately at the same time as the next state



(a) The outputs depend on the inputs



(b) The state decoder delay is too long

Fig. 4.16 Partitionning a SSC

signals and therefore the set-up and hold times describe a period around the clock transition. If on the other hand, the state decoder delay is longer, it will be necessary to separate it as in Fig. 4.16b to make sure that the new state signal is time aligned with the clock signal.

4.8 Conclusion

In modeling logic devices with CTA, we have been able to clearly formulate the mechanism to transform continuous change of state (continuous time) into a timed state sequence (discrete time) and formally justify the existence of timing constraints and the necessity of an undefined state. The mechanism meets the objectives enumerated in section 4.1. It processes the continuous sequence of input logic events, a timed state sequence is obtained, the processing of timing constraints is well integrated with the logic function, the undefined states and the undefined events are systematically processed and zero delay output logic events are generated.

CTAs are easy to implement and use. A CTA is a finite state machine and is therefore easily translated into a computer program. In addition, since the CTA and the processing of the timing constraints are hidden, the user only needs to fill a template with the required arguments (timing constraints and logic function) to create a new part. Therefore, very few CTAs are required to model thousands of logic parts and the validation of CTA based models is simplified.

Chapter 5

Implementation, results and comparative analysis

In the preceding chapters, a new logic model has been proposed. The objective of the work is to clarify the abstraction used and to improve the model correctness when modeling logic devices. As it will be demonstrated in this chapter, the resulting simulator is not faster or less memory hungry than comparable simulators. It is a logic simulator compatible with other logic simulators. The question is then: How is the modeling technique better or at least different for simulation? The object of this chapter is to answer this question and compare important aspects in the abstraction mechanism, the simulation process and the simulation results with other simulators. The comparison is made with respect to the initial objectives:

- modeling the transmission of information between devices,
- modeling the processing time (propagation delays) and
- modeling the transformation of continuous time into discrete time.

Then, implementation issues and hardware description languages are compared to assess the advantages and the disadvantages of our approach. Finally the relationship with engineering practice is studied.

5.1 Implementation

The proposed modeling technique was tested with a logic simulator. The simulation process is shown in Fig. 5.1. Our approach to modeling and simulation closely matches common engineering practice. First, the logic model parameters of the devices being modeled are extracted and the logic models are computed and placed in a library. A library model is similar to a data book specification, the model

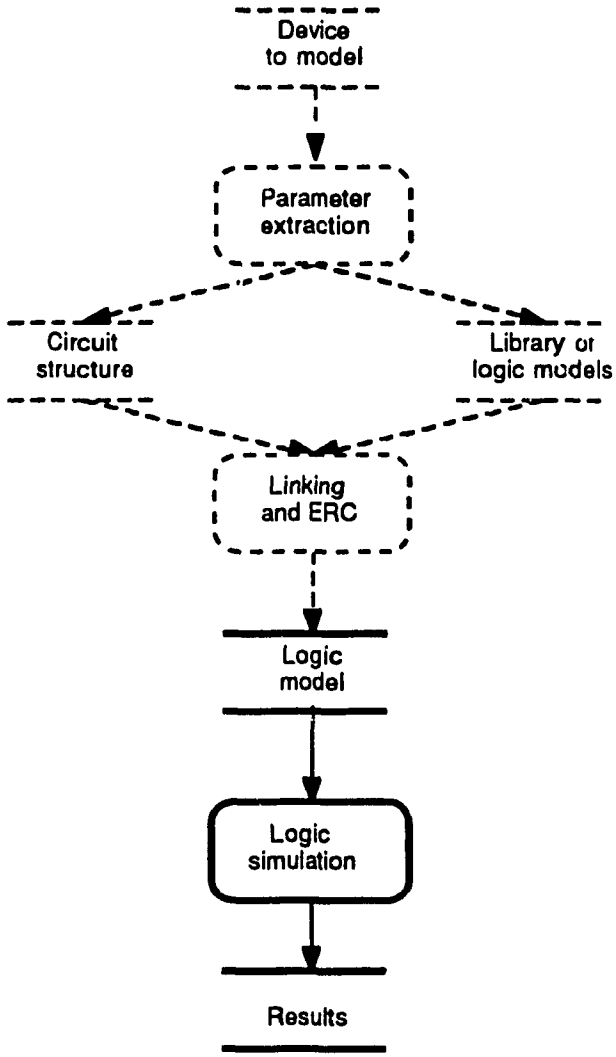


Fig. 5.1 Simulation process

parameters are expressed with respect with the source and the load. Examples are given in section 3.4. A linking algorithm including Electrical Rule Checking (ERC) is then used to ensure that the connections between the devices satisfies certain rules (e.g.: fan in and fan out). The logic model arguments (t_{PLH}, t_{PHL}, ...) for the actual sources and loads are computed. The sources are characterized by the input signal transition times and logic levels and the loads are characterized by the required input currents or the input impedances.

This is where our implementation begins. It is assumed that the logic model has been computed for a fixed and time invariant load and only the simulator is

implemented Even though parameter extraction and linking is not implemented, the corresponding algorithms will be reviewed in this section to demonstrate that our approach can use existing algorithms. Then a simulator based on the proposed modeling technique is described.

5.1.1 Parameter extraction

The first problem in any simulator is to obtain a model that faithfully represents the real device. In the case of a logic model using a CTA, the following parameters must be computed:

- input thresholds
- input timing constraints
- propagation delays
- output transition times
- output voltages levels

Logic model extraction consists of computing these parameters based on electrical parameters, such as capacitances, resistances and transistor characteristics. Computing the logic model from the electrical circuits is a difficult task if no information about the logic function is available [63] and generally it is assumed that the logic function is known. Many analysis techniques [57-64], have been developed to extract the logic model parameters, like propagation delay, set-up time and so on. A logic model extraction program analyses the network of transistors and capacitances and computes the charge or discharge characteristics. It then evaluates the propagation delay and other parameters. Parameters for logic models can also be obtained by simulation or by test by exercising the model or the actual circuit respectively. The logic model can also be parameterized for different sources and loads if necessary.

These parameters are similar to the parameters used in many logic simulators, therefore their extraction should not cause any particular problem and CTA based logic models and simulators should integrate easily into a CAD system.

Tolerances on the parameters

As in all logic simulators, the problem of interdependence of parameters still exists in our model. For example, the output transition times of the NAND gate in Fig. 5.2 (Real C) might be different depending on which input causes the output event. Propagation delays and transition times at the output not only depend on the output load, but also on the input signals. Keeping track of these relations during simulation is needlessly complicated and contradicts the objective of logic simulation which must be fast. Therefore, instead of keeping track of the complex relations between the parameters, typical values with tolerances are used, see Fig. 5.2 (Simulated C). Tolerances are used by manufacturers of logic devices [43-45] and in most logic

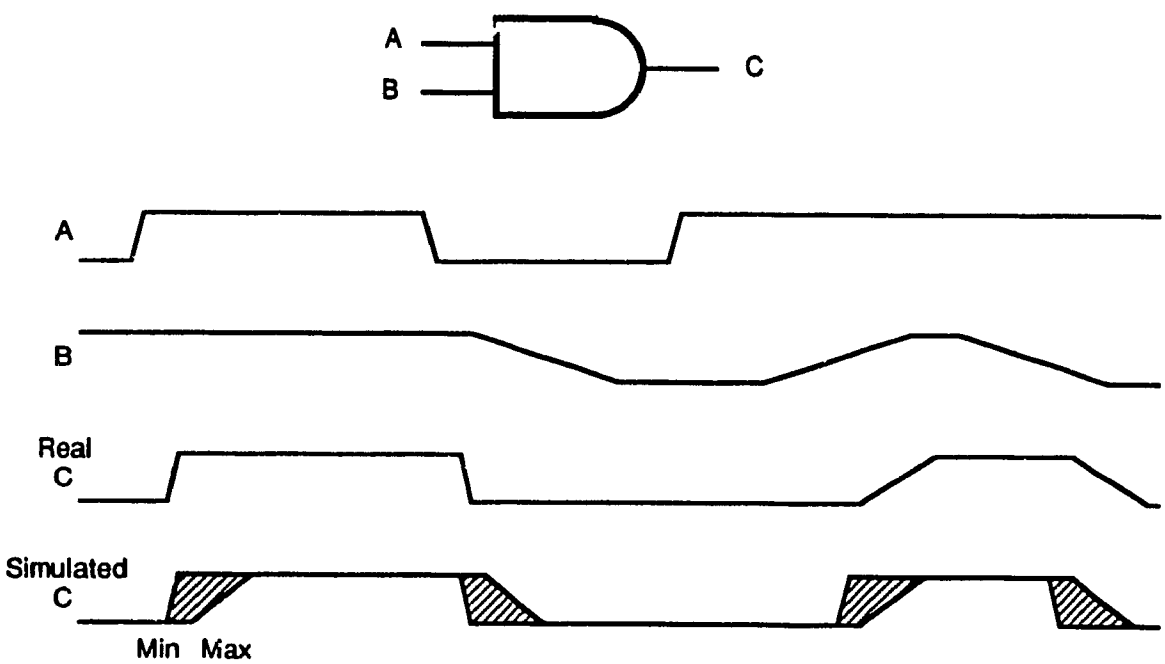


Fig. 5.2 Use of tolerances on propagation delays and transition times

simulators. Since logic devices are designed to minimize these dependencies, this model is acceptable in most cases.

5.1.2 Linking algorithm and Electrical Rule Checking (ERC)

We now briefly discuss the operation of a hardware linker. A linker has two functions: verify the linking rules and deparameterize the logic model. Deparameterizing the logic model involves computing the parameters (propagation delays, transition times, ...) for the actual sources and loads. For TTL, this is simple as the propagation delay and the output transition times normally do not depend on the loads. For CMOS, they are often approximated by a linear function of the load capacitance. Examples are given in section 3.4.

The linking rules describe the conditions required for an element to operate properly. For example, the linking rules for TTL [43] and discrete CMOS [44-45] are as follow:

<u>TTL</u>

Given:

I_{iL} = Required input current for low level

I_{iH} = Required input current for high level

C_i = Input capacitance

I_{oL} = Output drive capability for low level

I_{oH} = Output drive capability for high level

C_L = Specified maximum load capacitance

C_W = Wire capacitance

Rules for each output:

$I_{oL} > \Sigma I_{iL}$ for all inputs connected to it

$I_{oH} > \Sigma I_{iH}$ for all inputs connected to it

$C_L > C_W + \Sigma C_i$ for all inputs connected to it

CMOS

Given: C_i = Input capacitance

C_L = Specified maximum load capacitance

C_W = Wire capacitance

Rules for each output:

$C_L > C_W + \Sigma C_i$ for all inputs connected to it

The hardware linker algorithm is:

```
algorithm link
  begin
    for all outputs of all elements
      begin
        check rules; {violation to the rules indicates
                        an improper use of the device}
        compute logic model parameters;
      end
  end
```

Parameter extraction, linking and ERC will not be discussed further. These are important implementation issues but are well known [43-45].

5.1.3 Implementation of the logic simulator

This section describes the implementation of a CTA based logic simulator using master slave events and a continuity preserving delay model. The implementation could have been done using a simulator based on VHDL. This was not done for two reasons. First, when we started this work, VHDL simulators were not easily available and second, VHDL seemed too complex for our simple task. Today, a subset of VHDL could certainly be used and the required translators could be written. In this section, we will briefly describe the hardware description language, the simulator interface and the simulator organization. Simulation algorithms have been described in section 3.3 and logic device models in chapter 4.

Circuit description

Implementation of a logic simulator requires some means to input the circuit and the logic device model descriptions. A hardware description language was designed for this application. Using a custom language gave us some flexibility compared to a predefined language like VHDL. The language definition was changed to follow the continuously evolving requirements associated with the development of our new model. The resulting language is simple and precise and the specification can be translated into VHDL. The language is called HDIL (Hardware Description and Integration Language), so named to reflect its support of all phases of the design (compilation, linking and simulation) in an integrated environment. The language syntax is described in appendix B. One of the main characteristics of HDIL is its ability to describe views of different abstractions, including schematic, layout and simulation while maintaining the structural integrity between views.

Simulator interface

The need for a fast interactive design environment requires that most tools share the information through memory rather than through a file. To ease the sharing of information through memory, a Hardware Description and Integration Data Structure (HDIDS) has been defined. Each data element in HDIDS matches a syntactic construct in HDIL. Two simple procedures "parse" and "unparse" are used to translate HDIL circuit description (file) into HDIDS (memory) and vice versa. This leads to a strong resemblance between HDIL and HDIDS. The description of HDIDS is given in appendix C.

The use of HDIL and HDIDS is illustrated in Fig. 5.3. A circuit description (HDIL) is parsed and a data structure (HDIDS) and a symbol table are created. The simulator then uses the circuit description, performs a simulation and returns the state and the signal waveforms as different views attached to the circuit description.

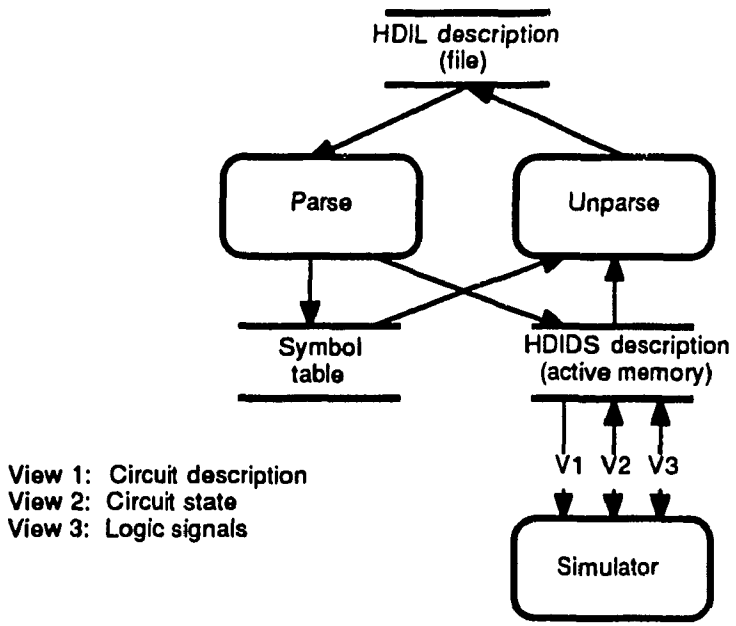


Fig. 5.3 Simulator environment

Simulator organization

As described in chapter 2, a logic simulator is an event driven simulator which uses the circuit description and the device models to predict the circuit behavior for a desired period of time from an initial state. Since a real circuit starts by itself, it was decided for simplicity and also for curiosity to make the simulator self starting without external events. Obviously, for practical reasons, signal sources are included. Nevertheless, the basic operation of the logic simulator does not require external events and its operation breaks down into three algorithms:

Algorithm build

This algorithm takes the circuit description (HDIDS) and creates the data structure required by the simulator. Typically, records are created to store the various pieces of information: signals (wires), devices, nodes, state, logic function, timing constraints (t_{SU}, t_{HD}) and propagation delays (t_{PLH}, t_{PHL}).

Algorithm initialize

This algorithm computes the initial values of the output nodes. It uses the logic function and the initial values of some nodes given by the user. During initialization, events are inserted in the predicted event list to allow the simulation to start if the *initial forced value* (IFV) is different from the *initial computed value* (ICV).

Algorithm simulate

This algorithm schedules the calls of the procedures that simulate various devices and computes the slave events.

5.2 Simulation results and simulator performances

The objective here is to verify the simulator algorithms based on master slave events and the device modeling procedure using CTA. This is accomplished by simulating circuits similar to the circuit in Fig. 5.4. The circuit is used to test the CTA for gates and consists of a self starting waveform generator circuit (U1 to U10) and the gate to verify (U11). The verification consists in running a simulation with different propagation delays for U1 to U6 in order to verify all branch conditions in the CTA. The results of the simulation are shown in Fig. 5.5.

The prototype simulator was written in PASCAL and run on a SUN workstation. A small circuit was simulated and the simulator processed 1500 slave events/sec. In the simulated circuit, the processing one slave events requires the evaluation of ten functions (not, and, or, ...). The listing of a typical test circuit and the simulation results are given in Appendix E. Obviously, the statistics presented here are based on longer simulation time. Considering that the functions are interpreted and not compiled and that no default values have been used to compute the slave events, there is plenty of room for improvement and 10k events/sec should be achievable. This is obviously on the low side for a logic simulator and is similar to

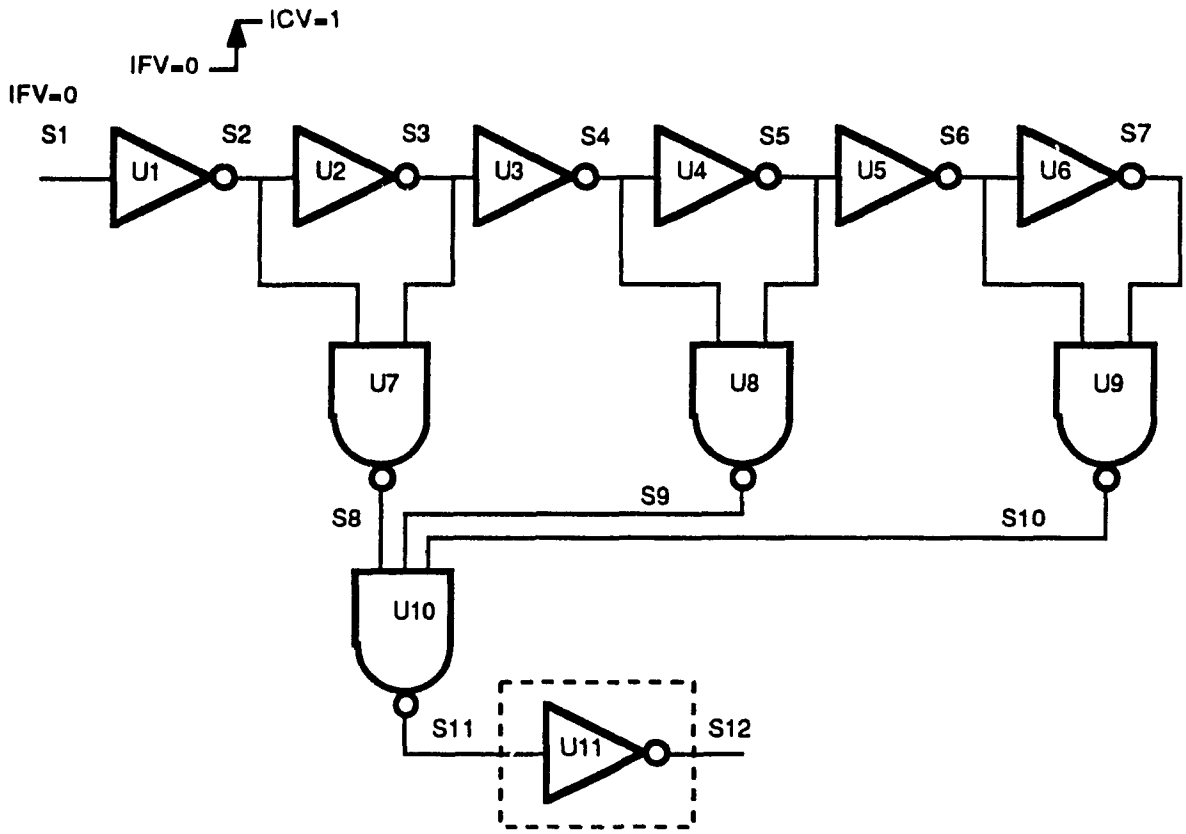
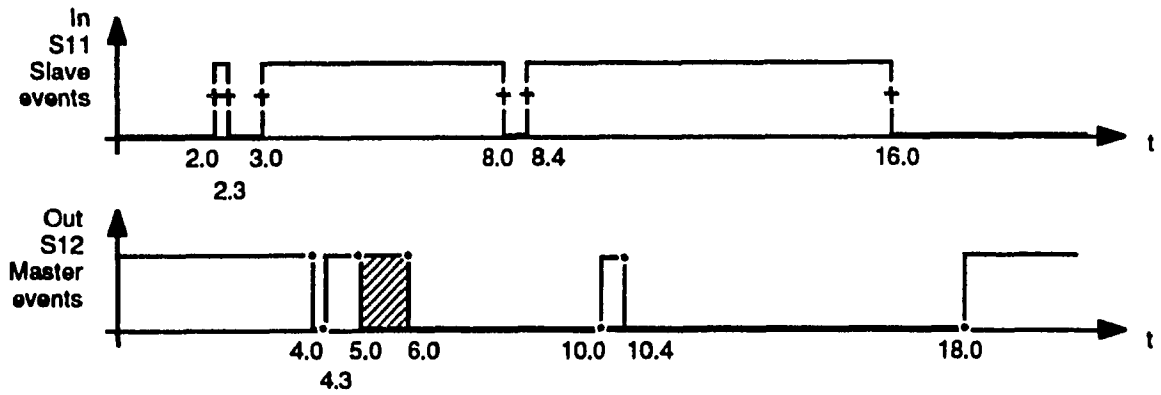


Fig. 5.4 Typical simulation circuit



Note: 1. Propagation delay is 2.0 and separation time is 1.0
2. For clarity, transition times are not shown

Fig. 5.5 Simulation results

logic simulator using transition times. Improving simulation speed was not our objective, we were more concerned with theoretical aspects. The memory requirements of a typical circuit are 260 bytes/device, 460 bytes/wire and 34 bytes/slave event. This is comparable to most logic simulators that use between 100 and 1000 bytes/device.

5.3 Comparative analysis

As shown in the previous section, the simulator performance is modest. The proposed modeling technique improves the global picture of a digital system in the form of a logic model abstraction. In this section we will bring up the characteristics of the proposed logic modeling technique that improves this picture. Having a more rigorous picture of a digital system is essential in a modern CAD environment where a large amount of information is shared by the clients and the designers and where computers are used in all phases of the design.

5.3.1 Comparing abstraction mechanisms and conceptual models

Logic modeling begins with an analog circuit. The abstraction mechanism as defined in section 1.7 corresponds to the engineer's intuitive view of digital systems. The three hypotheses state that logic devices are unidirectional, that input ports are threshold sensitives and that the logic signals are switching signals. We all have a clear picture of what is a logic device, but constructing a logic model for it is not as clear. The Fig. 5.6 shows some of the basic conceptual models used in the construction of logic models. Each model will be analysed with respect to the three modeling objectives related to signal representation, delay model and timed state sequences.

Hybrid model

Hybrid models combine the best of two worlds. Fast simulation speed is achieved by using logic functions to model the internal behavior and timing accuracy is

obtained using an analog model for the connections. Included in this category are the macro based models [24, 26], the switch based models [13-23] and their variations such as the current limited switch model [24]. Commercial mixed-mode simulators such as PSPICE [52] use A to D and D to A interfaces for their logic components. In hybrid model, Kirchhoff laws are used to solve the network equations. Hybrid models are clearly superior to master slave events in modeling the transmission of information between devices as they allow time varying loads. In summary, hybrid models are interesting for time varying loads but must be used with an appropriate logic model. Either of the device models in the Fig. 5.6 can be used with the wire model of Fig. 5.6a. The passage of analog signal to logic signal and vice versa is critical and the proposed master slave event definition is well suited as its purpose is to model the analog signal. For instance, the definition of master events which indicates a change in the output signal closely resembles the closing of the switches in Fig. 5.6a.

Classical logic model

Most logic simulators use the conceptual model of Fig. 5.6b or a simplified version of it. Modern logic simulators like SILOS [29], Verilog [48], SAMSON [30] and VHDL based simulators [31, 32, 41, 49, 50] supports the full conceptual model including inertial and similar delay models and strength factors.

In such simulators, the delays are included in the logic function. For example, a typical VHDL statement looks like:

```
portA <= 0 after 10ns;
```

Thus, the propagation delay model combines events from many logical statements executed at different times and sometimes some events have to be removed in order to preserve the consistency of the logic signal. In addition, part of the behavior associated with the transmission of information on wires, such as the effect of a large capacitance, is also integrated in the delay model. This is the function

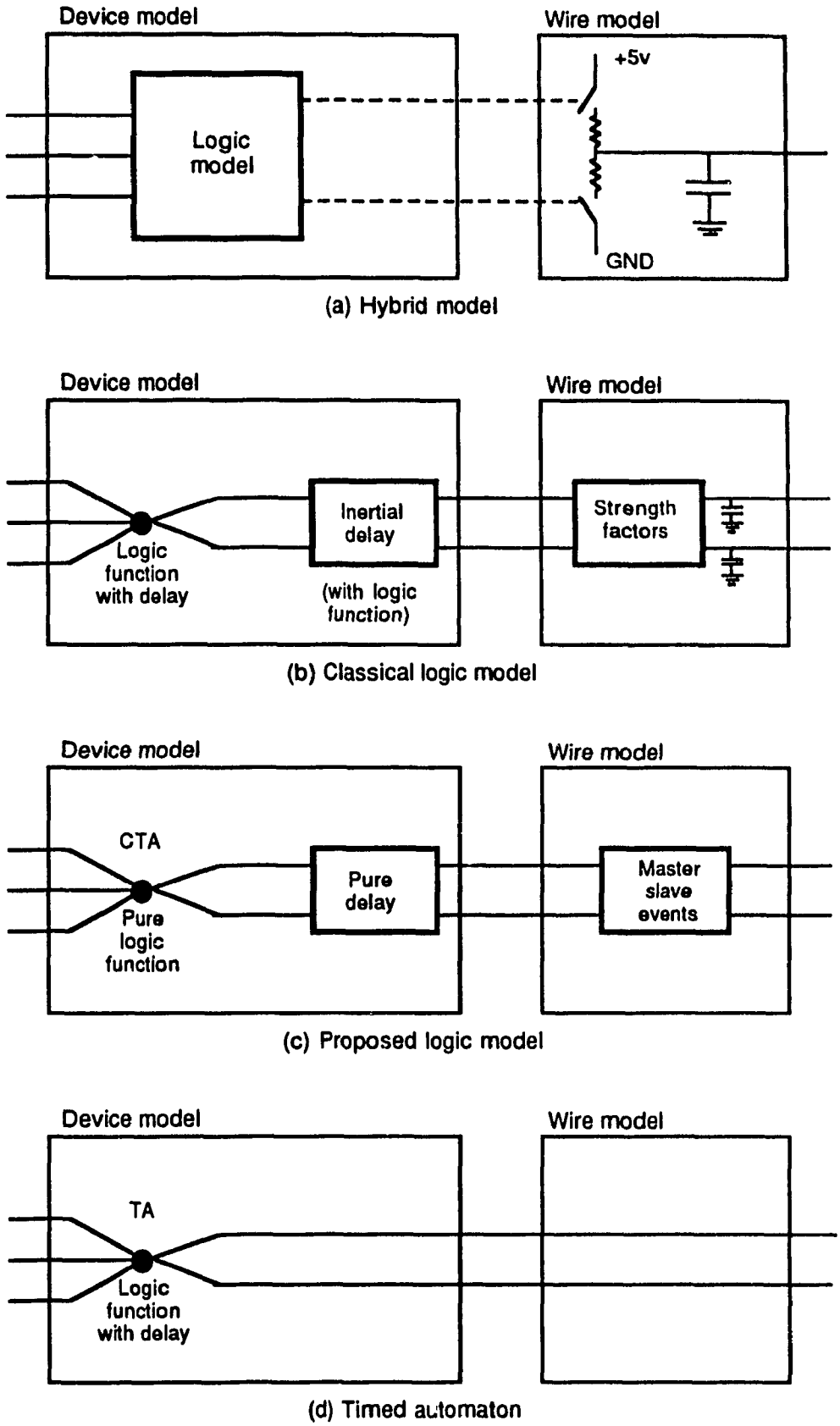


Fig. 5.6 Comparisons of conceptual models

of inertial delay or similar delay models. The conceptual model is ambiguous since some of the delay are part of the logic function and part of the logic function is performed by the propagation delay when combining the events.

The classical logic model is an ad hoc model where the logic function, the propagation delay and the transmission time model have been mixed up. The resulting model is inaccurate for close events. It does not preserve continuity, does not preserve the timing relationships between events from different logic statements and is far from a state based. In that respect, the CTA based logic model with master slave events is far superior as it elegantly and properly separates the modeling of state and the logic function from the propagation delay and from the transmission time.

Proposed logic model

The proposed conceptual model is shown in Fig. 5.6c. The three modeling objectives are met. The concept of master slave events models the transmission of information between devices for time invariant loads. The continuity preserving delay model is a pure delay model in the sense that it only delays the events without removing any. And the CTA integrates the timing constraints and the logic function and does not include any delay. Finally, the CTA has been designed to transform the continuous change of state into a timed state sequence. This model clearly separates the logic function, the propagation delay and the modeling of the transmission of information between devices. If time varying loads are used, the model can easily be interfaced with analog models as in hybrid simulation.

Timed automaton model

The conceptual model of the TA is shown in Fig. 5.6d. All aspects are included in a single state machine. As demonstrated in section 4.2, there are two problems associated with TA. First, combining the timing constraints, the propagation delays and the logic function makes TA difficult to use. Second, TA are not meant to model

the fine timing details associated with continuous devices. TAs are designed to formally describe the high level behavior of real time system. TAs alone are not adequate for low level modeling. Transmission of information between devices can not be accurately handled in a TA. Continuous function can not be used to model the propagation delays. And TAs are difficult to use to integrate timing constraints and generate a timed state sequence.

5.3.2 Signal representation

For any level of modeling, from analog to behavioral, we ought to adequately model the transfer of information from one component to another. Two cases must be considered: time varying loads and time invariant loads. In the general case of time varying load, it is necessary to resolve the Kirchhoff Laws and compute the voltages and the currents. In the case of time invariant load, the waveform of the signal can be computed with "logic level" accuracy for most output ports without solving the Kirchhoff laws. In this section, various signal representations will be analysed with respect to the objective about the modeling of the transfer of information between logic devices.

Analog simulator

At analog level, the transfer of information is accurately modeled, at least compared to a logic model. Analog models and Kirchhoff laws are used to accurately compute the voltages and the currents between devices.

Hybrid simulator: Verilog and SILOS (Time varying load)

Verilog [48] and SILOS [29] support logic models with a simplified analog model of the interconnections. This type of model is often used for dynamic MOS integrated circuits to model network of switches and to compute signal contention and node voltage decay. Each port is modeled using driving strength (supply, strong, pull and weak) and each node is characterized by a charge storage strength. In Verilog,

the charge storage strength is large, medium or small and in SILOS, the actual capacitance value is used.

These are simplified analog simulators for logic circuits with time varying loads. If only time invariant loads are used, this technique is needlessly complex. For instance, digital system designers use higher level (time invariant load) logic models [43-45].

Mixed-mode simulation: PSPICE and ELOGIC (Time varying load)

When it is required to connect a logic model to an analog model, special interface circuits are used. The logic states (0, 1, U or Z) are converted to analog signal using Boolean-Controlled Switches (BCS) and resistances. The circuit used by PSPICE is shown in Fig. 5.7. An improved version of this circuit called the Voltage-Controlled Switch (VCS) has been used in ELOGIC [51]. In the VCS, the sources and the impedances depend on the actual output voltage. In these mixed model simulators, the information is adequately transferred to the analog part of the circuit, but the problem remains the same within the logic part of the simulator.

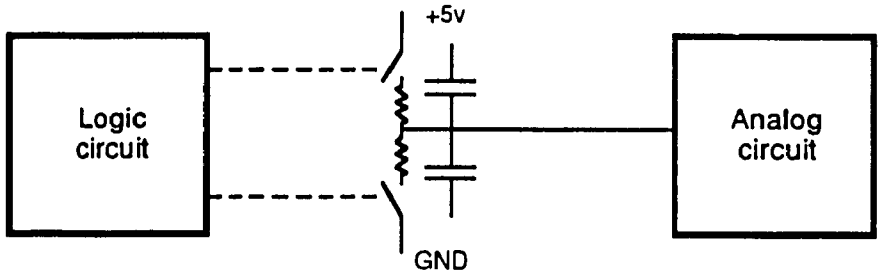


Fig. 5.7 Mixed mode simulation: Typical D/A interface circuit

Current-limited switch

An interesting approach [25] has been used in a switch level simulator where each transistor are replaced by a switch controlled current source. The basic model is shown in Fig. 5.8. Each node is driven by current sources and the resulting voltage is piecewise linear and is computed without numerical integration. This technique

directly leads to master slave events in the case of a unique source driving a time invariant load. Such a model could be interfaced easily with the proposed modeling technique to handle time varying loads.

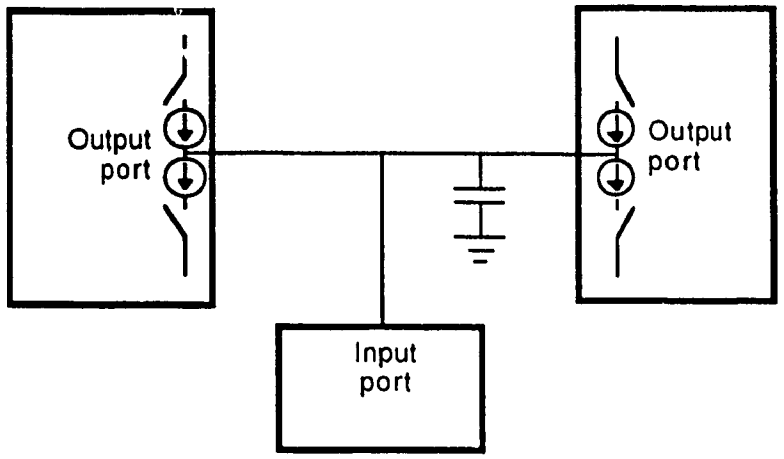


Fig. 5.8 Current limited switch model

Classic logic models (time invariant load)

Most logic simulators only generate a logic signal based on thresholds. As discussed in section 1.6.1, the transmission of information between logic devices is not accurate because the input and output events have the same definition. We have demonstrated that by using output events to describe analog signals and input events to describe the crossing of input thresholds, the transfer of information is adequately modeled. It is wrong to patch a logic model in order to take into account the effect of wire capacitances as in inertial delay. The concept of master slave events does it simply and elegantly.

Mixed-mode simulation: SAMSON (time invariant load)

In SAMSON [30], a formalism is used to model the information on the signal. The model incorporates threshold crossing, minima and maxima. A typical signal is shown in Fig. 5.9. The signal states are Rising, Falling, Low and High. There are two problems with this approach. First, the input and output events have the same

definitions leading to problems similar to those described in section 1.6.1, and second, there is redundant information leading to inefficiencies in the simulation. Master/slave events can describe the signal of Fig. 5.9 using only two events (0 and 1), thus leading to much simpler logic functions and propagation delay models.

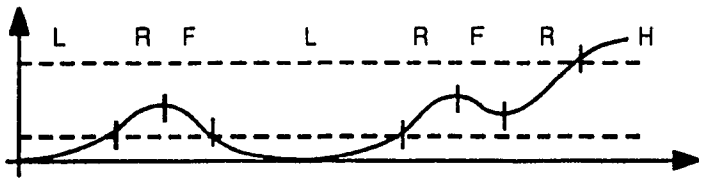


Fig. 5.9 Signal model in SAMSON

Summary

Many acceptable techniques have been developed to model the transfer of information in analog simulation, in switch simulation and between logic models and analog models. Unfortunately, when modeling the transfer of information between logic devices, threshold based events are alway used even when they do not work well. To our knowledge the concept of master slave events where output and input events are separated is the first two-valued (single threshold) defininition that adequately models the transfer of information between logic devices.

5.3.3 Propagation delay models

The objective of a delay model is to accurately model the propagation time of a device. In multiple inputs and multiple outputs devices, propagation delay exists in each input-output pair. In practice, it is assumed that input events are merged at some space point and that the delays are separated between input and output delays. Furthermore, it is assumed that the input delays are zero, leading to conceptual model of Fig. 1.10. If the input delays are not zero, most simulators will provide the necessary feature to solve the problem. In Verilog [48], MIPD (Module Input Port

Delay) can be used and in VHDL [49], intermediate signals are added with the appropriate delay. This has been described in section 4.7.1.

As shown in Fig. 5.6b, the propagation delay is usually attached to the logic function. For example, here are three typical statements:

G3 .AND/N 9 2.5 7 2.2 3.4* A B 3 SILOS [29]	G3 is declared as an and-gate /N indicates normal strength t_{PLH}=9 and t_{PHL}=7 A and B are the inputs
AND #10 G3 (A, B, OUT) Verilog [48]	G3 is declared as an and-gate #10 is the delay A and B are the inputs
OUTG3 <= A . B after 10n; VHDL [49]	#10 is the delay OUTG3 is the output of G3 A and B are the inputs

In such a case, when two or more statements with different propagation delays control the same output, the delay modeler might need to remove some events to ensure the consistency of the signal. As discussed in section 1.6.2, this has led to delay models such as inertial delay.

With master slave events, it is possible to isolate the delay function and a pure delay model was described in section 3.2. One of the characteristics of this delay model is to associate the delay model to an output port, thus allowing separate specification of delay and logic function, as demonstrated below below:

```
      circuit and2
         interface
            A B:  bit (input  3,0);
                              (VTh)
            OUT:  bit (output  0.5n  0.5n  0    5)
                              (tTLH, tTHL,  VL, VH)
-->                   (delay2 2n     3n);
                              (tPLH, tPHL)
         implementation  gate   (CTA to be used)
            (timing  0.1n)        (Input timing constraints)
                    (ts)
-->         (function (out X (nand A B))) (Logic function)
```

```
end and2
```

In this model, the integrity of the output signal is always maintained since the delay model preserves continuity and events are always processed in the proper chronological order, independent of the actual propagation delays.

5.3.4 Timing constraints and timed state sequence

Chapter 4 has established that timing constraints and undefined states were required to obtain a timed state sequence. Obtaining a timed state sequence is not normally a modeling objective in other simulators. However, it is necessary to handle continuous sequences of events and, as described in section 1.6.3, timing constraints are used. We shall now illustrate how they have been implemented in other simulators and how they differ from our approach.

Timing constraints and processing in Verilog

The Verilog language [48] provides the necessary features to insert event definitions such as @posedge, @negedge and @(event description). Event descriptions may include Boolean expressions and one of the following: r for rise, f for fall or (xy) for a change from x to y. Finally, the following system tasks are available: setup, hold, width, period, skew, recovery, setuphold and nochange. These system tasks are used to verify specific timing constraints. All the features required to generate a timed state sequence are supported and, as in most logic simulators, the timing constraints are only monitored. They do not influence the device state and are not used to generate a timed state sequence.

As shown in chapter 4, timing constraints directly affect the device state. Timing violation should produce undefined state and error recovery should be modeled. The transformation to a timed state sequence is partially obtained through the inertial delay, but it is not adequate since an undefined state is not syste used. In SILOS, a modified inertial delay model called spike delay is used. This model replaces close events with an undefined event but it is still an ad hoc technique.

In conclusion, none of the reviewed simulators provide a systematic technique to process the timing constraints and generate a timed state sequence.

Timing constraints and processing in VHDL

In VHDL, a device model is a process activated by an event arriving at one of the input ports, called sensitivity channel, and deactivated with wait statements like: wait on "signal-list", wait until "condition" or wait for "time-expression". Here are two process examples:

```
and_process process
   begin
      wait on A, B;
      OUT <= transport A and B after 20n;
   end process

or_process  process (A, B)  (Sensitivity list)
   begin
      OUT <= A or B after 20n;
   end process
```

VHDL also provides attributes and assertion statements to help program the detection of timing violations. The following is the model of a flip-flop including a process to check the timing violations:

```
entity TFF is
   generic
      (tPLH, tPHL : Time := 0ns;
       tH,   tL   : Time := 0ns;
       tSU,  tHD  : Time := 0ns);
   port
      (D,C : in    Bit := '0';
       Q   : inout Bit := '0');
   begin
      Check_tSU_tHD_tH_tL:
      process
         if not C'Stable and C='1' then
            assert D'Stable(tSU)
               report "Violation of set-up time"
               severity Warning;
         end if;
         if not D'Stable and C='1' then
            assert C'Stable(tHD)
               report "Violation of hold time"
               severity Warning;
         end if;
         if not C'Stable and C='1' then
            assert C'Stable(tL)
               report "Violation of low time"
```

```
                 severity Warning;
            end if;
            if not C'Stable and C='0' then
               assert C'Stable(tH)
                  report "Violation of high time"
                  severity Warning;
            end if;
         end process;
end tFF;

architecture behavior of TFF  is
   begin
      if C='1'and not C'Quiet then
         if not Q = '0' and D = '0' then
            Q = 0 after tPHL;
         if not Q = '1' and D = '1' then
            Q = 1 after tPLH;
   end BEHAVIOR;
```

Later on, in the circuit description, a device would be defined as follow:

```
G3 : TFF;
   generic map (0.5ns, 0.5ns, 5ns, 5ns, 0.5ns, 0.5ns);
   port map (DataIn, Clock, DataOut);
```

Again timing violation are only detected and not used to generate a timed state sequence.

Timing constraints and processing in HDIL

Here is the corresponding listing of a flip-flop model in HDIL:

```
circuit DFF
   interface
      D :bit (input   2);
                     (VT)
      C :bit (clock   2);
                     (VT)
      Q :bit (output 0.5n  0.5n 0   5)
                    (tTLH, tTHL, VL, VH)
             (delay2 2n    3n);
                    (tPLH, tPHL)
   implementation DFlipFlop    (CTA to be used)
      (timing 5n 5n  0.5n  0n)
             (tL tH  tSU   tHD)
      (function (out Q D))
end DFF
```

The VHDL description of the flip-flop illustrates the flexibility and the expressiveness of VHDL. But it also illustrates the problems associated with the described model. The VHDL description of a D flip-flop is a text book description and

is often used. The HDIL description with the underlying CTA model does more than its VHDL counterpart:

- Timing violations in HDIL are used by the CTA to transform the continuous change of state into a timed state sequence.
- Undefined states and undefined events are processed and timing violation recovery is built in the CTA.
- The logic function is much simpler when the propagation delays and the timing constraints are separately taken care internally.
- HDIL does not need to support any time related attributes or functions like `C'Quiet`, `C'Stable(tH)`, `wait`, `after` or `transport`. The sensitivity list used in VHDL models is also built in the CTA.

The VHDL model of a D flip-flop including a CTA would be more complicated.

5.3.5 Comparison of the outputs of a typical gate

The Fig. 5.10 shows the output signals of a typical gate using different logic models. The signal A is the input signal of a simple buffer and the signal B is the actual output signal. The transmission line delay model simply delays the input signal and generates the signal C. This signal does not resemble the actual signal. An inertial delay model would produce the signal D if the distance between the transitions is smaller than the propagation delay. Again this model does not give any indication of the ringing problem of the actual signal. With the CTA logic model, the signal E clearly indicates the problem by replacing some events by an undefined envelope. Finally, by using master events and waveform information the signal F is produced. This signal quite accurately provides the useful information associated with the actual signal.

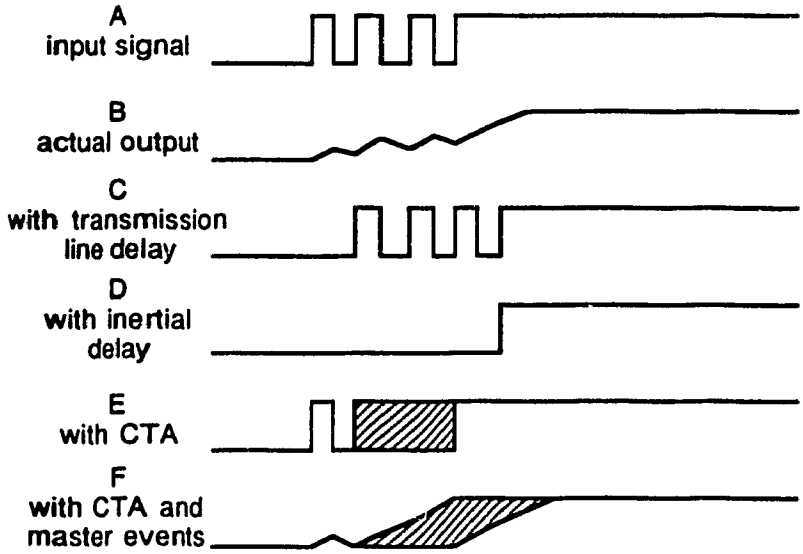


Fig. 5.10 Comparing various gate models

5.3.6 Adequacy with engineering practice

One of the most interesting characteristics of the HDIL-CTA combination is the correspondance with the engineering practice. The proposed model seems to be a simple formalization of modeling technique used by engineers. For example, the formalism used justifies:

- the need for all the information in data books,
- the need of transition times in logic models, as used by engineer for decades,
- the intuitive undefined state for unacceptable input sequences,
- the use of undefined events
- the separation of the logic function from the timing specification, as it is done in data books.

Modeling many devices

By separating the specification of propagation delays and of timing constraints from the logic function, the problem of modeling logic devices is greatly simplified. In

addition, it appears that only a few CTAs are needed in order to model useful devices. Up to now, we have developed CTAs for:

- gates
- combinational circuits (ROM, PLA)
- 3-states devices
- edge triggered flip-flops
- synchronous sequential devices
- synchronous sequential devices with asynchronous inputs

The construction of logic models for most devices consists in using or combining one or more of the above CTAs and adding the logic function. Although we have not done so, it would be interesting to construct the CTAs for devices such as:

- asynchronous sequential devices
- communicating sequential processes and communication interfaces [65]
- serial interface, such as RS-232
- parallel interface, such as VME
- neurons [66]
- self-timed systems [67]
- delay insensitive systems [68]
- Q-modules [69]

A CTA is therefore used for a class of devices that possess the same timing behavior. This class of devices can include complex devices. The CTA of a CSS is currently being used to model the 68000 microprocessor [70].

Model compatibility

The design of digital systems now requires the use of a variety of CAD tools. At logic level, it is essential that the interpretation of the specification of a system or a device be precise. There is incompatibility if the specification is incomplete or its interpretation ambiguous. Incompatibilities reported in VHDL logic models [32] are caused by logic level definition, logic event definition, signal strength, interpretation of timing specification and handling of unknown operating conditions. Unfortunately, the

only solution to the compatibility problem is to establish a standard for logic models. Here is a list of the features that should make the proposed event definition and the CTA acceptable candidates for standardization.

- The underlying formalism.
- Simple definitions for propagation delays and timing constraints.
- Simple two-valued logic (single threshold events).
- Glitches are both described and processed.
- Capability to mix different logic families and analog circuits.
- Clearly formulated mechanism (CTA) to transform continuous change of state into a timed state sequence.
- Small set of primitives (CTA).
- Built-in processing of timing constraints hidden from the end user.
- Separate specifications for propagation delays, timing constraints and logic function.
- Systematic processing of unknown conditions.
- Acceptable performances (accuracy, speed, memory requirements).
- Model parameters similar to the one in use today.

5.5 Conclusion

If we consider the logic level of modeling abstraction as defined in chapter 1, the combination of master slave events, continuous time automaton and continuity preserving delay model is a more formal and rigorous approach than the classical logic models. Inertial and similar delay models and logic models only monitoring timing constraints are patches added to the logic function that bears little resemblance with the continuous, causal and time invariant system they want to model. TA is a neat high level formalism but it is not adequate for modeling continuous causal and time invariant systems if fine timing details need be modeled. The proposed modeling

technique is a better and more accurate model than classical logic models and TA and is easy to use. A summary of its key points follows.

Accuracy

- Because of the use of mater slave events, the timing accuracy in modeling the transmission of information between devices is better than logic models with and without strength factors.
- Timing accuracy in the device model is similar to the timing accuracy in other logic simulator since the same parameters are used.
- Close events and the resulting behavior are more accurately modeled with CTA and continuity preserving delay model than with other delay models including TA.
- The correctness and the completeness in modeling the behavior under timing violation is improved compared with the other logic models and TAs. Although it is possible to construct adequate models with the actual tools, it has never been done.

Speed

- The simulation speed is slightly reduced compared to classical logic simulators. The processing of master slave events is comparable with the processing of strenght factors.
- The processing of timing constraints are as efficiently handled with a CTA than with monitor functions or TA.
- The processing time of logic functions is the same in all simulators and depends on the implementation.

Ease of use

- The logic model parameters used are similar to the parameters used by engineers and compiled in data books.
- The small number of primitives built in the simulator simplifies their validation.

- Factorization of the propagation delay, the timing constraints and the logic function greatly simplifies the construction of logic models for many parts.

In conclusion, master slave events, continuous time automata and continuity preserving delay models represent an interesting alternative for classic threshold based logic models. The resulting logic model is more accurate than classic models in modeling fine timing details and we believe that the more rigourous model proposed should be better in the long term for the specification, the analysis, the formal verification and the testing of digital systems. In this chapter we have tried to demonstrate that using the proposed modeling technique is practical and advantageous compared to current modeling techniques.

Chapter 6

Conclusion

In this thesis we have studied the problems associated with the modeling and the simulation of digital systems. Both theroetical and practical aspects have been considered and a new modeling technique based on master slave events and continuous time automaton is proposed. The new modeling technique neatly integrates:

- the transmission time between logic devices
- the processing time (propagation delays) and
- the timed state sequence.

We shall now conclude by reviewing the results and suggesting new research directions.

6.1 Summary

A digital system is a continuous, causal and time invariant system and preserving these properties while constructing logic models is necessary to model fine timing details. The problem of modeling digital systems begins with the definition of a logic circuit and then the description of a logic model for it.

The three hypotheses about input port, output port (logic signal) and logic network describe how to partition an analog circuit into logic devices. In short, a logic device is sensitive to threshold crossing. A logic signal neatly traverses the threshold and a logic network consists of logic devices connected with ideal wires carrying logic signals. In order to avoid using Kirchoff's laws in the analysis or the simulation of logic circuits, it is necessary to assume time invariant loads. According to these

hypotheses, time varying loads, like switch networks, are analog circuits and have not been considered.

Fig. 6.1 summarizes the key ideas proposed here for the construction of logic models. First, analog signals are transformed into sequences of slave logic events using input thresholds. Input events are then combined and processed using a CTA. Processing of the events consists in using timing constraints to transform continuous change of state into a timed state sequence and generate master output events as required. Then, for each output signal, a continuity-preserving delay model is used to generate the sequences of output master events which are used to reconstruct the analog signals.

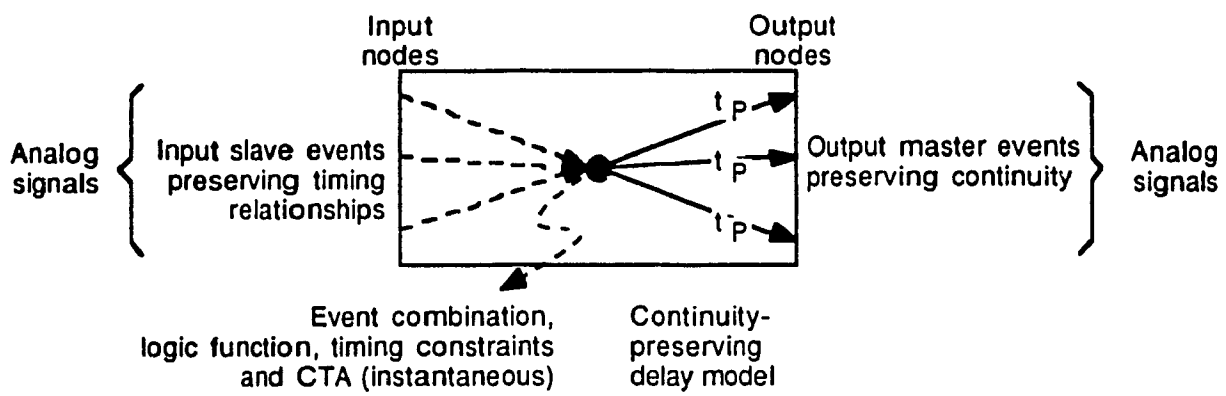


Fig. 6.1 Typical logic model for a logic device

On the surface, this model might seem very similar to the models used in existing logic simulators, but because of the rigorous abstraction process that preserves the relation with the physical reality, it differs on many important aspects:

- Use of analog signals is mandatory in logic simulation to model the fine timing details associated with continuous systems like glitches.
- Use of distinct definitions for output and input events is also necessary. Output events are used to describe the analog signals while the input

events are used to accurately model the event arrival time at each input node.

- The transformation of continuous change of state into timed state sequence absolutely requires input timing constraints and undefined state. Thus the processing of undefined events is mandatory.
- CTA seems the most appropriate tool to transform continuous time into discrete time. It neatly integrates the timing constraints with the logic function while hiding their processing from the end user.
- A continuity-preserving delay model is preferable to inertial and transport delay models to properly describe close events on an analog signal.

6.2 Results

The usefulness of the above theoretical results has been demonstrated through a prototype simulator. Use of signal based master slave events and CTA resulted in a simple and efficient 2-valued (single threshold) logic simulator capable of processing glitches. The CTA hides the processing of timing constraints and logic function from the end user, thus simplifying the construction of logic models. Few CTAs are required to model most logic devices making it an interesting candidate for primitive models in VHDL. Because of its sound theoretical basis, CTA based logic simulators clearly outperform threshold based simulators using adhoc timing constraints by providing a simple and accurate solution to most modeling problems. To our knowledge no other logic simulator preserves the continuity in the delay model and the signal model while using a simple mechanism to perform discrete time abstraction. The resulting simulator is comparable to logic simulators using transition times as far as memory requirements, speed and timing accuracy are concerned. This modeling technique provides a more accurate logic signal representation, a better model for the physical phenomenon associated with propagation delay and a systematic method to

specify and process input timing constraints. It is also easier to use than other logic modeling techniques and it closely matches the enginering approach. On the negative side, the modeling technique is limited to unidirectional signal. However it still permits the use of tri-state busses. Because it accepts and generates analog signals, the proposed model easily interfaces with other circuit models in mixed mode simulation. It was also demonstrated that CTA based models are easily extracted and should reduce the compatibility problems in modern CAD systems.

6.3 Future work

We hope that we have provided enough evidence of the importance of the theory presented here and enough hints on how to use it to allow the continuation of this work elsewhere. Both theoreticians and practitioners could benefit from using our approach which has a clear abstraction that bears a close resemblance to the physical reality, in particular, the causality among events and their timing relationships.

The master slave event concept is technology independent, produces simpler timing specifications and is therefore more appropriate than threshold based events for the specification of logic devices. Designers and manufacturers of logic devices, standard cells or discrete devices, will benefit from using master slave events.

Even if model extraction, mixed mode simulation, worst-case specification and logic simulation are directly affected, the most interesting avenue is the application of CTA to other devices such as: microprocessor, RAM, serial busses (RS-232), parallel busses (VME), communication protocol interfaces, communicating sequential processes, delay insensitive devices, networks and more. We are currently using CTAs to model neural networks and the 68000 microprocessor.

Theoretical research in timing analysis and formal verification will certainly benefit from the specification mechanism for logic devices proposed here. For example, use of CTA leads to a timed state sequence which in turn produces timed

constrained sequence of events. It will be interesting to study the relation between timed state sequence and constrained sequence of events. It will also be interesting to study how timed constrained sequences of events are combined depending on whether they are mutually constrained as in SSC or asynchronous systems.

References

General

[1] A. R. Newton, "Computer-aided design of VLSI circuits," *IEEE Proceedings*, Vol. 69, No. 10, pp. 1189-1199, Ocotber 1981.

[2] A. R. Newton and A. L. Sangiovanni-Vincentelli, "CAD tools for ASIC design," *IEEE Proceedings*, Vol. 75, No. 5, pp. 765-776, June 1987.

[3] A. C. Parker and S. Hayati, "Automating the VLSI design process using expert systems and silicon compilation," *IEEE Proceedings*, Vol. 75, No. 6, pp. 777-785, June 1987.

[4] "Special issue on the future of Computer-Aided Design", *IEEE Proceedings*, Vol. 78, No. 2, February 1990.

Device modeling and simulation

[5] "Special issue on numerical modeling of processes and devices for ICs (NUPAD I), *IEEE trans. on CAD*, Vol. 7, No. 2, 1988.

Circuit simulation

[6] L. W. Nagel, "SPICE2:A computer program to simulate semiconductor circuits," ERL memo ERL-M520, University of California, Berkeley, May 1975.

[7] W. T. Weeks, A. J. Jiminez, G. W. Mahoney, D. Mehta, H. Qassemzadeh and T. R. Scott, "Algorithms for ASTAP - A network analysis program," *IEEE Trans. on Circuit Theory*, Vol. 20, No. 11, pp. 628-634, Nov. 1973.

[8] G. D. Hatchel and A. L. Sangiovanni-Vincentelli, "A survey of third-generation simulation techniques," *IEEE Proceedings*, Vol. 69, No. 10, pp. 1264-1280, October 1981.

[9] A. R. Newton and A. L. Sangiovanni-Vincentelli, "Relaxation-based electrical simulation," *IEEE trans. on CAD*, Vol. 3, No. 4, pp. 308-330, October 1984.

[10] B. Hennion and P. Senn, " A new algorithm for third generation circuit simulators: The one-step relaxation method", 22nd Design Automation Conference, 1985.

[11] B. D. Ackland and R. A. Clark, "An event driven timing simulator for MOS VLSI circuits", *IEEE International Conferenece on Computer-Aided Design, ICCAD 89*.

[12] C. Visweswariah and R. A. Rohrer, "Piecewise approximate circuit simulator", *IEEE trans. on CAD*, Vol. 10, No. 7, July 1991.

Switch/circuit simulation

[13] R. Byrd, G.D. Hachtel, M.R. Lightner, and M. Heydeman, "Switch level simulation-models: Theory and algorithms," in *Computer Aided Design*, A. L. Sangiovanni-Vincentelli, Ed. Greenwich, CT: JAI press, 1985.

[14] J. P. Hayes, "A unified switching theory with applications to VLSI design," *IEEE Proceedings*, Vol. 70, No. 10, pp. 1140-1151, October 1982.

[15] D. Adler, "SIMMOS: a multiple-delay switch-level simulator," In the proceedings of the 23rd Design Automation Conference, pp. 159-163, 1986.

[16] Z. Barzilai, D. K. Beece, L. M. Huisman, V. S. Iyengar and G. M. Silberman, "SLS - a fast switch-level simulator," *IEEE Trans. on CAD*, Vol. 7, No. 8, pp. 838-849, August 1988.

[17] C. Roy, L.-P. Demers, E. Cerny and J. Gecsei, "An object-oriented switch-level simulator," In the proceedings of the 22nd Design Automation Conference, pp. 623-629, 1985.

[18] I. Spillinger and G. M. Silberman, "Improving the performance of a switch-level simulator targeted for a logic simulation machine," *IEEE Trans. on CAD*, Vol. 5, No. 3, pp. 396-404, July 1986.

[19] D. Tsao and C.-F. Chen, "A fast-timing simulator for digital MOS circuits," *IEEE trans. on CAD*, Vol. 5, No. 4, 1986.

[20] T. J. Schaefer, "A transistor-level logic-with-timing simulator for MOS circuits," In the proceedings of the 22nd Design Automation Conference, pp. 762-765, 1985.

[21] I. N. Hajj and D. Saab, "Switch-level logic simulation of digital bipolar circuits," *IEEE trans. on CAD*, Vol. 6, No. 2, 1987.

[22] R. Kao, B. Alverson, M. Horowitz and D. Stark, "Bisim: A simulator for Custom ECL Circuits", *IEEE International Conferenece on Computer-Aided Design, ICCAD 88*.

[23] D. G. Saab, A. T. Yang and I. N. Hajj, " Delay Modeling and Timing of Bipolar Digital Circuits", 25th ACM/IEEE Desing Automation Conferentce, 1988.

Macro-model/circuit simulation

[24] M. D. Matson and L. A. Glasser, "Macromodeling and optimization of digital MOS VLSI circuits," *IEEE trans. on CAD*, Vol. 5, No. 4, pp. 659-678, October 1986.

[25] G. Ruan, J. Vlach and J. A. Barby, "Current-limited switch-level timing simulator for MOS logic networks," *IEEE trans. on CAD*, Vol. 7, No. 6, 1988.

[26] L. M. Brocco, S. P. McCormick and J. Allen, "Macromodeling CMOS circuits for timing simulation", *IEEE trans. on CAD*, Vol. 12, No. 12, December 1988.

Logic simulation

[27] "Fundamentals of mathematics, Vol. III-Analysis," Edited by H. Behnke, F. Bachmann, K. Fladt and W. Süss, Translated by S. H. Gould, The MIT press, 1986.

[28] VLSI Systems Design staff, "1987 survey of logic simulator," VLSI Systems Design, February 1987, pp. 71-86.

[29] SILOS II User's manual, SIMUCAD Inc.,Union City, 1990.

[30] K. A. Sakallah, S. W. Director, "SAMSON: A mixed circuit-logic-level simulator," in *Computer Aided Design*, A. L. Sangiovanni-Vincentelli, Ed. Greenwich, CT: JAI press, 1985.

[31] D. R. Coelho, "The VHDL handbook", Kluwer Academic Publishers, Boston.

[32] D. R. Coelho, "VHDL: a call for standards", Proceedings of the 25th Design Automation Conference, 1988.

[33] M. R. Lightner, "Modeling and simulation of VLSI digital system", *Proc. IEEE*, vol. 75, no. 6, pp. 786-796, June 1987.

[34] H.-Y. Chen and S. Dutta, "Timing model for static CMOS gates", IEEE International Conference on Computer-Aided Design (ICCAD-89).

[35] D. Overhauser, I. Hajj, "Tabular macromodeling approach to fast timing simulation including parasitics", IEEE International Conference on Computer-Aided Design (ICCAD-88).

[36] A.-C. Deng, "Piecewise-linear timing delay modeling for digital CMOS circuits", IEEE transactions on circuits and systems, Vol. 35, No. 10, Oct. 1988.

[37] J. P. Caisso, E. Cerny, N. C. Rumin, "Interconnection delays in hierarchical timing simulation", IEEE International Symposium on Circuits and Systems, 1989.

[38] M. Bafleur, J. Buxo, J. P. Teixeirsa, I. C. Teixeira, "A logical timing simulator for CMOS circuits based on an accurate formulation of the propagation delay", European Conference on Circuit Theory and Design, 1989.

[39] C. Zukowski and D.-P. Chen, "Variable reduction in MOS timing model", IEEE International Conference on Computer Design (ICCD-88).

[40] B. P. Ziegler, "Multifacetted modelling and discrete event simulation," Academic Press, 1984.

[41] L. M. Augustin, "Timing models in VAL/VHDL", *IEEE International Conferenece on Computer-Aided Design, ICCAD-89* , Los-Alamitos.

[42] J. P. Hayes, "Digital Simulation with multiple logic values", *IEEE trans. on CAD*, Vol. 5, No. 4, April 1986.

[43] TTL data books, Texas Instruments Inc., 1984-90.

[44] CMOS data book, Motorola Semiconductor Products Inc., 1988.

[45] 2-μm CMOS Standard Cell data book, Texas Instruments Inc., 1986.

[46] J. Wyall et al., "Waveform bounding for VLSI timing," *Proc. IEEE Intl. Conf. on Comp. Des.*, IEEE, New-York, 1983.

[47] R. A. Saleh, A. R. Newton, "Mixed-Mode Simulation", Kluwer Academic Publishers, 1990.

[48] "Verilog-XL reference manual", Vol. 1 & 2, Cadence Design System Inc., 1991.

[49] James R. Armstrong, "Chip level modeling with VHDL", Prentice-Hall, 1989.

[50] R. Lipsett, C. Shaefer, C. Ussery, "VHDL: Hardware Description and Design", Intermetrics Inc., Kluwer Academic Publishers, 1989.

[51] Dwight D. Hill and David R. Coelho, "Multi-level simulation for VLSI design", Kluwer Academic Publishers, 1987.

[52] "PSPICE Circuit Analysis", MicroSim Corp., 1991.

Real time systems

[53] E. A. Emerson, "Temporal and modal logic", In the *Handbook of Theoretical Computer Science* (J. Van Leeuwen, ed.), Volume B, North-Holland, 1990.

[54] R. Alur, T. A. Henzinger, "Real-time logics: Complexity and expressiveness", in *Proc. of the 5th Annual IEEE Symp. on Logic in Computer Science*, 1990.

[55] R. Alur and D. Dill, "Automata For Modeling Real-Time Systems", 17th ICALP, *Lecture notes in Computer Science* 443, Speinger-Verlag, 1990.

[56] J. C. Mitchell, "Logic column 1: Time for logic", Computer Science Department, Standford University.

Model extraction

[57] F.-C. Chang, C.-F. Chen and P. Subramaniam, "An accurate and efficient gate level delay calculator for MOS circuits," In the proceedings of the 25th Design Automation Conference, pp. 282-287, 1988.

[58] M. Boehner, "LOGEX - An automatic logic extractor from transistor to gate level for CMOS technology," In the proceedings of the 25th Design Automation Conference, pp. 517-522, 1988.

[59] S. H. Hwang, Y. H. Kim and A. R. Newton, "An accurate delay modelling technique for switch-level timing verification," In the proceedings of the 23rd Design Automation Conference, pp. 227-233, 1986.

[60] D. G. Saab, A. T. Yang and I. N. Hajj, "Delay modeling and timing of bipolar digital circuits," In the proceedings of the 25th Design Automation Conference, pp. 288-293, 1988.

[61] J. J. Cherry, "Pearl: A CMOS timing analyser," In the proceedings of the 25th Design Automation Conference, pp. 148-153, 1988.

[62] T. Tokuda, K. Okazaki, K. Sakashita, I. Ohkura and T. Enomoto, "Delay-time modeling for ED MOS logic LSI," *IEEE trans. on CAD*, Vol. 2, No. 3, pp. 129-134, July 1983.

[63] R. E. Bryant, "Boolean analysis of MOS circuits," *IEEE trans. on CAD*, Vol. 6, No. 4, pp. 634-649, July 1987.

[64] J. K. Ousterhout, "A switch-level timing verifier for digital MOS VLSI," *IEEE trans. on CAD*, Vol. 4, No. 3, pp. 336-349, July 1985.

Special devices

[65] C. A. R. Hoare, "Communicating Sequential Processes", Prentice-Hall, 1985.

[66] J. J. Hopfield, "Neural Networks and Physical Systems with Emergent Computational Abilities", *Proc. of the National Academy of Sciences*, vol. 79, pp. 2554-2558, 1982.

[67] C. Mead and L. Conway, "Introduction to VLSI systems", pp. 242-261, Addison-Wesley, 1980.

[68] C. E. Molnar, T. P. Fang and F. U. Rosenberg, "Synthesis of delay-insensitive modules," *Proc. 1985 Chapel Hill Conf. VLSI*, Chapel Hill, NC, May 15-17, pp. 67-86, 1985.

[69] F. U. Rosenberg, C. E. Molnar, T. J. Chaney and T. P. Fang, "Q-modules: Internally clocked delay insensitive modules," *IEEE Trans. on computer*, Vol. 37, No. 9, pp. 1005-1018, September 1988.

[70] M68000 microprocessor user's manual, Motorola, Prentice-Hall, 1989.

Appendix A

Binary Decision Theorem (BDT)

Although synchronization of asynchronous inputs in a Synchronous Sequential Network (SSN) is not per se a modeling problem, it is nevertheless an important problem in digital system. It is necessary to understand it in the development of complete models for latches, flip-flops and synchronous sequential circuits. The formal approach used here allowed us to demonstrate a simple theorem about binary decision. The theorem does not change the well known truth about the impossibility of reliable synchronization, but it is simpler and more general than previous metastability theorems.

1.0 Introduction

The synchronization of asynchronous inputs in SSN has haunted digital system designers for decades. It is now a relatively well known problem, but its effects are still underestimated. In asynchronously interacting SSN, the system's reliability depends directly on the synchronizers. The synchronization problem can be resolved at three levels. One can take care of the problem by special design techniques like self-timed systems [1], delay insensitive systems [2] or Q-modules [3]. These approaches assume some communication protocol between circuit elements. The resulting circuits are correct by construction and the synchronization problem is eliminated. The synchronization problem can also be resolved at digital circuit level as seen in [4]. Typical successful circuits for solving the synchronization problem are variations of the stoppable clock [5, 6]. For fixed clock systems, it is demonstrated that the use of redundancy, masking [7] and schmitt trigger [4] to reduce the

probability of synchronization failure is not effective. For the special case of synchronizing two systems with clocks of approximately equal frequency, the problem has been successfully resolved [8]. Finally, the last approach is to design better synchronizers [31].

In studying the synchronization problem, researchers have identified what is believed to be the fundamental problem-maker: metastability. Metastability describes the behavior of a bistable device, like a flip-flop, that sits between two stable states for a prolonged period of time. Since a binary decision is associated with the stable states of the device, metastability leads to indecision. The metastable state of a synchronizer might then cause the failure of the sub-system using the signal, leading to what is called a *synchronization failure*. Synchronization failures might be caused by flip-flops if the set-up and hold time requirements are not met. Although the probability of a device entering a metastable state is extremely small, it is not zero. Therefore, reducing the failure rate is a prime concern in designing synchronizers.

The problem of synchronizing asynchronous inputs has evolved around metastability, so most of the past research has been devoted to it. In the early days, a heuristic approach was used and the hard-to-reproduce problem was merely observed and described. During this time, the metastable behavior was not well understood. For example, it has taken some time for people to accept the fact that perfect synchronization is impossible [9]. That was the dark age of metastability; it was controversial, subtle and elusive. In 1976, Couranz and Wann were the first to shed some light on this problem [10]. They explained the metastable behavior of bistable devices using a SUP (*single unstable pole*) model. Single pole instability is not the only model, nor is it very precise; but it does provide enough insight to predict metastable behavior. Chaney and Molnar [11] have observed sinusoidal metastability which can only be explained using a pair of unstable poles.

Nevertheless, the SUP model has been used by all subsequent authors. Couranz and Wann used a probabilistic model to explain the escape from the metastable state, but a simpler deterministic model was proposed soon afterwards [12]. The deterministic model is now widely accepted and used for the analysis and prediction of the MTBF (mean time between failures), a failure being characterized by an undecided output logic level (in the 'x' range), after a bounded decision time. The understanding of this phenomenon has evolved from the measurement of the metastable behavior[11-13], to simple analytical model, [10, 14-19], and finally to complex optimized models [20-22]. Early work on metastable behavior was focussed at TTL, ECL and tunnel diode flip-flops while most of the recent work has been done on MOS. Even if the problem was well understood, the avoidability of metastable behavior or the possibility of deciding on real value remained unanswered. In 1981, Marino[23], presented theoretical proof of the unavoidability of metastable behavior in bistable devices used as synchronizers.

Since then, better test setups have been used, and more data published on the metastable behavior. One especially interesting set-up was used by Rosenberg and Chaney[24]. They designed a complete IC including the device under test and the test set-up. They have shown that the evaluation of τ, can be done precisely, from external measurements. Basically, τ is a measure of the ability of a synchronizer to escape from the metastability region. Data from various sources have been compiled by Chaney[25]. In his compilation, the SUP model is used and the model parameters for the various synchronizers reviewed are also compared.

In this appendix, the theoretical aspects associated with metastability are reviewed and a new theorem is proposed about the impossibility of taking a binary decision in continuous systems.

2.0 Analysis of the metastability phenomenon

Metastability has always been associated with digital synchronization. In fact, it is a purely analog phenomenon. Metastability is a name given to the behavior of bistable devices under certain driving conditions. In this section it will be defined as such and an example of a typical metastable circuit will be analyzed. Metastability is a general behavior of all systems with two or more stable states. A stable state is the result of an unstable characteristic, in the linear system sense, and a nonlinearity, usually the saturation of a component. A simple linear system definition of metastability is:

> "A system is metastable at a time t_x, if the state of the system at t_x is part of a metastable solution. A *metastable solution* is a state trajectory for a specified input and an initial state which cancels the

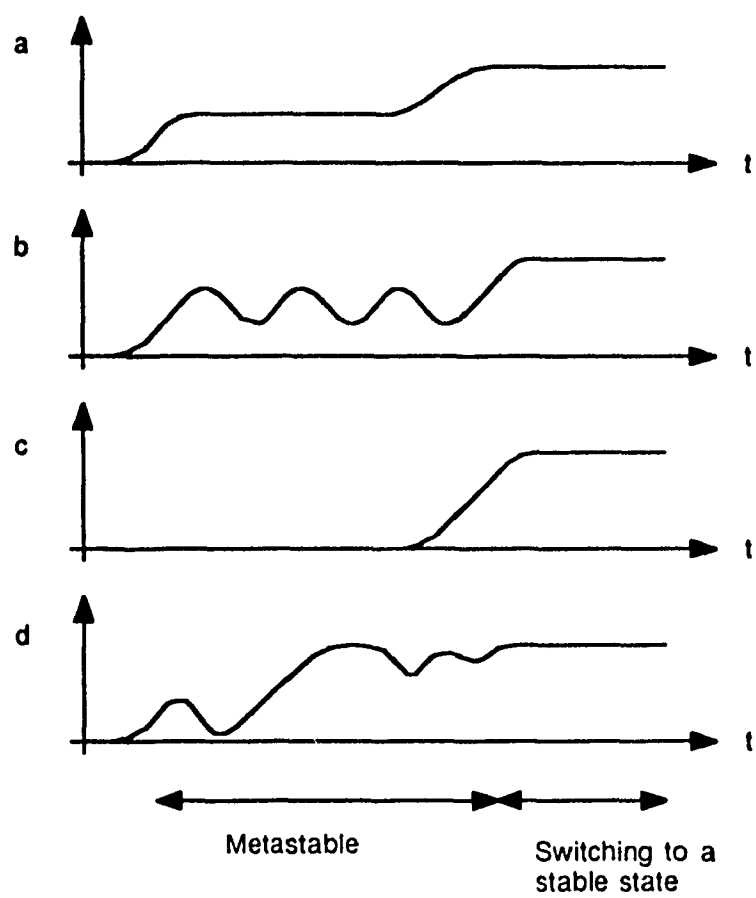


Figure 1. Examples of metastable behavior

unstable poles of the system. The initial state is called the *metastable condition.*"

The metastability phenomenon manifests itself in various forms. Fig. 1 illustrates four types of metastable behavior. The signal of Fig. 1a corresponds to the classic text book example as observed in flip-flops [29]. The signal on Fig. 1b illustrates sinusoidal metastability as described in [11] Metastability is not necessarily undefined, Fig. 1c or simple, Fig. 1d.

A more general definition will be given later. The hysteresis comparator of Fig. 2 is a bistable device and is therefore subject to metastability. Metastable conditions and the corresponding solutions have been computed analytically for three type of inputs: constant, ramp and exponential. Analytical results are given in Table 1, and numerical examples in table 2.

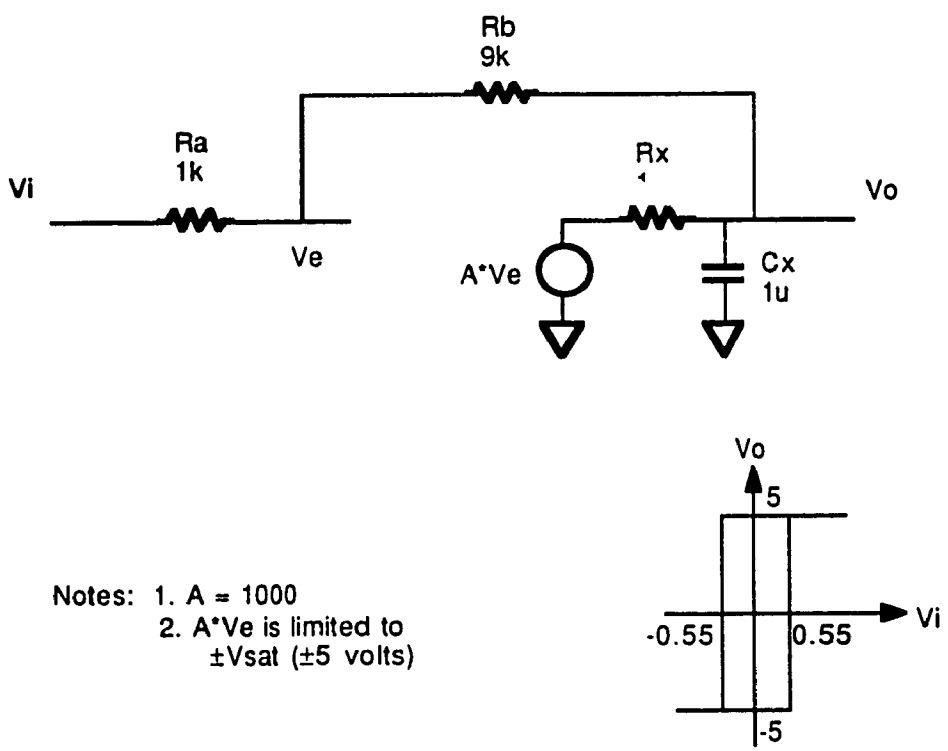


Fig. 2 Hysteresis comparator model and DC characteristic

Input signal $v_i(t)$	V_i	Metastable condition V_{o0} (initial state)	Metastable solution $v_o(t)$	V_o
V_{i0}	$\frac{V_{i0}}{s}$	$\frac{A_i}{1-A_o} V_{i0}$	V_{o0}	$\frac{V_{o0}}{s}$
$V_{i0} + m_i t$	$\frac{V_{i0}}{s} + \frac{m_i}{s^2}$	$A_i \left[\frac{V_{i0}}{1-A_o} - \frac{m_i T}{(1-A_o)^2} \right]$ $m_o = \frac{A_i}{1-A_o} m_i$	$V_{o0} + m_o t$	$\frac{V_{o0}}{s} + \frac{m_o}{s^2}$
$V_{i0}\, e^{-t/\tau}$	$\frac{V_{i0}}{s+\frac{1}{\tau}}$	$\frac{A_i V_{i0}}{1-A_o-\frac{T}{\tau}}$	$V_{o0}\, e^{-t/\tau}$	$\frac{V_{o0}}{s+\frac{1}{\tau}}$

Table 1. Metastability for constant, ramp and exponential inputs

Input signal $v_i(t)$	V_i	Metastable condition V_{o0} (initial state)	Metastable solution $v_o(t)$	V_o
0.1	$\frac{0.1}{s}$	$-\frac{10}{11}$	V_{o0}	$\frac{V_{o0}}{s}$
0.2 -0.1t	$\frac{0.1}{s} - \frac{0.1}{s^2}$	$-\frac{10}{11} \times \frac{197}{99}$ $\quad m_o = \frac{10}{11}$	$V_{o0} + m_o t$	$\frac{V_{o0}}{s} + \frac{m_o}{s^2}$
$0.2\, e^{-t}$	$\frac{0.1}{s+1}$	-1.8	$V_{o0}\, e^{-t}$	$\frac{V_{o0}}{s+1}$

Table 2 Numerical results, simple waveforms

The input signals in Table 2 produce simple linear metastable solutions that can be easily predicted. In fact, metastability can be very complex to analyze. To illustrate this, the hysteresis comparator of Fig. 2 has been exercised using a more complex waveform. A simple simulation has given the results of Table 3 and Fig. 3. In this simulation, the hysteresis comparator rises in a ramp, saturates and

desaturates. Note that at 1.6usec it gets out of metastability, because of the limited accuracy (16 digits) of the simulation.

Input signal $v_i(t)$	Metastability condition V_{o0} (initial state)	Metastable solution $v_o(t)$ V_o
0.5 -0.5t for $0 < t < 5.5/7$	$-\frac{10}{11} \times \frac{488}{99} + 1.484204884E-6$	See Fig. 11
-0.05 for $t > 5.5/7$		

Table 3 Numerical results, complex waveform

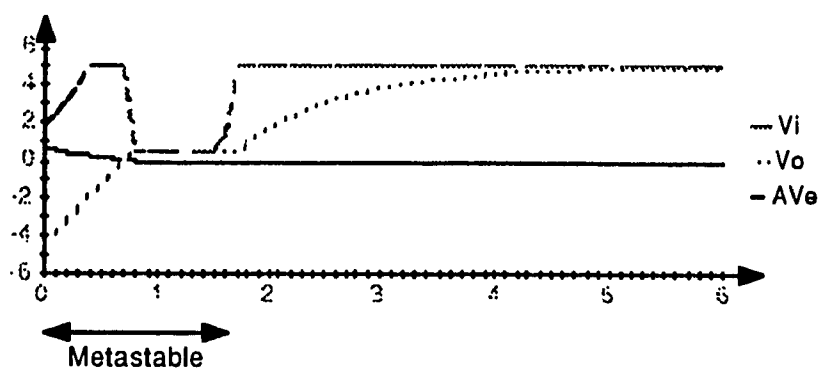


Fig. 3. Metastability of complex waveform

This forces us to define metastability in more general terms.

Metastability

"A system S is metastable at a time t_x, if the state of the system at t_x is part of a metastable solution. A *metastable solution* is a state trajectory for a specified input and initial state leading, at $t \rightarrow \infty$, to the output being in the x-range. The initial state is the *metastable condition*."

3.0 Practical problem description

A major design problem for synchronizers is the reduction of the failure rate. We will first explain the design requirements of a synchronizer and the notion of synchronization failure. The problem definition given here is based on a generally accepted behavioral model and follows the idea proposed by Marchegay [26]. A typical synchronizer is shown in Fig. 4.

There are two inputs, the *clock* C and the *asynchronous signal* D_a, and one output, the *synchronized signal* D_s. The two input signals interact asynchronously modelled roughly by operating frequencies f_c and f_d. The synchronizer is expected to

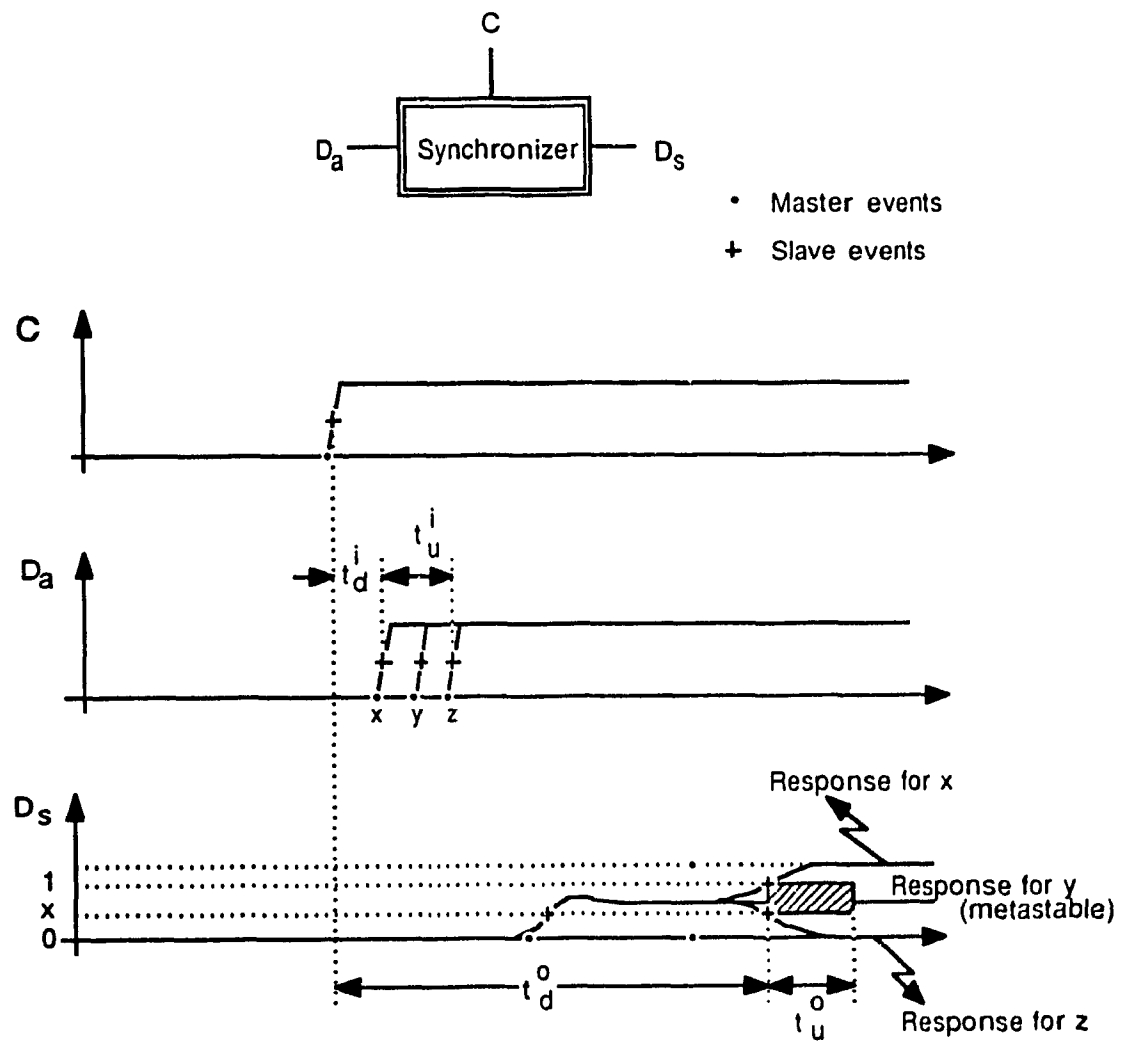


Fig. 4. Synchronization problem

provide a defined logic level (0 or 1), within a bounded decision time after D_a is sampled (clocked); otherwise, a *synchronization failure* has occurred. Fig. 4 illustrates the output signal for three different input timings. During the decision period, the signal is allowed to have any value, but it must settle to a known logic level afterwards. The middle waveform illustrates a synchronization failure.

3.1 Uncertainty interval and delay

In the following discussion, the slave events are used for inputs and a range of slave events are used for the output to model the devices that might use the synchronized signal. Two useful parameters associated with logic signals in synchronous systems are now defined. For any digital signal in a synchronous sequential system, there is a time interval during which an undefined signal (value in the x range) might cause a failure in the device using the signal. The delay from the system clock to the beginning of this interval is defined as the *uncertainty delay*, t_d, and the length of the interval is the *uncertainty interval*, t_u. In the case of a synchronizer, there are uncertainty delays and uncertainty intervals at the input, D_a, and at the output, D_s, respectively labelled with t_d^i, t_u^i, t_d^o and t_u^o in Fig. 4. At the output of a synchronizer, the uncertainty delay and uncertainty interval represent the interval during which use of D_s by a receiver may cause a failure. The uncertainty delay is often called the decision time, since it corresponds to the time allowed for the synchronizer to decide on the input signal. At the input, the uncertainty interval represents the interval during which synchronizer failure may occur. Of course, the output uncertainty interval, t_u^o, and delay, t_d^o, and the input uncertainty interval, t_u^i, and delay, t_d^i, are inter-related.

3.2 Coherence fault

When a synchronous signal is used by more than one sub-system, as exemplified in Fig. 5, the uncertainty interval of this signal depends on the uncertainty

intervals of all the sub-systems. A *coherence fault* may arise when the receiving subsystems do not see the same logic value. To avoid such a fault, the equivalent uncertainty interval and delay for the signal D_s can be derived:

$$t_d^o = \min_j \left[t_{dj}^i \right] \tag{1}$$

$$t_u^o = \max_j \left[t_{dj}^i + t_{uj}^i \right] - t_d^o \tag{2}$$

where t_{dj}^i and t_{uj}^i are the input uncertainty delay and interval for subsystem j.

In summary, to avoid synchronization failure, the synchronizer ensures that the output signal never traverses the undefined x region during the uncertainty interval.

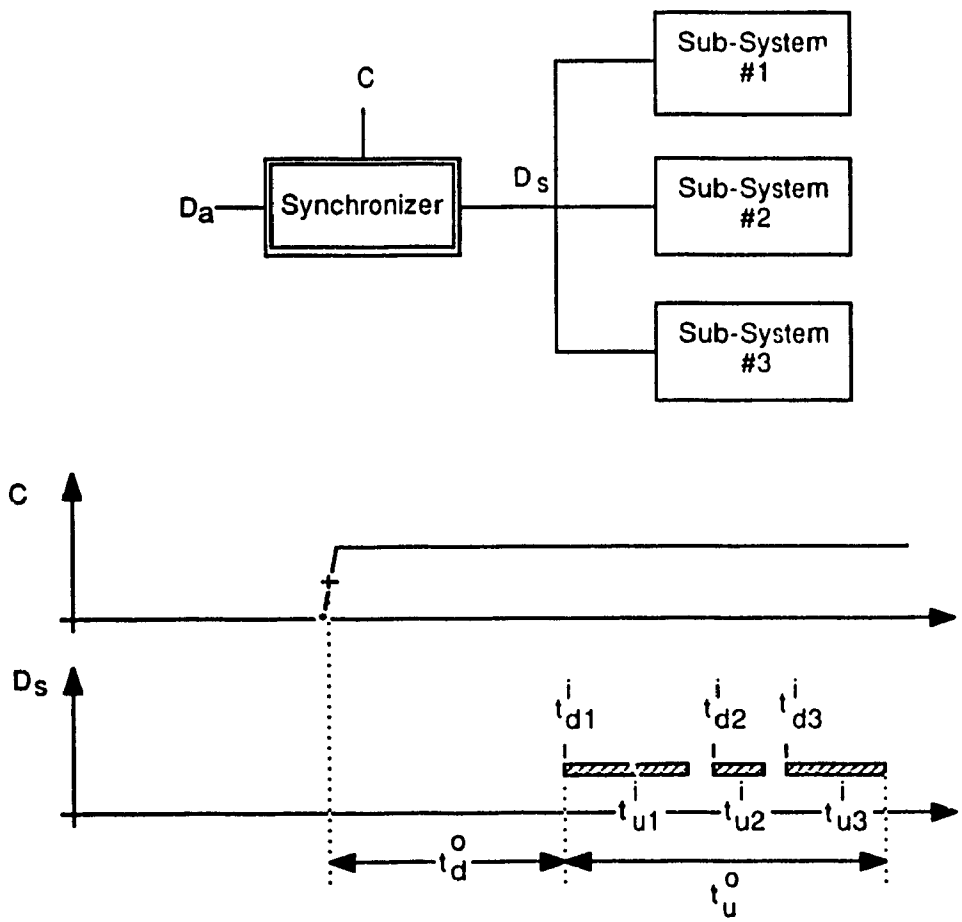


Fig. 5. Coherence faults

3.3 Mean time between failure

The MTBF of a synchronizer is given by [69]:

$$\mathrm{MTBF} = \frac{1}{t_u^i f_c f_d} \tag{3}$$

Hence the reliability of a synchronizer depends on the frequency of interactions $f_c f_d$ and the input uncertainty interval t_u^i. The obvious approach to improve the MTBF, is to reduce the input uncertainty interval of the synchronizer, t_u^i.

4.0 Theoretical problem: Unavoidability of metastable behavior

An important theoretical question is: "Can the probability of failure of a synchronizer be reduced to zero?" Many researchers had suggested that metastability could not be avoided, but it took some time before convincing proofs were published. The problem was studied in detail by Hurtado and Elliot [27], Marino [23] and Kleeman and Cantoni [28]. Marino proved that metastable behavior is unavoidable when bistable devices, such as flip-flops, are used as synchronizers. Marino's hypotheses and conclusions are reviewed in this section. Marino first assumes that the decision element satisfies four axioms regarding the system operation. These axioms together state that the decision element is a continuous, causal and time invariant system and that it possesses states which are sufficient to predict future behavior. In addition there are four important hypotheses, restricting the type of system to which his conclusions apply.

H1. The decision element is a bistable device (or multi-stable), like a flip-flop

H2. For the decision element, there exists a *common input idle range*. An input idle range is a range on the input signal(s) such that if the input(s) varies within it, the logic state will not change. It is common if the same range is valid for both logic states. For example, if the inputs of a SR-

latch are in the range from 0 to 0.8volt, then the latch will keep its state, this is a common input idle range since the range is valid for both logic states.

H3. The decision element is driven by a family of switching waveforms, for example, a set of pulses of varying width, generated by different asynchronous input timing.

H4. Some members of the family of switching waveforms induce a '0' decision and others a '1' decision. For example, short pulses don't set the latch and large pulses do.

In addition, Marino bases his analysis and proof on evaluating the state of the decision element rather than the output signal. It is assumed that the state is representative of the decision. The basic conclusion is that the logic state could always be undefined for any period of time. Even if these conclusions are useful in some areas, they cannot be generalized to the binary decision problem. It will be demonstrated that there exist decision elements using bistable devices not meeting the above hypotheses.

Under these conditions, it was proven that metastability could not be avoided. The assertion is insufficient, as it does not address the overall synchronization problem. Moreover, it only covers bistable devices (multi-stable indirectly), assumes the existence of a common idle range and uses a specific family of waveforms. These hypotheses weaken the result significantly. Kleeman and Cantoni [74] have generalized H3 to all waveforms having the topological property of connectivity, but this is still insufficient.

Counter-example

Here is an example where Marino's proof does not apply and where a more general model is required. In the example shown in Fig. 6, Marino's hypotheses are not met and yet it is a digital synchronizer built around a bistable device. The circuit

consists of an analog sampler, a waveform control circuit and an hysteresis comparator as a decision element. The capacitor is reset to zero before a sample and the family of waveforms that drive the decision element are shown on Fig. 6 This circuit is obviously capable of decision, since for certain sampled values the output is -5v (L_0) and for some others it will be +5v (L_1).

The hypothesis H2 concerning the common input idle range cannot be applied. No common input idle range to the synchronizer can be defined. The family of waveforms is simply not composed of switching waveforms like those in Marino's

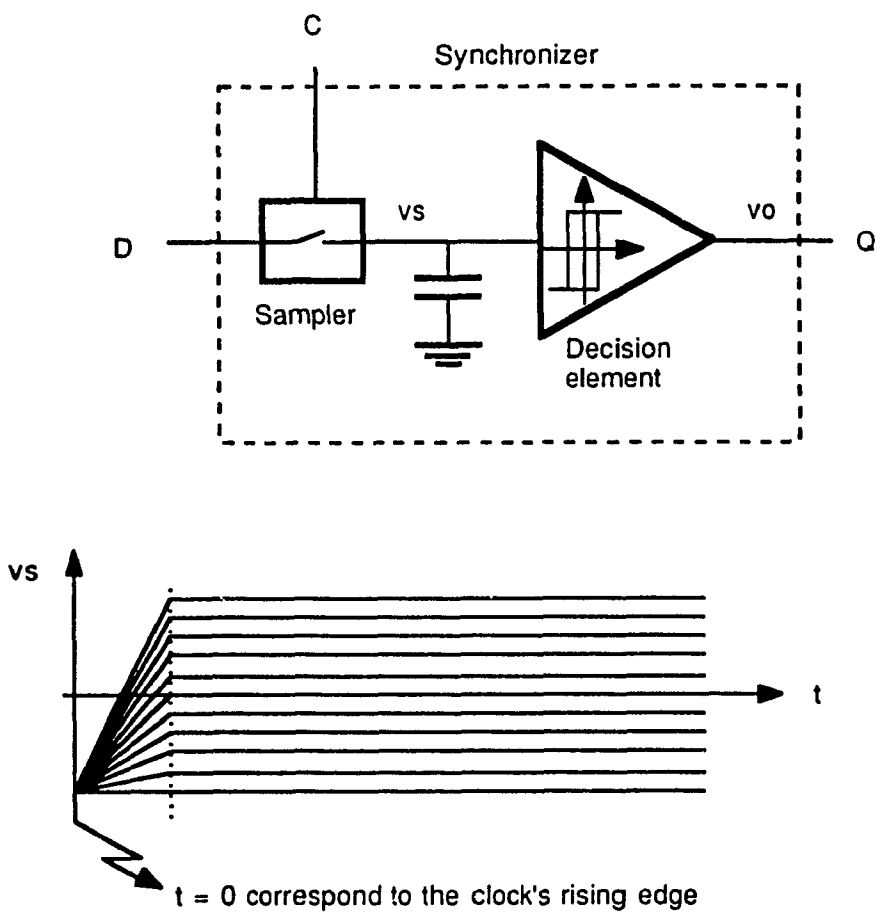


Fig. 6. Example of a synchronizer not meeting Marino's hypothesis

paper. In addition, the metastable solution for this family of waveform can not be distinguished from the normal logic state. Therefore Marino's theorem does not apply.

More formally, Marino has first defined a region of attraction, $A(L_i, \bar{u})$, for each logic state L_i, as the states for which any input $\bar{u}$ eventually bring the system to the state L_i. Marino has then defined a region of indecision (RID) as the set of system states Σ, minus the sets $A(L_i, \bar{u})$.

$$RID(L_0, L_1, \bar{u}) = \Sigma - (A(L_0, \bar{u}) \cup A(L_1, \bar{u})) \qquad (4)$$

Or in simple terms, the above equation divides the initial state space Σ of the decision element in three parts. For the stable state 0, there is a region of attraction, the set of states called $A(L_0, \bar{u})$; similarly, for the state 1, there is a set of states called $A(L_1, \bar{u})$. The region of indecision consist in all the states being in neither set. In our example, it happens that the region of indecision (metastability) is the same as the region of attraction toward 0. Thus $RID(L_0, L_1, \bar{u}) \in A(L_0, \bar{u})$, which contradicts equation (4).

5.0 Binary decision theorem (BDT)

It was just demonstrated that Marino's conclusions do not apply to all systems and an example of a circuit not meeting all hypotheses was given. Our approach is to construct the class of binary decision problems, and prove that the class cannot be solved. Then instances from the class are examined: decision on real value, synchronization and existence of metastability.

In digital systems, binary decisions are often required, for example in the synchronization of asynchronous inputs in synchronous systems or in arbiter circuits. A binary decision element is a device that possesses two stable states corresponding to the decision outcome. In general, the output of a decision element indicates the decision by taking the appropriate value (0 or 1). The binary decision problem is:

Given some input parameter on which a decision must be made, a binary decision element and an output variable to indicate the decision, a binary decision is obtained if the output variable range is such that for some values of the parameter the range of the variable is R_0 and for the other values the range is R_1 and that the ranges R_0 and R_1 do not overlap, even at the boundaries. The question is: Can we construct a binary decision element?

The Binary Decision Theorem (BDT) proves that if the parameter is continuous and if the decision element is a continuous, causal and time invariant system, then indecision is always possible.

Since all devices are continuous, causal and time invariant, the binary decision problem will be studied with the model shown in Fig. 7. The system S represents the decision element. It is driven by the input signal $u_p(t)$ and the decision appears on the output signal $v_p(t)$.

Assumptions: 1. S is a continuous, causal and time invariant system as defined in [23].

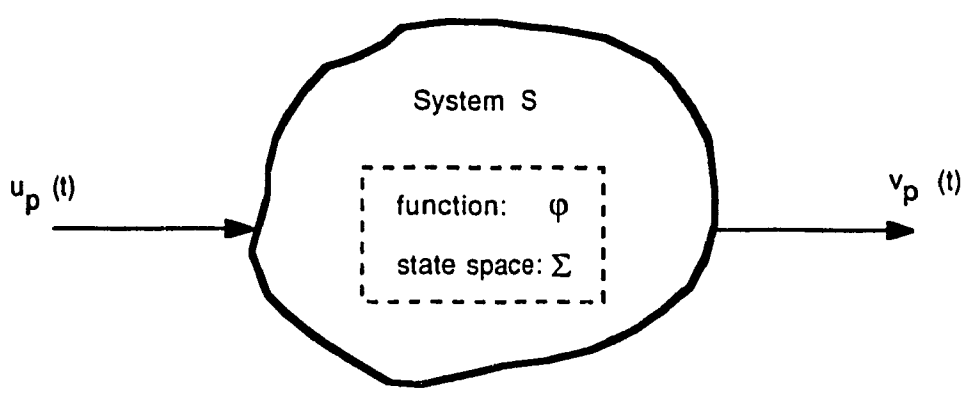


Fig. 7. Logic device

2. $u_p(t)$ is a function dependent on the continuous parameters p and t (time). Varying p generates a family of parameterized input "waveforms".
3. $v_p(t)$ is the corresponding output of S for a given $u_p(t)$. In particular, from causality, we assert that $v_p(t')$ may depend on $u_p(t)$ $\forall t < t'$ but does not depend on $u_p(t)$ $\forall t \geq t'$.

The device S must make a binary decision based on the arbitrary parameter p and the outcome of the decision must appear in the output variable v. The decision at time t' is based on the input $u_p(t)$ for time up to t' and appear at the output as $v_p(t')$. The range of the output variable must consist of two non-overlapping sub-ranges. The continuous range of the output variable is divided into three sub-ranges: 0-range, 1-range and X-range. For a given $u_p(t) \in U_p$, a decision is obtained if $v_p(t') \in$ 0-range or $v_p(t') \in$ 1-range and indecision occurs if $v_p(t') \in$ x-range.

The question we wish to answer: Can we prove that the x-range is necessarily non-empty? The proof is divided in three parts: mathematical premises, system model and the theorem itself.

1) Mathematical premises: The mathematics of continuous functions will be required in the proof. We will state the pertinent theorems without proof[30].

Theorem A. Continuous function on compact space

Let M be a compact space, i.e. a bounded closed set in a Cartesian R^n, and f a continuous function, then the image f(M) is also compact.

Theorem B. Continuous function on compact connected space

In addition, if M is connected, then f(M) is also connected.

Theorem C. Continuous mapping

If a real continuous function on a connected space has an interval for its image, then, between any two of its values, α and β, it assumes every intermediate value γ ($\alpha < \gamma < \beta$).

2) System model: The model assumed in the proof is the system model used for continuous physical system. It has a state $s \in \Sigma$ and is modeled by a continuous function φ, or more formally:

$\Sigma : \{ s \mid s \text{ is the state of } S \}$ $\qquad \Sigma \equiv R \times R \times R \ .. \times R \equiv R^m$

Each Real element (R) in Σ is a continuous state variable.

$U_p = \{ u_p(t) \mid u_p(t) \text{ is a parameterized input function} \}$

Each element of U_p is a function of the parameter p and the time t:

$$u_p(t): R \times R^+ \rightarrow R$$

$\varphi : \Sigma \times U_p \times R^+ \rightarrow R$

The function φ maps an initial state, $s \in \Sigma$, an input motion $u_p(t) \in U_p$ on the interval [0, t], and the time, $t \in R^+$, into R, the value of $v_p(t)$ at time t.

$$v_p(t) = \varphi (s, u_p, t)$$

3) Theorem: the Binary Decision Theorem (BDT) is:

"For an arbitrary t', if $\exists\, p_0, p_1$ such that $v_{p_0}(t') \in$ 0-range and $v_{p_1}(t') \in$ 1-range, then $\forall\, x \in$ x-range, $\exists\, p_x$ such that $v_{p_x}(t')=x$."

In other words, if there are inputs which lead to 0 and 1 decision at any time t', then any output value in the x-range (indecision range) is also possible at time t' due to some other input waveform. Notice that the theorem only assumes hypotheses H3 and H4 used in Marino's proof.

Proof - Since we are only interested on the mapping of the parameter into an output value at time t, nothing need be assumed about the initial state and the actual t. So, for all initial state $s \in \Sigma$, and for any time $t \in R^+$:

1. $p \in R$, and the range of p is a bounded connected space, by hypothesis.

2. $v_p = \varphi(s, u_p, t)$ is a continuous function, according to the system model.

3. The image f(p) is an interval, by hypothesis.

4. $\exists\ p_0, p_1$ such that $v_{p_0}(t') \in$ 0-range and $v_{p_1}(t') \in$ 1-range, else there is no decision.

Since f(p) is a real continuous function having an interval for its image, by theorem C, between any two of its values, $v_{p_0}(t')$ and $v_{p_1}(t')$, it assumes every intermediate value $v_{px}(t') \in$ x-range and $v_{p_0}(t') < v_{px}(t') < v_{p_1}(t')$. Therefore, $\forall\ x \in$ x-range, $\exists\ p_x$ such that $v_{p_x}(t')=x$.

Q.E.D.

6.0 Application of the BDT

Here a few instances of the class of binary decision problem are reviewed.

6.1 BDT and Decision on Real Value (DRV)

Suppose the object of the decision is the initial value $u_p(0)$ (relative to t') and U_p is the set of all possible signals such that $u_p(0)$ forms a continuous range by

varying p. Clearly there are inputs $u_{p0}(t)$ and $u_{p1}(t)$ that lead to $v_{p0}(t')$ and $v_{p1}(t')$ in the 0 and 1 ranges respectively, else no decision is required. Then from BDT, we get:

DRV: $\forall$ x $\in$ x-range, $\exists$ p_x such that $v_{p_x}(t')=x$, and indecision is inevitable.

It means there is an input function $u_{p_x}(t)$ such that $v_{p_x}(t')=x$ for all x in the x-range. The above scenario is illustrated in Fig. 8.

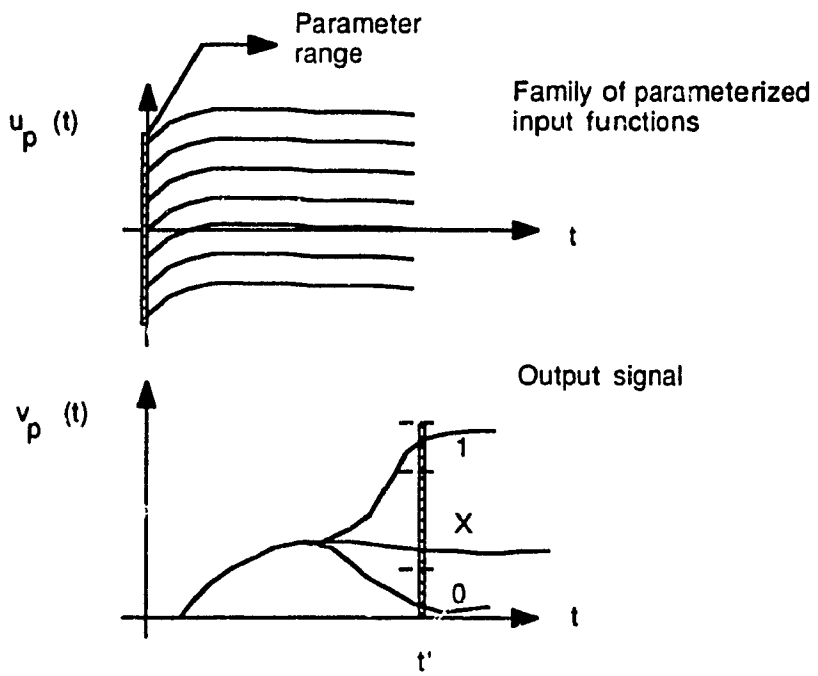


Fig. 8. Decision on Real Value (DRV)

6.2 BDT and synchronization

Synchronization deals with digital events and is defined as follows. On an asynchronous signal, the events can occur at any time with respect to a reference event (clock). On a synchronous signal there exists an empty uncertainty interval in respect to the reference event during which no events will occur. A perfect synchronizer would generate a synchronized signal from an asynchronous signal. The synchronization problem is formally analyzed using the BDT as follows.

Returning to the synchronization problem previously introduced (Fig. 4), the signal D_a corresponds to $u_p(t)$ and the signal D_s to $v_p(t)$. The occurrence of the clock

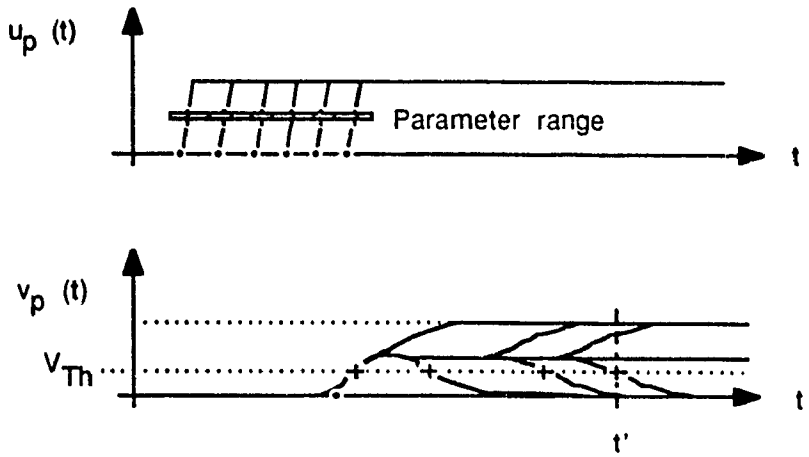


Fig. 9. Formal model for synchronization

event (reference event) is assumed to be at t=0. The object to be decided on is the delay between the occurrence of the clock event (t=0) and the occurrence of an asynchronous event on $u_p(t)$, as shown in Fig. 9.

So in this application of the BDT, we treat p as the parameter representing the variations of the delay. Accordingly, if by varying p, we get $v_p(t')$ in the 0-range for some $p=p_0$ and in the 1-range for some $p=p_1$, and if V_{Th} separates the 0 and 1 ranges, then:

$\forall\ V_{TH}, \exists\ p_x$ such that $v_{p_x}(t')=V_{Th}$

SYNCHRONIZATION:

Therefore, $\exists\ p_x$ such that there is an event at t=t'.

Because BDT is valid for arbitrary t', this further implies an output event can occur at any time. Therefore an empty uncertainty interval cannot be produced and D_s cannot be synchronized with respect to the reference event.

6.3 BDT and the existence of metastability

According to the definition of metastability given in section 2, a metastable solution exists if the output can be in the x-range at $t\rightarrow\infty$. In the BDT, the value of t' can be arbitrarily large; therefore, the output can be in the x-range at $t\rightarrow\infty$ and there exists a metastable solution.

7.0 References

[1] C. Mead and L. Conway, "Introduction to VLSI systems", pp. 242-261, Addison-Wesley, 1980.

[2] C. E. Molnar, T. P. Fang and F. U. Rosenberg, "Synthesis of delay-insensitive modules," *Proc. 1985 Chapel Hill Conf. VLSI*, Chapel Hill, NC. May 15-17, pp. 67-86, 1985.

[3] F. U. Rosenberg, C. E. Molnar, T. J. Chaney and T. P. Fang, "Q-modules: Internally clocked delay insensitive modules," *IEEE Trans. on computer*, Vol. 37, No. 9, pp. 1005-1018, September 1988.

[4] L. Kleeman and A. Cantoni, "Metastable behavior in digital systems," *IEEE Design and test of Comp.*, December 1987, pp. 4-19.

[5] Y-P. W. Lim and J. R. Cox, "Clocks and the performance of synchronizers," *Proceedings of the IEE-part E*, Vol. 130, pp. 57-64, 1983.

[6] D. M. Chapiro, "Globally-asynchronous locally-synchronous systems reliability," Ph. D. Dissertation, Standford University, 1984.

[7] L. Kleeman and A. Cantoni, "Can redundancy and masking improve the performance of synchronizers," *IEEE Trans. on computer*, Vol. 35, pp. 643-646, 1986.

[8] W. K. Stewart and S. A. Ward, "A solution to a special case of the synchronization problem', *IEEE Trans.* on computer, Vol. 37, pp. 123-125, 1988.

[9] E. G. Wormald, "A note on synchronizer or interlock maloperation," *IEEE Trans. on computer*, Vol 26, pp. 317-318, 1977.

[10] G. R. Couranz and D. F. Wann, "Theoretical and experimental behavior of synchronizers operating in the metastable region," *IEEE Trans.* on computer, Vol. 24, pp. 604-616, 1975.

[11] T. J. Chaney and C. E. Molnar, "Anomalous behavior of synchronizer and arbiter circuits," *IEEE Trans. on computer*, Vol 22, pp. 421-422, 1973.

[12] B. Liu and N. C. Gallagher, "On the metastable region of flip-flop circuits,"*IEEE Procedings*, Vol. 65, pp. 583-585, 1977.

[13] G. Elineau and W. Wiesbeck, "A new J-K flip-flop for synchronizers," *IEEE Trans. on computer*, Vol. 26, pp. 1277-1279, 1977.

[14] D. J. Kinninment and J. V. Woods, "Synchronization and arbitration circuits in digital systems," *Proceedings of the IEE.*, Vol. 123, pp. 961-966, 1976.

[15] S. T. Flannagan, "Synchronization reliability in CMOS technology," *IEEE J. Solid-State Circuits*, Vol. 20, pp. 880-882, 1985.

[16] W. Fleischhammer and O. Dortok, "The anomalous behavior of Flip-Flops in synchronizer circuits," *IEEE Trans. on computer*, Vol. 28, pp. 273-276, 1979.

[17] H. J. M. Veendrick, "The behavior of Flip-Flops used as synchronizers and prediction of their failure rate'," *IEEE J. Solid-State Circuits*, Vol. 15, pp. 169-176, 1980.

[18] A. Albicki and T. A. Jackson, "Simulation of NMOS flip-flops under asynchronous inputs," In the proceeding of the IEEE 1983 Custom Integrated Circuit Conference, Rochester, pp. 239-242.

[19] N. A. Hossri, P. Marchegay and G. Lacroix, "Formulation of the uncertainty graph of a CMOS bistable circuit," *Modelling and Simulation & Control*, Vol. 8-2, 1986.

[20] H. J. M. Veendrick, "Design aspects and reliability of a synchronizer made in MOS technology," In the proceeding of the Fifth European solid state circuits conference - ESSCIRC 79, Southampton, England.

[21] T. A. Jackson and A. Albiki, "Reduction of the probability of synchronization failure in NMOS systems through proper Flip-Flop design," In the proceedings of the Sixth Biennal University/Government/Industry Microelectronics Symposium, 1985, Auburn.

[22] J. H. Hohl, W. R. Larsen and L. C. Schooley, "Prediction of error probabilities for integrated digital synchronizers," *IEEE J. Solid-State Circuits*, Vol. 19, pp. 236-244, 1984.

[23] L. R. Marino, "General Theory of Metastable Operation," *IEEE Trans. on computer*, Vol. 30, pp. 107-115, 1981.

[24] F. Rosenberg and T. J. Chaney, "Flip-flop resolving time test circuit," *IEEE J. Solid-State Circuits*, Vol. 17, pp. 731-738, 1982.

[25] T. J. Chaney, "Measured flip-flop responses to marginal triggering," *IEEE Trans. on computer*, Vol. 32, pp. 1207-1209, 1983.

[26] P. Marchegay, P. Nouel, E. Garnier and G. Lacroix, "Coherence faults of the random access sequential network," In the proceeding of the FTCS 13th annual international symposium on fault-tolerant computing (IEEE), 1983, Milan.

[27] M. Hurtado and D. L. Elliot, "Ambiguous behavior of logic bistable systems," Proceedings of the 13th annual Allerton conference on circuit and system theory, University of Illinois at Urbana-Champaign, Oct. 1975, pp. 605-611.

[28] L. Kleeman and A. Cantoni, "On the Unavoidability of Metastable Behavior in Digital Systems," *IEEE Trans. on computer*, Vol. 36, pp. 109-112, 1987.

[29] W. I. Fletcher, "An engineering approach to digital design," Prentice-Hall, Englewood Cliffs, N.J., 1980.

[30] "Fundamentals of mathematics, Vol. III-Analysis," Edited by H. Behnke, F. Bachmann, K. Fladt and W. Süss, Translated by S. H. Gould, The MIT press, 1986.

[31] L. Morin and H. F. Li, "Thedesing of synchronizers: A review", IEE Proceedings, Vol. 136, Pt. E, No. 6, Nov. 1989.

Appendix B

HDIL reference manual

The Hardware Description and Integration Language (HDIL) is a language for circuit description. The simple syntax of HDIL and the general purpose LISP like constructs make it easy to learn and use in small research and development projects. The language is supported by a powerful schematic editor, PADS-LOGIC [1]. Using "parse" and "unparse" procedures, a circuit description can be loaded into memory and stored into an HDIL file. HDIDS (Hardware Description and Integration Data Structure) is the format for the data representation in the memory and is described in appendix C. This trio (HDIL, HDIDS and PADS-LOGIC) forms a universal front end for the construction of design tools. The programmer can use this front end and implement his algorithms directly on validated data in active memory. Even if they were designed for circuit description, this front end can be used for the description of any graph made of objects connected with arcs.

1.0 Introduction

This language has three important characteristics. First it includes features to support the three phases of design: hardware compilation, hardware linking and hardware simulation. This is accomplished by a syntactic construct called "SystemCall" that captures the names of the software tools used in the design. This is important for configuration control of the designs.

Second HDIL was designed to be used with schematic editors, all abstractions used in schematic diagrams are supported by HDIL. The language is based on elements, nodes and wires and general purpose symbols are used to allow the

description of complex or repetitive circuits. Off-page connections and buses are easily handled, but busses are not currently supported by the parser.

And third, the perfect match between HDIL and HDIDS allows both internal and external integration. Internal integration is particularly interesting in a CAD system for rapid exchange of information between tools through memory.

Basically HDIL and HDIDS support the structural information like elements, nodes, wires and hierarchical circuits using specific syntactic constructions. The structural information is used to link various aspects or views of a digital circuit. For each view, such as the layout, the schematic diagram, the electrical model or the logic model, general purpose LISP like syntactic constructions are used to insert the required information. The structural information in HDIL and HDIDS acts like a common skeleton that maintains the coherence between each view of the design.

1.1 Data model and definition

The data model on which the language is based, is similar to the data model used in most systems, see Fig. 1. An object in a digital system is a *circuit*. A circuit description includes an interface and an implementation to describe either hierarchical circuits or logic devices. A hierarchical circuit is made of *elements* and *wires*. An element is an instance of a circuit and wires are composed of *wiring nodes*. A wiring node is used to describe the connection to the node of an element. Each wire and each node carries a *signal* of a declared type.

A *system call* indicates the type of information contained in the circuit description or the CAD tool that will process it. For example, system call indicates which tool will compile the circuit, which tools are used to edit the layout, which tool will link the circuit and compute the effects of the wire capacitances and loads and which tool, or more precisely which model is used during the simulation. The

parameters are general purpose LISP like data structures used to insert any information required during the design process.

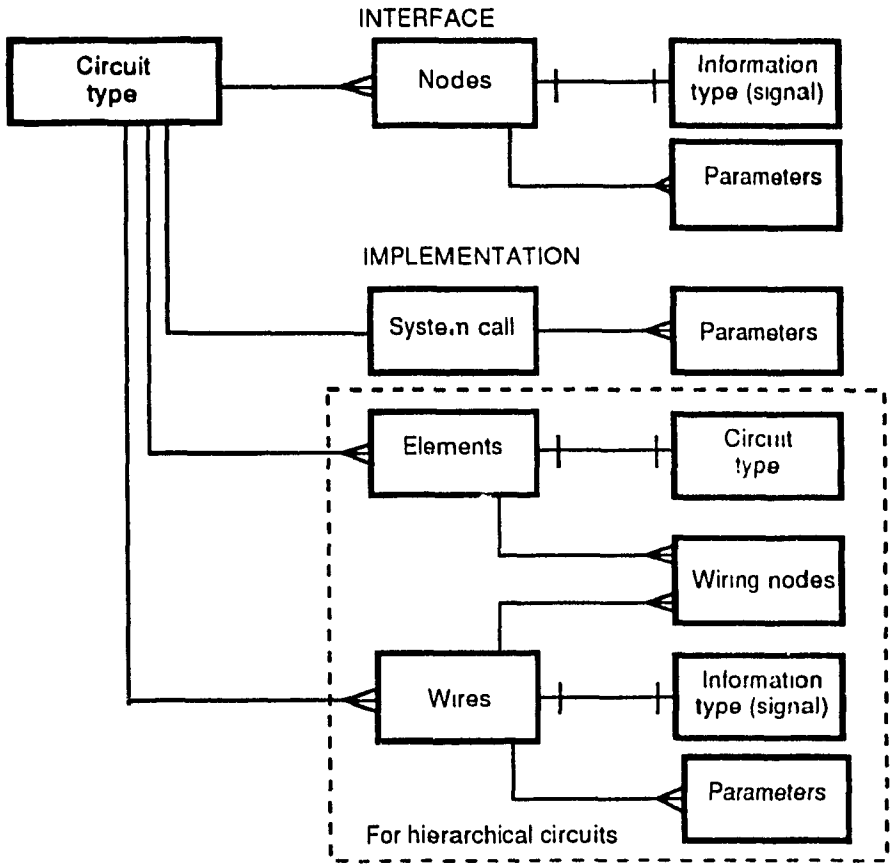


Fig. 1. Data model

1.2 Design philosophy

The design philosophy for hardware description languages is the same as for programming languages. The design of HDIDS and HDIL was greatly inspired by papers from N. Wirth [2] and C.A.R. Hoare [3]. N. Wirth has well summarized his and our philosophy as follow: "A language should be simple, and that simplicity must be achieved by transparence and clarity of its features and by a regular structure, rather than by utmost conciseness and unwanted generality". Even if the design of a HDL is more complicated because it must describe a real object, many ideas have

been imported from software programming languages. The CAD system and the language should also be designed to work with each other [4] and provide a coherent design environment.

2.0 Language description

2.1 Simplicity, transparence and clarity

Simplicity has been achieved by defining a language with syntactic construct matching the designer's view and by using a simple and regular structure. A typical hierarchical circuit is organized as follow:

```
circuit  xxx

   symbol
      ......   {symbol declaration statements}
   type
      .....    {type declaration statements}
   interface
      ......   {node declaration statements}
   variable
      ......   {variable declaration statements}

   implementation  SystemCall
         ......   {SystemCall parameters}

      circuit  CircuitX
         ......   {circuit description,
      end  CircuitX
      circuit  CircuitY
         ......   {circuit description}
      end  CircuitY
      circuit  CircuitZ
         ......   {circuit description}
      end  CircuitZ

      element
         ......   {element declaration statements}

      wire
         ......   {wire declaration statements}

end  xxx
```

If the circuit is a device, the description becomes:

```
circuit  xxx

   symbol
      ......   {symbol declaration statements}
```

```
    type
       .....     {type declaration statements}
    interface
       ......    {node declaration statements}
    variable
       ......    {variable declaration statements}

    implementation  SystemCall {Model of the device}
       ......    {SystemCall parameters}

end  xxx
```

The description of a circuit is enclosed by the key words **circuit** and **end** followed by the circuit's name. A circuit consists of three parts, the interface nodes, the state variables and the implementation with declaration statements for symbols and types. There is a system call associated with the implementation. Symbol statements declare the symbols used in the description of the circuit. A symbol is nothing more than an identifier or a name given to an object in the circuit. It can be used for replacing string and list of symbols or to defined structured symbol. As the above listing shows, the language has an easy to understand structure. Here are a few examples for the various types of statements:

Node statements, used to declare all interface nodes:

```
a b c : bit;        {declares nodes a, b and c}
```

LISP like functions used for the description of parameters:

```
(out c (nand a b)){declares a 2-inputs nand function}
```

Element statements, used to declare all elements:

```
G1 G2: nand2s4;    {declares the elements G1 and G2 as
                   circuits of the type nand2s4}
```

Wire statements, used to declare all wires:

```
reset : G1#1 G2#3;    {declares a wire connecting
                      node 1 of element G1 with
                      node 3 of element G2, the
                      name of the signal is reset}
```

2.2 Schematic diagram and hierarchical constructs

The language directly supports graphical representations like schematic diagrams. The Fig. 2 is the schematic diagram corresponding to the circuit description on the next page. This example also illustrates how the language supports the hierarchical structure of digital circuits. In this example, multiple levels of implementation are used. A printed circuit board, NandBoard, is a circuit implemented using a PCBcompiler, a program that performs placement and routing at board level. The NandBoard is composed of one element, the NandIC. The NandIC element is an integrated circuit with one element, the NandChip. The NandIC is constructed using a PackageCompiler. Finally the NandChip is constructed using an ICcompiler that performs the placement of the four cells.

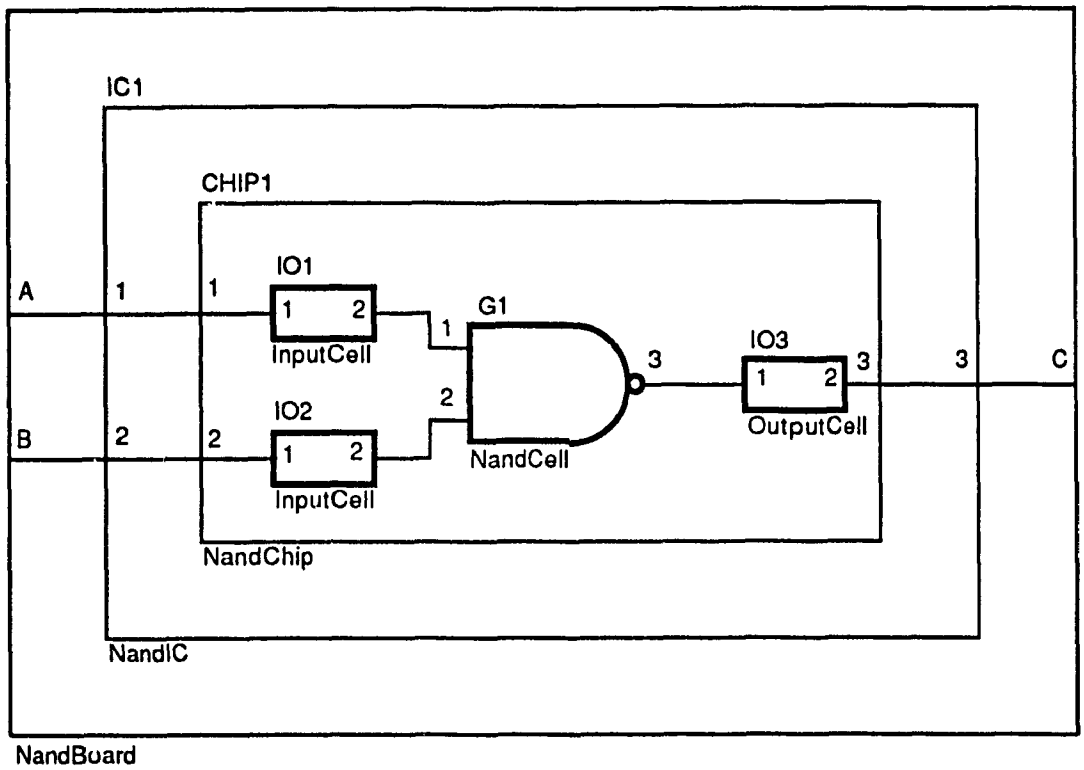


Fig. 2. Example of a hierarchical circuit

```
circuit  NandBoard
   interface
      a b c: bit;
   implementation   PCBcompiler

      circuit  NandIC
         interface
            1 2 3: bit;
         implementation   PackageCompiler

            circuit  NandCHIP
               interface
                  1 2 3: bit;
               implementation   ICcompiler

                  circuit  NandCell
                     interface
                        1 2: bit;
                  circuit  InputCell
                     interface
                        1 2 3 : bit;
                     external  "Library"
                  circuit  OutputCell
                     interface
                        1 2 3 : bit;
                     external  "Library"

                  element
                     G1: NandCell;
                     IO1 IO2: InputCell;
                     IO3: OutputCell;
                  wire
                     G1#1 IO1#2;
                     G1#2 IO2#2;
                     G1#3 IO3#1;
                     IO1#1 1;
                     IO2#1 2;
                     IO3#2 3;
            end  NandCHIP

            element
               CHIP1: NandChip;
            wire
               CHIP1#1 1;
               CHIP1#2 2;
               CHIP1#3 3;
      end  NandIC

      element
         IC1: NandIC;
      wire
         IC1#1 a;
         IC1#2 b;
         IC1#3 c;
end  NandBoard
```

2.3 Compilation, linking and simulation

The following fictitious example illustrates how the language can be used for the three phases of design. The schematic diagram of a simple clock circuit made of three inverters is given in Fig. 3.

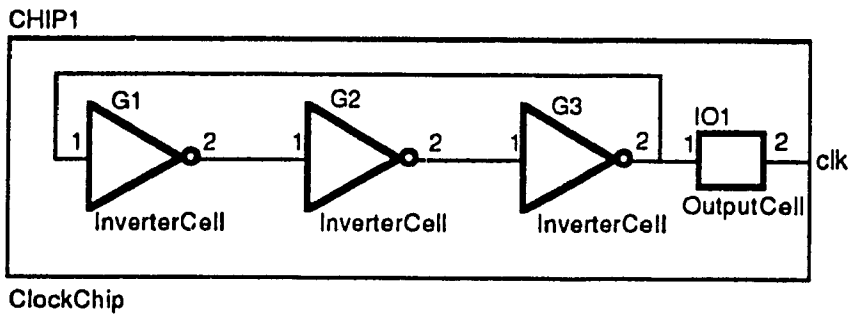


Fig. 3. Clock chip circuit

The corresponding listing, before hardware compilation would look like:

```
circuit  Clock
   node
      clk: bit;

   implementation ChipCompiler (data +3);
      circuit  InverterCell
         interface
            1 2:bit;
         implementation GateCompiler
            (data +2)
            (function (out 2 (not 1)))
      end  InverterCell
      circuit  OutputCell
         interface
               1 2:bit;
         implementation IOcellCompiler
            (data +2)
            (function (out 2 1))
      end OutputCell

      element
         G1 G2 G3: InverterCell;
         IO1: OutputCell;

      wire
         G1#2 G2#1;
         G2#2 G3#1;
         G3#2 IO1#1 G1#1;
         IO1#2 clk;
end  Clock
```

The above listing is the source file for the compilation phase. In a typical design environment, the user would draw the schematic diagram and the above file would be automatically generated. This listing describes how to construct the chip clock. First, the InverterCell must be produced. To generate the InverterCell, the GateCompiler is used. The GateCompiler reads the function not, counts the number of inputs, and reads the arguments for the size of the transistors, here +2. With this information, the GateCompiler generates the layout for the InverterCell and produces its linkable model. Then, the OutputCell must be produced. Let us suppose it is a manually designed cell stored in a library. Therefore the IOcellCompiler reads the parameters +2 which tells which cell to fetch from the library. The layout and the linkable model are simply copied. After compilation of the two cells, the circuit clock will be modified as follow:

```
circuit  Clock
      ......
   implementation ChipCompiler (data +3);

      circuit  InverterCell
               ......
--->     implementation GateCompiler (data +120 +30 +90);
      end  InverterCell

      circuit  OutputCell
            .......
--->     implementation IOcellCompiler (data +160 +10);
      end OutputCell
      ......
end  Clock
```

The new information required by the ChipCompiler has been added. For the InverterCell, information like the cell width (+120) and the positions of the input and output pins (+30 +90) are required. For the OutputCell the coordinates of the input pin (+160 +10) are required. Once all the cells have been compiled, the chip can be compiled. The ChipCompiler will read its parameter (+3) to determine the frame for the chip. Knowing the frame which is used, all pad positions are known and the OutputCell can be placed. The size and shape of the usable area on the chip is also

known and placement and routing of the three instances of the InverterCell and the OutputCell can be done. In most IC technology, wire capacitance is important and must be computed during chip compilation. After compilation, the following model will be available.

```
circuit  Clock
   interface
         clk: bit;
   implementation DeviceLinker

      circuit  InverterCell
         interface
            1:bit (data +0.9e-12);  {Input capacitance}
            2:bit (data +0.3e-12);  {Output capacitance}
         implementation
            (function  (out 2 (not 1)))
      end  InverterCell

      circuit  OutputCell
         interface
            1: bit (data +0.4e-12); {Input capacitance}
            2: bit (data +12e-12);  {Output capacitance}
         implementation
            (function (out 2 1))
      end OutputCell

      element
         G1 G2 G3: InverterCell;
         IO1: OutputCell;
      wire
         G1#2 G2#1 (data +0.3e-12);        {Wire capacitance}
         G2#2 G3#1 (data +0.7e-12);        {Wire capacitance}
         G3#2 IO#1 G1#1 (data +0.1e-12);   {Wire capacitance}
         IO#2 clk (data +0.9e-12);         {Wire capacitance}
end  Clock
```

The link phase consist in calling the appropriate linker to compute the effects of the wires and the loads on the model. To do so, information like input and output capacitances and wire capacitances are required. In the above listing, input and output capacitances are located in node statements and wire capacitances in wire statements. After linking, a complete logic model will be available. Because each instances has different loads, new circuit types are created with different propagation delay. The circuit description becomes:

```
circuit  Clock
   interface
      clk: bit;
   implementation
      circuit  InverterCell_1
         interface
            1:bit;
            2:bit (delay +5e-9); {Prop. delay}
         implementation gate
            (function (out 2 (not 1)))
      end  InverterCell_1

      circuit  InverterCell_2
         interface
            1:bit ;
            2:bit (delay +12e-9);   {Prop. delay}
         implementation gate
            (function (out 2 (not 1)))
      end  InverterCell_2

      circuit  InverterCell_3
         interface
            1:bit;
            2:bit (delay +7e-9); {Prop. delay}
         implementation gate
            (function (out 2 (not 1)))
      end  InverterCell_3

      circuit  OutputCell
         interface
            1: bit;
            2: bit (delay +15e-9);  {Prop. delay}
         implementation gate
            (function (out 2 1))
      end OutputCell

      element
         G1: InverterCell_1;
         G2: InverterCell_2;
         G3: InverterCell_3;
         IO1: OutputCell;

      wire
         G1#2 G2#1;
         G2#2 G3#1;
         G3#2 IO#1 G1#1;
         IO#2 clk;

end  Clock
```

2.4 Handling of special tools

The following example shows how HDIL can be used with a tool that does not accept HDIL syntax. The circuit is a ROM containing the program of a microcontroller.

```
circuit  ROMcircuit
   interface
         Data: byte;
         Address: byte; {For a 256 bytes ROM}
         Enable Read Write:bit;
-->implementation 6805assembler (use "SourceFile" "ListFile")
end ROMcircuit
```

In this example, the SourceFile contains the assembly language listing of a program for the 6805 microprocessor. During compilation, the 6805 assembler first assembles the SourceFile and determines the ROM content. Then it generates the layout and the linkable model. This might be useful in integrating existing tools not using HDIL format.

2.5 Complex circuits description

The following example illustrates how a complex and repetitive circuit can be described using HDIL. The Fig. 4 is a simple systolic array whose internal function is unknown. The corresponding listing is:

```
circuit SystolicArray
   symbol
      a b: structure 1 2 3;
      c: structure 1 2 3 4 5;
      SM1: structure 1 2 3  structure 1 2 3;
   interface
      a.1 a.2 a.3: bit;
      b.1 b.2 b.3: bit;
      c.1 c.2 c.3 c.4 c.5: bit;

   implementation
      circuit  cell
         symbol
            direction: in out;
            a b c: structure direction;
         interface
            a.in a.out b.in b.out c.in c.out: bit;
         implementation
            ......
      end cell

      element
         SM1.1.1 SM1.1.2 SM1.1.3:  cell;
         SM1.2.1 SM1.2.2 SM1.2.3:  cell;
         SM1.3.1 SM1.3.2 SM1.3.3:  cell;
```

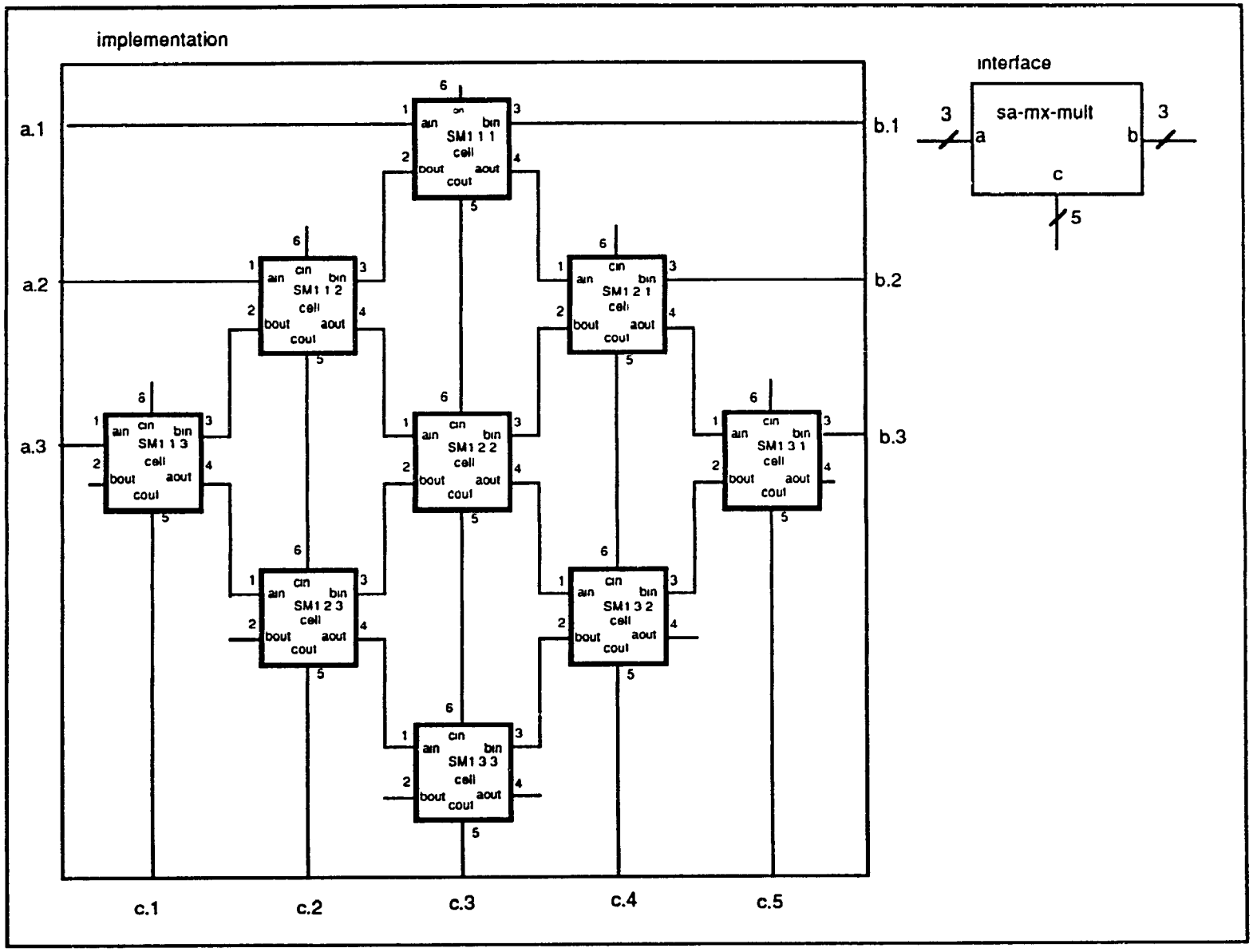


Figure 4. Systolic array example

```
        wire
           a.1 SM1.1.1#1;
           a.2 SM1.1.2#1;
           a.3 SM1.1.3#1;
           b.1 SM1.1.1#3;
           b.2 SM1.2.1#3;
           b.3 SM1.3.1#3;
           c.1 SM1.1.3#5;
           c.2 SM1.2.3#5;
           c.3 SM1.3.3#5;
           c.4 SM1.3.2#5;
           c.5 SM1.3.1#5;
           SM1.1.1#5 SM1.2.2#6;
           SM1.1.2#5 SM1.2.3#6;
           SM1.2.1#5 SM1.3.2#6;
           SM1.2.2#5 SM1.3.3#6;
           SM1.1.1#2 SM1.1.2#3;
           SM1.1.2#2 SM1.1.3#3;
           SM1.2.1#2 SM1.2.2#3;
           SM1.2.2#2 SM1.2.3#3;
           SM1.3.1#2 SM1.3.2#3;
           SM1.3.2#2 SM1.3.3#3;
           SM1.1.1#4 SM1.2.1#1;
           SM1.2.1#4 SM1.3.1#1;
           SM1.1.2#4 SM1.2.2#1;
           SM1.2.2#4 SM1.3.2#1;
           SM1.1.3#4 SM1.2.3#1;
           SM1.2.3#4 SM1.3.3#1;
    end  SystolicArray
```

The above listing illustrates the flexibility of structured symbols. For large arrays repeat statements will be added.

3.0 Formal definition

In this section HDIL is formally defined. The following abbreviations are used:

- **bold** words are terminals symbols

- < > are used to indicate non-terminals symbols

- 2 to 4 letters are used as general prefix for non-terminals symbols:

<u>Lexical analyzer related</u>

id	identifier
int	integer
dec	decimal integer
hex	hexadecimal integer
ch	character integer
real	real number
exp	exponent

suf	real number suffix
str	string
esc	escape sequence
cmt	comment
let	letters
dig	digits
hexd	hexadecimal digits
std	standard characters
spc	space characters
prt	printable characters
esch	escape characters

Symbol processing related (not implemented)

tkn	token
itm	item
rpt	repeat
rng	range

Parser related

cir	circuit
imp	implementation
ele	element
wir	wire
sym	symbol
als	alias
ssy	structured symbol
nod	node
var	variable
fct	function
exp	expression
sys	system call

- letters used as a qualifier suffix:

s	indicates plural
d	declaration
l	list
sts	statements
st	statement

3.1 Lexical analyzer

3.1.1 Characters set

HDIL uses the standard ascii character set.

Hexadecimal	ASCII	Key	Escape code
00	<NUL>	ctrl-@	\00
01..06		ctrl-a to ctrl-f	\01 to \06
07	<BEL>	ctrl-g	\a
08	<BS>	ctrl-h	\b
09	<TAB>	ctrl-i	\t
0A	<LF>	ctrl-j	\n
0B	<VT>	ctrl-k	\v
0C	<FF>	ctrl-l	\f
0D	<CR>	ctrl-m	\r
0E..10		ctrl-n to ctrl-p	\0E to \10
11	<XON>	ctrl-q	\11
12		ctrl-r	\12
13	<XOFF>	ctrl-s	\13
14..19		ctrl-t to ctrl-y	\14 to \19
1A	<EOF>	ctrl-z	\1A
1B	<ESC>	ctrl-[	\1B
1C..1F		ctrl-\ ctrl-] ctrl-^ ctrl- _	\1C to \1F
20	<SP>		
21..2F	characters	! " # $ % & ' ( ) * + , - . /	\" \'
30..39	numbers	0 to 9	
3A..40	characters	: ; < = > ? @	
41..5A	letters	A to Z	
5B..60	characters	[\ ] ^ _ `	\\
61..7A	letters	a to z	
7B..7E		{ | } ~	
7F	<DEL>		\7F
80..FF			\80 to \FF

3.1.2 Special symbols

Single character: , : ; . \ $ + - # ( ) [ ] { } ' " < >

Characters pair: .. << >>

Paired characters: ( ) [ ] { } '' "" << >>

Reserved words:

circuit	**alias**	
symbol	**structure**	**repeat**
type	**variable**	**interface**
implementation	**element**	**wire**
end	**external**	

3.1.3 Predeclared identifiers

Predefined macro-symbols

Z U

Build-in types

bit byte sbyte word16 word32 sword16 sword32

Predifined system calls

gate CC SSC 3SD ASC flipflop AR

Predefined functions

input	output	hvinput	hvoutput		
not	and	nand	nor	xor	xnor
set	out	in	table	defunc	timing
if	exe	case	defcyc	data	

3.1.4 Comments

Comments are enclosed in braces {}. Any printable characters <prn> or space characters <spc> are allowed.

3.1.5 Token syntax

The lexical analyzer recognizes special characters (, . : ; # [ ] ( )), character pair (..), identifier <id>, integer constant <ct>, real constant <real> and string <str>, as follow:

<id> --> (<std>| _)$^+$

<int> --> <dec> | <hex> | <ch>

<dec> --> (-|+)<dig>$^+$

<hex> --> $<hexd>$^+$

<ch> --> '(<prt> | <esc>)*'

<real> --> (<dec> . <dig>* <exp>)

<exp> --> <sufch>

```
                | (e|E) (-|+)?<dig>+)
<str>    -->  " (<prt> | <esc>)* "
<esc>    -->  \<esch> | \ $ <hexd><hexd>
<cmt>    -->  { (<prt> | <spc>)* }
<prt>    -->  char (21 .. 7E)
<std>    -->  <let> | <dig>
<let>    -->  a..z | A..Z
<dig>    -->  0..9
<sufch>  -->  a | f | p | n | u | m | K | M | G | T | P | E
<hexd>   -->  <dig> | A | B | C | D | E | F
<esch>   -->  a | b | t | n | v | f | r | \ | ' | "
<spc>    -->  <SP> | <BS> | <TAB> | <LF> | <VT>| <FF> | <CR>
```

Legend:

|	-->	choice
( )?	-->	zero or one
( )+	-->	one or more
( )*	-->	zero or more

Where required, white space <spc> separates the tokens. An identifier is a sequence of letters, digits or underscore, beginning with any one of them. This language is case sensitive. There are three types of integer constant: decimal, hexadecimal and character. A decimal integer consists of a sequence of decimal digits, preceded by a sign (+ or -). An hexadecimal integer consists of a dollar sign ($) followed by a sequence of hexadecimal digit. A character integer consists of one to four character enclosed in simple quotes. The escape character backslash (\) can be used to enter non-printable characters, the quote or the backslash. Real numbers must also begin with a sign and are composed of a mantissa and an exponent. The mantissa consists of one or more decimal digits followed by a dot and zero or more

digits. The exponent consists of a suffix letter (a, f, p, n, u, m, K, M, G, T, P, E) or begins by the letters e or E, followed by an optional sign and a sequence of decimal digits. The following suffixes are accepted:

Suffix	Name	Exponent
E	exa	18
P	peta	15
T	tera	12
G	giga	9
M	mega	6
K	kilo	3
m	milli	-3
u	micro	-6
n	nano	-9
p	pico	-12
f	femto	-15
a	atto	-18

3.2 Language syntax and semantics

3.2.1 Circuit organization

<cir>	-->	**circuit**	<id_1>	{circuit name}
			<symd>	{symbol declaration}
			<typd>	{type declaration}
			<nodd>	{node declaration}
			<vard>	{variable declaration}
		external	<str>	{library name}
	-->	**circuit**	<id_1>	{circuit name}
			<symd>	{symbol declaration}
			<typd>	{type declaration}
			<nodd>	{node declaration}
			<vard>	{variable declaration}
			<imp>	{implementation}
		end	<id_2>	
<imp>	-->	**implementation**	<sys>	{system call}
			<cird>	{circuit declaration}
			<elel>	{element list}
			<wirl>	{wire list}

A circuit begins with the word **circuit** followed by its name <id_1>, the symbol declaration, the type declaration, the node declaration and the variable declaration. If

external follows, the circuit description is stored in the library named string <str>. Otherwise, the implementation follows and the circuit description terminates with **end** followed by the circuit name <id_2>. The implementation begins with **implementation** and a system call. If the circuit is a device, the system call describes the model. In the case of a hierarchical circuit, the circuit used are declared in <cird>, followed by the list of element <elel> and the list of wires <wirl> describing how the circuit is constructed.

3.2.2 Symbol declaration

<symd>	-->	**nil**	
	-->	**symbol** <symsts>	{symbol statements}
<$symsts_1$>	-->	<symst> ;	
	-->	<symst> ; <$symsts_2$>	
<symst>	-->	<id> : <ids>	{list of identifiers}
	-->	<id> : <ints>	{list of integers}
	-->	<id> : <reals>	{list of reals}
	-->	<id> : <strs>	{list of strings}
	-->	<sym> <als>	{alias symbols}
	-->	<sym> <ssy>	{structured symbol}
<als>	-->	**alias** <syms>	
<ssy_1>	-->	**structure** <ids>	
	-->	**structure** <ids> <ssy_2>	

Symbols are names given by the user to ease the circuit description. There are three types of symbols: simple and structured. A simple symbol is a single identifier, *test* for example. A structured symbol consists of identifiers separated by dots such as *a.2.3*. Simple symbols do not need to be explicitly declared, they are declared at the time they are first encountered. Structured symbol must be declared prior to their use. So before using the symbol *a.2.3*, it is necessary to declare the following structured symbol:

```
a : structure 1  2  3  structure 0  1  2  3  4;
```

Once declared, a structured symbol can be used as any simple symbol. Symbol declaration begins with **symbol** followed by symbol statements <symsts>. There are three types of symbol statements: macro definition, structure definition and alias definition. A macro definition declares an identifier as a macro replacement, for example:

```
symbol
   state: reset  run  stop  wait;
   BegAddr: $100;
   library: "\usr\vlsi\lib";
   esc: '\1A';
```

Each time an identifier corresponding to a macro definition is encountered, it is replaced by its definition. Structured symbols are declared as follow:

```
symbol
  a : structure  0  1  2 ;
  a.2 :  structure  on  off;      {structured symbols
                  a.2.on and a.2.off are created}
  Data : structure 0 1 2 3 4 5 6 7;
```

This allows multiple access to the same object. For example, CPUbus.Data.3 accesses the same object as Data.3. In addition, it is possible to access the same object with different names using alias declaration. In the following declaration, the symbols on the right are alternate names for the object declared on the left. This is specially useful in hierarchical circuits description.

```
symbol
   reset alias x.r x.y.reset;
```

3.2.3 Type declaration

<typd>	-->	**nil**	
	-->	**type** <typsts>	{type statements}
<typsts$_1$>	-->	<typst> ;	
	-->	<typst> ; <typsts$_2$>	
<typst>	-->	<id> : <int> <int>	{range of values, integer type}
	-->	<id> : <ids>	{enumerated}

Type declarations are used to define the type of nodes and the type of variables in a circuit. A type statement begins with the name of the type <id>, followed by either two integers or a list of identifiers. Both are integer type, in the first case, the two integers indicate the range of values and in the second case the list of values is given. If the enumerated type is used, each identifier in the list is given a constant value starting at +0. The range of values is limited to 16 bits. If the first number is negative, two's complement is assumed, otherwise, unsigned numbers are used. In addition to the listed values, nodes and variables can take the undefined value U and nodes can take the high impedance value Z. The following are valid type declarations:

```
bit: +0 +1;     {a variable of type bit
                   can take the value +0 and +1}
boolean:false true;
letter:'A' 'Z';       {range of integers from 65 to 90}
byte:+0 +255;         {unsigned byte}
long:$0 $FFFF         {word}
state:begin undefined halted stop;      {range +0 to +3
            encoded as: begin=      +0
                        undefined=  +1
                        halt=       +2
                        stop=       +3}
```

3.2.4 Node and variable declaration

<nodd> --> **interface** <nodsts> {node declaration}

<nodsts$_1$> --> **nil**
--> <nodst> <fcts> ; <nodsts$_2$>

<nodst> --> <syms> : <id> {id is a predefined type}

Each symbol on the left is declared as an interface node of the type declared on the right side. The type of a node defines the values the signal can have. Information like input capacitance can be inserted using function <fcts>. The following are valid node declarations:

```
node
   a : bit;              {a can take the values +0 and +1}
```

```
d1 d2 d3 : byte;      {d1 d2 and d3 can take
                          a value between +0 or +255}
next : state;         {next can take the values
                        begin, undefined, halted and stop}
```

The types bit, bytes and state must be declared types. The declaration could have been done anywhere before, signal type declarations are global. The directions of the nodes are not explicitly stated. The information about whether a node is an input or an output is added using <fcts> if desired. A node can be connected to many outputs, conflict are detected at simulation time.

Except for the keyword variable, the declaration of variables is similar to the declaration of nodes.

<vard> --> **nil**
--> **variable** <varsts> {variable declaration}

<varsts$_1$> --> **nil**
--> <varst> <fcts> ; <varsts$_2$>

<varst> --> <syms> : <id> {predefined type}

3.2.5 Circuit declaration

<cird> --> **nil**
--> <cir> <cird>

In a hierarchical circuit description, the circuit consists of elements connected with wires. Each element is a circuit that must be declared and the circuit declaration gives the list of the circuits used.

3.2.6 Element list

<elel> --> **nil**
--> **element** <elests>

<elests$_1$> --> <elest> ;
--> <elest> ; <elests$_2$>

<elest> --> <syms> : <id> {list of elements: circuit name}

The element list begins with **element** followed by the element statements <elests>. Each statement <elest> declares the symbols on the left as elements of the type specified on the right. For example,

element

G1 G2 G3 : nand2s4;

declares G1, G2 and G3 as elements of the type nand2s4. The type nand2s4 must be a declared circuit. The declaration could have been done anywhere before, circuit declarations are global.

3.2.7 Wire list

```
<wirl>      -->  nil
            -->  wire <wirsts>

<wirsts1>   -->  <wirst> ;
            -->  <wirst> ; <wirsts2>

<wirst>     -->  <nods> <fcts>                    {list of nodes}
            -->  <sym> : <nods> <fcts>            {wire name: list of nodes}

<nods>      -->  <nod>
            -->  <nod> <nods>

<nod>       -->  <sym>                            {node name of current circuit}
            -->  <sym> # <sym>                    {circuit name # node name}
```

The wire list starts with **wire** followed by wire statements. A wire statement <wirst> consists of an optional signal name <sym>, a list of nodes <nods> and wire information <fcts> like wire capacitance. The list of nodes defines the nodes connected to the wire. The nodes consists of an element name and a node name separated by #. The element must have been declared. A connection to an interface node can also be specified. For example,

```
wire
  test : G1#2  G3#1  4;
```

declares a wire connecting node 2 of G1, node 1 of G3 and interface node 4 of the current circuit, test is the optional signal name. Wire names are useful for off-page connections, supply lines, grounds and busses.

3.2.8 System call, function and expressions

<sys>	-->	**nil**	
	-->	<id> <fcts>	{name and parameters}
<fcts>	-->	**nil**	
	-->	<fct> <fcts>	
<fct>	-->	(<id> <exps>)	{LISP like function}
<exps>	-->	**nil**	
	-->	<exp> <exps>	
<exp>	-->	<sym>	{symbol: node or variable}
	-->	<int>	{integer}
	-->	<real>	{real}
	-->	<str>	{string}
	-->	<fct>	{function}

A system call corresponds normally to a tool or a model in the CAD system. A system call <sys> has a name <id> and optional parameters <fcts>. The name must be an existing system call. The flexible LISP like construct makes it possible to insert all kind of information, like electrical capacitances, propagation delay and logic functions. Here is a list of acceptable functions:

```
(data +2  -3 +2e-12)
(initial +0)
(event  +1  +1.2e-12  +0  +1.4e-12  Z  +1.6e-12)
(if  <exp>  <fct>  <fct>)
(case  <exp>
   val1  <fct>
   val2  <fct>
   val3  <fct>)
(exe  <fct>  <fct>  <fct>)
(table (input <variables or nodes>) (output <variables or nodes>)
   .......tabular information.......)
(input  ...)
(output ...)
(clock  ...)
(set  <variable> <exp>)
(out  <node> <exp>)
```

3.2.9 Symbols and identifiers

```
<syms>    -->  <sym>                          {list of symbols}
          -->  <sym>  <syms>

<sym>     -->  <id>
          -->  <id> . <sym>

<ids>     -->  <id>                           {list of identifiers}
          -->  <id>  <ids>

<ints>    -->  <int>                          {list of integers}
          -->  <int>  <ints>

<reals>   -->  <real>                         {list of reals}
          -->  <real>  <reals>

<strs>    -->  <str>                          {list of strings}
          -->  <str>  <strs>
```

Symbols are the basic naming abstraction in HDIL. Most object, like node, element or wire, are named using symbols. A symbol is either an identifier or a structured symbol composed of identifiers separated by dots. All structured symbols must be declared.

3.3 References

[1] The PADS-LOGIC user manual (Version 2.0), CAD SOFTWARE, INC. 119 Russell Street, Suite #6, Littleton, MA 01460.

[2] N. Wirth, "On the design of programming languages," In the proceedings of IFIP Congress, pp. 386-393, 1974.

[3] C. A. R. Hoare, "Hints on programming language design," In the proceedings of the Sigact/Sigplan Symposium on Principles of Programming Languages, 1973.

[4] C. Ghezzi and M. Jazayeri, "Programming language concepts," John Wiley & Sons, pp. 256, 1982.

Appendix C

HDIDS reference manual

The Hardware Description and Integration Data Structure (HDIDS) is the format of the information about a circuit when in active memory. Normally, an HDIL circuit description is loaded from a file into memory using the parse procedure and stored back into a file using the unparse procedure. There is a one to one correspondence between HDIL and HDIDS. Having defined a standard format for in memory information greatly facilitates and speeds up the transfer of information between various CAD tools. HDIDS does not support symbolic information, all symbols are stored in a symbol table which is constructed while parsing the HDIL source file. The PASCAL type declaration for HDIDS follows and a C type declaration is also available:

```
type
  CircuitPtr=^Circuit;
  NodePtr=^Node;
  VariablePtr=^Variable;
  RangePtr=^Range;
  WirePtr=^Wire;
  ElementPtr=^Element;
  WiringNodePtr=^WiringNode;
  LISPfctPtr=^LISPfct;
  ExpressionPtr=^Expression;
  LibraryPtr=^Library;
  Library=
     record
        name:string;
        next:LibraryPtr;
     end;

  Circuit=
    record
      Nbr:integer;
      TypeList:RangePtr;
      NodeList:NodePtr;
      VariableList:VariablePtr;
      ExtFlag:boolean;
      LibName:LibraryPtr;
      SysCall:SystemCallPtr;
      SysCallArg:LISPfctPtr;
```

```
    CircuitList:CircuitPtr;
    ElementList:ElementPtr;
    WireList:WirePtr;
    next:CircuitPtr;
 end;


Node=
  record
    Nbr:integer;
    Range:RangePtr;
    Parameters:LISPfctPtr;
    Next:NodePtr;
  end;


Variable=
  record
    Nbr:integer;
    Range:RangePtr;
    Parameters:LISPfctPtr;
    Next:VariablePtr;
  end;


Range=
  record
    Min:integer;
    Max:integer;
    next:RangePtr;
  end;


Element=
  record
    Nbr:integer;
    typ:CircuitPtr;
    NodeList:WiringNodePtr;
    Next:ElementPtr;
end;


Wire=
  record
    Nbr:integer;
    Range:RangePtr;
    NodeList:WiringNodePtr;
    Parameters:LISPfctPtr;
    Next:WirePtr;
end;


WiringNode=
  record
    NodeOf:ElementPtr;
    Node:NodePtr;
    Wire:WirePtr;
    NextE:WiringNodePtr;
```

```
    NextW:WiringNodePtr;
  end;


LISPfctPtr=
  record
    func:FunctionType;
    Arguments:ExpressionPtr;
    next:LISPfctPtr;
  end;


ExpressionType=(NodeExp,VariabelExp,FunctionExp,
                IntNumExp,RealNumExp,StringExp);
Expression=
  record
    next:ExpressionPtr;
    case typ:ExpressionType of
      NodeExp:(NodVal:NodePtr);
      VariabelExp:(VarVal:VariablePtr);
      FunctionExp:(fct:LISPfctPtr);
      IntNumExp:(IntVal:integer);
      RealNumExp:(RealVal:real);
      StringExp:(StrVal:string);
  end;
```

The above type declaration also uses the following user defined and application specific types:

```
type
  FunctionType=(BUFFERfct,NOTfct,NORfct,NANDfct,ERRORfct);
  SysCallType=(gate,CC,SSC,flipflop,AR);
```

FunctionType declares the functions type allowed in the data structure statements. For each of these function types, there is a string to be used in the HDIL. Corresponding to the above declaration, the following strings are allowed as function names: buffer, not, nor and nand. SysCallType declares the type of system call installed in a given design environment. Corresponding to the above declaration, the following system call names are used in HDIL: gate, CC, SSC, flipflop and AR. Current declared functions and system calls depend on the application.

A circuit is either a device or a hierarchical circuit. A device description does not have the CircuitList, ElementList and WireList fields. The first field in the circuit record, Nbr, is an identification integer. It is unique and automatically assigned by the

parser. The field ExtFlg, indicates when true that this portion of the circuit has not been parsed and is still in a library file whose name is pointed to by the LibName field. The last field (next) allows the construction of list of circuits.

The fields in HDIDS correspond to each part of the HDIL circuit description. The Fig. 1 illustrates the LISP like functions and expressions data structures. The LISP like structure allows the representation of simple arguments, lists of arguments, generalized list of arguments, vectors and arrays of any dimensions. A simple example is also given showing the relation between a LISP function in HDIL and its representation in HDIDS. The LISPfct record has three fields, func contains the function name, Arguments the LISP like list of arguments and next a pointer to the remaining functions. Evaluating a LISP like function consists in applying the function in the func field to the argument list pointed to by the Arguments field. Each argument is computed by evaluating the corresponding expression. Evaluating an expression consists in either reading the value of the interface node or a variable, using integer, real or string values or recursively evaluating a function.

The Fig. 2 illustrates the data structure of a typical hierarchical circuit. The Circuit record contains the following fields: Nbr, TypeList, NodeList, VariableList, ExtFlag, LibName, SysCall, SysCallArg, CircuitList, ElementList, WireList and next. The TypeList field points to the list of declared types. The NodeList field points to the list of interface nodes for the circuit. Each interface node is a record composed of a Nbr field, a Range field, a Parameters field and a Next field. The Nbr field is a unique identification integer assigned during parsing. The Range field is a pointer to a Range record. The Range record contains the range of values allowed for this interface node. The range record is filled by the parser based on the type of the node. For example, a byte type is stored as Min field set to 0 and Max field set to 255. The Parameters field points to a LISPfct data structure which contains the node characteristic, for example input capacitance or TTL fan-out.

The CircuitList field points to the list of all types of circuit used. The ElementList enumerates all instances of circuits and the WireList describes how they are interconnected. The ElementRecord has four fields. The first field, Nbr, is a unique identification integer assigned during parsing. The Typ field points to the type of circuit, the NodeList points to the list of wiring nodes attached to this element and the Next field allows the construction of a list. The Wire record has five fields. The Nbr field is a unique identification integer assigned during parsing. The Next field allows the construction of a list. The Range field points to the range of possible values for the signal. The Parameters field points to the LISPfct data structure containing the parameter, like wire capacitance. Finally, the NodeList points to a list of WiringNode record giving all nodes connected to the wire. A WiringNode record has five fields. The NodeOf field points to the element to which it is attached. The Node field points to the corresponding node in the circuit. The Wire field points to the corresponding wire. The field NextE allows the construction of the list of nodes for each element and the field NextW allows the construction of the list of nodes for each wire. For example, the list of wiring nodes for the first wire in Fig. 2 indicates that E1#2 is connected to E2#1. An empty NodeOf field indicates a connection to an interface node of the current circuit.

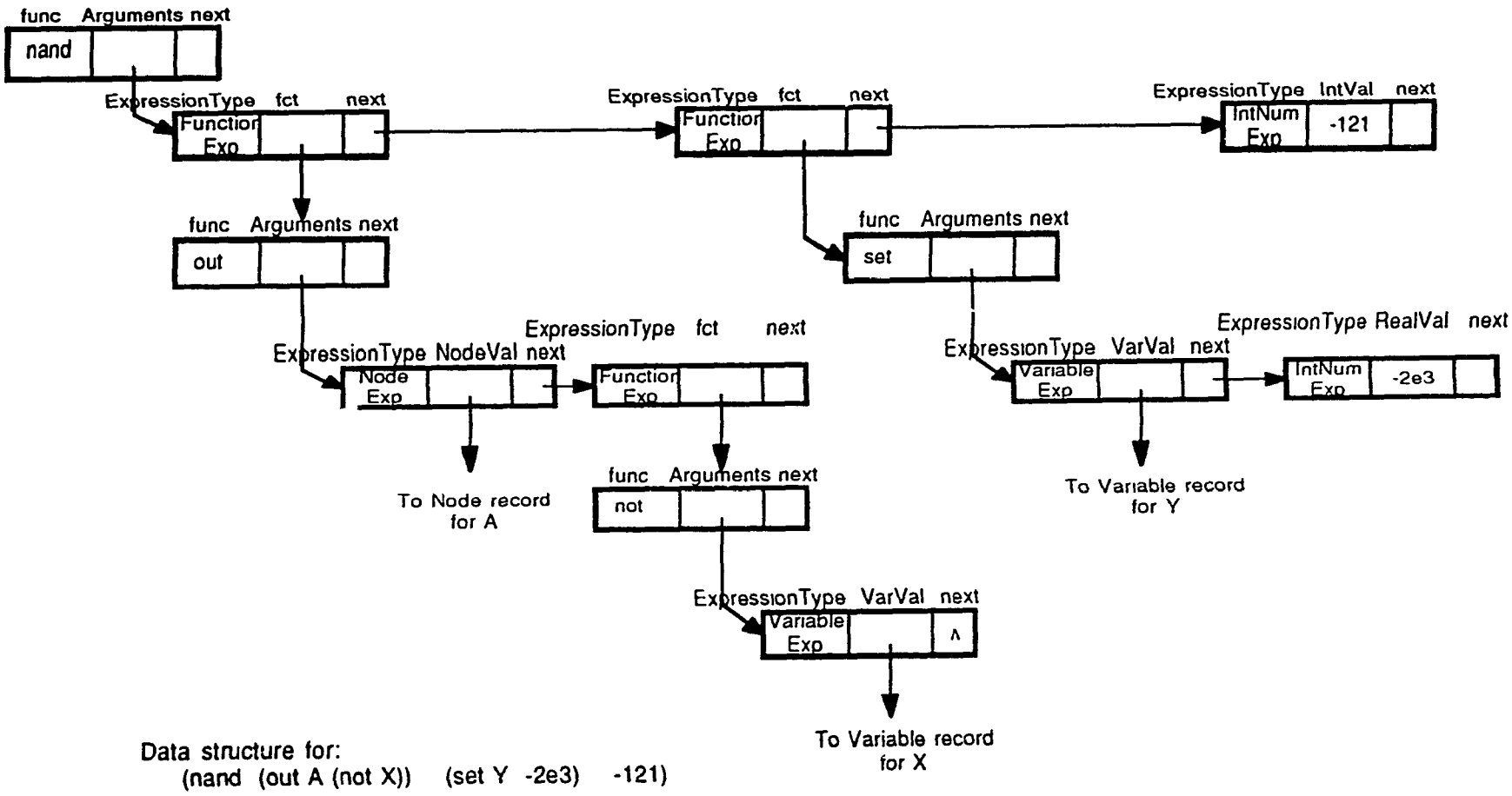


Fig. 1 Data structure for LISP like functions and expressions

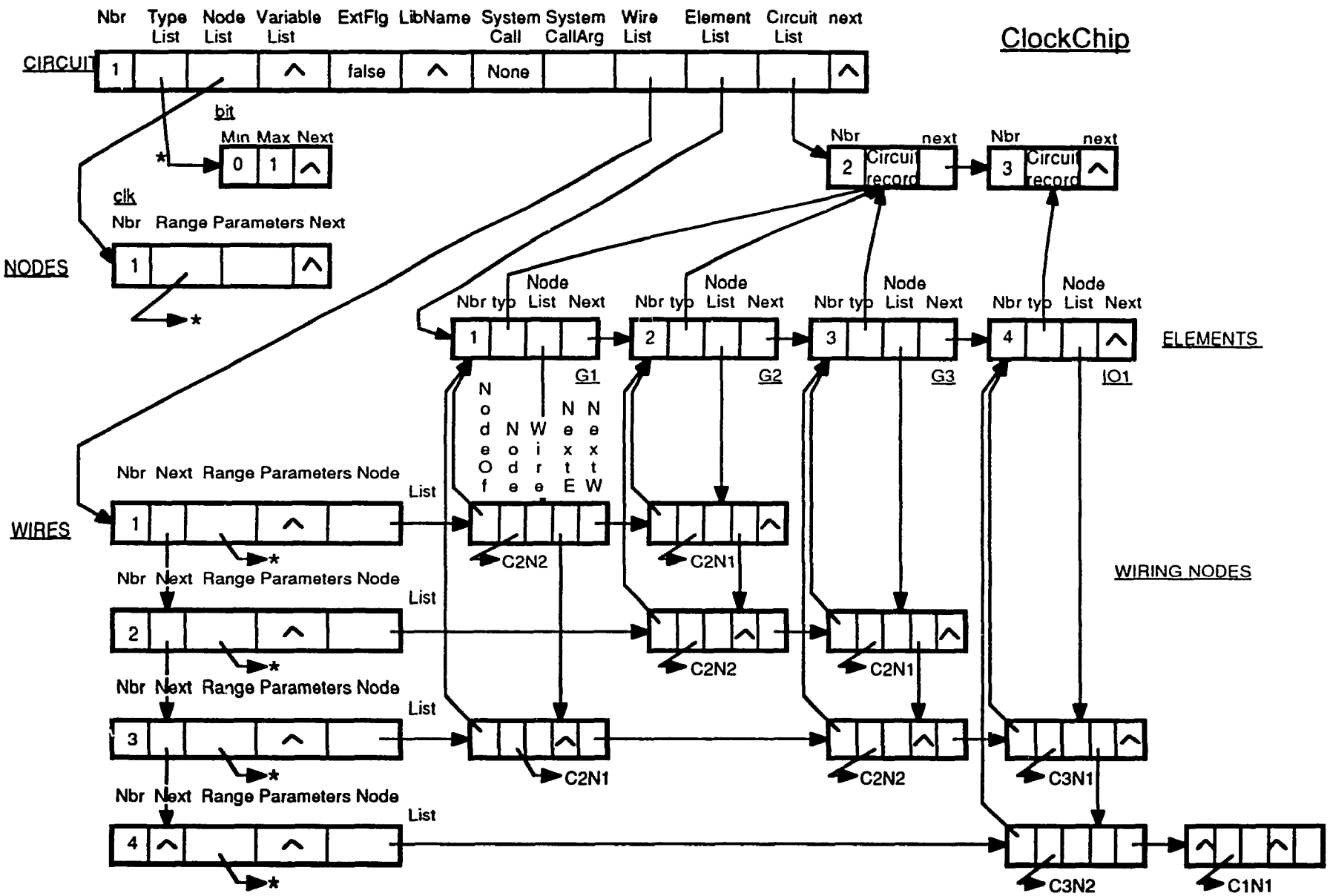


Fig. 2 HDIDS of ClockChip.

Appendix D

CTA based logic models

The most interesting result in this thesis is the abstraction of continuous time to discrete time obtained by the CTA. It was demonstrated that a few CTAs are required to model most logic devices and some of them are described in this appendix. Each CTA was carefully verified but it is always possible that another structure of automaton might be better. The proposed CTAs constitute a working set that should be exercised. Eventually, this should lead to an acceptable set of primitives that could be integrated into VHDL to process timing constraints. A continuity preserving delay model should also replace the transport and inertial delay models used in VHDL. CTA based logic models have the advantage of integrating the timing specifications with the logic function while hiding its processing from the user.

1. gate

Description

As shown in Fig. 1, a gate is a combinational circuit with one or more inputs and one output. A gate behaves properly, i.e. no hazards or glitches are produced on the output for separate input events. A gate is characterized by a logic function which is evaluated to determine the new output. Even if the verification of the separation of input events is not implemented in current logic simulators, it is necessary to specify a minimum separation time in order to perform the continuous time to discrete time abstraction. A gate is also characterized by its propagation delay. Notice that the propagation delay model assumes discrete change of state which is obtained by the CTA and the separation time. The gate class of logic devices includes buffers,

inverters and simple gates (and, or, nand, nor, exor and exnor). It also includes combinational circuits with adequate cover term, for example $X = B.\overline{C} + A.C + A.B$, where the term A.B insures a glitch free transition when C changes.

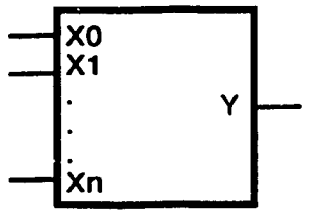


Fig. 1. Typical gate

HDIL description

The description of a typical gate is:

```
circuit nand2
   interface                  {Note this is a comment}
      A B:bit (input  +3,0);
                      {VT}
      X:  bit (output +0.5n +0.5n +0   +5)
                      {tTLH, tTHL,  VL, VH}
              (delay2 +2n   +3n);{tPLH, tPHL}
   implementation  gate  {CTA to be used}
      (timing +0.1n)        {Input timing constraints}
              {ts}
      (function (out X (nand A B))) {Logic function}
end nand2
```

This description includes the list of interface nodes, the simulator arguments, the type of CTA (gate), the device timing arguments and the logic function. For a gate, there are two classes of signals: inputs (A, B) and output (C). Each class of node has simulator arguments: thresholds (V_{TH}) for input nodes, transition times (t_{TLH}, t_{THL}) and logic levels (V_L, V_H) for output nodes. The description also includes the propagation delays (t_{PLH}, t_{PHL}), the input timing constraints (t_S) and the logic function (nand). If the propagation delays t_{PLH} and t_{PHL} are differents, a continuity-preserving delay model such as the propagation delay model described in chapter 4 must be used.

The separation time (t_s) defines the input timing constraint used for the discretization of the change of state. All events on input signals that cause a change in the output must be separated by t_s. except for pair of 01 or 10 events. A typical sequence of events that will be accepted by the CTA without changes is shown in Fig. 2. The output becomes undefined if the function evaluates to undefined or if more than two consecutives events are close (timing violation). Undefined (U) or open (Z) events are both treated as undefined. The output returns to a normal state after a delay equal to the separation time t_s. Since t_s indicates the separation time required for adequate operation it seems natural to allow the same time to return to a normal state.

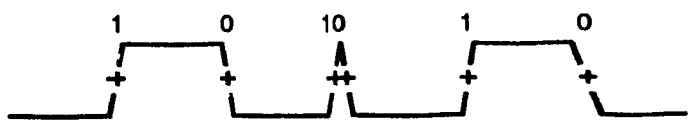


Fig. 2. Acceptable sequence of events for the gate CTA

Continuous Time Automaton

The CTA of Fig. 3 models the behavior of a gate. The basic function of the automaton is to transform the continuous change of state into a discrete change of state or a timed state sequence. The continuous stream of input events is parsed and divided into a finite number of sequences leading to normal behavior and an infinite number of sequences leading to an unspecified or abnormal behavior. The CTA is made of states and arcs and uses the timing constraint and the logic function as arguments. Solid arcs indicates a lower bounded transition time and dotted arcs do not have lower bound. Therefore cycles with at least one solid arc result in a timed state sequence and since there is a unique undefined state associated with the dotted cycles, cycles with only dotted arcs also result in a timed state sequence.

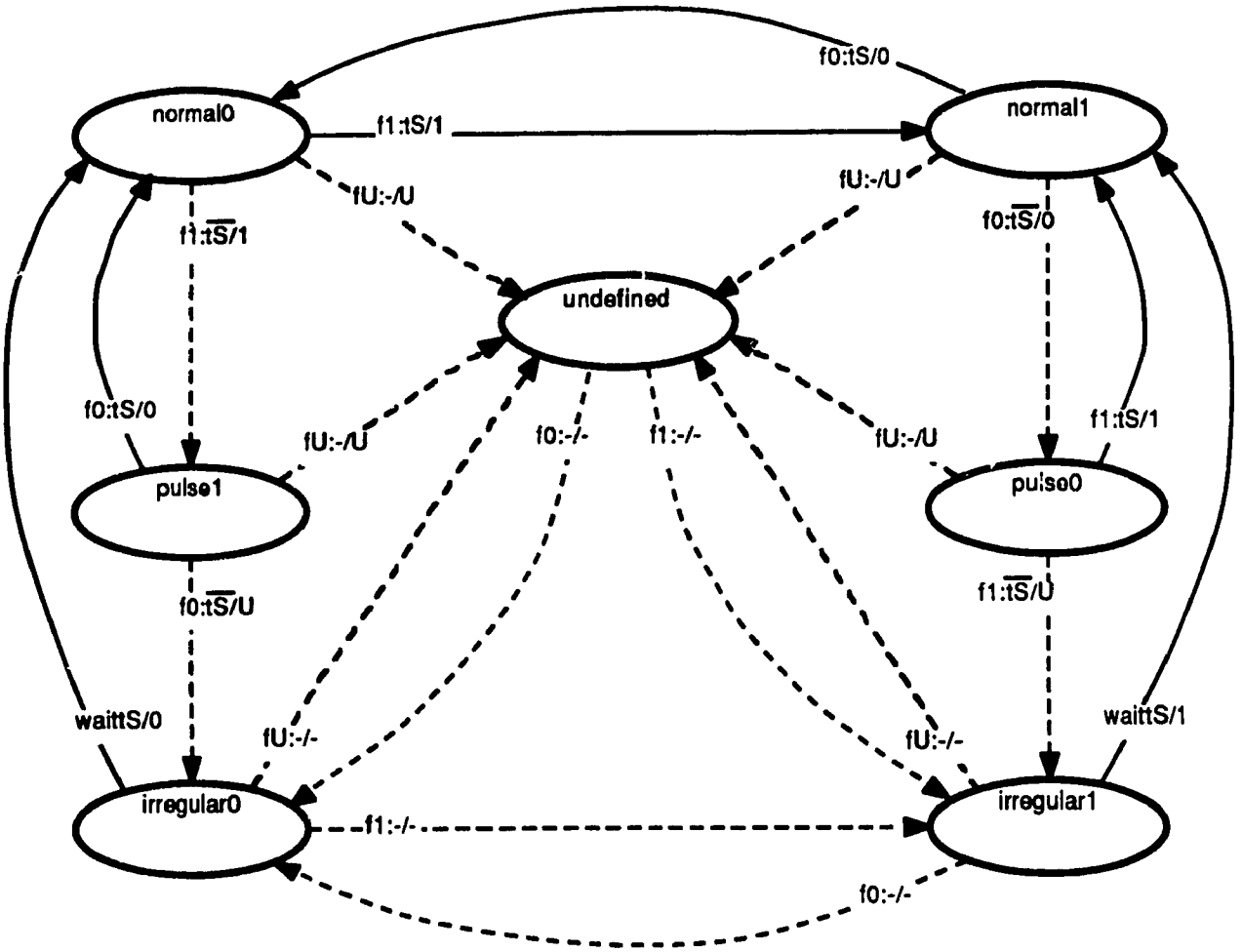


Fig. 3. CTA of a gate

In the case of a gate the function is first evaluated and if the result is different from the current output, a change of state of the CTA is performed. The separation time argument (t_S) is compared with the actual separation between the current and the last change of state to determine if the input sequence leads to normal or undefined behavior. Depending on the new value and the separation time, the next state and the output events are computed. The delay model and the propagation delay arguments (t_{PLH}, t_{PHL}) are then used to generate the output event, outside the CTA.

There are seven states in the gate CTA. In the remainder of this section, we consider the input events as the result of evaluating the function. The normal0 and normal1 states indicate that separate 0 and 1 events have been received. The pulse0 and pulse1 states indicate that a pair of 10 or 01 events separate from preceding events has been received. The states irregular0 and irregular1 indicate that two or more close 0 or 1 events were received. Finally, the undefined state indicates that the result of evaluating the function is undefined.

Normal operation consists of cycles with at least one solid arc. There are three such cycles. The first cycle between normal0 and normal1 consists of sequence of separate 0 and 1 events. The cycles between normal0 and pulse0 and also between normal1 and pulse1 are triggered by separated pairs of 01 or 10. While in pulse0 or pulse1, if a second close event is received, the output is made undefined and the CTA state changes to irregular0 or irregular1. In order to exit from this abnormal or undefined condition, the input must be stable for at least t_S. If a sequence of close 0 and 1 events are received, the CTA state toggles between irregular0 and irregular1 and the output remains undefined. Finally, if the function evaluates to undefined, the CTA state changes to undefined and will return to a normal state after t_S (through irregular0 and irregular1 states) as soon as a 0 or a 1 event is received.

2. Combinational circuit

Description

A combinational circuit is a circuit similar to a gate, except that for each input events, the output goes through an undefined state before it settles to its stable value. As shown in Fig. 4, a combinational circuit has one or more inputs and one output. It is characterized by a logic function, a propagation delay t_P and a timing constraint t_U. Each change on the input generates an undefined period of length t_U after t_P. The transformation of continuous time into discrete time is obtained by extending the

undefined period. This class of logic devices includes ROMs and most PLAs and decoders. It also includes combinational circuits without adequate cover term, for example $X = B.\overline{C} + A.C$, where a change on the signal C may cause a glitch.

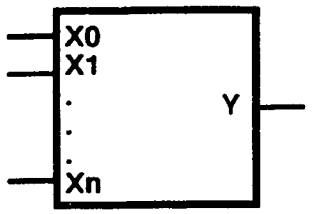


Fig. 4. Typical combinational circuit

HDIL description

The description of a typical combinational circuit is:

```
circuit dec
   interface
      A B C:bit (input  +3,0);
                        {VT}
      Y:  bit (output +0.5n +0.5n +0  +5)
                        {tTLH, tTHL,  VL, VH}
              (delay1 +10n);
                        {tP}
   implementation CC
      (timing +40n)
             {tU}
      (function (out Y (or (and B (not C))(and A C))))
end dec
```

This description includes the list of interface nodes, the simulator arguments, the type of CTA (CC), the device timing arguments and the logic function. As for a gate, there are two classes of signals: inputs (A, B, C) and outputs (Y). The description also includes the timing constraint (t_U), the propagation delays (t_P) and the logic function (or ...). A typical sequence of events is shown in Fig. 5. The output becomes undefined for each event on any input nodes and remains undefined if input events are close. Notice that the output signal is time constrained.

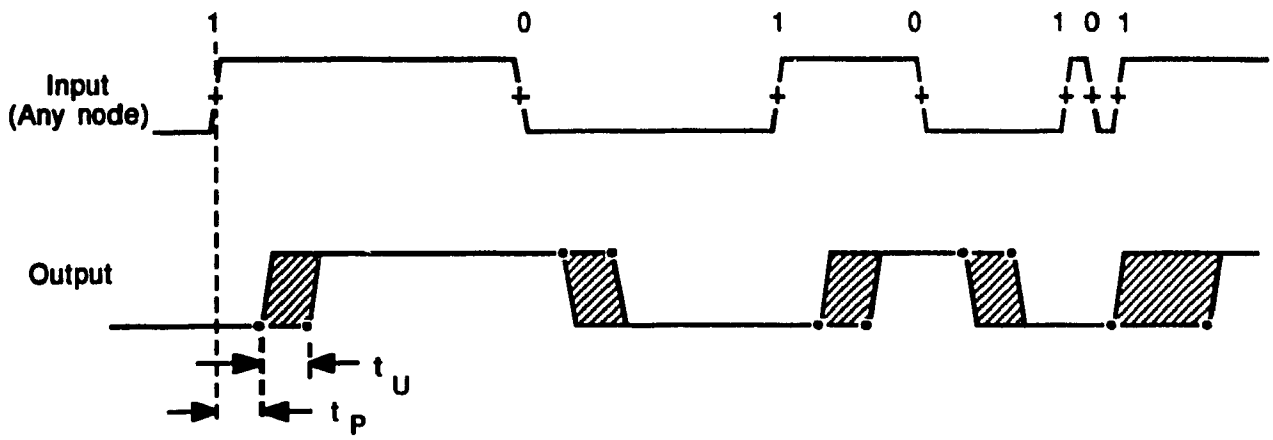


Fig. 5. Typical sequence of events for a combinational circuit

Continuous Time Automaton

The CTA of Fig. 6 models the behavior of a combinational circuit (CC). In the case of a combinational circuit, each input event causes a change of state in the CTA. The undefined time is used as input timing constraint to discriminate acceptable sequences from unacceptable ones. An unacceptable sequence is a sequence of events closer than t_U that will produce a continuous undefined output. Unacceptable sequences are simply useless.

There are four states in the combinational circuit CTA. The normal state indicates that the undefined period has elapsed and that the output is set to 0 or 1. The pending0 and pending1 states indicate that the output is undefined and that a 0 or a 1 event should be produced after t_U. The undefined state indicates that the function evaluates to undefined.

Starting in the normal state, the function is first evaluated for each new event and an undefined event is produced. The next state depends on the function new value. If the new state is pending0, a 0 event will be produced after t_U, unless another event is received. In such case, the function is evaluated and depending on the new value, the next state will be pending0, pending1 or undefined. When in state pending0

or pending1, the CTA waits t_U from the time of the last event before producing the pending event. This effectively extends the undefined period for close events.

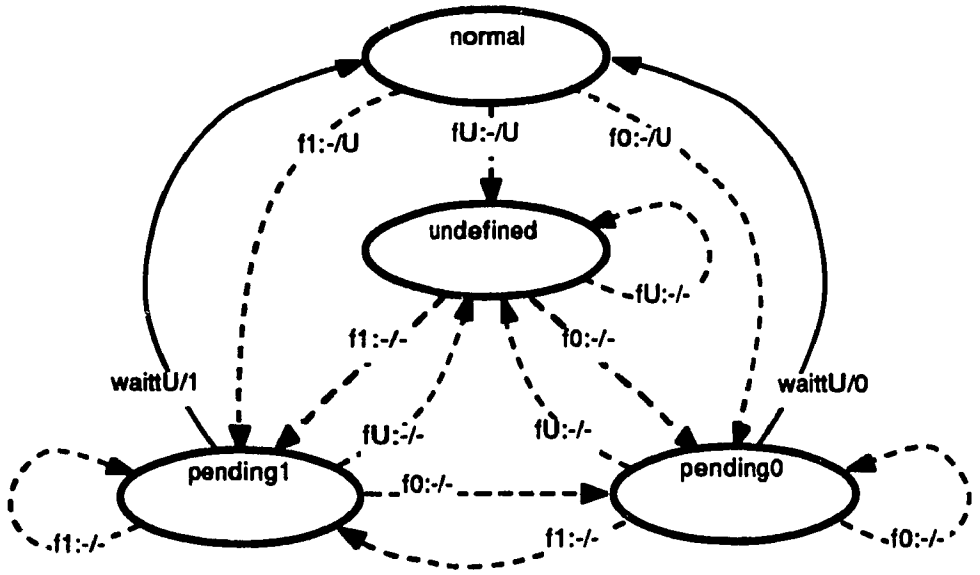


Fig. 6. CTA for a combinational circuit

Normal operation consists of cycles with at least one solid arc. There are two such cycles: between normal and pending0 and between normal and pending1. Finally, if the function evaluates to undefined, the CTA state changes to undefined.

3. 3-state devices

Description

Many logic devices have a special output stage that allows the output to become high impedance. A dedicated input signal is used to control the high impedance state of the output. The model of a typical device with this capability is illustrated in Fig. 7. As it is the case with real devices, like ROM, the enable line

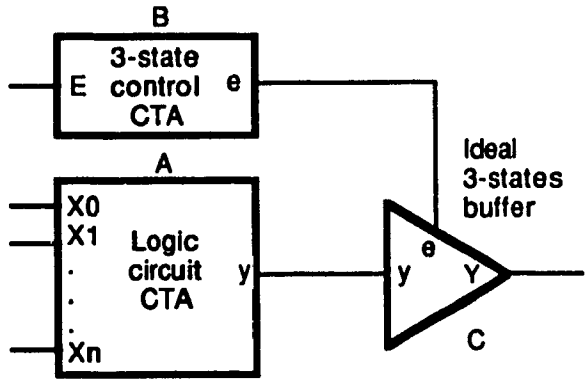


Fig. 7. Typical 3-state device

does not interfere with the logic function and it is assumed that the logic circuit can be separated from the 3-state control circuit and that a perfect 3-state buffer combines the signals y and e. The logic circuit CTA performs the continuous to discrete time abstraction for signal y and the 3-state control circuit CTA does the same for the signal e. Since both signals y and e are discrete, their combination Y will be. The CTA for the 3-state control circuit is described in this section. The ideal 3-state buffer does not have timing constraints, it might have a propagation delay and it has the following logic function:

y	e	Y
-	0	Z
-	U	U
0	1	0
1	1	1
U	1	U

HDIL description

The ideal 3-state buffer does not need a CTA and the following description is used for the 3-state control circuit:

```
circuit 3_state_control_circuit
   interface
      E: bit (input +2.5);
                  {VT}
      e: bit (output +1n   +1n  +0  +5);
                  {tTLH, tTHL, VL, VH}
            (delay2 +2n   +3n);
                  {tPLH, tPHL}
   implementation 3SE
      (timing +5n +4n)
            {tL, tH}
      (function (out e E))
end 3_state_control_circuit
```

There are two types of interface nodes: input (E) and output (e). The description contains the usual simulator arguments for the interface nodes and the timing specification arguments used by the CTA (3SE). In this case, the logic function is simple but one could imagine two or more signals anded to control the 3-state output. If the propagation delays t_{PLH} and t_{PHL} are differents, the delay model described in chapter 4 must be used. The input timing constraints t_L and t_H shown in Fig. 8 are used for the discretization of time, being respectively the minimum low time and minimum high time for the signal E.

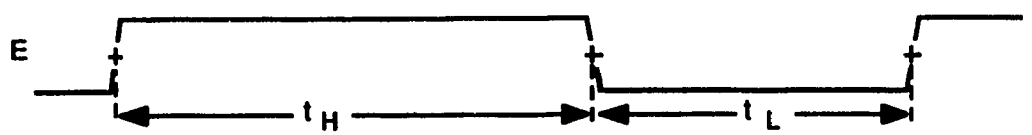


Fig. 8. Input timing constraints on E

Continuous Time Automaton

The CTA for the 3-state control circuit is given in Fig. 9. The CTA uses the input timing constraints t_L and t_H to differentiate acceptable and unacceptable sequences of events. An acceptable sequence is simply a sequence where the signal E is low for at least t_L and high for at least t_H. There are two normal states HiZ and Enable corresponding to E=0 and E=1, one Undefined state for E=U and two irregular states HiZirregular and EnableIrregular. The 3-state control circuit operates normally with separate events and the CTA toggles between HiZ and Enable states. If either

t_L or t_H is not met, the output is made undefined (U) and the next state is one of the irregular states. The output will remain undefined as long as the timing specifications are not met. The output also becomes undefined if the input is undefined. Following an undefined input, the output will return to a normal state if the input returns to 0 or 1, but after going through an irregular state. This adds a delay to let the device move out of an undefined condition and insure the discrete change of state.

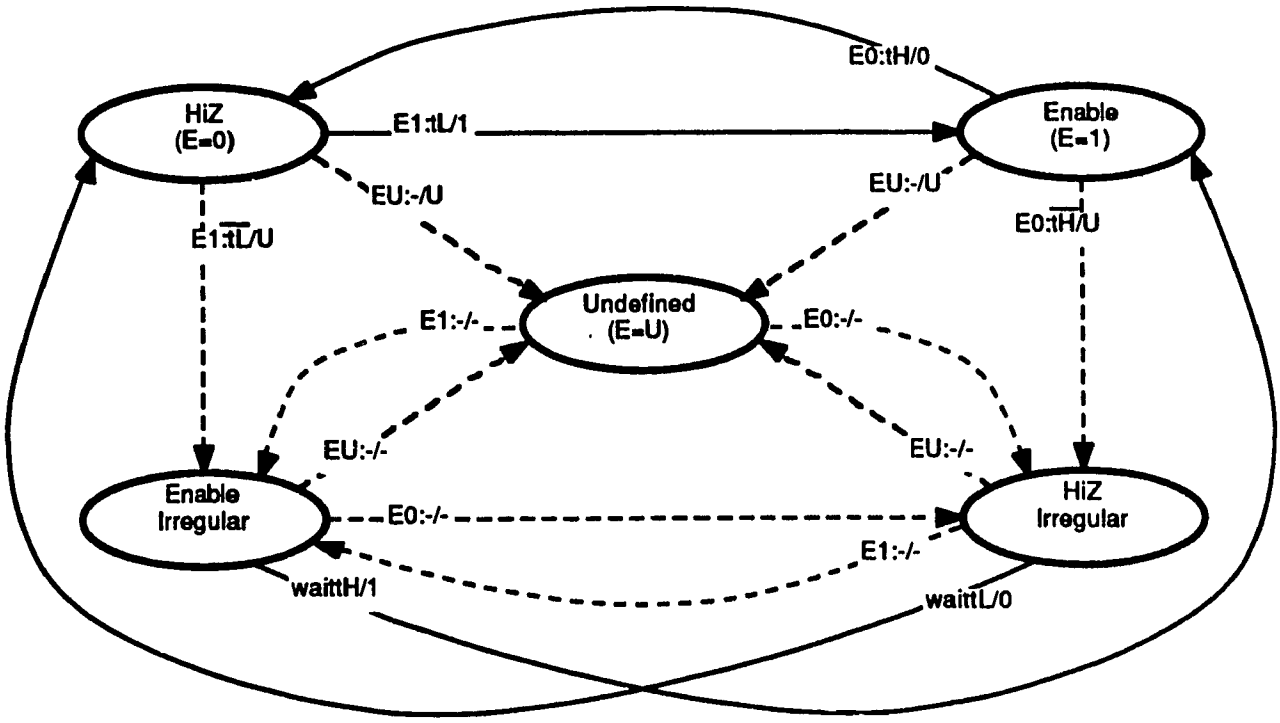


Fig. 9. CTA for a 3-state control circuit

4. Flip-flops

Description

The flip-flop is the device that stores one bit of information. As shown in Fig. 10, the flip-flop has zero or more inputs, one or two outputs and a clock. Assuming an edge triggered flip-flop, the output changes on the rising edge of the

clock to a value determined by a function of the inputs (SR, D, JK) and the present state. Flip-flops require stable input values at the clock rising edge and the clock signal is limited to a specified frequency.

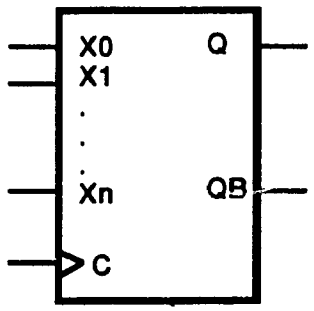


Fig. 10. Typical flip-flop

HDIL description

The following is the description of a typical D flip-flop:

```
circuit Dflipflop
   interface
      D: bit (input  +2,0);
                     {VT}
      C: bit (clock  +2,0);
                     {VT}
      Q QB: bit (output +0.5n +0.5n +0  +5)
                        {tTLH, tTHL, VL, VH}
                 (delay2 +2n   +3n);
                          {tPLH, tPHL}
   implementation FlipFlop
      (timing 5n 5n +0.5n +0n)
             {tL tH  tSU   tHD}
      (function (exe (out Q D) (out Qb (not D))))
end Dflipflop
```

A flip-flop has three types of interface nodes: inputs, outputs and clock. Simulator arguments are specified for the clock as for an input. The description includes the type of CTA, the timing specification and the logic function. The logic function describe how the new value of Q and Qb are computed. The timing specification includes two specification to limit the clock frequency (t_L and t_H), the set-up time (t_{SU}) and the hold time (t_{HD}). Except for the use of master-slave

events, these definitions are similar to those normally used to specify flip-flops. This is illustrated in Fig. 11.

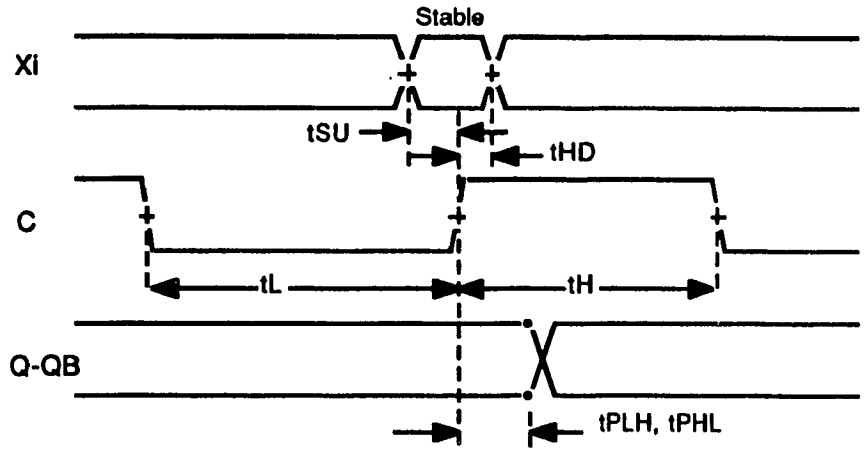


Fig. 11. Timing constraints and propagation delays for a SSC

Continuous Time Automaton

The CTA of the flip-flop is given in Fig. 12. It consists of six states corresponding to the state of the clock signal. In a normal cycle, the clock toggles between 0 and 1 and the inputs are stable near the rising edge of the clock. Starting in state clock0, if the clock goes to 1 and the set-up time (t_{SU}) and the low time (t_L) are met, the CTA state changes to clock1a, the function is evaluated and the output events produced if required (C1:$t_{SU}t_{HD}$/f). Once in state clock1a, the CTA waits for the hold time (t_{HD}) and then moves to state clock1b (waitt_{HD}). This effectively checks the hold time. It is assumed that $t_{HD}<t_H$. Then the CTA state changes to clock0 if the clock signal goes to 0 and if the high time is met (C0:t_H/-).

If any of the timing constraints are not met, the CTA state changes to one of the irregular states and makes the output undefined. The CTA will return to a normal cycle if the clock events meets the timing constraints. Finally, the CTA state becomes undefined if the clock signal is undefined (clockU). Most of the events on the

inputs do not have any effect (X-:-/-), except in state clock1a where an event on X violates the hold time. If for any reason, a new value does not change the output, no event will be produced. This will occur every time the function f evaluates to the same value on consecutive positive clock edges.

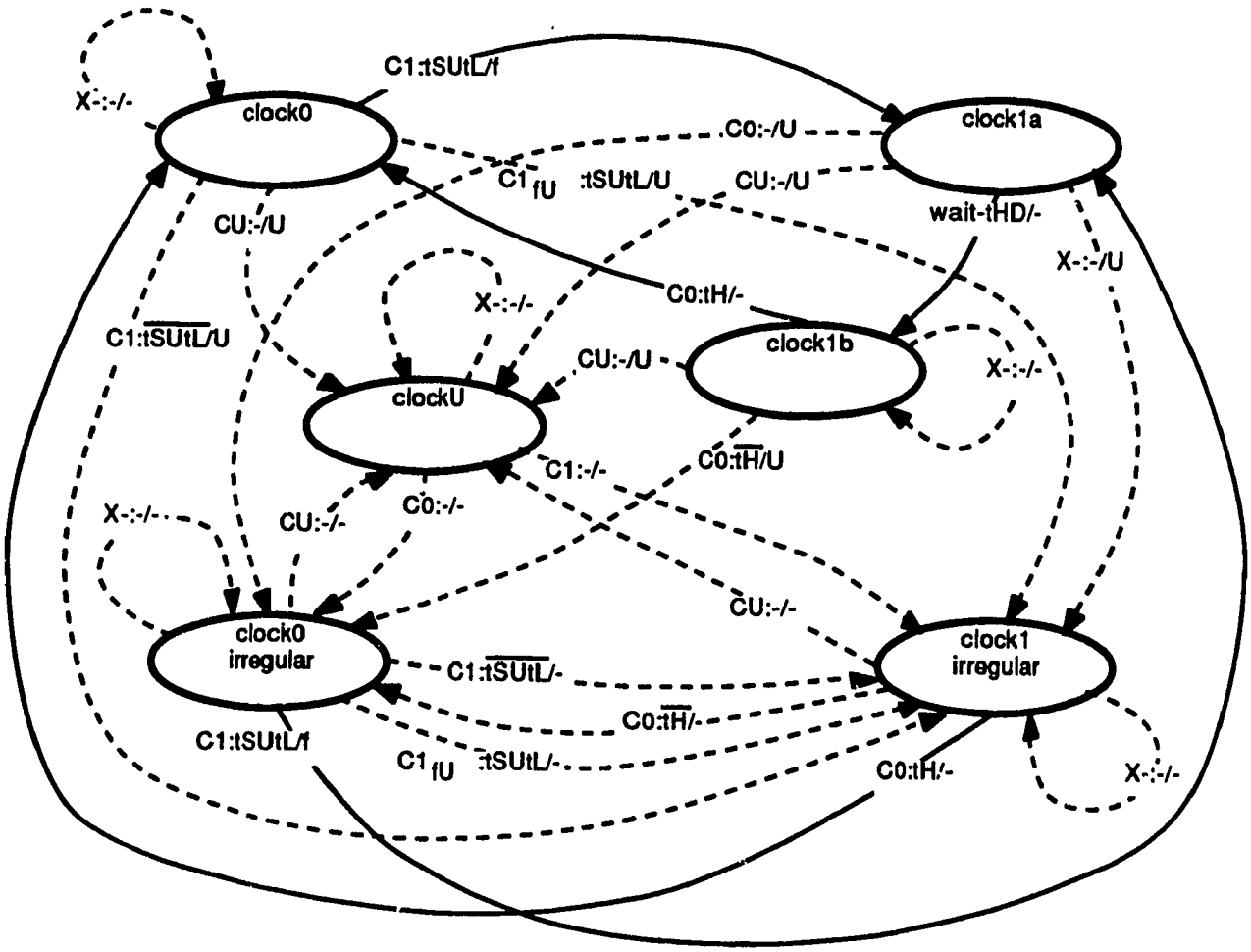


Fig. 12. CTA for a flip-flop

5. Synchronous sequential circuit

Description

As depicted in Fig. 13, a Synchronous Sequential Circuit (CSS) is similar to a flip-flop. The flip-flop does not have an explicit logic state whereas the the user can specify one in a SSC. In addition, the SSC may have many outputs. A SSC is classified as Mealy machine if the output depends on the inputs otherwise it is a Moore machine. The SSC described here is a Moore machine and the output only depends on the state. As discussed in chapter 4, Mealy machines require a separate CTA for the output decoder. A SSC is composed of an input decoder, an output decoder and a memory element. The SSC modeled in this section has synchronous inputs, a clock and synchronous outputs. The inputs must be stable during state change and the change of state occurs on the rising edge of the clock. In addition, the output decoder might behave like a gate or like a combinational circuit. It is assumed that it behaves like a gate and therefore the outputs will change without glitches. Variations of SSC are described in chapter 4.

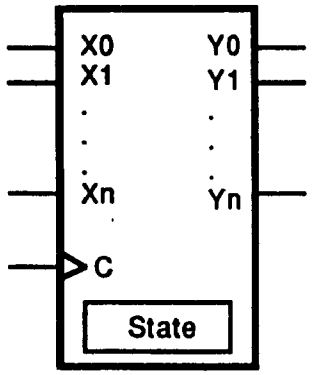


Fig. 13. Typical synchronous sequential circuit

HDIL description

The following is a description of a synchronous sequential circuit. It is a divide by three circuit with two inputs and two outputs.

```
circuit div3
   interface
      UP RESET: bit (input +2.5);
                            {VT}
      C: bit  (clock +2.5);
                     {VT}
      Q0 Q1: bit (output +1n  +1n   +0  +5);
                         {tTLH, tTHL, VL, VH}
            (delay1 +2n);
                   {tP}
   variable
      count:0 1 2;

   implementation SSC
      (timing +5n +5n +2n +0n)
            {tL  tH  tSU tHD}
      (function
         (case RESET
            U (exe (set count U)(out Q0 U)(out Q1 U))
            1 (exe (set count 0)(out Q0 0)(out Q1 0))
            0 (case UP
                  U (exe (set count U)(out Q0 U)(out Q1 U))
                  0 ()
                  1 (case count
                        U (exe (set count U)(out Q0 U)(out Q1 U))
                        0 (exe (set count 1)(out Q0 1)(out Q1 0))
                        1 (exe (set count 2)(out Q0 0)(out Q1 1))
                        2 (exe (set count 0)(out Q0 0)(out Q1 0))
                     )
               )
         )
      )
end div3
```

The SSC description is very similar to the flip-flop description. It includes input nodes, output nodes and a clock. In addition, a SSC includes a state variable (count). Notice that the function out is used for output and the function set is used for state variables. The timing specification is identical to the flip-flop and so is its processing.

Continuous Time Automaton

The CTA of the SSC is identical to the CTA of the flip-flop given in Fig. 12.

6. Asynchronous inputs in flip-flop

Description

While discussing event combination in a flip-flop (section 4.7.5), it was demonstrated that a separate CTA was required to model the asynchronous inputs such as the set and the reset inputs of a flip-flop. The first CTA that combines the clock and the data inputs is similar to a SSC except for the output. As shown in Fig. 14, a new output cq is used. The signal cq can take three declared values 0, S0, S1 and the undefined value U. This signals combines the information of the flip-flop inputs C and Xi. If the clock falls there is a 0 event on cq and if the clock rises, the events on cq will either be S0, S1 or U, depending on the D input. In addition, the signal cq is U if the clock is U or if there is a timing violation. The CTA for the first part of the flip-flop (A) is similar to the CTA of a flip-flop.

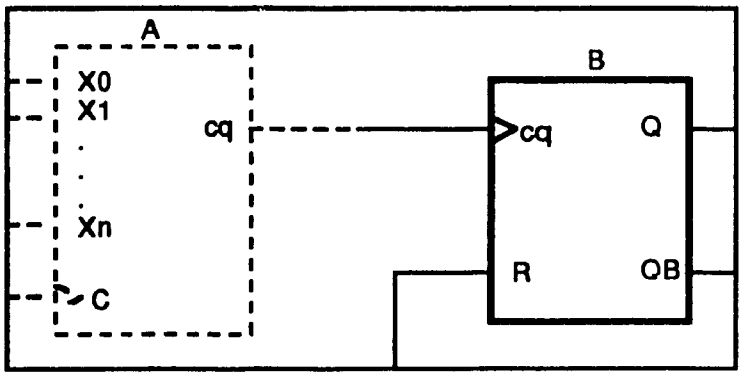


Fig. 14. Model of a flip-flop with asynchronous reset

The device that models the combination of the asynchronous input (R) and the special clock (cq) is shown in Fig. 14 (B). The operation of such a device is simple, unless there is an overriding condition (R=1), the output Q is controlled by the special input clock cq. As usual, if the input R equals to 1, it overrides the input cq and forces a 0 on Q. It is extremely important to realize that the signal cq has been parsed by a CTA and is therefore a timed sequence of events. Therefore, there is no need for any

input timing constraints on cq. Input timing constraints are required for the overriding signal R and between R and cq. The description and the processing of these constraints are described in this section. The device operation is illustrated in Fig. 16. There are two timing constraints associated with the asynchronous reset (R). First, the signal R must be active for at least t_W and second, a release time (t_R) is necessary after the overriding input becomes inactive. During the release time, the signal cq must not change to S1 or U.

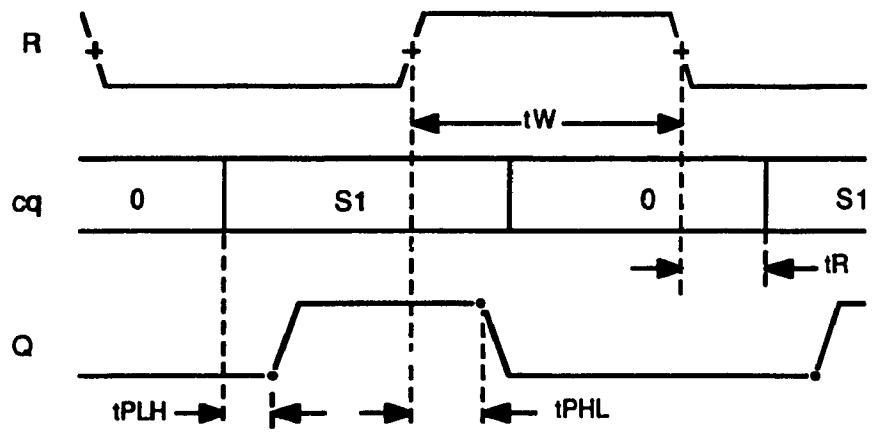


Fig. 15. Timing constraints and propagation delays for the asynchronous reset (R)

HDIL description

This is the circuit that combines the special clock (cq) and the reset (R):

```
circuit ResetCircuit
   interface
      cq: 0 S0 S1 (clock +2.5);
                        {VT}
      R: bit (input +2.5);
                  {VT}
      Q QB: bit (output +1n +1n +0 +5);
                        {tTLH, tTHL, VL, VH}
                (delay2 +2n   +3n);
                        {tPLH, tPHL}
   implementation AR
      (timing +20n +5n)
            {tW    tR}
```

```
        (function (if R (exe (out Q +0) (out Qb +1))))
end ResetCircuit
```

The HDIL description includes the list of interface nodes, the simulator arguments, the type of CTA and the device arguments. There are three classes of nodes: overriding input (R), special clock (cq) and outputs (Q and QB). As discussed earlier, there are two timing constraints (t_W, t_R) and two propagation delays (t_{PLH}, t_{PHL}).

Continuous time automaton

The CTA of Fig. 16 models the behavior of the asynchronous reset of a flip-flop. There are five states in the CTA. The normal state indicates that the overriding input is inactive and that the output is controlled by cq. As soon as an overriding condition is detected (R1:-/0), the output is reset and the state changes to forced0. Once in forced0, the input cq does not have any effect (cq-:-/-). If the R input returns to 0 and the pulse width is large enough (R0:t_W/-), the state changes to releasing0.

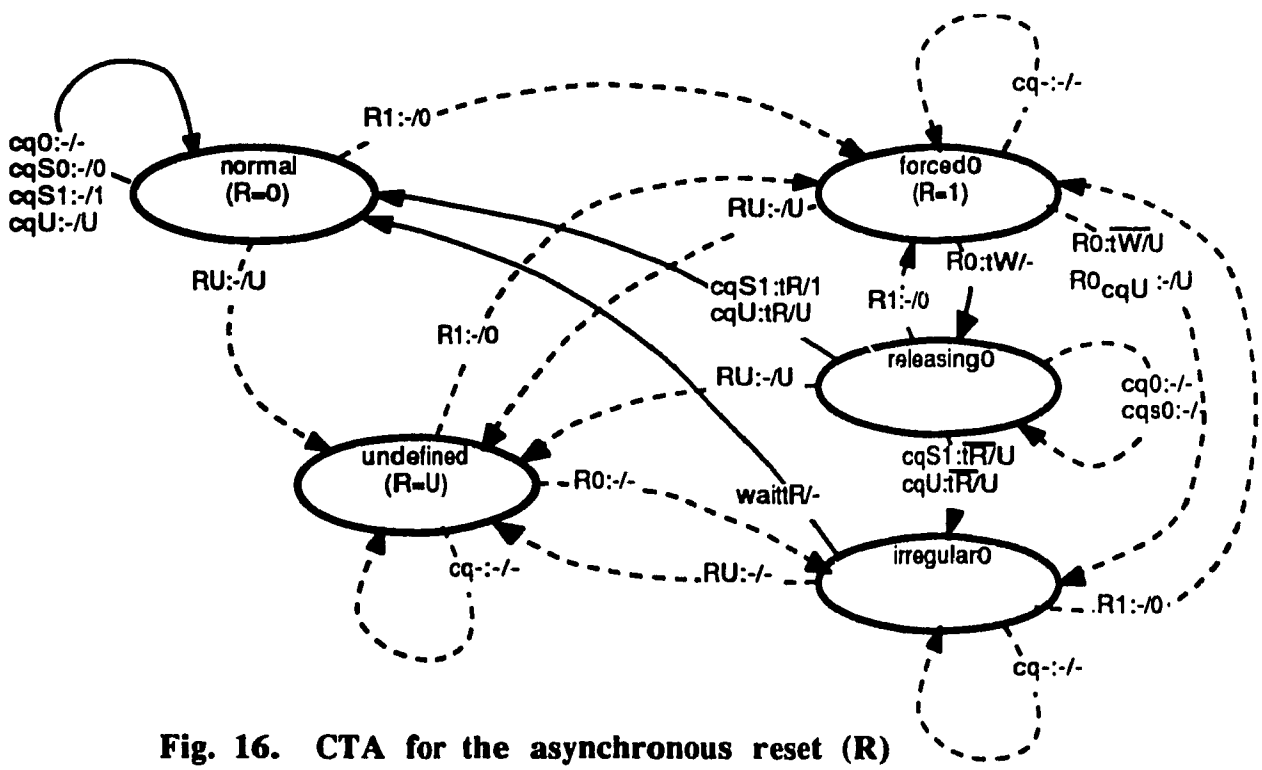


Fig. 16. CTA for the asynchronous reset (R)

The CTA returns to normal if cq is either S1 or U and if the release time is met ($cqS1{:}t_R/1$ and $cqU{:}t_R/U$). While in state releasing0, if the clock is S1 or SU, the timing constraint t_R is violated and the state changes to irregular0. The state also changes to irregular0 from forced0, when the reset is removed while the clock is already undefined ($R0_{cqU}{:}\text{-}/U$) or when the timing constraint t_W is violated ($R0{:}\ \overline{tW}\ /U$). The CTA returns from irregular0 state to normal state after t_R ($waitt_R$). From any state, the CTA state changes to undefined if the reset signal is U.

7. General Synchronous Sequential Circuit (GSSC)

Description

The SSC described in section 5 has synchronous inputs and synchronous outputs. A synchronous input is an input that must be stable during an interval determined by the set-up and hold times. A synchronous output is an output that neatly changes, without producing glitches. The GSSC of Fig. 17 also has synchronized inputs (SXi) and asynchronous outputs (AYi). Synchronized inputs have been internally synchronized with one or more synchronization latches and can

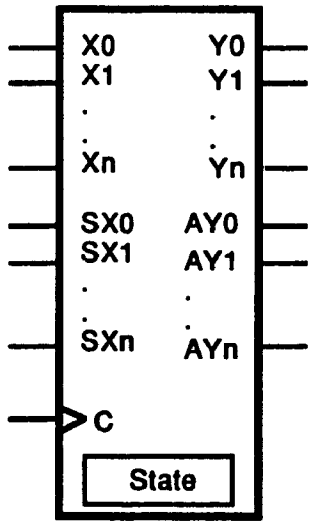


Fig. 17. Typical general synchronous sequential circuit (GSSC)

change without restriction with respect to the clock. The value at the edge of the clock is used in the logic function.

The behavior of asynchronous outputs is similar to the behavior of a combinational circuit. Every change in the output is preceded by an undefined period. In a GSSC, it is also assumed that changes of state and output events occur on both edges of the clock. This is illustrated in Fig. 18. The logic function must therefore include references to the clock. The 68000 microprocessor is being modeled and it uses the GSSC.

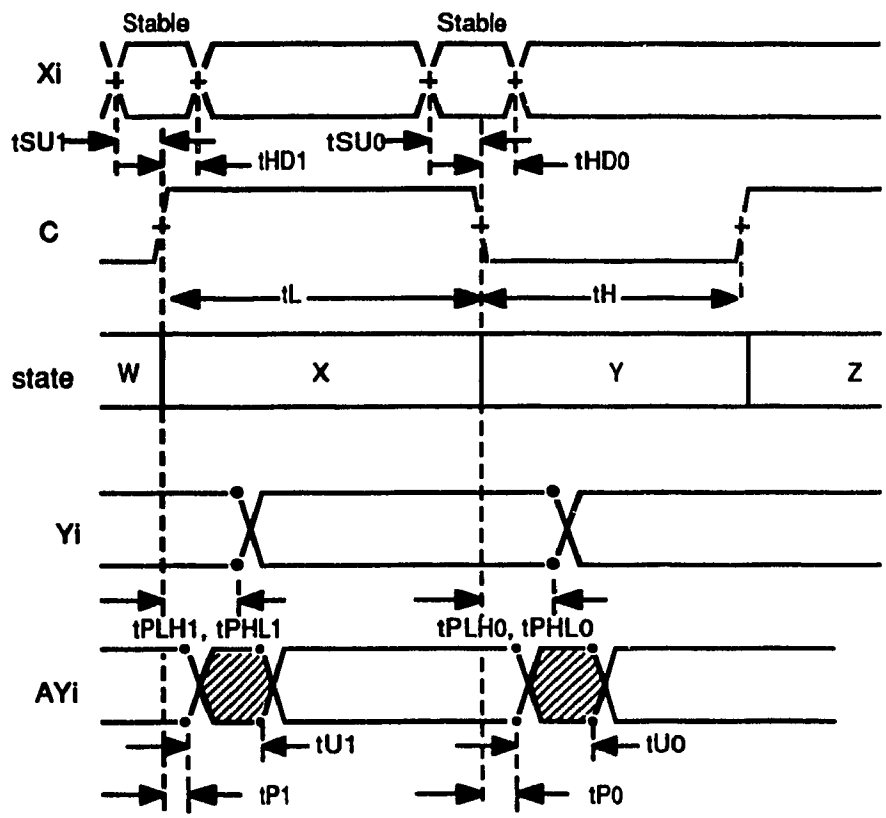


Fig. 18. Timing constraints and propagation delays for a GSSC

Continuous Time Automaton

The CTA for the GSSC is similar to the CTA of a flip-flop or a SSC where the function is used on both edge of the clock and the clock0 state is also divided into

clock0a and clock0b to verify the hold time on the falling edge. The use of GSSC to model complex devices is still under development.

Appendix E

Listing and simulation results

This appendix contains the listing of the simulation of two circuits. The circuits were used to verify the simulator's performance in term of events/sec. Each listing has two parts. The first part titled LISTING is the circuit description. The second part titled EVENTS is the list of events computed during simulation. The list of events is provided for each signal. A signal begins with its number (<1>) followed by the initial forced value (IFV) and the initial computed value (ICV). For each signal the master events (M) followed by the associated slave events (S) are given on separate lines. Information like the event value, the type of event ('d'ummy, 'n'ormal and 'i'rregular), the father event (signal number - slave event number), the time, the node on which it occurred and the previous state of the corresponding device are shown. The actual simulation performance was based on a much larger number of events. The simulation proved that algorithms based on master slave events and using continuous time automaton are possible and relatively efficient.

1.0 Simple clock made of three inverter

```
*************************************************************************
                                LISTING
*************************************************************************
circuit clock
  interface
  model
  implementation

    circuit inverter
      interface
        node
          a:+0 +1 (input +3,0 {threshold});
          b:+0 +1 (output +1,2e-9 +2,4e-9 {rise and fall time});
      model Gate
        (timing +1,2e-9 +2,4e-9 +8,2e-9 +9,4e-9)
```

```
13          (lambda (set b (not a)))
14        implementation
15      end inverter
16
17      element
18        G1 G2 G3:inverter;
19      wire
20        G1#b G2#a (IFV +1);
21        G2#b G3#a ;
22        G3#b G1#a ;
23  end clock


clock=                  0
sysclock=             183
event number=           8
clock=                  0
sysclock=             183

*************************************************************************
                              EVENTS
*************************************************************************


===== <1> ===== IFV =    1 ===== ICV =    0 ============================
   nbr  value typ   father        time              node  previous state
M           1  d
S                                                    G2#a

M    0:     0  s                    .        0       G1#b
S    1:                                     960      G2#a  FSMstate =   0
M    3:     1  n   2 -   3                 6240      G1#b
S    4:                                    6960      G2#a  FSMstate =   1
                                                           t1 = 960
M    6:     0  i   5 -   6                12240      G1#b
S    7:                                   13200      G2#a  FSMstate =   0


===== <2> ===== IFV =   IU ===== ICV =    0 ============================
   nbr  value typ   father        time              node  previous state
M           0  d
S                                                    G3#a

M    1:     1  n   0 -   1                 2160      G2#b
S    2:                                    2880      G3#a  FSMstate =   0
M    4:     0  i   3 -   4                 8160      G2#b
S    5:                                    9120      G3#a  FSMstate =   1
                                                           t1 = 2880


===== <3> ===== IFV =   IU ===== ICV =    1 ============================
   nbr  value typ   father        time              node  previous state
M           1  d
S                                                    G1#a

M    2:     0  n   1 -   2                 4080      G3#b
```

```
S    3:                          5040          G1#a  FSMstate =   0
M    5:     1  i   4 -   5      10320          G3#b
S    6:                         11040          G1#a  FSMstate =   1
                                                     t1 = 5040
```

2.0 Ten clocks made of one inverter each having ten functions to evaluate

```
**********************************************************************
                              LISTING
**********************************************************************
    1  circuit clock1
    2    interface
    3    model
    4    implementation
    5
    6      circuit inverter
    7        interface
    8          node
    9            a:+0 +1 (input +2,5 {threshold});
   10            b:+0 +1 (output +1,0e-9 +1,0e-9 {rise and fall time});
   11        model Gate
   12          (timing +1,0e-9 +1,0e-9 +0,5e-9 +0,0e-9)
   13                {tp1     tp0     tsep    tmin}
   14          (lambda (set b (not(not(not(not(not(not(not(not(not(not(not
a))))
)))))))))
   15        implementation
   16      end inverter
   17
   18      element
   19        G1 G2 G3 G4 G5 G6 G7 G8 G9 G10:inverter;
   20      wire
   21        G1#a G1#b (IFV +1);
   22        G2#a G2#b (IFV +1);
   23        G3#a G3#b (IFV +1);
   24        G4#a G4#b (IFV +1);
   25        G5#a G5#b (IFV +1);
   26        G6#a G6#b (IFV +1);
   27        G7#a G7#b (IFV +1);
   28        G8#a G8#b (IFV +1);
   29        G9#a G9#b (IFV +1);
   30        G10#a G10#b (IFV +1);
   31  end clock1


clock=                 0
sysclock=            166
event number=         82
clock=             16666
sysclock=            183


**********************************************************************
                              EVENTS
```

```
*****************************************************************************

===== <1> ===== IFV =     1 ===== ICV =     0 ============================
   nbr  value typ   father        time              node  previous state
M           1  d
S                                                    G1#a

M    9:     0  s                    0               G1#b
S   10:                           500               G1#a  FSMstate =   0
M   19:     1  n   9 -  10       1500               G1#b
S   20:                          2000               G1#a  FSMstate =   1
                                                           t1 = 500
M   29:     0  n  19 -  20       3000               G1#b
S   30:                          3500               G1#a  FSMstate =   0
                                                           t1 = 2000
M   39:     1  n  29 -  30       4500               G1#b
S   40:                          5000               G1#a  FSMstate =   0
                                                           t1 = 3500
M   49:     0  n  39 -  40       6000               G1#b
S   50:                          6500               G1#a  FSMstate =   0
                                                           t1 = 5000
M   59:     1  n  49 -  50       7500               G1#b
S   60:                          8000               G1#a  FSMstate =   0
                                                           t1 = 6500
M   69:     0  n  59 -  60       9000               G1#b
S   70:                          9500               G1#a  FSMstate =   0
                                                           t1 = 8000
M   79:     1  n  69 -  70      10500               G1#b
S   80:                         11000               G1#a  FSMstate =   0
                                                           t6 = 4096

===== <2> ===== IFV =     1 ===== ICV =     0 ============================
   nbr  value typ   father        time              node  previous state
M           1  d
S                                                    G2#a

M    8:     0  s                    0               G2#b
S    9:                           500               G2#a  FSMstate =   0
M   18:     1  n   8 -   9       1500               G2#b
S   19:                          2000               G2#a  FSMstate =   1
                                                           t1 = 500
M   28:     0  n  18 -  19       3000               G2#b
S   29:                          3500               G2#a  FSMstate =   0
                                                           t1 = 2000
M   38:     1  n  28 -  29       4500               G2#b
S   39:                          5000               G2#a  FSMstate =   0
                                                           t1 = 3500
M   48:     0  n  38 -  39       6000               G2#b
S   49:                          6500               G2#a  FSMstate =   0
                                                           t1 = 5000
M   58:     1  n  48 -  49       7500               G2#b
S   59:                          8000               G2#a  FSMstate =   0
                                                           t1 = 6500
M   68:     0  n  58 -  59       9000               G2#b
S   69:                          9500               G2#a  FSMstate =   0
```

```
                                                                    t1 = 8000
M   78:     1  n  68 -  69        10500              G2#b
S   79:                          11000              G2#a  FSMstate =   0

===== <3> ===== IFV =    1 ===== ICV =    0 ============================
   nbr  value typ   father        time              node  previous state
M           1  d
S                                                   G3#a

M    7:     0  s                     0              G3#b
S    8:                            500              G3#a  FSMstate =   0
M   17:     1  n   7 -   8        1500              G3#b
S   18:                           2000              G3#a  FSMstate =   1
                                                            t1 = 500
M   27:     0  n  17 -  18        3000              G3#b
S   28:                           3500              G3#a  FSMstate =   0
                                                            t1 = 2000
M   37:     1  n  27 -  28        4500              G3#b
S   38:                           5000              G3#a  FSMstate =   0
                                                            t1 = 3500
M   47:     0  n  37 -  38        6000              G3#b
S   48:                           6500              G3#a  FSMstate =   0
                                                            t1 = 5000
M   57:     1  n  47 -  48        7500              G3#b
S   58:                           8000              G3#a  FSMstate =   0
                                                            t1 = 6500
M   67:     0  n  57 -  58        9000              G3#b
S   68:                           9500              G3#a  FSMstate =   0
                                                            t1 = 8000
M   77:     1  n  67 -  68       10500              G3#b
S   78:                          11000              G3#a  FSMstate =   0

===== <4> ===== IFV =    1 ===== ICV =    0 ============================
   nbr  value typ   father        time              node  previous state
M           1  d
S                                                   G4#a

M    6:     0  s                     0              G4#b
S    7:                            500              G4#a  FSMstate =   0
M   16:     1  n   6 -   7        1500              G4#b
S   17:                           2000              G4#a  FSMstate =   1
                                                            t1 = 500
M   26:     0  n  16 -  17        3000              G4#b
S   27:                           3500              G4#a  FSMstate =   0
                                                            t1 = 2000
M   36:     1  n  26 -  27        4500              G4#b
S   37:                           5000              G4#a  FSMstate =   0
                                                            t1 = 3500
M   46:     0  n  36 -  37        6000              G4#b
S   47:                           6500              G4#a  FSMstate =   0
                                                            t1 = 5000
M   56:     1  n  46 -  47        7500              G4#b
S   57:                           8000              G4#a  FSMstate =   0
                                                            t1 = 6500
M   66:     0  n  56 -  57        9000              G4#b
```

```
S   67:                                  9500            G4#a  FSMstate =   0
                                                               t1 = 8000
M   76:     1  n  66 -  67              10500            G4#b
S   77:                                 11000            G4#a  FSMstate =   0

===== <5> ===== IFV =    1 ===== ICV =    0 ==============================
   nbr  value typ   father          time                node  previous state
M           1  d
S                                                       G5#a

M    5:     0  s                        0               G5#b
S    6:                                500               G5#a  FSMstate =   0
M   15:     1  n   5 -   6             1500               G5#b
S   16:                                2000               G5#a  FSMstate =   1
                                                               t1 = 500
M   25:     0  n  15 -  16             3000               G5#b
S   26:                                3500               G5#a  FSMstate =   0
                                                               t1 = 2000
M   35:     1  n  25 -  26             4500               G5#b
S   36:                                5000               G5#a  FSMstate =   0
                                                               t1 = 3500
M   45:     0  n  35 -  36             6000               G5#b
S   46:                                6500               G5#a  FSMstate =   0
                                                               t1 = 5000
M   55:     1  n  45 -  46             7500               G5#b
S   56:                                8000               G5#a  FSMstate =   0
                                                               t1 = 6500
M   65:     0  n  55 -  56             9000               G5#b
S   66:                                9500               G5#a  FSMstate =   0
                                                               t1 = 8000
M   75:     1  n  65 -  66            10500               G5#b
S   76:                                11000              G5#a  FSMstate =   0

===== <6> ===== IFV =    1 ===== ICV =    0 ==============================
   nbr  value typ   father          time                node  previous state
M           1  d
S                                                       G6#a

M    4:     0  s                        0               G6#b
S    5:                                500               G6#a  FSMstate =   0
M   14:     1  n   4 -   5             1500               G6#b
S   15:                                2000               G6#a  FSMstate =   1
                                                               t1 = 500
M   24:     0  n  14 -  15             3000               G6#b
S   25:                                3500               G6#a  FSMstate =   0
                                                               t1 = 2000
M   34:     1  n  24 -  25             4500               G6#b
S   35:                                5000               G6#a  FSMstate =   0
                                                               t1 = 3500
M   44:     0  n  34 -  35             6000               G6#b
S   45:                                6500               G6#a  FSMstate =   0
                                                               t1 = 5000
M   54:     1  n  44 -  45             7500               G6#b
S   55:                                8000               G6#a  FSMstate =   0
                                                               t1 = 6500
```

```
M   64:     0  n  54 -  55        9000            G6#b
S   65:                            9500            G6#a  FSMstate =   0
                                                         t1 = 8000
M   74:     1  n  64 -  65       10500            G6#b
S   75:                           11000            G6#a  FSMstate =   0

===== <7> ===== IFV =    1 ===== ICV =    0 ==============================
   nbr  value typ   father        time            node  previous state
M           1  d
S                                                  G7#a

M    3:     0  s                     0            G7#b
S    4:                            500            G7#a  FSMstate =   0
M   13:     1  n   3 -   4        1500            G7#b
S   14:                           2000            G7#a  FSMstate =   1
                                                         t1 = 500
M   23:     0  n  13 -  14        3000            G7#b
S   24:                           3500            G7#a  FSMstate =   0
                                                         t1 = 2000
M   33:     1  n  23 -  24        4500            G7#b
S   34:                           5000            G7#a  FSMstate =   0
                                                         t1 = 3500
M   43:     0  n  33 -  34        6000            G7#b
S   44:                           6500            G7#a  FSMstate =   0
                                                         t1 = 5000
M   53:     1  n  43 -  44        7500            G7#b
S   54:                           8000            G7#a  FSMstate =   0
                                                         t1 = 6500
M   63:     0  n  53 -  54        9000            G7#b
S   64:                           9500            G7#a  FSMstate =   0
                                                         t1 = 8000
M   73:     1  n  63 -  64       10500            G7#b
S   74:                          11000            G7#a  FSMstate =   0

===== <8> ===== IFV =    1 ===== ICV =    0 ==============================
   nbr  value typ   father        time            node  previous state
M           1  d
S                                                  G8#a

M    2:     0  s                     0            G8#b
S    3:                            500            G8#a  FSMstate =   0
M   12:     1  n   2 -   3        1500            G8#b
S   13:                           2000            G8#a  FSMstate =   1
                                                         t1 = 500
M   22:     0  n  12 -  13        3000            G8#b
S   23:                           3500            G8#a  FSMstate =   0
                                                         t1 = 2000
M   32:     1  n  22 -  23        4500            G8#b
S   33:                           5000            G8#a  FSMstate =   0
                                                         t1 = 3500
M   42:     0  n  32 -  33        6000            G8#b
S   43:                           6500            G8#a  FSMstate =   0
                                                         t1 = 5000
M   52:     1  n  42 -  43        7500            G8#b
S   53:                           8000            G8#a  FSMstate =   0
```

```
                                                                    t1 = 6500
M   62:     0  n  52 -  53        9000            G8#b
S   63:                           9500            G8#a  FSMstate =   0
                                                        t1 = 8000
M   72:     1  n  62 -  63       10500            G8#b
S   73:                          11000            G8#a  FSMstate =   0

===== <9> ===== IFV =    1 ===== ICV =    0 ==============================
   nbr  value typ   father        time            node  previous state
M           1  d
S                                                G9#a

M    1:     0  s                     0            G9#b
S    2:                            500            G9#a  FSMstate =   0
M   11:     1  n   1 -   2        1500            G9#b
S   12:                           2000            G9#a  FSMstate =   1
                                                        t1 = 500
M   21:     0  n  11 -  12        3000            G9#b
S   22:                           3500            G9#a  FSMstate =   0
                                                        t1 = 2000
M   31:     1  n  21 -  22        4500            G9#b
S   32:                           5000            G9#a  FSMstate =   0
                                                        t1 = 3500
M   41:     0  n  31 -  32        6000            G9#b
S   42:                           6500            G9#a  FSMstate =   0
                                                        t1 = 5000
M   51:     1  n  41 -  42        7500            G9#b
S   52:                           8000            G9#a  FSMstate =   0
                                                        t1 = 6500
M   61:     0  n  51 -  52        9000            G9#b
S   62:                           9500            G9#a  FSMstate =   0
                                                        t1 = 8000
M   71:     1  n  61 -  62       10500            G9#b
S   72:                          11000            G9#a  FSMstate =   0

===== <10> ===== IFV =    1 ===== ICV =    0 ==============================
   nbr  value typ   father        time            node  previous state
M           1  d
S                                               G10#a

M    0:     0  s                     0           G10#b
S    1:                            500           G10#a  FSMstate =   0
M   10:     1  n   0 -   1        1500           G10#b
S   11:                           2000           G10#a  FSMstate =   1
                                                        t1 = 500
M   20:     0  n  10 -  11        3000           G10#b
S   21:                           3500           G10#a  FSMstate =   0
                                                        t1 = 2000
M   30:     1  n  20 -  21        4500           G10#b
S   31:                           5000           G10#a  FSMstate =   0
                                                        t1 = 3500
M   40:     0  n  30 -  31        6000           G10#b
S   41:                           6500           G10#a  FSMstate =   0
                                                        t1 = 5000
M   50:     1  n  40 -  41        7500           G10#b
```

S	51:					8000	G10#a	FSMstate =	0
								t1 = 6500	
M	60:	0	n	50 -	51	9000	G10#b		
S	61:					9500	G10#a	FSMstate =	0
								t1 = 8000	
M	70:	1	n	60 -	61	10500	G10#b		
S	71:					11000	G10#a	FSMstate =	0
								t1 = 9500	
M	80:	0	n	70 -	71	12000	G10#b		
S	81:					12500	G10#a	FSMstate =	0