July 1983

Report No. STAN-CS-83-970

# Reasoning about Digital Circuits

by

Benjamin C. Moszkowski

Department of Computer Science

Stanford University
Stanford, CA 94305

# REASONING ABOUT DIGITAL CIRCUITS

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

By

Benjamin Charles Moszkowski

June 1983

© Copyright 1983

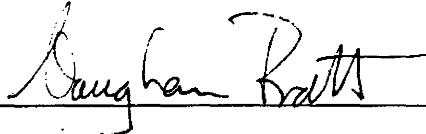
by

Benjamin Charles Moszkowski

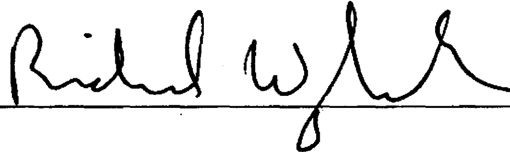
I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

(Principal Adviser)

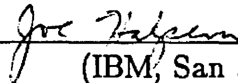
I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



(IBM, San Jose)

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies & Research

*To my family*

# Abstract

Predicate logic is a powerful and general descriptive formalism with a long history of development. However, since the logic's underlying semantics have no notion of time, statements such as "*I increases by* 2" and "*The bit signal X rises from* 0 *to* 1" can not be directly expressed. We present a formalism called *interval temporal logic* (ITL) that augments standard predicate logic with time-dependent operators. ITL is like discrete linear-time temporal logic but includes time intervals. The behavior of programs and hardware devices can often be decomposed into successively smaller intervals of activity. State transitions can be characterized by properties relating the initial and final values of variables over intervals. Furthermore, these time periods provide a convenient framework for introducing quantitative timing details.

After giving some motivation for reasoning about hardware, we present the propositional and first-order syntax and semantics of ITL. We demonstrate ITL's utility for uniformly describing the structure and dynamics of a wide variety of timing-dependent digital circuits. Devices discussed include delay elements, adders, latches, flip-flops, counters, random-access memories, a clocked multiplication circuit and the Am2901 bit slice. ITL also provides a means for expressing properties of such specifications. Throughout the dissertation, we examine such concepts as device equivalence and internal states. Propositional ITL is shown to be undecidable although useful subsets are of relatively reasonable computational complexity.

---

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

Acknowledgements

Professors John McCarthy and Zohar Manna gave much support and guidance as this research developed. Zohar's insistence on clear writing and notation has set a standard I will always aim for. Joseph Halpern provided valuable insights into the logic's theoretical complexity. His friendliness, curiosity and desire for mathematical simplicity combined to make him a wonderful collaborator. The thought-provoking lectures of Professor Vaughan Pratt provided an endless source of stimulation and challenge. Professors Michael Melkanoff and Mickey Krieger helped get me started when I was an undergraduate. My fellow students Russell Greiner, Yoni Malachi and Pierre Wolper are thanked for the numerous conversations we had on all matter of things temporal.

Many thanks also go to the following people for discussions and suggestions concerning the notation's readability (or lack thereof): Martin Abadi, Patrick Barkhordarian, Kevin Karplus, Amy Lansky, Fumihiro Maruyama, Gudrun Polak, Richard Schwartz, Alex Strong, Carolyn Talcott, Moshe Vardi, Richard Waldinger, Richard Weyhrauch and Frank Yellin. If it had not been for my friends at Siemens AG and the Polish Academy of Sciences, it is unlikely I would have undertaken this investigation. Late-night transatlantic discussions with Mike Gordon helped provide a sense of intrigue. Highest-quality chocolate and enthusiasm were always available from the Trischlers.

## Table of Contents

**Abstract** . . . . . . . . . . . . . . . . . . . . . . . . . . . v

**Acknowledgements** . . . . . . . . . . . . . . . . . . . . . . . vi

**Table of Contents** . . . . . . . . . . . . . . . . . . . . . . . vii

**Chapter 1 — Introduction** . . . . . . . . . . . . . . . . . . . . 1

1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.2 Contributions of Thesis . . . . . . . . . . . . . . . . . . . 5

1.3 Organization of Thesis . . . . . . . . . . . . . . . . . . . . 6

**Chapter 2 — Propositional Interval Temporal Logic** . . . . . . . . . 7

2.1 The Basic Formalism . . . . . . . . . . . . . . . . . . . . . 7

Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . 7

Models . . . . . . . . . . . . . . . . . . . . . . . . . . 8

Interpretation of formulas . . . . . . . . . . . . . . . . . 9

2.2 Expressing Temporal Concepts in Propositional ITL . . . . . . . . 10

Some properties of *next* and *semicolon* . . . . . . . . . . . . 11

Examining subintervals . . . . . . . . . . . . . . . . . . . 11

Initial and terminal subintervals . . . . . . . . . . . . . . . 12

The *yields* operator . . . . . . . . . . . . . . . . . . . . 13

Temporal length . . . . . . . . . . . . . . . . . . . . . . 15

Initial and final states . . . . . . . . . . . . . . . . . . . 16

The operators *halt* and *keep* . . . . . . . . . . . . . . . . 17

2.3 Propositional ITL with Quantification . . . . . . . . . . . . . 17

The *until* operator . . . . . . . . . . . . . . . . . . . . 18

Iteration . . . . . . . . . . . . . . . . . . . . . . . . . 19

2.4 Some Complexity Results . . . . . . . . . . . . . . . . . . . 20

Undecidability of propositional ITL . . . . . . . . . . . . . . 20

Decidability of a subset of ITL . . . . . . . . . . . . . . . . 23

Lower bound for satisfiability . . . . . . . . . . . . . . . . 24

**Chapter 3 — First-Order Interval Temporal Logic** . . . . . . . . . . 26

3.1 The Basic Formalism . . . . . . . . . . . . . . . . . . . . . . 26

Syntax of expressions . . . . . . . . . . . . . . . . . . . . 26

Syntax of formulas . . . . . . . . . . . . . . . . . . . . . 26

Models . . . . . . . . . . . . . . . . . . . . . . . . . . 27

Interpretation of expressions and formulas . . . . . . . . . . . 27

Arithmetic domain . . . . . . . . . . . . . . . . . . . . . 28

Temporal domain . . . . . . . . . . . . . . . . . . . . . 29

Naming conventions for variables . . . . . . . . . . . . . . . 29

3.2 Some First-Order Temporal Concepts . . . . . . . . . . . . . . 30

Temporal assignment . . . . . . . . . . . . . . . . . . . . 30

Temporal equality . . . . . . . . . . . . . . . . . . . . . 32

Temporal stability . . . . . . . . . . . . . . . . . . . . . 33

Iteration . . . . . . . . . . . . . . . . . . . . . . . . . 33

Measuring the length of an interval . . . . . . . . . . . . . . 34

Expressions based on *next* . . . . . . . . . . . . . . . . . 36

Initial and terminal stability . . . . . . . . . . . . . . . . 36

Blocking . . . . . . . . . . . . . . . . . . . . . . . . . 37

Rising and falling signals . . . . . . . . . . . . . . . . . . 38

Smoothness . . . . . . . . . . . . . . . . . . . . . . . . 40

**Chapter 4 — Delays and Combinational Elements** . . . . . . . . . . 41

4.1 Unit Delay . . . . . . . . . . . . . . . . . . . . . . . . . 41

4.2 Transport Delay . . . . . . . . . . . . . . . . . . . . . . 42

4.3 Functional Delay . . . . . . . . . . . . . . . . . . . . . . 43

4.4 Delay Based on Shift Register . . . . . . . . . . . . . . . . 44

4.5 Variable Transport Delay . . . . . . . . . . . . . . . . . . 45

4.6 Delay with Sampling . . . . . . . . . . . . . . . . . . . . 46

4.7 An Equivalent Delay Model with an Internal State . . . . . . . . 46

4.8 Delay with Separate Propagation Times for 0 and 1 . . . . . . . . 47

4.9 Smooth Delay Elements . . . . . . . . . . . . . . . . . . . 48

4.10 Delay with Tolerance to Noise . . . . . . . . . . . . . . . . 48

4.11 Gates with Input and Output Delays . . . . . . . . . . . . . . 49

4.12 High-Impedance . . . . . 49

Chapter 5 — Additional Notation . . . . . 51

5.1 Reverse Subscripting . . . . . 51

5.2 Conversion from Bit Vectors to Integers . . . . . 52

5.3 Tuples and Field Names . . . . . 52

5.4 Types for Lists and Tuples . . . . . 52

5.5 Temporal Conversion . . . . . 53

Chapter 6 — Adders . . . . . 55

6.1 Basic Adder . . . . . 56

Formal specification of addition circuit . . . . . 56

Combining two adders . . . . . 58

6.2 Adder with Internal Status Bit . . . . . 59

6.3 Adder with More Detailed Timing Information . . . . . 60

6.4 Adder with Carry Look-Ahead Outputs . . . . . 62

Chapter 7 — Latches . . . . . 65

7.1 Simple Latch . . . . . 65

7.2 Conventional SR-Latch . . . . . 66

Constructing an SR-latch . . . . . 68

7.3 Smooth SR-Latch . . . . . 69

7.4 D-Latch . . . . . 69

Building a D-latch . . . . . 71

Combining the interface with SR-latch . . . . . 73

Introducing a hold time . . . . . 74

Chapter 8 — Flip-Flops . . . . . 75

8.1 Simple D-Flip-Flop . . . . . 75

8.2 A Flip-Flop with More Timing Information . . . . . 77

Comparison of the predicates *SimpleDFlipFlop* and *DFlipFlop* . . 79

Simplifying the predicate *Store* in *DFlipFlop* . . . . . 79

8.3 Implementation of D-Flip-Flip . . . . . 80

Specification of the master latch . . . . . 80

Combining the latches . . . . . 82

8.4 D-Flip-Flops with Asynchronous Initialization Signals . . . . . 83

**Chapter 9 — More Digital Devices** . . . 86
9.1 Multiplexer . . . 86
Alternative specifications . . . 88
9.2 Memory . . . 88
9.3 Counters . . . 92
Clocked counter . . . 92
9.4 Shift Register . . . 94
Variant specifications . . . 98
Combining shift registers . . . 99

**Chapter 10 — Multiplication Circuit** . . . 101
10.1 Specification of Multiplier . . . 101
Variants of the specification . . . 104
10.2 Development of Multiplication Algorithm . . . 104
Deriving the predicate *Init* . . . 106
Deriving the predicate *Step* . . . 106
10.3 Description of Implementation . . . 107
Implementation theorem . . . 109

**Chapter 11 — The Am2901 Bit Slice** . . . 111
11.1 Behavior of Ramdom-Access Memory . . . 114
11.2 Behavior of Q-Register . . . 116
11.3 Behavior of Arithmetic Logic Unit . . . 117
11.4 Behavior of Bus Interface . . . 121
11.5 Composition of Two Bit Slices . . . 121
11.6 Timing Details . . . 122

**Chapter 12 — Discussion** . . . 123
12.1 Related Work . . . 123
12.2 Future Research Directions . . . 125
Proof theory . . . 125
Some variants of temporal logic . . . 125
Temporal types and higher-order temporal objects . . . 128
Temporal logic as a programming language . . . 128

Hardware . . . . . . . . . . . . . . . . . . . . . . . . 130
12.3 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . 130

**Bibliography** . . . . . . . . . . . . . . . . . . . . . . . . **131**

CHAPTER 1

# INTRODUCTION

## §1.1 Motivation

Computer systems continue to grow in complexity and the distinctions between hardware and software keep on blurring. Out of this has come an increasing awareness of the need for behavioral models suited for specifying and reasoning about both digital devices and programs. Contemporary hardware description languages (for example [5,35,46]) are not sufficient because of various conceptual limitations:

- Most such tools are intended much more for simulation than for mathematically sound reasoning about digital systems.
- Difficulties arise in developing circuit specifications that out of necessity must refer to different levels of behavioral abstraction.
- Existing formal tools for such languages are in general too restrictive to deal with the inherent parallelism of circuits.

Consider now some of the advantages of using predicate logic [12] as a tool for specification and reasoning:

- Every formula and expression in predicate logic has a simple semantic interpretation.
- Concepts such as recursion can be characterized and explored.

- Subsets of predicate logic can be used for programming (e.g., Prolog [24]).
- Theorems about formulas and expressions can themselves be stated and proved within the framework of predicate logic.
- Reasoning in predicate logic can often be reduced to propositional logic. Propositional logic also provides a means for reasoning about bits in digital circuits.
- Decades of research lie behind the overall predicate logic formalism.

One problem with predicate logic is that it has no built-in notion of time and therefore cannot directly express such dynamic tasks as

"*I increases by* 2"

"*The values of A and B are exchanged*"

or

"*The bit signal X rises from* 0 *to* 1."

Here are some ways to handle this limitation:

- We can simply try to ignore time. For example, the statement "*I increases by* 2" can be represented by the formula

  $$I = I + 2.$$

  Similarly, the statement "*The values of A and B are exchanged*" can be expressed as

  $$(A = B) \land (B = A).$$

  Unfortunately, this technique doesn't work since neither of these formulas has the intended meaning.

- Each variable can be represented as a function of time. Thus, we might express the statement "*I increases by* 2" as the formula

  $$I(t_f) = I(t_0) + 2,$$

where $t_0$ designates the initial time and $t_f$ is the final time. In an analogous manner, we can express the statement "*The values of A and B are exchanged*" as

$$[A(t_f) = B(t_0)] \wedge [B(t_f) = A(t_0)].$$

Because of the extra time variables such as $t_0$, this approach rapidly becomes tedious and lacks both clarity and modularity. For example, it is not straightforward to alter the above formulas to concisely express the statements "*I increases by* 2 *and then by* 3" and "*The values of A and B are exchanged n times in succession.*"

- Variables can be represented as lists or histories of values. Thus, the statement "*I increases by* 2" corresponds to the formula

$$last(I) = first(I) + 2$$

where $first(I)$ equals $I$'s first element and $last(I)$ equals $I$'s last element. This technique is very much like the previous one and suffers from similar problems.

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

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

Using the formalism, we can describe and reason about qualitative and quantitative properties of signal stability, delay and other fundamental aspects of circuit operation.

We present an extension of linear-time temporal logic [31,39] called *interval temporal logic* (ITL). The behavior of programs and hardware devices can often be

decomposed into successively smaller periods or intervals of activity. These intervals provide a convenient framework for introducing quantitative timing details. State transitions can be characterized by properties relating the initial and final values of variables over intervals of time. The principle feature of ITL is that every formula refers to some implicit interval of time. The dissertation will later examine the logic's formal syntax and semantics in great depth. Below are a few English-language statements and corresponding formulas in ITL. These examples are meant to give an feel for what ITL looks like.

- *I increases by* 2:

$$I + 2 \rightarrow I$$

- *The values of A and B are exchanged:*

$$(A \rightarrow B) \;\wedge\; (B \rightarrow A)$$

- *I increases by* 2 *and then by* 3:

$$(I + 2 \rightarrow I); (I + 3 \rightarrow I)$$

- *The values of A and B are exchanged n times in succession:*

$$([A \rightarrow B] \;\wedge\; [B \rightarrow A])^n$$

- *The bit signal X rises from* 0 *to* 1:

$$(X \approx 0); skip; (X \approx 1)$$

As in conventional logic, we can express properties without the need for a separate "assertion language." For example, the formula

$$[(I + 1 \rightarrow I); (I + 1 \rightarrow I)] \supset (I + 2 \rightarrow I)$$

states that if the variable $I$ twice increases by 1 in an interval, then the overall result is that $I$ increases by 2.

ITL's applicability is not limited to the goals of computer-assisted verification and synthesis of circuits. This type of notation, with appropriate "syntactic sugar," can provide a fundamental and rigorous basis for communicating, reasoning or teaching about the behavior of digital devices, computer programs and other discrete systems. We apply it to describing and comparing devices ranging from delay elements up to a clocked multiplication circuit and the Am2901 ALU bit slice developed by Advanced Micro Devices, Inc. Interval temporal logic also provides a basic framework for exploring the computational complexity of reasoning about time. Simulation-based languages can perhaps use such a formalism as a vehicle for describing the intended semantics of delays and other features. In fact, we feel that ITL provides a sufficient basis for directly describing a wide range of devices and programs. For our purposes, the distinctions made in dynamic logic [19,37] and process logics [11,20,38] between programs and propositions seem unnecessary. Manna and Moszkowski [29,30] show how ITL can itself serve as the basis for a programming language.

## §1.2 Contributions of Thesis

Here is a summary of the key ideas developed in this thesis:

- The propositional and first-order syntax and semantics of interval temporal logic are presented.
- We give complexity results regarding satisfiability of formulas in propositional ITL.
- We demonstrate the utility of ITL for uniformly describing and reasoning about the structure and dynamics of a wide variety of timing-dependent digital circuits. Devices discussed include delay elements, adders, latches, flip-flops, counters, random-access memories, a clocked multiplication circuit and the Am2901 bit slice.
- The overall approach used indicates that multi-valued logics and partial values are such as $\perp$ are not necessary in the treatment of timing-dependent hardware.

## §1.3 Organization of Thesis

Chapter 2 introduces the propositional form of interval temporal logic. The logic's basic syntax and semantics are given. In addition, ITL serves to express a number of general temporal concepts and properties. The chapter concludes with some results on the theoretical complexity of propositional ITL.

In chapter 3, we present first-order ITL. A variety of useful predicates are introduced to capture dynamic notions such as assignment and signal transitions.

The next few chapters show how to formalize specifications and properties of a number of digital devices. Chapter 4 describes and compares a number of delay models that arise in digital systems. In chapter 5 we introduce some extra notation for dealing with subscripting, conversion and tuples. Chapter 6 looks at adders, chapter 7 discusses latches and chapter 8 examines flipflops. Chapter 9 contains descriptions and properties of multiplexers, random-access memories, counters and shift registers.

Chapter 10 discusses a clocked multiplication circuit and shows one way to derive a suitable multiplication algorithm in ITL. In chapter 11, we use ITL to describe and reason about the functional behavior of the Am2901 bit slice, a large-scale integrated circuit. The dissertation concludes with chapter 12 containing a discussion of some related work and future research directions.

CHAPTER 2

# PROPOSITIONAL INTERVAL TEMPORAL LOGIC

We first present propositional ITL; this later provides a basis for first-order ITL.

## §2.1 The Basic Formalism

### Syntax

Propositional ITL basically consists of propositional logic with the addition of modal constructs to reason about intervals of time.

Formulas are built inductively out of the following:

- A nonempty set of propositional variables:

  $P, Q, \ldots$

- Logical connectives:

  $\neg w$ and $w_1 \wedge w_2$, where $w$, $w_1$ and $w_2$ are formulas.

- Next:

  $\bigcirc\, w$ (read "*next* $w$"), where $w$ is a formula.

- Semicolon:

  $w_1; w_2$ (read "$w_1$ *semicolon* $w_2$" or "$w_1$ *followed by* $w_2$"),

  where $w_1$ and $w_2$ are formulas.

*Examples:*

Here are some sample formulas:

$$P$$
$$P \land Q$$
$$\bigcirc(P \land \neg R)$$
$$Q;(P \land R)$$
$$\neg Q \land \bigcirc[P; \neg \bigcirc(Q;R)]$$

Notice that all constructs, including $\bigcirc$ and *semicolon*, can be arbitrarily nested.

## Models

Our logic can be viewed as linear-time temporal logic with the addition of the "chop" operator of process logic [11,20]. The truth of variables depends not on states but on intervals. A model is a pair $(\Sigma, \mathcal{M})$ consisting of a set of states $\Sigma = \{s, t, \dots\}$ together with an interpretation $\mathcal{M}$ mapping each propositional variable $P$ and nonempty interval $s_0 \dots s_n \in \Sigma^+$ to a some truth value $\mathcal{M}_{s_0 \dots s_n}[\![P]\!]$. In what follows, we assume $\Sigma$ is fixed.

The length of an interval $s_0 \dots s_n$ is $n$. An interval consisting of a single state has length 0. It is possible to permit infinite intervals although for simplicity we will omit them here. An interval can also be thought of as the sequence of states of a computation. In the language of Chandra et al. [11], our logic is "non-local" with intervals corresponding to "paths."

Here is a sample model:

- States:

$$\Sigma = \{s, t, u\}$$

- Assignments:

| Variables | Where $\mathcal{M}$ is true |
|---|---|
| $P$ | $s, t, tus, tt, ts, su$ |
| $Q$ | $t, ts, tst, tsts$ |
| $R$ | — |
| $\vdots$ | $\vdots$ |

### Interpretation of formulas

We now extend the meaning function $\mathcal{M}$ to arbitrary formulas:

- $\mathcal{M}_{s_0\ldots s_n}[\![\neg w]\!] = true$ *iff* $\mathcal{M}_{s_0\ldots s_n}[\![w]\!] = false$
  The formula $\neg w$ is true in an interval $s_0 \ldots s_n$ iff $w$ is false.

- $\mathcal{M}_{s_0\ldots s_n}[\![w_1 \wedge w_2]\!] = true$ *iff* $\mathcal{M}_{s_0\ldots s_n}[\![w_1]\!] = true$ and $\mathcal{M}_{s_0\ldots s_n}[\![w_2]\!] = true$
  The conjunction $w_1 \wedge w_2$ is true in $s_0 \ldots s_n$ iff $w_1$ and $w_2$ are both true.

- $\mathcal{M}_{s_0\ldots s_n}[\![\bigcirc\, w]\!] = true$ *iff* $n \geq 1$ and $\mathcal{M}_{s_1\ldots s_n}[\![w]\!] = true$
  The formula $\bigcirc\, w$ is true in an interval $s_0 \ldots s_n$ iff $w$ is true in the subinterval $s_1 \ldots s_n$. If the original interval has length 0, then $\bigcirc\, w$ is false.

- $\mathcal{M}_{s_0\ldots s_n}[\![w_1; w_2]\!] = true$ *iff* $\mathcal{M}_{s_0\ldots s_i}[\![w_1]\!] = true$ and $\mathcal{M}_{s_i\ldots s_n}[\![w_2]\!] = true$, for some $i$, $0 \leq i \leq n$.
  Given an interval $s_0 \ldots s_n$, the formula $w_1; w_2$ is true if there is at least one way to divide the interval into two adjacent subintervals $s_0 \ldots s_i$ and $s_i \ldots s_n$ such that the formula $w_1$ is true in the first one, $s_0 \ldots s_i$, and the formula $w_2$ is true in the second, $s_i \ldots s_n$.

*Examples:*

We now given the interpretations of some formulas with respect to the particular model discussed earlier:

- $\mathcal{M}_{ts}[\![P \wedge Q]\!] = true$ since $\mathcal{M}_{ts}[\![P]\!] = true$ and $\mathcal{M}_{ts}[\![Q]\!] = true$.

- $\mathcal{M}_{tsu}[\![Q; P]\!] = true$ since $\mathcal{M}_{ts}[\![Q]\!] = true$ and $\mathcal{M}_{su}[\![P]\!] = true$.

- $\mathcal{M}_{t}[\![\neg(P \wedge Q)]\!] = false$ since $\mathcal{M}_{t}[\![P \wedge Q]\!] = true$.

- $\mathcal{M}_{ts}[\![\bigcirc(P \wedge \neg R)]\!] = true$ since $\mathcal{M}_{s}[\![P \wedge \neg R]\!] = true$.

A formula $w$ is *satisfied* by a pair $(\mathcal{M}, s_0 \ldots s_n)$ iff

$$\mathcal{M}_{s_0\ldots s_n}[\![w]\!] = true$$

This is denoted as follows:

$$(\mathcal{M}, s_0 \ldots s_n) \models w.$$

We sometimes make $\mathcal{M}$ implicit and write

$$s_0 \ldots s_n \models w.$$

If all pairs of $\mathcal{M}$ and $s_0 \ldots s_n$ satisfy $w$ then $w$ is *valid*, written $\models w$.

## §2.2 Expressing Temporal Concepts in Propositional ITL

We illustrate propositional ITL's descriptive power by giving a variety of useful temporal concepts. The connectives $\neg$ and $\wedge$ clearly suffice to express other basic logical operators such as $\vee$ and $\equiv$:

- $w_1 \vee w_2$ – logical-or:

$$w_1 \vee w_2 \quad \equiv_{\text{def}} \quad \neg(\neg w_1 \wedge \neg w_2)$$

- $w_1 \supset w_2$ – implication:

$$w_1 \supset w_2 \quad \equiv_{\text{def}} \quad \neg w_1 \vee w_2$$

- $w_1 \equiv w_2$ – equivalence:

$$w_1 \equiv w_2 \quad \equiv_{\text{def}} \quad (w_1 \supset w_2) \wedge (w_2 \supset w_1)$$

- *if* $w_1$ *then* $w_2$ *else* $w_3$ – conditional formula:

$$\textit{if } w_1 \textit{ then } w_2 \textit{ else } w_3 \quad \equiv_{\text{def}} \quad (w_1 \supset w_2) \wedge (\neg w_1 \supset w_3)$$

- *true* – truth:

$$\textit{true} \quad \equiv_{\text{def}} \quad P \vee \neg P$$

- *false* – falsity:

$$\textit{false} \quad \equiv_{\text{def}} \quad \neg \textit{true}$$

## Some properties of *next* and *semicolon*

Throughout this thesis, numerous sample formulas are given in order to convey the utility of ITL for expressing temporal and digital concepts. The reader need not look at every single formula. Here are some representative properties of the operators *next* and *semicolon*. All follow from the semantic model just covered.

$$\models \quad (P;Q);R \;\equiv\; P;(Q;R)$$

*Semicolon* is associative. Therefore a formula such as $P;Q;R$ is unambiguous.

$$\models \quad [(P \lor Q);R] \;\equiv\; [(P;R) \lor (Q;R)]$$

The left of *semicolon* distributes with logical-or. An analogous property applies to the right of *semicolon*.

$$\models \quad [P;(Q \land R)] \;\supset\; [(P;Q) \land (P;R)]$$

A logical-and can be removed from *semicolon*'s right. The left of *semicolon* has a similar property.

$$\models \quad (\bigcirc P);Q \;\equiv\; \bigcirc(P;Q)$$

The operator $\bigcirc$ commutes with the left of *semicolon*.

We now introduce a variety of other useful temporal concepts that are expressible by means of the constructs just defined.

## Examining subintervals

For a formula $w$ and an interval $s_0 \ldots s_n$, the construct $\diamond\!\!\!\cdot\; w$ is true if $w$ is true in at least one subinterval $s_i \ldots s_j$ contained within $s_0 \ldots s_n$ and possibly the entire interval $s_0 \ldots s_n$ itself. Note that the "a" in $\diamond\!\!\!\cdot$ simply stands for "any" and is not a variable.

$$\mathcal{M}_{s_0 \ldots s_n}[\![\diamond\!\!\!\cdot\; w]\!] = \mathit{true} \quad \mathit{iff} \quad \mathcal{M}_{s_i \ldots s_j}[\![w]\!] = \mathit{true}, \text{ for some } 0 \le i \le j \le n$$

Similarly, the formula $\boxdot\, w$ is true if the formula $w$ itself is true in all subintervals of $s_0 \ldots s_n$:

$$\mathcal{M}_{s_0 \ldots s_n}[\![\boxdot\, w]\!] = \mathit{true} \quad \mathit{iff} \quad \mathcal{M}_{s_i \ldots s_j}[\![w]\!] = \mathit{true}, \text{ for all } 0 \le i \le j \le n$$

These constructs can be expressed as follows:

$$⟐\, w \quad \equiv_{\text{def}} \quad (\mathit{true}; w; \mathit{true})$$

$$⊡\, w \quad \equiv_{\text{def}} \quad \neg ⟐ \neg w$$

Because *semicolon* is associative, the definition of $⟐$ is unambiguous. Together, $⟐$ and $⊡$ fulfill all the axioms of the modal system *S4* [23], with $⟐$ interpreted as *possibly* and $⊡$ as *necessarily*.

*Properties:*

$\models \quad ⊡ P \supset P$

If the proposition $P$ is true in all subintervals then it is true in the primary interval.

$\models \quad ⊡(P \wedge Q) \equiv [⊡ P \wedge ⊡ Q]$

The logical-and of two propositions $P$ and $Q$ is true in every subinterval if and only if both propositions are true everywhere.

$\models \quad ⟐ P \equiv ⟐ ⟐ P$

A proposition $P$ is somewhere true exactly if there is some subinterval in which $P$ is somewhere true.

$\models \quad [⊡ P \wedge ⟐ Q] \supset ⟐(P \wedge Q)$

If $P$ is true in all subintervals and $Q$ is true in some subinterval then both are simultaneously true in at least one subinterval.

## Initial and terminal subintervals

For a given interval $s_0 \ldots s_n$ the operators $◇⃒$ and $⎅$ are similar to $⟐$ and $⊡$ but only look at *initial* subintervals of the form $s_0 \ldots s_i$ for $i \le n$. We can express $◇⃒\, w$ and $⎅\, w$ as shown below:

$$◇⃒\, w \quad \equiv_{\text{def}} \quad (w; \mathit{true})$$

$$⎅\, w \quad \equiv_{\text{def}} \quad \neg ◇⃒ \neg w$$

For example, the formula $\Box_i(P \wedge Q)$ is true on an interval if $P$ and $Q$ are both true in all initial subintervals. The connectives $\Diamond_t$ and $\Box_t$ refer to *terminal* subintervals of the form $s_i \ldots s_n$ and are expressed as follows:

$$\Diamond_t\, w \quad \equiv_{\text{def}} \quad (\mathit{true}; w)$$

$$\Box_t\, w \quad \equiv_{\text{def}} \quad \neg \Diamond_t \neg w$$

Both pairs of operators satisfy the axioms of *S4*. The operators $\Diamond_t$ and $\Box_t$ correspond directly to $\Diamond$ and $\Box$ in linear-time temporal logic [31].

*Properties:*

$$\models \quad (\Box_a P \equiv \Box_i \Box_t P) \wedge (\Box_a P \equiv \Box_t \Box_i P)$$

The proposition $P$ is true in all subintervals exactly if $P$ is true in all initial subintervals of all terminal subintervals. In fact, the operators $\Box_i$ and $\Box_t$ commute.

$$\models \quad [\Box_i(P \supset Q) \wedge (P;R)] \supset (Q;R)$$

If $P$ implies $Q$ in all initial subintervals and $P$ is followed by $R$, then $Q$ is followed by $R$.

$$\models \quad (\Diamond_t P);Q \equiv \Diamond_t(P;Q)$$

The operator $\Diamond_t$ commutes with the left of *semicolon*.

### The *yields* operator

It is often desirable to say that within an interval $s_0 \ldots s_n$ whenever some formula $w_1$ is true in any initial subinterval $s_0 \ldots s_i$, then another formula $w_2$ is true in the corresponding terminal interval $s_i \ldots s_n$ for any $i$, $0 \le i \le n$. We say that $w_1$ *yields* $w_2$ and denote this by the formula $w_1 \rightsquigarrow w_2$:

$$\mathcal{M}_{s_0 \ldots s_n}[\![w_1 \rightsquigarrow w_2]\!] = \mathit{true}$$
$$\textit{iff} \quad \mathcal{M}_{s_0 \ldots s_i}[\![w_1]\!] = \mathit{true} \text{ implies } \mathcal{M}_{s_i \ldots s_n}[\![w_2]\!] = \mathit{true}, \quad \text{for all } 0 \le i \le n$$

The yields operator can be viewed as ensuring that no counterexample of the form $w_1; \neg w_2$ exists in the interval:

$$(w_1 \rightsquigarrow w_2) \quad \equiv_{\text{def}} \quad \neg(w_1; \neg w_2)$$

This is similar to interpreting the implication $w_1 \supset w_2$ as the formula $\neg(w_1 \wedge \neg w_2)$.

*Examples:*

| *Concept* | *Formula* |
|---|---|
| After $P$, both $Q$ and $R$ are true | $P \rightsquigarrow (Q \wedge R)$ |
| After $P$, $Q$ yields $R$ | $P \rightsquigarrow (Q \rightsquigarrow R)$ |
| $P$ always yields $Q$ | $\boxed{\scriptstyle t}(P \rightsquigarrow Q)$ |
| After $P$ and $Q$, $R$ is false | $(P \wedge Q) \rightsquigarrow (\neg R)$ |

*Properties:*

$$\models \quad ([P;Q] \rightsquigarrow R) \equiv (P \rightsquigarrow [Q \rightsquigarrow R])$$

The formula $P;Q$ yields $R$ exactly if after $P$ is true, $Q$ yields $R$. This is analogous to the propositional tautology

$$\models \quad [(P \wedge Q) \supset R] \equiv [P \supset (Q \supset R)]$$

$$\models \quad \mathit{false} \rightsquigarrow P$$

After *false*, anything can happen. Since *false* never occurs, this is a vacuous assertion.

When combined with other temporal operators, *yield* exhibits a number of interesting properties based on the underlying behavior of *semicolon*. Here are some examples:

$$\models \quad \boxed{\scriptstyle t}\, P \equiv (\mathit{true} \rightsquigarrow P)$$

The proposition $P$ is true in all terminal subintervals exactly if $P$ is true after any initial subinterval satisfying *true*.

$$\models \quad (P \rightsquigarrow \boxed{\scriptstyle t}\, Q) \equiv (\Diamond_{i}\, P \rightsquigarrow Q)$$

After $P$, $Q$ is true in all terminal subintervals iff the result of $P$ being true in some initial subinterval yields $Q$.

$$\models \quad (P \rightsquigarrow \boxed{\scriptstyle i}\, Q) \equiv \boxed{\scriptstyle i}(P \rightsquigarrow Q)$$

After any initial subinterval where $P$ is true, the formula $\boxed{\scriptstyle i}\, Q$ results iff in all initial subintervals, $P$ yields $Q$.

**Temporal length**

The construct *empty* checks whether an interval has length 0:

$$\mathcal{M}_{s_0 \ldots s_n}[\![empty]\!] \equiv true \quad \textit{iff} \quad n = 0$$

Similarly, the construct *skip* checks whether the interval's length is exactly 1:

$$\mathcal{M}_{s_0 \ldots s_n}[\![skip]\!] \equiv true \quad \textit{iff} \quad n = 1$$

These operators are expressible as shown below:

$$empty \quad \equiv_{\text{def}} \quad \neg \bigcirc true$$

$$skip \quad \equiv_{\text{def}} \quad \bigcirc empty$$

Combinations of the operators *skip* and *semicolon* can be used to test for intervals of some fixed length. For example, the formula

$$skip; skip; skip$$

is true exactly for intervals of length 3. Alternatively, the connective *next* suffices:

$$\bigcirc \bigcirc \bigcirc empty$$

*Examples:*

| *Concept* | *Formula* |
|---|---|
| After two units of time, $P$ holds | $skip; skip; P$ |
| $P$ is true in some unit subinterval | $\diamondsuit(skip \wedge P)$ |

*Properties:*

$\models \quad \diamondsuit\, empty$

Eventually time runs out because intervals are finite.

$$\models \quad (\mathit{skip}; P) \equiv \bigcirc P$$

The operators *skip* and *semicolon* can be used instead of *next*.

$$\models \quad (\mathit{empty}; P) \equiv P$$

The construct *empty* disappears on the left of *semicolon*. An analogous theorem applies to the right of *semicolon* as well.

### Initial and final states

The construct *beg* $w$ tests if the formula $w$ is true in an interval's starting state:

$$\mathcal{M}_{s_0 \ldots s_n}[\![\mathit{beg}\, w]\!] \equiv \mathcal{M}_{s_0}[\![w]\!]$$

The connective *beg* can be expressed as follows:

$$\mathit{beg}\, w \quad \equiv_{\text{def}} \quad \Diamond(\mathit{empty} \wedge w)$$

This checks that $w$ holds for an initial subinterval of length 0, *i.e.*, the interval's first state. By analogy, the final state can be examined by the operator *fin* $w$:

$$\mathit{fin}\, w \quad \equiv_{\text{def}} \quad \Diamond(\mathit{empty} \wedge w)$$

This checks that $w$ holds for a terminal subinterval of length 0, *i.e.*, the interval's final state. The construct *beg* corresponds directly to the construct **f** in the process logic of Harel et al. [20]. Similarly, *fin* corresponds to the process logic's construct **last**.

*Examples:*

| *Concept* | *Formula* |
|---|---|
| If $P$ is initially true, it ends true | $\mathit{beg}\, P \supset \mathit{fin}\, P$ |
| $P$ and $Q$ end true | $\mathit{fin}(P \wedge Q)$ |

*Properties:*

$$\models \quad beg\,P \;\equiv\; \neg\, beg(\neg P)$$

$P$ is true in the first state iff $\neg P$ is not.

$$\models \quad fin(P \lor Q) \;\equiv\; [fin\,P \;\lor\; fin\,Q]$$

The logical-or of $P$ and $Q$ ends up true exactly if either $P$ ends true or $Q$ ends true.

### The operators *halt* and *keep*

Various other useful operators can be expressed in propositional ITL. For example, the construct $halt\,w$ is true for intervals that terminate the first time the formula $w$ is true:

$$halt\,w \quad \equiv_{\text{def}} \quad \boxdot(w \;\equiv\; empty)$$

Thus $halt\,w$ can be thought of as forcing an interval to wait until $w$ occurs.

The construct $keep\,w$ is true if the formula $w$ is true in all nonempty terminal intervals:

$$keep\,w \quad \equiv_{\text{def}} \quad \boxdot([\neg empty] \;\supset\; w)$$

## §2.3 Propositional ITL with Quantification

It is very useful to extend propositional ITL to permit existential and universal quantification over variables. In order for quantification to properly work, we require that $\Sigma$, the model's set of states, be varied enough so that any possible combined behavior of variables is represented by some interval. More precisely, let $P$ be a propositional variable, $s_0 \ldots s_n$ be an interval and $\alpha(i,j)$ be a function mapping ordered pairs $0 \le i \le j \le n$ to truth values. We require some interval $s'_0 \ldots s'_n$ exist such that $s'_i \ldots s'_j$ agrees with the corresponding subinterval $s_i \ldots s_j$ on assignments to all variables with the exception that each subinterval $s'_i \ldots s'_j$ gives $P$ the value $\alpha(i,j)$:

$$\begin{aligned} \mathcal{M}_{s'_i \ldots s'_j}[\![Q]\!] &= \mathcal{M}_{s_i \ldots s_j}[\![Q]\!], \quad \text{for } Q \ne P \text{ and } 0 \le i \le j \le n \\ \mathcal{M}_{s'_i \ldots s'_j}[\![P]\!] &= \alpha(i,j), \quad \text{for } 0 \le i \le j \le n \end{aligned}$$

We denote the interval $s'_0 \ldots s'_n$ as

$$(s_0 \ldots s_n)[P/\alpha]$$

The construct

$$\exists P.\, w$$

represents existential quantification and has the semantics

$$\mathcal{M}_{s_0 \ldots s_n}[\![\exists P.\, w]\!] = \mathit{true} \quad \mathit{iff} \quad \text{for some } \alpha, \;\; \mathcal{M}_{s'_0 \ldots s'_n}[\![w]\!] = \mathit{true},$$

where $s'_0 \ldots s'_n = (s_0 \ldots s_n)[P/\alpha]$.

Universal quantification is expressed as the dual of existential quantification:

$$\forall P.\, w \quad \equiv_{\text{def}} \quad \neg \exists P.\, \neg w$$

*Property:*

$$\models \;\; (\neg \mathit{empty}) \supset \exists P.\, [\mathit{beg}\, P \;\wedge\; \mathit{fin}(\neg P)]$$

In a nonempty interval, a variable can be constructed that starts true and ends false.

### The *until* operator

Linear-time temporal logic has the *until* operator $w_1 \,\mathcal{U}\, w_2$ which is true in an interval if the formula $w_2$ is eventually true and $w_1$ is true until then:

$$\mathcal{M}_{s_0 \ldots s_n}[\![w_1 \,\mathcal{U}\, w_2]\!] \;\equiv\; \mathit{true} \quad \mathit{iff}$$

$$\mathcal{M}_{s_i \ldots s_n}[\![w_2]\!] \text{ for some } 0 \le i \le n \text{ and } \mathcal{M}_{s_j \ldots s_n}[\![w_1]\!] \text{ for all } 0 \le j < i$$

We can express *until* as follows:

$$w_1 \,\mathcal{U}\, w_2 \quad \equiv_{\text{def}} \quad \exists\, P.\, \big[\mathit{beg}\, P \;\wedge\; \boxdot\big(\mathit{beg}\, P \supset [w_2 \;\vee\; (w_1 \;\wedge\; \bigcirc\, \mathit{beg}\, P)]\big)\big]$$

where $P$ does not occur free in $w_1$ or $w_2$. In essence, $P$ is initially true and inductively remains so until $w_2$ is true.

**Iteration**

An interval can be broken up into an arbitrary number of successive subintervals, each satisfying some formula $w$. We can use, for example, the construct $w^3$ as an abbreviation for

$$w; w; w$$

In general, we abbreviate repetition by induction:

$$w^0 \quad \equiv_{\text{def}} \quad empty$$

$$w^{i+1} \quad \equiv_{\text{def}} \quad w; w^i$$

Thus, for the case of $i = 0$, an interval $s_0 \ldots s_n$ satisfies the operator exactly if the interval's length is 0. We can extend propositional ITL to include the Kleene closure of *semicolon*:

$$\mathcal{M}_{s_0 \ldots s_n}[\![w^*]\!] = true \quad \textit{iff} \quad \mathcal{M}_{s_0 \ldots s_n}[\![w^i]\!] = true, \quad \text{for some } i \geq 0.$$

Iteration can be expressed by quantifying over a variable $P$ that is true at the end-points of the steps:

$$w^* \quad \equiv_{\text{def}} \quad \exists P.\big(beg\,P \;\wedge\; \boxdot\big[beg\,P \supset \big(empty \;\vee\; \diamondsuit[w \;\wedge\; \bigcirc\, halt(beg\,P)]\big)\big]\big)$$

where $P$ does not occur free in $w$. Other constructs such as while-loops can also be expressed within ITL:

$$while\,P\;do\,Q \quad \equiv_{\text{def}} \quad \big[(beg[P] \;\wedge\; Q)^* \;\wedge\; fin(\neg P)\big]$$

*Properties:*

$\models \quad P^* \equiv (P \;\wedge\; \neg empty)^*$

During iteration, each step can be assumed to have length $\geq 1$.

$\models \quad false^* \equiv empty$

An interval in which *false* is iterated must be empty.

## §2.4 Some Complexity Results

We prove that satisfiability for arbitrary formulas in propositional ITL is undecidable but demonstrate the decidability of a useful subset.

### Undecidability of propositional ITL

*Theorem* (Halpern and Moszkowski): Satisfiability for propositional ITL is undecidable.

*Proof*: Our proof is very similar to the one presented by Chandra et al. [11]. for showing the undecidability of satisfiability for a propositional process logic. We strengthen their result since we do not require programs in order to obtain undecidability.

Given two context-free grammars $G_1$ and $G_2$, we can construct an propositional ITL formula that is satisfiable iff the intersection of the languages generated by $G_1$ and $G_2$ is nonempty. Since this intersection problem is undecidable [22], it follows that satisfiability for propositional ITL is also.

Without lose of generality, we assume that $G_1$ and $G_2$ contain no $\epsilon$-productions, use 0 and 1 as the only terminal symbols and are in Greibach normal form (that is, the right-hand side of each production starts with a terminal symbol).

For a given an interval $s_0 \ldots s_n$ and an interpretation $\mathcal{M}$, we form the *trace* $\sigma_{s_0 \ldots s_n}(P)$ of a variable $P$ by observing $P$'s behavior over the states $s_0, \ldots, s_n$. We define $\sigma$ as follows:

$$\sigma_s(P) = \begin{cases} 0 & \text{if } \mathcal{M}_s[\![P]\!] = \mathit{false} \\ 1 & \text{if } \mathcal{M}_s[\![P]\!] = \mathit{true} \end{cases}$$

$$\sigma_{s_0 \ldots s_n}(P) = \sigma_{s_0}(P) \ldots \sigma_{s_n}(P)$$

Suppose that $G$ is a context-free grammar consisting of a list $\pi$ of $m$ production sets $\pi_1, \ldots, \pi_m$, one for each nonterminal symbol $A_i$:

$$\pi_1 : \; A_1 \to \pi_{11} \mid \pi_{12} \mid \cdots \mid \pi_{1,|\pi_1|}$$

$$\pi_2 : \; A_2 \to \pi_{21} \mid \pi_{22} \mid \cdots \mid \pi_{2,|\pi_2|}$$

$$\vdots$$

$$\pi_m : \; A_m \to \pi_{m1} \mid \pi_{m2} \mid \cdots \mid \pi_{m,|\pi_m|}$$

Let $L(G, A_i)$ be the language generated by $G$ with $A_i$ as the start symbol. We give a translation $f(G, A_i)$ into ITL such that an interval $s_0 \dots s_n$ satisfies $f(G, A_i)$ iff $P$'s trace in $s_0 \dots s_n$ is in $L(G, A_i)$:

$$s_0 \dots s_n \models f(G, A_i) \quad \textit{iff} \quad \sigma_{s_0 \dots s_n}(P) \in L(G, A_i). \qquad (*)$$

For each of the production sets $\pi_i$, the associated translation $f(\pi_i)$ is the ITL formula

$$\boxdot(A_i \equiv [f(\pi_{i1}) \lor f(\pi_{i2}) \lor \cdots \lor f(\pi_{i,|\pi_i|})])$$

Each production string $\pi_{ij} = V_1 V_2 \dots V_{|\pi_{ij}|}$ has the translation

$$f(V_1 V_2 \dots V_m) = f(V_1); \textit{skip}; f(V_2); \textit{skip}; \dots \textit{skip}; f(V_{|\pi_{ij}|})$$

where

$$f(0) = (\neg P \land \textit{empty})$$
$$f(1) = (P \land \textit{empty})$$
$$f(A_i) = A_i, \quad \text{for each nonterminal symbol } A_i$$

Recall that the variable $P$ determines whether a state maps to 0 or 1. In order to avoid conflicts, we require that $P$ not occur in the grammar. The overall translation $f(G, A_i)$ is

$$A_i \land f(\pi)$$

It is now easy to show (*) by induction on the size of the interval $s_0 \dots s_n$. We need the grammar to be in Greibach normal form in order for the inductive step to go through. See Chandra et al. [11] for details.

Given two context-free grammars $G_1$ and $G_2$ with disjoint sets of nonterminals and respective start symbols $S_1$ and $S_2$, the ITL formula

$$f(G_1, S_1) \land f(G_2, S_2)$$

is satisfiable iff the intersection of the languages $L(G_1)$ and $L(G_2)$ is nonempty. Because this emptiness problem is undecidable [22], it follows that satisfiability in propositional ITL is also. ∎

*Corollary*: **Validity for propositional ITL is undecidable.**

*Remark*: Undecidability can be shown to hold even if we are restricted to just using *empty* instead of *skip*. To do this, we use propositional variables $P$ and $Q$. We introduce an operator $group(P, Q)$ which is true in intervals satisfying the formula

$$(\Box^{i}\, beg\, Q);\ skip;\ (\Box^{a}\, beg[P \wedge \neg Q]);\ skip;\ (\Box^{i}\, beg\, Q)$$

Such intervals are in effect delimited on both sides by states with $Q$ true and contain internal states with $P \wedge \neg Q$ true. Hence, $Q$ acts as a delimiter around a group of states where $P$ is true. The following is a sample 5-state interval $s_0 \ldots s_4$ satisfying $group(P, Q)$:

$$\begin{array}{ccccc} s_0 & s_1 & s_2 & s_3 & s_4 \\ Q & P & P & Q & Q \\ & \wedge & \wedge & & \\ & \neg Q & \neg Q & & \end{array}$$

Similarly, $group(\neg P, Q)$ denotes a delimited group of states with $\neg P$ true in the interior. If we take *empty* as a primitive operator, the operator *group* can be expressed without the use of *next*:

$$group(P, Q) \quad \equiv_{\text{def}} \quad [grp(P, Q) \ \wedge\ \neg(grp(P, Q); grp(P, Q))]$$

where $grp(P, Q)$ has the definition

$$grp(P, Q) \quad \equiv_{\text{def}} \quad [beg\, Q \ \wedge\ fin\, Q \ \wedge\ \Box^{a}(beg(P \wedge \neg Q) \ \vee\ Q) \ \wedge\ \Diamond^{a}\, beg\, P]$$

Recall that *beg* and *fin* are defined using *empty* and *semicolon*:

$$beg\, w \quad \equiv_{\text{def}} \quad \Diamond^{i}(empty \ \wedge\ w)$$

$$fin\, w \quad \equiv_{\text{def}} \quad \Diamond^{t}(empty \ \wedge\ w)$$

The modified translation $f'$ is like $f$ with the following exceptions:

$$f'(V_1 V_2 \ldots V_m) = f'(V_1); f'(V_2); \ldots; f'(V_m)$$
$$f'(0) = group(\neg P, Q)$$
$$f'(1) = group(P, Q)$$

∎

**Decidability of a subset of ITL**

In *local* ITL (LITL), we restrict each variable $P$ to be true of an interval $s_0 \ldots s_n$ iff $P$ is true of the first state $s_0$:

$$\mathcal{M}_{s_0 \ldots s_n}[\![P]\!] = \mathcal{M}_{s_0}[\![P]\!]$$

*Theorem* (Halpern and Moszkowski): Propositional local ITL with quantification is decidable.

*Proof*: We give a linear translation from formulas in propositional ITL to formulas in a temporal logic that is known to be decidable. This is the *quantified propositional temporal logic* (QPTL) described and analyzed in Wolper [50] and Wolper et al. [51]. Formulas are built from propositional variables $P$, $Q$, ... and the constructs

$$\neg w \quad w_1 \wedge w_2 \quad \bigcirc w \quad \boxdot w \quad \exists P.\, w$$

where $w$, $w_1$ and $w_2$ are themselves QPTL formulas. The interpretation of variables and formulas is identical to that of local ITL with quantification. The particular QPTL used by us restricts intervals to be finite and is known as *weak* QPTL (WQPTL). Weak QPTL can express such constructs as $\Diamond w$, $w_1 \, \mathcal{U} \, w_2$, and *empty*. For a given variable $P$ and local ITL formula $w$, we now give a translation $g(P, w)$ which is true of an interval $s_0 \ldots s_n$ in weak QPTL iff the variable $P$ is true for the first time in some state $s_i$ and $w$ is true over the initial interval $s_0 \ldots s_i$. Thus, $g(P, w)$ is semantically like the ITL formula

$$\Diamond([\mathit{halt}\, P] \wedge w)$$

Here is the definition of $g$:

$$\begin{aligned}
g(P, Q) &= (\Diamond P) \wedge Q \\
g(P, \neg w) &= [\neg g(P, w) \wedge \Diamond P] \\
g(P, [w_1 \wedge w_2]) &= [g(P, w_1) \wedge g(P, w_2)] \\
g(P, \bigcirc w) &= [\neg P \wedge \bigcirc g(P, w)] \\
g(P, [w_1; w_2]) &= \exists R.\, [g(R, w_1) \wedge ([\neg P] \, \mathcal{U} [R \wedge g(P, w_2)])], \\
&\qquad \text{where } R \text{ does not occur free in either } w_1 \text{ or } w_2. \\
g(P, \exists Q.\, w) &= \exists Q.\, g(P, w)
\end{aligned}$$

A formula $w$ in local ITL has the same semantics as $g(empty, w)$ in weak QPTL:

$$s_0 \ldots s_n \models_{\text{LITL}} w \quad \textit{iff} \quad s_0 \ldots s_n \models_{\text{WQPTL}} g(empty, w)$$

Wolper [50] and Wolper et al. [51] show that the theory of QPTL over infinite intervals is decidable but nonelementary; this result easily extends to weak QPTL. The complexity is elementary in the alternation of $\neg$ and $\exists$. ∎

*Remark*: The translation can be extended to handle local ITL over infinite intervals. ∎

### Lower bound for satisfiability

The decision procedure just given is essentially the best that can be done since D. Kozen (private communication) has proved the following theorem:

*Theorem* (Kozen): Satisfiability for propositional local ITL is nonelementary.

*Proof*: Stockmeyer [44] shows that the problem of deciding the emptiness of an arbitrary regular expression over the alphabet $\{0, 1\}$ and with operators $+$, $\cdot$ and $\neg$ is nonelementary. Given a regular expression $e$, we construct an ITL formula $h(e)$ which is satisfiable iff the language generated by $e$ is nonempty. The definition of $h$ given by induction on the syntactic structure of $e$:

$$\begin{aligned} h(0) &= (\neg P \ \wedge \ empty) \\ h(1) &= (P \ \wedge \ empty) \\ h(e_1 + e_2) &= [h(e_1) \ \vee \ h(e_2)] \\ h(\neg e) &= \neg h(e) \\ h(e_1 \cdot e_2) &= [h(e_1); skip; h(e_2)] \end{aligned}$$

For example, the translation of the regular expression $(01) + \neg 1$ is

$$[(\neg P \wedge empty); skip; (P \wedge empty)] \ \vee \ \neg(P \wedge empty)$$

Note that the length of $h(e)$ is linear in that of $e$.

A formal proof relating nonemptiness of a regular expression $e$ and satisfiability of the ITL formula $h(e)$ would use a straightforward induction of the syntactic structure of $e$. ∎

*Remark*: **We can show nonelementary complexity even with the operator *empty* instead of *skip*. We use a modified translation $h'$ defined as follows:**

$$h'(0) = group(P, Q)$$
$$h'(1) = group(\neg P, Q)$$
$$h'(e_1 + e_2) = [h'(e_1) \lor h'(e_2)]$$
$$h'(\neg e) = \neg h'(e)$$
$$h'(e_1 e_2) = [h'(e_1); skip; h'(e_2)]$$

Again, the language $L(e)$ generated by $e$ is nonempty iff $h'(e)$ is satisfiable. ∎

CHAPTER 3

# FIRST-ORDER INTERVAL TEMPORAL LOGIC

## §3.1 The Basic Formalism

We now give the syntax and semantics of first-order ITL. This subsequently serves as our hardware description language.

### Syntax of expressions

Expressions and formulas are built inductively as follows:

- Individual variables: $U, V, \ldots$
- Functions: $f(e_1, \ldots, e_k)$, where $k \geq 0$ and $e_1, \ldots, e_k$ are expressions. In practice, we use functions such as $+$ and $\vee$ (bit-or). Constants like 0 and 1 are treated as zero-place functions.

### Syntax of formulas

- Predicates: $p(e_1, \ldots, e_k)$, where $k \geq 0$ and $e_1, \ldots, e_k$ are expressions. Predicates include $\leq$ and other basic relations.
- Equality: $e_1 = e_2$, where $e_1$ and $e_2$ are expressions.
- Logical connectives: $\neg w$ and $w_1 \wedge w_2$, where $w$, $w_1$ and $w_2$ are formulas.
- Existential quantification: $\exists V.\, w$, where $V$ is a variable and $w$ is a formula.

- Next: $\bigcirc\, w$, where $w$ is a formula.

- Semicolon: $w_1; w_2$, where $w_1$ and $w_2$ are formulas.

### Models

A model consists of a set of states $\Sigma = \{s, t, \ldots\}$ and domain $D$ together with an interpretation $\mathcal{M}$ mapping each variable $V$ and interval $s_0 \ldots s_n$ to some value $\mathcal{M}_{s_0 \ldots s_n}[\![V]\!]$ in $D$. Furthermore, each function and predicate symbol is given some meaning. As in propositional ITL, for quantification to properly work, there must be some interval for every possible behavior of variables. Each $k$-place function symbol $f$ has an interpretation $\mathcal{M}[\![f]\!]$ which is a function mapping $k$ elements in $D$ to a single value:

$$\mathcal{M}[\![f]\!] \in (D^k \to D)$$

Interpretations of predicate symbols are similar but map to truth values:

$$\mathcal{M}[\![p]\!] \in (D^k \to \{true, false\})$$

The semantics given here keep the interpretations of function and predicate symbols independent of intervals and thus time-invariant. The semantics can however be extended to allow for functions and predicates that take into account the dynamic behavior of parameters.

### Interpretation of expressions and formulas

We now extend the interpretation $\mathcal{M}$ to arbitrary expressions and formulas:

- $\mathcal{M}_{s_0 \ldots s_n}[\![f(e_1, \ldots, e_k)]\!] = \mathcal{M}[\![f]\!](\mathcal{M}_{s_0 \ldots s_n}[\![e_1]\!], \ldots, \mathcal{M}_{s_0 \ldots s_n}[\![e_k]\!])$,
  The interpretation of the function symbol $f$ is applied to the interpretations of $e_1, \ldots, e_k$.

- $\mathcal{M}_{s_0 \ldots s_n}[\![p(e_1, \ldots, e_k)]\!] = \mathcal{M}[\![p]\!](\mathcal{M}_{s_0 \ldots s_n}[\![e_1]\!], \ldots, \mathcal{M}_{s_0 \ldots s_n}[\![e_k]\!])$

- $\mathcal{M}_{s_0 \ldots s_n}[\![e_1 = e_2]\!] = true \quad \textit{iff} \quad \mathcal{M}_{s_0 \ldots s_n}[\![e_1]\!] = \mathcal{M}_{s_0 \ldots s_n}[\![e_2]\!]$

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

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

- $\mathcal{M}_{s_0\ldots s_n}[\![\exists V.\, w]\!] = \mathit{true}$ *iff* for some $\alpha$, $\mathcal{M}_{s'_0\ldots s'_n}[\![w]\!] = \mathit{true}$, where $s'_0\ldots s'_n = (s_0\ldots s_n)[V/\alpha]$ and the function $\alpha(i,j)$ maps pairs $0 \le i \le j \le n$ to values in the data domain $D$.

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

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

Satisfiability and validity of formulas are as in the propositional case. All the other related temporal operators mentioned earlier are expressible as before. If the data domain $D$ includes at least two values, the iterative construct $w^*$ can also be expressed.

### Arithmetic domain

We will assume that the data domain $D$ contains natural numbers as well as nested finite lists. Both 0 and 1 serve as numbers and bits, with 0 standing for low voltage and 1 standing for high voltage. The data domain does not contain any intermediate voltages or "undefined" values. We permit finite sets and represent them by lists. The following are sample values:

$$0, \quad 3, \quad \langle 0\rangle, \quad \{1,2\}, \quad \langle\rangle, \quad \langle 6,3,\langle\rangle,9\rangle, \quad \langle 4,\{3,2\}\rangle$$

We adapt the convention that an $n$-element list $L$ has subscripts ranging from 0 on the left to $n-1$ on the right:

$$L = \langle L[0],\ldots,L[n-1]\rangle, \quad \text{where } n = |L|$$

It is assumed that $\mathcal{M}$ contains standard interpretations of function and predicate symbols such as $+$, $\le$ and $\vee$ (bit-or). We also include conditional expressions and conventional operators for constructing, combining, subscripting and determining the length of finite lists and sets.

The unary predicate $nat(U)$ is true if $U$'s value is a natural number (i.e., nonnegative integer).

$$\mathcal{M}_{s_0 \ldots s_n}[\![nat(U)]\!] = true \quad \mathit{iff} \quad \mathcal{M}_{s_0 \ldots s_n}[\![U]\!] \in \{0, 1, 2, \ldots\}$$

Sometimes we use the predicate *time* instead of *nat* when the associated parameter is used as a time. The two predicates are however semantically equivalent. The predicate *bit* checks if a value is either 0 or 1 and the predicate *positive* checks for positive integers (that is, integers $\geq 1$).

### Temporal domain

A variable $V$ is *static* in an interval $s_0 \ldots s_n$ if $V$ has a single interpretation over all subintervals:

$$\mathcal{M}_{s_0 \ldots s_n}[\![V]\!] = \mathcal{M}_{s_i \ldots s_j}[\![V]\!], \quad \text{for all } 0 \leq i \leq j \leq n$$

Just as *nat* and *bit* look at the type of a value, the predicate *static* checks that its parameter is static in an interval. We give *static* the following interpretation:

$$\begin{aligned} &\mathcal{M}_{s_0 \ldots s_n}[\![static(V)]\!] = true \\ &\qquad\qquad \mathit{iff} \quad \text{for some } d \in D, \text{ for all } 0 \leq i \leq j \leq n,\ \mathcal{M}_{s_i \ldots s_j}[\![V]\!] = d, \end{aligned}$$

Within an interval $s_0 \ldots s_n$, a *signal* has a unique value for all subintervals starting with a given state. Thus, signals are local in the sense of LITL. The predicate $signal(V)$ is true iff the variable $V$ behaves as a signal. We define *signal* as follows:

$$signal(V) \quad \equiv_{\text{def}} \quad \boxdot\, \exists U.\, [static(U) \wedge \boxdot(V = U)]$$

The predicate *Bit* checks that its parameter is always bit-valued:

$$Bit(V) \quad \equiv_{\text{def}} \quad \boxdot\, bit(V)$$

### Naming conventions for variables

For convenience, we will associate sorts with variables:

- **Interval variables: $\mathcal{A}, \mathcal{N}, \mathcal{X}, \ldots$**
  **These can vary in value from interval to interval and are also known as *non-local* or *path* variables.**

- Signal variables: $A, N, X, \ldots$
  Signal variables can also be referred to as *local* or *state* variables.

- Static variables: $a, n, x, \ldots$
  Static variables can also be called *global* or *frame* variables. All static variables are signals.

In general, variables such as $\mathcal{A}$, $B$ and $c$ range over all elements of the data domain $D$. On the other hand, $\mathcal{J}$, $K$ and $n$ range over natural numbers. The variables $\mathcal{X}$, $Y$ and $z$ always equal one of the bit values 0 and 1. If desired, the naming style suggested here can also be used in propositional ITL.

As in conventional first-order logic, sort information can always be made explicit. For example, a formula $\forall b.\, w$ containing a static variable $b$ is equivalent to the formula

$$\forall V.\, [static(V) \supset w_b^V]$$

where the formula $w_b^V$ results from replacing all free occurrences of $b$ in $w$ by the sort-free variable $V$.

## §3.2 Some First-Order Temporal Concepts

Within the framework of first-order temporal logic, we can explore a variety of qualitative and quantitative timing issues. The constructs given below are useful for describing and reasoning about circuits.

### Temporal assignment

The formula $A \rightarrow B$ is true for an interval if the signal $A$'s initial value equals $B$'s final value:

$$A \rightarrow B \quad \equiv_{\text{def}} \quad \forall c.\, \big[beg(A = c) \supset fin(B = c)\big]$$

We call this *temporal assignment.* Unlike in conventional programming languages, it is perfectly acceptable to have an arbitrary expression on the receiving end of the arrow. Furthermore, temporal assignment only affects variables explicitly mentioned; the values of other variables do not necessarily remain fixed. Incidentally, because the variables $A$ and $c$ are signals, the subformula $beg(A = c)$ used in the definition could be replaced by $A = c$.

*Examples:*

| *Concept* | *Formula* |
|---|---|
| $Z$ gets the initial value of $\neg Y$ | $(\neg Y) \rightarrow Z$ |
| $I$ doubles | $2I \rightarrow I$ |
| $M + N$ doesn't change | $(M + N) \rightarrow (M + N)$ |
| $A$ and $B$ swap values | $(A \rightarrow B) \wedge (B \rightarrow A)$ |

As noted above, temporal assignment specifies nothing about the behavior of those variables that are not referenced. Thus, the formulas

$$[(I + 2) \rightarrow I]$$

and

$$[(I + 2) \rightarrow I] \wedge [J \rightarrow J]$$

are not equivalent.

*Properties:*

$$\models \quad a \rightarrow a$$

A static variable's initial and final values agree.

$$\models \quad [(A \rightarrow B); (B \rightarrow C)] \supset (A \rightarrow C)$$

If $B$ gets $A$'s value and then $C$ gets $B$'s, the net result is that $C$ gets $A$'s initial value.

$$\models \quad empty \supset (A \rightarrow A)$$

In an empty interval, the first and last states are identical. Therefore, a variable's initial and final values agree.

$$\models \quad (A \to B) \supset [f(A) \to f(B)]$$

If $A$ is assigned to $B$, then any time-invariant function application $f(A)$ is passed to $f(B)$.

$$\models \quad [(\neg Z \to Z);(\neg Z \to Z)] \supset (Z \to Z)$$

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

### Temporal equality

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

$$A \approx B \quad \equiv_{\text{def}} \quad \boxdot(A = B)$$

Because $A$ and $B$ are signals, the formula $A \approx B$ can also be expressed using the linear-time temporal operator $\boxdot$:

$$\models \quad A \approx B \quad \equiv \quad \boxdot(A = B)$$

*Examples:*

| *Concept* | *Formula* |
|---|---|
| The signal $A$ is 0 throughout the interval | $A \approx 0$ |
| The bit-and of $X$ and $Y$ everywhere equals 0 | $(X \wedge Y) \approx 0$ |
| $X$ agrees everywhere with the complement of $Y$ | $X \approx \neg Y$ |

*Property:*

$$\models \quad [\langle A, B\rangle \approx \langle A', B'\rangle] \equiv (A \approx A' \wedge B \approx B')$$

The pair $\langle A, B\rangle$ temporally equals $\langle A', B'\rangle$ exactly if the signal $A$ temporally equals $A'$ and $B$ temporally equals $B'$.

## Temporal stability

A signal $A$ is *stable* if it has a fixed value. The notation used is $\mathit{stb}\, A$ and can be expressed as shown below:

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

It follows from this that every static variable is stable.

*Properties:*

$\models \quad \mathit{stb}\, X \equiv [X \approx \mathbf{0} \lor X \approx \mathbf{1}]$

A bit signal $X$ is stable iff it is always $\mathbf{0}$ or always $\mathbf{1}$.

$\models \quad \mathit{stb}\langle A, B\rangle \equiv [\mathit{stb}\, A \land \mathit{stb}\, B]$

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

## Iteration

The propositional constructs $w^*$ and *while* $w_1$ *do* $w_2$ can be expressed as in propositional ITL with quantification. We can also augment the first-order logic with iteration of the form $w^e$ where $w$ is a formula and $e$ is an arithmetic expression. We first define the construct $\mathit{cycle}^e\, w$ which iterates $w$ the number of times specified by $e$:

$$\mathit{cycle}^e\, w \quad \equiv_{\text{def}} \quad \exists I.\, \big[\mathit{beg}(I = e) \land \mathit{while}\, (I \neq 0)\, \mathit{do}\, (w \land [I - 1 \to I])\big]$$

where the quantified variable $I$ does not occur free in $e$ or $w$. We initially set $I$ to $e$ and then decrement $I$ by 1 over each iteration. The semantics of *cycle* are such that the individual iterations of $w$ take at least one unit of time since $I$ cannot decrease in an empty interval. Thus the formulas

$$\mathit{cycle}^e\, w$$

and

$$\mathit{cycle}^e(w \land \neg \mathit{empty})$$

are semantically equivalent.

In order for the formula $w^e$ to permit possibly empty steps, we define it as follows:

$$w^e \quad \equiv_{\mathrm{def}} \quad cycle^e\, w \;\vee\; \exists\, i, j.\big(i + j < e \;\wedge\; [(cycle^i w); (w \;\wedge\; empty); (cycle^j w)]\big)$$

where the static variables $i$ and $j$ do not occur free in $w$ or $e$. By introducing extra quantified variables that always equal $w$ and $e$, we can modify this definition to be linear in the size of $w$ and $e$.

*Examples:*

| *Concept* | *Formula* |
|---|---|
| $Z$ is complemented $n$ times | $(\neg Z \rightarrow Z)^n$ |
| $N$ doubles some number of times | $(2N \rightarrow N)^*$ |
| $I$ keeps halving itself | $(I \rightarrow 2I)^*$ |
| While construct | $while\,(I < n)\; do\,(I + 1 \rightarrow I)$ |

*Properties:*

$$\models \quad (f(A) \rightarrow A)^3 \;\supset\; [f^3(A) \rightarrow A]$$

After a series of three applications of $f$, $A$ ends up with the initial value of $f^3(A)$, where $f^3(A) = f(f(f(A)))$.

$$\models \quad ([I + 1 \rightarrow I]^m)^n \;\supset\; ([I + mn] \rightarrow I)$$

This property illustrates how to nest iteration.

## Measuring the length of an interval

We will view the formula

$$len = e$$

as an abbreviation for the iterative construct

$$skip^e$$

This is true exactly of intervals with length $e$. The construct $len \geq e$ expands to

$$\exists i \geq e.\,(len = i)$$

We can similarly use formulas such as $len < e$.

Alternatively, we can introduce $len$ as an interpreted 0-place temporal function whose value for any interval $s_0 \ldots s_n$ equals the length $n$:

$$\mathcal{M}_{s_0 \ldots s_n}[\![len]\!] = n$$

*Examples:*

| *Concept* | *Formula* |
|---|---|
| The signal $A$ is stable and the interval has $\geq m + n$ units | $stb\,A \;\wedge\; (len \geq m + n)$ |
| In some subinterval of length $\geq m$, $X$ is stable | $\diamond([len \geq m] \;\wedge\; stb\,X)$ |
| $I$ doubles in $\leq I$ steps | $(2I \rightarrow I) \;\wedge\; (len \leq I)$ |

*Properties:*

$$\models \quad empty \;\equiv\; (len = 0)$$

The predicate *empty* is true exactly if the interval has length 0.

$$\models \quad skip \;\equiv\; (len = 1)$$

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

$$\models \quad (len = m + n) \;\equiv\; [(len = m); (len = n)]$$

An interval of length $m+n$ can be subdivided into two adjacent intervals of lengths $m$ and $n$. The converse is also true.

### Expressions based on *next*

We extend the operator *next* to handle expressions. The construct $\bigcirc\, e$ for an interval $s_0 \ldots s_n$ equals the value of the expression $e$ in the subinterval $s_1 \ldots s_n$:

$$\mathcal{M}_{s_0 \ldots s_n}[\![\bigcirc\, e]\!] = \mathcal{M}_{s_1 \ldots s_n}[\![e]\!]$$

If the length of the interval is 0, the resulting value is left unspecified. The following natural extension of *next* facilitates looking at values some specified number of units in the future:

$$\mathcal{M}_{s_0 \ldots s_n}[\![\bigcirc^{e_1} e_2]\!] = \mathcal{M}_{s_i \ldots s_n}[\![e_2]\!], \quad \text{where } i = \mathcal{M}_{s_0 \ldots s_n}[\![e_1]\!]$$

This definition results in the following properties:

$$\models \quad \bigcirc^0 e \quad = \quad e$$

$$\models \quad \bigcirc^1 e \quad = \quad \bigcirc\, e$$

We can analogously permit formulas of the form $\bigcirc^e w$, where $w$ is itself a formula and $e$ is an expression.

We now show how to eliminate these constructs. The formula $\bigcirc^e w$ abbreviates

$$\exists i.\,\big([i = e] \ \wedge\ [(len = i); w]\big),$$

where $i$ does not occur free in $w$ or $e$. A formula of the form $A = \bigcirc^{e_1} e_2$ becomes

$$\exists b.\,\big[(\bigcirc^{e_1}[b = e_2]) \ \wedge\ (A = b)\big]$$

where $b$ does not occur free in $e_1$ or $e_2$.

### Initial and terminal stability

The predicate $istb^m A$ is true for an interval $s_0 \ldots s_n$ if the signal $A$ is stable in the initial states $s_0 \ldots s_m$. The next definition has this meaning:

$$istb^m A \quad \equiv_{\text{def}} \quad \diamond\!\!\!\!\cdot\,(stb\, A \ \wedge\ len = m)$$

Note that the formula is false on an interval of length less than $m$. By analogy, $tstb^m A$ is true if $A$ ends up stable for at least $m$ units of time:

$$tstb^m A \quad \equiv_{\text{def}} \quad \diamond\!\!\!\!\circ\,(stb\, A \ \wedge\ len = m)$$

*Property:*

$$\models \quad istb^{m+n} A \supset istb^{m} A$$

The time factor can be reduced.

### Blocking

It is useful to specify that as long as a signal $A$ remains stable, so does another signal $B$. We say that $A$ *blocks* $B$ and write this as $A$ *blk* $B$. The predicate *blk* can be expressed using the temporal formula

$$A \; blk \; B \quad \equiv_{\rm def} \quad \boxdot(stb\, A \supset stb\, B)$$

*Examples:*

| *Concept* | *Formula* |
|---|---|
| While $A$ remains stable, so do $B$ and $C$ | $A\ blk\ \langle B, C\rangle$ |
| As long as the pair $\langle A, B\rangle$ is stable, so is C | $\langle A, B\rangle\ blk\ C$ |

*Properties:*

$$\models \quad [A\ blk\ B \ \wedge\ stb\, A] \supset \ stb\, B$$

If $A$ blocks $B$ and $A$ is stable, then so is $B$.

$$\models \quad [A\ blk\ B \ \wedge\ B\ blk\ C] \supset A\ blk\ C$$

Blocking is transitive.

$$\models \quad A\ blk\ \langle B, C\rangle \equiv [A\ blk\ B \ \wedge\ A\ blk\ C]$$

The signal $A$ blocks the pair $\langle A, B\rangle$ exactly if $A$ blocks both $B$ and $C$. This and the next property generalize to lists of arbitrary length.

$$\models \quad \langle A, B\rangle\ blk\ C \equiv [A\ blk\ C \ \vee\ B\ blk\ C]$$

The pair $\langle A, B\rangle$ blocks $C$ iff $A$ blocks $C$ or $B$ blocks $C$.

$\models \quad A\ blk\ B \supset (stb\, A \rightsquigarrow A\ blk\ B)$

If $A$ blocks $B$, then after $A$ is stable it continues to block $B$.

The predicate $A\ blk\ B$ can be extended to allow for quantitative timing. When describing the behavior of digital circuits, it is often useful to state that in any initial interval where $A$ remains stable up to within the last $m$ units of time, $B$ is stable throughout:

$$A\ blk^m\ B \quad \equiv_{\text{def}} \quad \boxdot\big[(stb\, A;\ len \le m) \supset stb\, B\big]$$

This modification has utility in situations where $B$ is known to be slow in responding to changes in $A$.

*Properties:*

$\models \quad A\ blk\ B \equiv A\ blk^0\ B$

The original notation is equivalent to the quantitative one with blocking factor 0.

$\models \quad [A\ blk^m\ B \wedge B\ blk^n\ C] \supset A\ blk^{m+n}\ C$

Transitivity accumulates blocking factors. Other properties of the predicate *blk* can also be extended to include quantitative timing.

$\models \quad A\ blk^1\ A \supset stb\, A$

If a signal $A$ won't change until after it does then $A$ is stable. This is a form of induction over time. The converse is also true.

## Rising and falling signals

A rising bit signal can be described by the predicate $\uparrow X$:

$$\uparrow X \quad \equiv_{\text{def}} \quad \big[(X \approx 0);\ skip;\ (X \approx 1)\big]$$

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

$$\downarrow X \quad \equiv_{\text{def}} \quad \big[(X \approx 1);\ skip;\ (X \approx 0)\big]$$

*Examples:*

| *Concept* | *Formula* |
| --- | --- |
| $X$ is stable and $Y$ goes up | $stb\, X \land \uparrow Y$ |
| The bit-or of $X$ and $Y$ falls | $\downarrow(X \lor Y)$ |
| In every subinterval where $X$ rises, $Y$ falls | $\boxdot(\uparrow X \supset \downarrow Y)$ |
| X goes up and then back down | $\uparrow X; \downarrow X$ |
| $X$ twice goes up and down | $(\uparrow X; \downarrow X)^2$ |

*Properties:*

$$\models \;\; (\uparrow X \land \uparrow Y) \supset [\uparrow(X \land Y) \land \uparrow(X \lor Y)]$$

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

$$\models \;\; \downarrow X \equiv \uparrow(\neg X)$$

A bit signal falls exactly if its complement rises.

$$\models \;\; [\uparrow X \land beg(Y = 0) \land (X\; blk\; Y)] \supset \uparrow(X \lor Y)$$

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

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

$$\uparrow^{m,n} X \quad \equiv_{\text{def}} \quad [(X \approx 0 \land len \geq m);\; skip;\; (X \approx 1 \land len \geq n)]$$

This can also be expressed as shown below:

$$\models \;\; \uparrow^{m,n} X \equiv (\uparrow X \land istb^m X \land tstb^n X)$$

Thus, the extended form of $\uparrow$ can be reduced to the original one with separate details concerning initial and terminal stability.

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

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

Positive pulses of the form $\uparrow\downarrow^{l,m,n} X$ are similarly defined. These constructs can be further modified to provide for noninstantaneous rise and fall times.

### Smoothness

A bit signal $X$ is smooth if it is either stable or has a single transition. The following definition illustrates one way to express smoothness:

$$sm\, X \quad \equiv_{\text{def}} \quad (stb\, X \vee \uparrow X \vee \downarrow X)$$

The next property gives two equivalent ways to say that a bit signal raises or falls:

$$\models \quad (\uparrow X \vee \downarrow X) \equiv (sm\, X \wedge [\neg X \rightarrow X])$$

Since digital devices often require clock inputs to be smooth, it is sometimes important to ensure that a signal has this property. The predicate $sm$ can be extended to include quantitative timing details similar to those given for the predicates $\uparrow$ and $\downarrow$:

$$sm^{m,n}\, X \quad \equiv_{\text{def}} \quad (sm\, X \wedge istb^{m}\, X \wedge tstb^{n}\, X)$$

The notion of smoothness generalizes to arbitrary signals. A scalar-valued signal $A$ is smooth if it is either stable or has a single transition:

$$sm\, A \quad \equiv_{\text{def}} \quad [stb\, A \vee (stb\, A;\ skip;\ stb\, A)]$$

A list $L$ is inductively defined to be smooth if all its components are smooth:

$$sm\, L \quad \equiv_{\text{def}} \quad \forall 0 \leq i < |L|.\, (sm\, L[i])$$

The individual components of $L$ need not all change at the same instant.

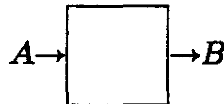
CHAPTER 4

# DELAYS AND COMBINATIONAL ELEMENTS

Delay is a fundamental phenomenon in dynamic systems and an examination of it touches upon basic issues ranging from feedback and parallelism to implementation and internal device states. In addition, a key design decision in building any hardware simulator centers around the treatment of delay. For example, Breuer and Friedman [10] and Blunden et al. [8] present a number of models of propagation. For these and other reasons, it is worth taking a detailed look at various forms of signal propagation.

## §4.1 Unit Delay

One of the simplest and most important types of delay elements can be modeled as having the following structure:




Here $A$ is the input signal and $B$ is the associated output. The following statement uses intervals to characterize the desired behavior:

*In every subinterval of length exactly one unit, the initial value of the input $A$ equals the final value of the output $B$.*

The next predicate *del* formalizes this:

$$A \; del \; B \quad \equiv_{\text{def}} \quad \boxdot[(len = 1) \supset (A \to B)]$$

*Properties:*

$$\models \quad (A\ del\ B) \equiv (skip \wedge [A \rightarrow B])^*$$

Unit delay can also by viewed as the successive iteration of atomic assignments. This suggests how to implement unit delay by means of looping.

$$\models \quad (A\ del\ B) \equiv keep(A = \bigcirc B)$$

The concept of unit delay can be expressed in *semicolon*-free linear-time temporal logic.

$$\models \quad (A\ del\ A) \equiv stb\ A$$

If a signal is feed back to itself, it is stable. The converse is also true.

## §4.2 Transport Delay

It is natural to extend the predicate *del* to cover delays over $m$-unit intervals:

$$A\ del^m\ B \quad \equiv_{\mathsf{def}} \quad \boxdot(len = m \supset [A \rightarrow B])$$

Breuer and Friedman [10] refer to this as *transport delay.*

*Properties:*

$$\models \quad (A\ del^0\ B) \equiv (A \approx B)$$

Zero delay is equivalent to temporal equality.

$$\models \quad A\ del^0\ A$$

A signal has zero delay to itself.

$$\models \quad (A\ del^m\ B \wedge B\ del^n\ C) \supset A\ del^{m+n}\ C$$

Delay is cumulative.

$$\models \quad \langle A, B\rangle\ del^m\ \langle A', B'\rangle \equiv (A\ del^m\ A' \wedge B\ del^m\ B')$$

Delay between pairs is equivalent to component-wise delay. This generalizes to lists of arbitrary length.

## §4.3 Functional Delay

Often, one signal receives a delayed function of another. The following examples illustrate this and are based on the predicate *del* although the other delay models later presented can also be used.

*Examples:*

| *Concept* | *Formula* |
|---|---|
| $X$ keeps on being complemented | $(\neg X)\ del\ X$ |
| $B$ either accepts $A$ or itself, depending on $X$ | $[\mathit{if}\ (X = 1)\ \mathit{then}\ A\ \mathit{else}\ B]\ del\ B$ |
| $N$ keeps on doubling | $2N\ del\ N$ |
| $A$ receives a delayed $f(A, B)$ | $f(A, B)\ del\ A$ |
| $I$ keeps decrementing by 1 | $I\ del\ (I + 1)$ |

Here is the description of a system that runs the variable $I$ from 0 to $n$ and simultaneously sums $I$ into $J$:

$$beg(I = 0\ \wedge\ J = 0)\ \wedge\ [(I + 1)\ del\ I]\ \wedge\ [(J + I)\ del\ J]\ \wedge\ halt(I = n)$$

*Properties:*

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

Functional composition applies.

$$\models\ \ (\neg X)\ del^m\ Y\ \equiv\ X\ del^m\ (\neg Y)$$

Bit inversion can occur either on the input or output.

$$\models\ \ [(\neg X)\ del^m\ Y\ \wedge\ (\neg Y)\ del^n\ Z]\ \supset\ X\ del^{m+n}\ Z$$

Two inverters cancel.

## §4.4 Delay Based on Shift Register

An $(m+1)$-bit vector $R$ acting as a shift register can be specified as follows:

$$R[0] \; del \; R[1] \;\land\; \cdots \;\land\; R[m-1] \; del \; R[m]$$

Over each unit of time, the contents of $R$ shift right by one element. That is, the value of $R[0]$ is passed to $R[1]$ and so forth. This description is more formally expressed by means of quantification:

$$\forall i \in [0, m-1].\,(R[i] \; del \; R[i+1])$$

The next formula has the same meaning but is more concise:

$$R[0 \,\mathrm{to}\, m-1] \; del \; R[1 \,\mathrm{to}\, m],$$

where the vector $R[0 \,\mathrm{to}\, m-1]$ by definition equals $\langle R[0], \ldots, R[m-1] \rangle$.

The following property shows how to achieve an $m$-unit delay by means of such a shift register:

$$\models \quad R[0 \,\mathrm{to}\, m-1] \; del \; R[1 \,\mathrm{to}\, m] \;\supset\; R[0] \; del^m \; R[m] \qquad (*)$$

This suggests an implementation of $A \; del^m \; B$ of the form $A \; shdel^m_R \; B$:

$$A \; shdel^m_R \; B \quad \equiv_{\mathrm{def}} \quad (A \approx R[0] \;\land\; R[m] \approx B \;\land\; R[0 \,\mathrm{to}\, m-1] \; del \; R[1 \,\mathrm{to}\, m])$$

Here, the value of $A$ is fed into $R[0]$ and $B$ receives the value $R[m]$. The correctness of this implementation is given by the following property:

$$\models \quad A \; shdel^m_R \; B \;\supset\; A \; del^m \; B$$

We can localize $R$ in the formula $A \; shdel^m_R \; B$ by defining a variant $A \; shdel^m \; B$ that existentially quantifies over $R$:

$$A \; shdel^m \; B \quad \equiv_{\mathrm{def}} \quad \exists R.[(R\colon signal^{m+1}) \;\land\; (A \; shdel^m_R \; B)]$$

Here the construct

$$R\colon signal^{m+1}$$

constrains $R$ to being a vector of $m+1$ signals. This notation will be described in more detail in the next chapter. Note that $R$ is assumed to exist without necessarily being externally visible to an observer. The quantifier's effect on scoping is similar to that of a begin-block in a conventional block-structured programming language. We call $A\ shdel^m\ B$ an *external* specification of the implementation. In fact, this is logically equivalent to the basic delay predicate $A\ del^m\ B$ as the next property states:

$$\models \quad A\ shdel^m\ B \equiv A\ del^m\ B$$

Basically, the proof that *shdel* implies *del* follows from the property (*) given above. The converse requires demonstrating that some $R$ exists. Perhaps the easiest way to do this is by direct construction. At each instant of time, the values of the $m+1$ elements of $R$ can be those of the next $m+1$ values of $B$ in appropriate order:

$$R[i] \approx \bigcirc^{m-i} B, \quad \text{for } 0 \le i \le m$$

The output value $R[m]$ always equals the expression $\bigcirc^0 B$, which is defined to be $B$'s current value. Similarly, $R[0]$ always equals $\bigcirc^m B$, that is, the value $B$ will have $m$ units later. This technique works even if the interval has length less than $m$.

## §4.5 Variable Transport Delay

A batch of delay elements may have varying characteristics although each individual device is rather fixed in its timing behavior. The predicate $A\ vardel^{m,n}\ B$ specifies that $A$'s value is propagated to $B$ by transport delay with some uncertain factor between $m$ and $n$:

$$A\ vardel^{m,n}\ B \quad \equiv_{\text{def}} \quad \exists i \in [m,n].\ (A\ del^i\ B)$$

## §4.6 Delay with Sampling

Digital circuits often require that inputs remain stable and be sampled for some minimum amount of time in order to ensure proper device operation. The delay model $A\ sadel\ B$ has this characteristic:

$$A\ sadel^m\ B \quad \equiv_{\text{def}} \quad \boxdot[(stb\ A \ \wedge\ len \geq m) \supset\ fin(A = B)]$$

Here the input $A$ must be stable at least $m$ units of time for the output $B$ to equal $A$. Behavior during changes in $A$ is left unspecified. The properties below illustrate two other ways of expressing *sadel*. We present them to demonstrate other possible styles:

$$\models \quad A\ sadel^m\ B \equiv \boxdot(tstb^m\ A \supset\ fin(A = B))$$

$$\models \quad A\ sadel^m\ B \equiv [tstb^m\ A \leadsto beg(A = B)]$$

*Properties:*

$$\models \quad A\ del^m\ B \supset A\ sadel^m\ B$$

Basic delay implements sampling-time delay.

$$\models \quad A\ sadel^m\ B \equiv \big(tstb^m\ A \leadsto [beg(A = B) \ \wedge\ A\ blk\ B]\big)$$

Once the device stabilizes, the input $A$ blocks the output $B$.

The predicate *sadel* can be extended to associate some factor with the blocking of $B$ by $A$:

$$A\ sadel^{m,n}\ B \quad \equiv_{\text{def}} \quad \big(tstb^m A \leadsto [beg(A = B) \ \wedge\ A\ blk^n\ B]\big)$$

In a sense, $m$ is the maximum delay and $n$ is the minimum delay.

## §4.7 An Equivalent Delay Model with an Internal State

A related delay model $A\,stdel_X^{m,n}\,B$ is based on a bit flag $X$ that is set to 1 after the input $A$ has been held stable $m$ units. Whenever $X$ is 1, the input $A$ equals the

output $B$ and blocks $X$, which in turn blocks $B$ by the factor $n$:

$$\begin{aligned} A\ stdel_X^{m,n}\ B \quad &\equiv_{\text{def}} \\ &\boxdot([stb\,A \wedge len \geq m] \supset fin(X = 1)) \\ &\quad \wedge \boxdot(beg(X = 1) \supset [beg(A = B) \wedge A\ blk\ X \wedge X\ blk^n\ B]) \end{aligned}$$

In the manner described earlier, we internalize $X$ by existentially quantifying over it:

$$A\ stdel^{m,n}\ B \equiv \exists X.\,(A\ stdel_X^{m,n}\ B)$$

This external form is in fact logically equivalent to $A\ sadel^{m,n}\ B$:

$$\models \quad A\ stdel^{m,n}\ B \equiv A\ sadel^{m,n}\ B$$

The following construction for $X$ can be used:

$$X \approx \big(if\,\big[beg(A = B) \wedge A\ blk^n\ B\big]\ then\ 1\ else\ 0\big)$$

The right hand expression is not a signal but is converted to one as outlined in the next chapter.

There are a variety of specifications that use different internal signals such as $X$ and yet are externally equivalent.

## §4.8 Delay with Separate Propagation Times for 0 and 1

Sometimes it is important to distinguish between the propagation times for 0 and 1. The following variant of *sadel* does this by having separate timing values for the two cases. The delay's input and output are both bit signals.

$$\begin{aligned} X\ sadel01^{m,n}\ Y \quad &\equiv_{\text{def}} \\ &\boxdot([X \approx 0 \wedge len \geq m] \supset fin(X = Y)) \\ &\wedge \boxdot([X \approx 1 \wedge len \geq n] \supset fin(X = Y)) \end{aligned}$$

*Property:*

$$\models \quad X\ sadel01^{m,n}\ Y \supset X\ sadel^{\max(m,n)}\ Y$$

The separate propagation times can be reduced to those for the more general form of sampling-time delay by using the larger of the two parameters.

## §4.9 Smooth Delay Elements

It is possible to specify that between times when the delay element is steady, if the input changes smoothly, then so does the output. We call such a device a *smooth* delay element. This type of delay has utility in systems that must propagate clock signals without distortion. Here is a predicate based on the earlier specification *stdel*:

$$
\begin{aligned}
&A\ smdel_X^{m,n}\ B \quad \equiv_{\text{def}}\\
&\quad A\ stdel_X^{m,n}\ B\\
&\qquad \land\ \boxdot\big([beg(X=1)\ \land\ fin(X=1)\ \land\ sm\,A]\ \supset\ sm\,B\big)
\end{aligned}
$$

The external form quantifies over $X$:

$$A\ smdel^{m,n}\ B \quad \equiv_{\text{def}} \quad \exists X.\,(A\ smdel_X^{m,n}\ B)$$

## §4.10 Delay with Tolerance to Noise

Sometimes it is important to consider the affects of transient noise during signal changes. A signal $A$ is *almost smooth* with factor $l$ if $A$ is continuously stable all but at most $l$ contiguous units of time:

$$stb\,A; (len \leq l); stb\,A$$

The delay model *toldel* is similar to *smdel* but has an additional timing coefficient $l$ for showing how almost smooth input changes result in smooth output transitions:

$$
\begin{aligned}
&A\ toldel_X^{m,n,l}\ B \quad \equiv_{\text{def}}\\
&\quad A\ stdel_X^{m,n}\ B\\
&\qquad \land\ \boxdot\big[(beg(X=1)\ \land\ fin(X=1)\ \land\ [stb\,A; (len \leq l); stb\,A]) \ \supset\ sm\,B\big]
\end{aligned}
$$

From this we can obtain the external form

$$A\ toldel^{m,n,l}\ B$$

The predicate *smdel* is a special case of *toldel* with a noise tolerance of 1 time unit:

$$\models \quad A\ smdel^{m,n}\ B \equiv A\ toldel^{m,n,1}\ B$$

## §4.11 Gates with Input and Output Delays

One might specify an and-gate with both input and output delays as follows:

$$(X, X')\,saand^{m,n}Y \quad \equiv_{\text{def}} \quad \exists Z, Z'.\big[X\,sadel^m Z \wedge X'\,sadel^m Z' \wedge (Z \wedge Z')\,sadel^n Y\big]$$

Here a delay exists from the input $X$ to an internal signal $Z$ and another delay exists from $X'$ to $Z'$. The bit-and of $Z$ and $Z'$ is propagated to $Y$. The input delays are given by $m$ and the output delay by $n$. If we choose to ignore input delays, the model reduces to a single occurrence of *sadel*:

$$\models \quad (X, X')\,saand^{0,n} \;\equiv\; (X \wedge X')\,sadel^n\,Y$$

If the internal propagation is modeled by transport delay, things are even simpler. Here is an and-gate specified in this manner:

$$(X, X')\,tand^{m,n}\,Y \quad \equiv_{\text{def}} \quad \exists Z, Z'.\big[X\;del^m\;Z \;\wedge\; X'\;del^m\;Z' \;\wedge\; (Z \;\wedge\; Z')\;del^n\,Y\big]$$

The predicate *tand* simplifies even if the internal input delay $m$ is not zero:

$$\models \quad (X, X')\,tand^{m,n}\,Y \;\equiv\; (X \;\wedge\; X')\;del^{m+n}\,Y$$

## §4.12 High-Impedance

Digital devices sometimes use the phenomenon of high-impedance as a decentralized means for sharing a common output among several sources. Each source has its own enabling signal which, when on, causes data to pass to the output. When the enable signal is off, the connection "disconnects" or "floats." Pass transistors in MOS semiconductor technology and tri-state drivers in TTL exhibit this kind of behavior. See Gschwind and McCluskey [17] or Mead and Conway [32] for details.

The predicate $A\;pass_X\;B$ specifies the connection of the signals $A$ and $B$ when the bit signal $X$ is 1:

$$A\;pass_X\;B \quad \equiv_{\text{def}} \quad \boxdot[(X = 1) \;\supset\; (A = B)]$$

Thus the pair of devices

$$(A\ pass_X\ B) \land (A'\ pass_{\neg X}\ B)$$

will pass the signal $A$ to $B$ when $X$ is 1 and will pass the signal $A'$ to $B$ when $X$ is 0. The following formula has the same semantics:

$$(\mathit{if}\ [X = 1]\ \mathit{then}\ A\ \mathit{else}\ A') \approx B$$

The predicate *pass* shows that the key feature of high impedance can be modeled in ITL without the introduction of extra bit values.

*Properties:*

$$\models \quad A\ pass_X\ B \equiv B\ pass_X\ A$$

A pass transistor is commutative.

$$\models \quad [A\ pass_X\ B \land (X \approx 1)] \supset (A \approx B)$$

During intervals when the pass transistor is enabled, the input and output are equal.

$$\models \quad [A\ pass_X\ B \land B\ pass_Y\ C] \supset A\ pass_{(X \land Y)}\ C$$

Pass transitor behavior is transitive.

CHAPTER 5

# ADDITIONAL NOTATION

This chapter introduces some useful notation that we need before looking at more complicated devices.

## §5.1 Reverse Subscripting

Because some of the devices we present deal with numbers and their representation as bit vectors, it is convenient to occasionally adapt an alternative subscripting order. Subscripts on a vector $V = \langle v_0, \ldots, v_n \rangle$ normally range from 0 on the left to $n$ on the right. The construct $V[i]$ follows this style. However, in order to simplify reasoning about the correspondence between a bit vector and its numerical equivalent, a slightly different convention is sometimes used. The alternative notation $V\{i\}$ indexes $V$ from the *right* with the right-most element having subscript 0. For example:

$$\langle 1, 0, \underset{\uparrow}{5} \rangle\{0\} = 5, \quad \langle 1, \underset{\uparrow}{0}, 5 \rangle\{1\} = 0, \quad \langle \underset{\uparrow}{1}, 0, 5 \rangle\{2\} = 1$$

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

$$\langle 0, 9, 4, 2 \rangle\{3 \text{ to } 1\} = \langle 0, 9, 4 \rangle, \quad \langle 0, 1 \rangle\{0 \text{ to } 0\} = \langle 1 \rangle, \quad \langle 3, 1, 0, 1 \rangle\{1 \text{ to } 2\} = \langle\rangle$$

## §5.2 Conversion from Bit Vectors to Integers

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

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

The following definition of *nval* can be used:

$$nval(\vec{X}) \quad =_{\text{def}} \quad \sum_{0\leq i<|\vec{X}|} (2^i \cdot \vec{X}\{i\})$$

## §5.3 Tuples and Field Names

We also permit composite values with field names. For example, the pair

$$\langle X\colon 3, Y\colon 4\rangle$$

has one element accessed by the field $X$ and another by accessed by $Y$. A given field name cannot be used twice in a tuple. For an given expression $e$, the value in field $X$ can be referenced to as

$$e.X.$$

Thus, if a variable $A$ equals the tuple above, the value of $A.X + A.Y$ is 7. Arbitrary nesting of such references is permitted.

Sometimes it is desired to let the particular field selected be variable. In that case we use *field names* such as $'X$ and $'Y$ which can be used like numerical subscripts. For example, the expressions $A['X]$ and $A.X$ are equivalent. Thus, if the variable $b$ equals either $'X$ or $'Y$, the expression $A[b]$ equals either $A.X$ or $A.Y$. Note that the expression $A.b$ is *not* equivalent to $A[b]$ but rather $A['b]$. Rather than extend the data domain, We view each field name as representing a distinct numerical constant. Thus, $'X$ might stand for 23. We view a construct such as $'\{A, B\}$ as an abbreviation for the set $\{'A, 'B\}$.

## §5.4 Types for Lists and Tuples

Given two predicates $p$ and $q$, we form the predicate $p \times q$ which is true for any pair whose first element satisfies $p$ and whose second element satisfies $q$. For

example, the formula

$$(nat \times bit)(\langle 3,1\rangle)$$

is true. In general, we write such a test as

$$\langle 3,1\rangle : (nat \times bit)$$

This can be considered an abbreviation for the formula

$$|\langle 3,1\rangle| = 2 \;\wedge\; nat(\langle 3,1\rangle[0]) \;\wedge\; bit(\langle 3,1\rangle[1])$$

The operator $\times$ extends to $n$-element tuples:

$$p_1 \times \cdots \times p_n,$$

where $p_1, \ldots, p_n$ are unary predicates. In addition, the construct $p^n$ is equivalent to $n$ repetitions of $p$. For instance, the test

$$a\colon nat^3$$

is true if $a$ is a triple of natural numbers.

The predicate $struct(X_1{:}p_1, \ldots, X_n{:}p_n)$ checks for tuples whose elements have field names $X_1, \ldots, X_n$ and satisfy the respective types $p_1, \ldots, p_n$. For example, the predicate

$$struct(X\colon nat, Y\colon bit^2)$$

is true for tuples such as

$$\langle X\colon 3, Y\colon \langle 1,0\rangle\rangle.$$

## §5.5 Temporal Conversion

Sometimes a formal parameter of a predicate or function has a sort that is slightly incompatible with that of the corresponding actual parameter. For example, in the formula

$$A\; del^N\; B$$

the signal variable $N$ is in a place requiring a static delay factor. We handle this by *temporally converting* the occurrence of $N$ to a static variable. Thus, the formula just given is considered a syntactic abbreviation for

$$\exists i.\, [(i = N) \land (A\ del^{i}\ B)].$$

In essence, the initial value of $N$ is used as the delay factor. This convention corresponds to the technique of *call-by-value* parameter passing in standard programming languages. The formula

$$A \approx \mathcal{B}$$

expands to

$$\exists C.\, [\boxdot(C = \mathcal{B}) \land (A \approx C)]$$

The occurrence of the interval variable $\mathcal{B}$ is replaced by a signal $C$ that agrees with $\mathcal{B}$ in all terminal subintervals.

CHAPTER 6

# ADDERS

In many computations involving arithmetic operations, it is advantageous to directly reason about numbers. We will now concentrate on addition. To express that the numerical variable $I$ always equals the sum of $J$ and $K$, we write the temporal formula

$$I \approx (J + K)$$

If there is, say, a unit delay, this might be given as the formula

$$(J + K) \; del \; I$$

Even though actual computers possess only finite capacity, it is quite natural to assume an unbounded range of values. When finite precision must be accounted for, modular arithmetic can be used. For example, if it is known that $I$, $J$ and $K$ all range between 0 and $2^n - 1$, then we can represent addition in the manner shown below:

$$I \approx [(J + K) \bmod 2^n]$$

Such descriptive techniques are sufficient for many purposes. However, in specifications of actual digital circuits we must often descend to the level where numbers are implemented by bit vectors. For instance, given that *In1*, *In2* and *Out* are all $n$-bit vectors, the following formula specifies that *Out* always equals the $n$-bit sum of *In1* and *In2*:

$$nval(Out) \approx \big([nval(In1) + nval(In2)] \bmod 2^n\big)$$

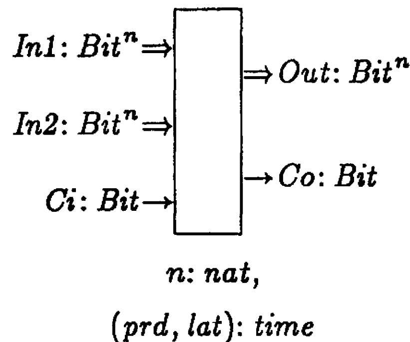
Bit signals for carry-in and carry-out can be included in the manner below:

$$nval(\langle Co \rangle \parallel Out) \approx [nval(In1) + nval(In2) + Ci]$$

The list operator $\parallel$ appends the lists $\langle Co \rangle$ and *Out* together. Since the carry-in *Ci* is a single bit (i.e., 0 or 1), it can be used directly in arithmetic expressions without reference to *nval*.

## §6.1 Basic Adder

Let us now consider an adder specification which includes some timing information regarding propagation delay. The diagram below gives the device's various fields:




*n*: *nat*,

(*prd*, *lat*): *time*

In this and further diagrams, we generally use a single arrow ($\rightarrow$) to indicate a bit input or output and a double arrow ($\Rightarrow$) to indicate a vector signal. The variables at the bottom of the diagram are static and usually determine the device's size or timing coefficients. Here, *prd* stands for the adder's *propagation delay* and *lat* stands for the adder's *latency* or blocking factor. The temporal specification makes this more precise.

### Formal specification of addition circuit

The predicate *BasicAdder* formally characterizes the circuit's desired structure and behavior. The device's various inputs, outputs and timing coefficients are represented as fields of the single parameter *A*. For example, the expression *A.Ci* equals the carry input. The predicate's definition makes reference to other predicates given

later.

$$
\begin{aligned}
&BasicAdder(A) \quad \equiv_{\text{def}} \\
&\quad BasicAdderStructure(A) \\
&\quad \wedge \boxdot\, Add(A)
\end{aligned}
$$

The predicate *BasicAdderStructure* presents $A$'s fields. The predicate *Add* gives the control sequencing required to perform an addition. The operator $\boxdot$ indicates that *Add* must be true in all subintervals.

*Definition of BasicAdderStructure:*

The definition below of *BasicAdderStructure* contains information on the physical structure of the adder. Fields starting in upper case represent signals while lower-case ones are static. Constructs such as "%Inputs" are comments included to classify the various circuit fields. For example, *A.In1* is an input bit vector. The input bit vectors *In1* and *In2* are of length $n$ as is the output vector *Out* which yields the sum. The input bit *Ci* determines the carry input and *Co* receives the carry output. The values *lat* and *prd* are the latency and propagation times.

$$
\begin{aligned}
&BasicAdderStructure(A) \quad \equiv_{\text{def}} \\
&\quad A\colon struct[ \\
&\qquad (In1, In2)\colon Bit^n, && \text{\%Inputs} \\
&\qquad Ci\colon Bit \\
&\qquad Out\colon Bit^n, && \text{\%Outputs} \\
&\qquad Co\colon Bit \\
&\qquad n\colon nat, (prd, lat)\colon time && \text{\%Parameters} \\
&\quad ]
\end{aligned}
$$

For brevity, the prefix "*A.*" is omitted when a field is referenced below.

*Definition of Add:*

After the inputs *In1*, *In2* and *Ci* are held stable long enough, the combined numerical value of the outputs *Out* and *Co* equals the inputs' numerical sum. In

addition, there is a certain amount of latency. Recall that the function *nval* converts a bit sequence to the corresponding numerical value.

$$\begin{aligned} Add(A) \quad & \equiv_{\mathrm{def}} \\ & (stb\langle In1, In2, Ci\rangle \wedge len \geq prd) \\ & \quad \rightsquigarrow [nval(\langle Co\rangle \,\|\, Out) = (nval(In1) + nval(In2) + Ci) \\ & \qquad \wedge \langle In1, In2, Ci\rangle\, blk^{lat}\, \langle Co, Out\rangle] \end{aligned}$$

It is possible to modify the predicate *BasicAdder* to handle other combinational logic elements with similar timing characteristics.

## Combining two adders

Two such adders can be used to build a bigger one by appending the corresponding vector inputs and outputs and using the carry-out of one adder as the carry-in of the other. The following property formally expresses this:

$$\models \quad [BasicAdder(A) \wedge BasicAdder(B) \wedge (A.Ci \approx B.Co)] \supset BasicAdder(C)$$

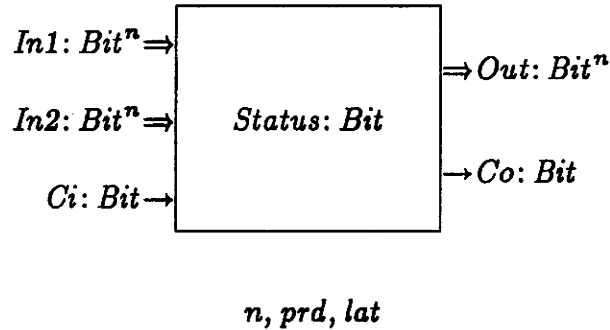
where the tuple $C$ has exactly the following fields and connections to $A$ and $B$:

$$\begin{aligned} C.In1 &\approx A.In1 \,\|\, B.In1 \\ C.In2 &\approx A.In2 \,\|\, B.In2 \\ C.Ci &\approx B.Ci \\ C.Out &\approx A.Out \,\|\, B.Out \\ C.Co &\approx A.Co \\ C.n &= A.n + B.n \\ C.lat &= \min(A.lat, B.lat) \\ C.prd &= A.prd + B.prd \end{aligned}$$

Here $A$ contains the most significant bits and $B$ contains the least significant ones. The operator $\|$ appends two lists together.

## §6.2 Adder with Internal Status Bit

An adder of length $n$ can be defined to include an internal status bit in the manner of the delay model *stdel*. Here is the device structure:




$n$, *prd*, *lat*

The specification given below is externally equivalent to *BasicAdder*.

*Definition of StatusAdder:*

$$\begin{aligned}&\mathit{StatusAdder}(A) \quad \equiv_{\mathbf{def}}\\ &\quad \mathit{StatusAdderStructure}(A)\\ &\quad \wedge\ \boxdot\ \mathit{Add}(A)\\ &\quad \wedge\ \boxdot\ \mathit{Steady}(A)\end{aligned}$$

*Definition of StatusAdderStructure:*

$$\begin{aligned}&\mathit{StatusAdderStructure}(A) \quad \equiv_{\mathbf{def}}\\ &\quad A\colon \mathit{struct}[\\ &\qquad (\mathit{In1}, \mathit{In2})\colon \mathit{Bit}^n,\ \mathit{Ci}\colon \mathit{Bit} && \%\text{Inputs}\\ &\qquad \mathit{Out}\colon \mathit{Bit}^n,\ \mathit{Co}\colon \mathit{Bit} && \%\text{Outputs}\\ &\qquad \mathit{Status}\colon \mathit{Bit} && \%\text{Internal}\\ &\qquad n\colon \mathit{nat}, (\mathit{lat}, \mathit{prd})\colon \mathit{time} && \%\text{Parameters}\\ &\quad ]\end{aligned}$$

*Definition of Add:*

After the inputs remain stable long enough, their sum is propagated to the outputs and the status bit equals 1.

$$
\begin{aligned}
&Add(A) \quad \equiv_{\text{def}} \\
&\quad (stb\langle In1, In2, Ci\rangle \land len \geq prd) \\
&\qquad \supset fin\big([Status = 1] \\
&\qquad\qquad \land \big[nval(\langle Co\rangle \parallel Out) = nval(In1) + nval(In2) + Ci\big]\big)
\end{aligned}
$$

*Definition of Steady:*

Whenever the signal *Status* is 1, there is a certain amount of blocking from the inputs to it and the outputs.

$$
\begin{aligned}
&Steady(A) \quad \equiv_{\text{def}} \\
&\quad beg(Status = 1) \\
&\qquad \supset [\langle In1, In2, Ci\rangle\ blk\ Status \land \langle In1, In2, Ci\rangle\ blk^{lat}\ \langle Out, Co\rangle]
\end{aligned}
$$

## §6.3 Adder with More Detailed Timing Information

Further timing details can be accomodated as we now demonstrate. Suppose each input has its own propagation time. This can be specified as follows:

*Definition of DetailedAdder:*

$$
\begin{aligned}
&DetailedAdder(A) \quad \equiv_{\text{def}} \\
&\quad DetailedAdderStructure(A) \\
&\quad \land \boxdot\, Add(A)
\end{aligned}
$$

*Definition of DetailedAdderStructure:*

In this adder, there is a separate parameter for each input's propagation time.

$$
\begin{array}{ll}
DetailedAdderStructure(A) & \equiv_{\mathbf{def}} \\
\quad A\colon struct[ & \\
\qquad (In1, In2)\colon Bit^n,\ Ci\colon Bit & \text{\%Inputs} \\
\qquad Out\colon Bit^n,\ Co\colon Bit & \text{\%Outputs} \\
\qquad n\colon nat, & \text{\%Parameters} \\
\qquad prd\colon (In1, In2, Ci)\colon time, & \\
\qquad lat\colon time & \\
\quad ] &
\end{array}
$$

We use the construct

$$prd\colon (In1, In2, Ci)\colon time$$

to indicate that *prd* has three subfields accessible as *prd.In1*, *prd.In2* and *prd.Ci*.

*Definition of Add:*

Here each input has its own time for stabilizing.

$$
\begin{array}{l}
Add(A) \quad \equiv_{\mathbf{def}} \\
\quad (tstb^{prd.In1}\, In1 \ \wedge\ tstb^{prd.In2}\, In2 \ \wedge\ tstb^{prd.Ci}\, Ci) \\
\qquad \rightsquigarrow [nval(\langle Co\rangle \,\|\, Out) = (nval(In1) + nval(In2) + Ci) \\
\qquad\qquad \wedge\ \langle In1, In2, Ci\rangle\ blk^{lat}\ \langle Co, Out\rangle]
\end{array}
$$

The sampling requirements can also be given in a less redundant form:

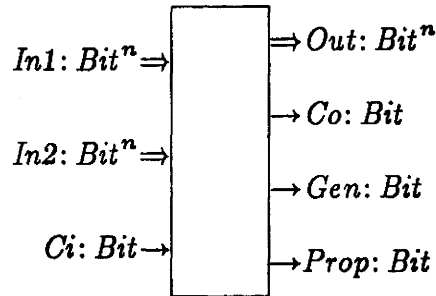
$$\forall field \in {}'\{In1, In2, Ci\}.\ (tstb^{prd[field]}\, A[field])$$

Recall that $'\{In1, In2, Ci\}$ represents the set

$$\{'In1, 'In2, 'Ci\}.$$

## §6.4 Adder with Carry Look-Ahead Outputs

Long adders usually have extra control signals to speed up the propagation of carry bits. One technique is called *carry look-ahead* (see [17]) and produces sum and carry outputs as well as two bit signals *Gen* and *Prop*. The structure is as follows:




$n, prd, lat$

The bit signal *Gen* is 1 iff the result of adding *In1* and *In2* will *generate* 1 as carry no matter what the carry input *Ci* is. The bit signal *Prop* is 1 iff the carry input *Ci* will be *propagated* unchanged to the carry output *Co*. Because both *Gen* and *Prop* can be computed without the carry input, they need not wait for carry rippling.

*Definition of CarryLookAheadAdder:*

$$\begin{aligned} & CarryLookAheadAdder(A) \quad \equiv_{\mathbf{def}} \\ & \quad CLAAdderStructure(A) \\ & \quad \wedge \boxdot\, Add(A, output), \quad \text{for } output \in {'}\{Out, Co, Gen, Prop\} \end{aligned}$$

The last line is equivalent to

$$\boxdot\, Add(A, {'}Out) \wedge \boxdot\, Add(A, {'}Co) \wedge \boxdot\, Add(A, {'}Gen) \wedge \boxdot\, Add(A, {'}Prop)$$

*Definition of CLAAdderStructure:*

$$
\begin{aligned}
&\mathit{CLAAdderStructure}(A) \quad \equiv_{\text{def}} \\
&\quad A\colon \mathit{struct}\Big[ \\
&\qquad (\mathit{In1}, \mathit{In2})\colon \mathit{Bit}^n,\ \mathit{Ci}\colon \mathit{Bit} && \text{\%Inputs} \\
&\qquad \mathit{Out}\colon \mathit{Bit}^n, (\mathit{Co}, \mathit{Gen}, \mathit{Prop})\colon \mathit{Bit} && \text{\%Outputs} \\
&\qquad n\colon \mathit{nat}, && \text{\%Parameters} \\
&\qquad \mathit{prd}\colon (\mathit{Out}, \mathit{Co}, \mathit{Gen}, \mathit{Prop})\colon \mathit{time}, \\
&\qquad \mathit{lat}\colon (\mathit{Out}, \mathit{Co}, \mathit{Gen}, \mathit{Prop})\colon \mathit{time} \\
&\quad \Big]
\end{aligned}
$$

The specification gives various propagation and latency times by making *prd* and *lat* each have a subfield for every output.

The function *inputs* shows the inputs used by each output:

| *output* | *inputs(A, output)* |
|---|---|
| *Out* | $\langle Ci, In1, In2\rangle$ |
| *Co* | $\langle Ci, In1, In2\rangle$ |
| *Gen* | $\langle In1, In2\rangle$ |
| *Prop* | $\langle In1, In2\rangle$ |

As noted earlier, the generate and propagate signals can be computed without reference to the carry input.

*Definition of Add:*

For any selected output, after the appropriate input fields remain stable long enough, the device satisfies the predicate *result* and the output depends on its associated inputs.

$$
\begin{aligned}
&\mathit{Add}(A, \mathit{output}) \quad \equiv_{\text{def}} \\
&\quad \big(\mathit{stb}\ \mathit{inputs}(A, \mathit{output}) \ \wedge\ \mathit{len} \geq \mathit{prd}[\mathit{output}]\big) \\
&\qquad \rightsquigarrow \big[\mathit{result}(A, \mathit{output}) \ \wedge\ \mathit{inputs}(A, \mathit{output})\ \mathit{blk}^i\ A[\mathit{output}]\big]
\end{aligned}
$$

where $i = \mathit{lat}[\mathit{output}]$ and the predicate *result* has the following definition:

| *output* | $result(A, output)$ |
|---|---|
| *Out* | $nval(Out) = (nval(In1) + nval(In2) + Ci) \bmod 2^n$ |
| *Co* | $Co = carry(n, nval(In1), nval(In2), Ci)$ |
| *Gen* | $Gen = carrygen(n, nval(In1), nval(In2))$ |
| *Prop* | $Prop = carryprop(n, nval(In1), nval(In2))$ |

The functions *carry*, *carrygen* and *carryprop* compute appropriate values:

$$carry(n, j, k, ci) \quad =_{\text{def}} \quad (j + k + ci) \div 2^n$$

$$carrygen(n, j, k) \quad =_{\text{def}} \quad \mathit{if}\ (\forall ci \in \{0, 1\}.\ carry(n, j, k, ci) = 1)\ \mathit{then}\ 1\ \mathit{else}\ 0$$

$$carryprop(n, j, k) \quad =_{\text{def}} \quad \mathit{if}\ (\forall ci \in \{0, 1\}.\ carry(n, j, k, ci) = ci)\ \mathit{then}\ 1\ \mathit{else}\ 0$$

Both *carrygen* and *carryprop* can be simplified:

$$carrygen(n, j, k) = \mathit{if}\ (j + k \geq 2^n)\ \mathit{then}\ 1\ \mathit{else}\ 0$$

$$carryprop(n, j, k) = \mathit{if}\ (j + k = 2^n - 1)\ \mathit{then}\ 1\ \mathit{else}\ 0$$

Thus, a carry is generated exactly when the sum of the two numbers $j$ and $k$ exceeds the capacity of $n$ bits. Similarly, the incoming carry is propagated if the sum of $j$ and $k$ is the "borderline" value $2^n - 1$. In practice, a carry look-ahead adder may output *Gen* and *Prop* in complemented form as the signals $\overline{Gen}$ and $\overline{Prop}$.

If we ignore propagation delay, the adder has the following behavior:

$$\forall output \in {}'\{Out, Co, Gen, Prop\}.\ [\boxdot\ result(A, output)]$$

CHAPTER 7

# LATCHES

A latch is a simple memory element for storing and maintaining a single bit of data. The two inputs $S$ and $R$ determine what value is stored with $S$ standing for *Set* and $R$ standing for *Reset*. When the latch is steady, the outputs $Q$ and $\overline{Q}$ are complements. Note that the bar in "$\overline{Q}$" is part of the name and not an operator. Such elements are among the simplest storage devices that can be constructed out of TTL gates and provide a basis for building counters and other sequential components.

## §7.1 Simple Latch

Here is one possible latch specification:

$$
\begin{aligned}
&(S,R)\ \mathit{latch}^{m,n}\ (Q,\overline{Q}) \quad \equiv_{\mathrm{def}} \\
&\quad \boxdot\big[(S \approx 0 \ \land\ R \approx 1 \ \land\ \mathit{len} \geq m) \\
&\qquad\qquad \leadsto (\mathit{beg}[Q = 0 \ \land\ \overline{Q} = 1] \ \land\ S\ \mathit{blk}^n\ \langle Q, \overline{Q}\rangle)\big] \\
&\ \land\ \boxdot\big[(S \approx 1 \ \land\ R \approx 0 \ \land\ \mathit{len} \geq m) \\
&\qquad\qquad \leadsto (\mathit{beg}[Q = 1 \ \land\ \overline{Q} = 0] \ \land\ R\ \mathit{blk}^n\ \langle Q, \overline{Q}\rangle)\big]
\end{aligned}
$$

For example, the specification states that after $S$ is 1 and $R$ is 0 for at least $m$ units of time, $Q$ equals 1, $\overline{Q}$ equals 0 and $R$ blocks both with factor $n$. That is, the outputs are stable as long as $R$ remains "inactive" at 0, independent of $S$'s behavior.

Such a latch can be constructed out of two nor-gates that feed back to one another:

$$\models \quad \left[\neg(R \lor \overline{Q})\ sadel^{m,n}\ Q \ \land\ \neg(S \lor Q)\ sadel^{m,n}\ \overline{Q} \ \land\ n \geq 1\right] \supset \left[(S,R)\ latch^{2m,n}\ (Q,\overline{Q})\right]$$

For example, to set $Q$ to 1 and $\overline{Q}$ to 0, we keep $R$ at 0 and $S$ at 1. After $m$ units of time, $\overline{Q}$ equals 0 and after $2m$ units of time, $Q$ equals 1. At this point both $Q$ and $\overline{Q}$ are stable as long as $R$ remains equal to 0. The gates' blocking factor $n$ must be nonzero in order to achieve a feedback loop that maintains the values of $Q$ and $\overline{Q}$.

## §7.2 Conventional SR-Latch

The latch specification now given has separate parts for entering and maintaining a value in the device. The following sort of table is often given to describe operation for various input values:

| $S$ | $R$ | $Q$ | $\overline{Q}$ |
|---|---|---|---|
| 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 0 | unchanged | |
| 1 | 1 | unspecified | |

For example, assuming unit delay, the behavior of $Q$ can be expressed by the formula

$$\boxdot\left[skip \supset \left([beg(S = \neg R) \supset (S \rightarrow Q)] \ \land\ [beg(S = 0 \land R = 0) \supset stb\, Q]\right)\right]$$

The following predicate *SRLatch* goes into more details on timing.

*Definition of SRLatchStructure:*

The latch includes the internal bit flag *Status*:

$SRLatchStructure(L) \quad \equiv_{\text{def}}$

$L$: *struct*[

| | |
|---|---|
| $(S, R)$: *Bit* | %Inputs |
| $(Q, \overline{Q})$: *Bit* | %Outputs |
| *Status*: *Bit* | %Internal |
| $(prd, lat)$: *time* | %Parameters |

]

We use *Status* to indicate when the device is steady.

*Definition of SRLatch:*

The latch can be set to 1, cleared to 0, disabled or kept steady.

$$
\begin{aligned}
&SRLatch(L) \quad \equiv_{\mathrm{def}} \\
&\quad SRLatchStructure(L) \\
&\quad \wedge \boxdot\, Store(L, i), \quad \text{for } i \in \{0,1\} \\
&\quad \wedge \boxdot\, Disable(L) \\
&\quad \wedge \boxdot\, Steady(L)
\end{aligned}
$$

The formula

$$\boxdot\, Store(L, i), \quad \text{for } i \in \{0,1\}$$

is equivalent to

$$\boxdot\, Store(L, 0) \wedge \boxdot\, Store(L, 1)$$

*Definition of Store:*

This definition uses the static variable $i$ to determine the value to be stored:

$$
\begin{aligned}
&Store(L, i) \quad \equiv_{\mathrm{def}} \\
&\quad \big[(S \approx i) \wedge (R \approx \neg i) \wedge (len \geq prd)\big] \\
&\qquad\qquad \supset \ fin\big[(Status = 1) \wedge (Q = i)\big]
\end{aligned}
$$

Alternatively we can omit $i$ by using a formula such as

$$\big[stb\langle S, R\rangle \wedge beg(S = \neg R) \wedge (len \geq prd)\big] \supset fin\big[(Status = 1) \wedge (Q = S)\big]$$

This works because $S$ and $R$ must be complements when setting or resetting and $S$ matches the value stored in $Q$.

*Definition of Disable:*

If the device is initially steady and the two inputs $S$ and $R$ smoothly become 0 for a period of sufficient length, the device remains steady and the outputs are

stable.

$$Disable(L) \quad \equiv_{\text{def}}$$
$$\left[beg(Status = 1) \ \wedge \ sm^{0,prd}\langle S, R\rangle \ \wedge \ fin(S = 0 \ \wedge \ R = 0)\right]$$
$$\supset \ \left[fin(Status = 1) \ \wedge \ stb\langle Q, \overline{Q}\rangle\right]$$

*Definition of Steady:*

When the flag *Status* equals 1, the outputs $Q$ and $\overline{Q}$ are complements. In addition, the flag and outputs depend on the two inputs $S$ and $R$.

$$Steady(L) \quad \equiv_{\text{def}}$$
$$beg(Status = 1)$$
$$\supset [beg(\overline{Q} = \neg Q) \ \wedge \ \langle S, R\rangle \ blk \ Status \ \wedge \ \langle S, R\rangle \ blk^{lat} \ \langle Q, \overline{Q}\rangle]$$

## Constructing an SR-latch

The next property shows how the first latch described implements a conventional SR-latch:

$$\models \quad [(S, R) \ latch^{m,n} \ (Q, \overline{Q})] \ \supset \ SRLatch(L)$$

where the tuple $L$ has exactly the following fields and connections:

$$L.S \approx S$$
$$L.R \approx R$$
$$L.Q \approx Q$$
$$L.\overline{Q} \approx \overline{Q}$$
$$L.prd = m$$
$$L.lat = n$$

and $L.Status$ is constructed as follows:

$$L.Status \approx$$
$$if \ \left(\exists i \in \{0,1\}. \left[Q = i \ \wedge \ \overline{Q} = \neg i \ \wedge \ \langle S, R\rangle[i] = 0 \ \wedge \ (\langle S, R\rangle[i]) \ blk^{n} \ \langle Q, \overline{Q}\rangle\right]\right) \ then \ 1 \ else \ 0$$

At all times, $L.Status$ is set to 1 if $Q$ and $\overline{Q}$ have complementary values and are blocked by $S$ if $Q = 0$ and by $R$ if $Q = 1$. The quantified variable $i$ is used to determine the values of $Q$ and $\overline{Q}$.

## §7.3 Smooth SR-Latch

The predicate *Store* in the specification *SRLatch* can be modified to include additional details regarding smooth transitions. As before, *Store* shows how to enter 0 or 1 into the latch. In addition, if the status bit is initially 1 and the inputs $S$ and $R$ are smooth, the outputs are also smooth. Notice that there is no requirement that $Q$ and $\overline{Q}$ change at exactly the same time.

$$
\begin{aligned}
&Store(L, i) \quad \equiv_{\text{def}} \\
&\quad \left[tstb^{prd}\langle S, R\rangle \ \wedge\ fin[(S = i)\ \wedge\ (R = \neg i)]\right] \\
&\qquad \supset \left[fin(Status = 1\ \wedge\ Q = i)\right. \\
&\qquad\qquad \left.\wedge\ ([sm\langle S, R\rangle\ \wedge\ beg(Status = 1)]\ \supset\ sm\langle Q, \overline{Q}\rangle)\right]
\end{aligned}
$$

## §7.4 D-Latch

A simple D-latch has one input pin to selectively enable the latch to accept data and another to indicate the actual value to be stored. The operation corresponds roughly to the following table, where $E$ and $D$ are the enable and data inputs, and $Q$ and $\overline{Q}$ are the outputs:

| $E$ | $D$ | $Q$ | $\overline{Q}$ |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |
| 0 | – | unchanged | |

When $E$ is held active at 1, $D$'s value is propagated through the device as through a delay element. When $E$ is 0, the device maintains whatever value is stored, independent of $D$. The formula below uses unit-delay to describe this:

$$(if\ [E = 1]\ then\ \langle D, \neg D\rangle\ else\ \langle Q, \overline{Q}\rangle)\ del\ \langle Q, \overline{Q}\rangle$$

If we just look at the behavior of $Q$, this reduces to

$$(if\ [E = 1]\ then\ D\ else\ Q)\ del\ Q$$

The D-latch is also referred to as a *transparent latch* because when $E$ is enabled, the input data passes through to the output.

*Definition of DLatch:*

As with the SR-latch, the specification has predicates for examining, modifying and disabling the device:

$$
\begin{aligned}
&\mathit{DLatch}(L) \quad \equiv_{\text{def}} \\
&\quad \mathit{DLatchStructure}(L) \\
&\quad \wedge\ \boxdot\ \mathit{Store}(L) \\
&\quad \wedge\ \boxdot\ \mathit{Disable}(L) \\
&\quad \wedge\ \boxdot\ \mathit{Steady}(L)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{DLatchStructure}(L) \quad \equiv_{\text{def}} \\
&\quad L\text{: } \mathit{struct}\Big[ \\
&\qquad (E, D)\text{: } \mathit{Bit} && \text{\%Inputs} \\
&\qquad (Q, \overline{Q})\text{: } \mathit{Bit} && \text{\%Outputs} \\
&\qquad \mathit{Status}\text{: } \mathit{Bit} && \text{\%Internal} \\
&\qquad (\mathit{prd}, \mathit{lat})\text{: } \mathit{time} && \text{\%Parameters} \\
&\quad \Big]
\end{aligned}
$$

*Definition of Store:*

When the latch is enabled, the data signal $D$'s value propagates to the output $Q$.

$$
\begin{aligned}
&\mathit{Store}(L) \quad \equiv_{\text{def}} \\
&\quad \big[(E \approx 1)\ \wedge\ \mathit{stb}\, D\ \wedge\ (\mathit{len} \geq \mathit{prd})\big]\ \supset\ \mathit{fin}\big[(\mathit{Status} = 1)\ \wedge\ (Q = D)\big]
\end{aligned}
$$

*Definition of Disable:*

If the enable signal drops to 0 and the data remains stable, the latch becomes disabled and retains the value it was set to.

$$
\begin{aligned}
&\mathit{Disable}(L) \quad \equiv_{\text{def}} \\
&\quad \big[\downarrow^{0,\mathit{prd}} E\ \wedge\ \mathit{stb}\, D\ \wedge\ \mathit{beg}(\mathit{Status} = 1)\big] \\
&\qquad\qquad \supset\ \big[\mathit{fin}(\mathit{Status} = 1)\ \wedge\ \mathit{stb}\langle Q, \overline{Q}\rangle\big]
\end{aligned}
$$

*Definition of Steady:*

Whenever the signal *Status* equals 1, the outputs $Q$ and $\overline{Q}$ are complements of each other. If $E$ is disabled, it blocks the status flag and outputs. When $E$ is enabled, the flag and outputs are blocked by $E$ and the incoming data signal $D$.

$$Steady(L) \quad \equiv_{\text{def}}$$

$$beg(Status = 1)$$

$$\supset \left[beg(\overline{Q} = \neg Q) \ \wedge\ V\ blk\ Status\ \wedge\ V\ blk^{lat}\ \langle Q, \overline{Q}\rangle\right]$$

where $V$ is a function of the enable signal's initial value:

| $E$ | $V$ |
|---|---|
| 0 | $\langle E\rangle$ |
| 1 | $\langle E, D\rangle$ |

**Building a D-latch**

A D-latch can be implemented by connecting a suitable combinational interface to the inputs of an SR-latch. The interface has inputs $E$ and $D$ and outputs $S$ and $R$ with stable-state behavior given by the following table:

| $E$ | $D$ | $S$ | $R$ |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |
| 0 | – | 0 | 0 |

When the interface is enabled with $E$ at 1, the data signal $D$ controls $S$ and $R$ for clearing or setting. If $E$ is 0, both $S$ and $R$ are deactivated. The interface has the following description:

*Definition of DLInterface:*

$$DLInterface(A) \quad \equiv_{\text{def}}$$

$$DLInterfaceStructure(A)$$

$$\wedge\ \boxdot\ Store(A)$$

$$\wedge\ \boxdot\ Disable(A)$$

$$\wedge\ \boxdot\ Steady(A)$$

*Definition of DLInterfaceStructure:*

$$DLInterfaceStructure(A) \quad \equiv_{\mathbf{def}}$$

$A$: *struct*[

| | |
|---|---|
| $(E, D)$: *Bit* | %Inputs |
| $(S, R)$: *Bit* | %Outputs |
| *Status*: *Bit* | %Internal |
| $(prd, lat)$: *time* | %Parameters |

]

*Definition of Store:*

When the device is enabled, the outputs eventually reflect $D$ and its complement. This is done so that any connected SR-latch will be actively set to $D$'s value.

$$Store(A) \quad \equiv_{\mathbf{def}}$$

$$[E \approx 1 \ \wedge \ stb\, D \ \wedge \ (len \geq prd)] \supset fin[(Status = 1) \ \wedge \ (S = D) \ \wedge \ (R = \neg D)]$$

*Definition of Disable:*

When the interface is disabled, both outputs smoothly change to 0 so that any connected SR-latch retains its value.

$$Disable(A) \quad \equiv_{\mathbf{def}}$$

$$[\downarrow^{\mathbf{0}, prd} E \ \wedge \ stb\, D \ \wedge \ beg(Status = 1)] \supset [fin(Status = 1 \ \wedge \ S = \mathbf{0} \ \wedge \ R = \mathbf{0}) \ \wedge \ sm\langle S, R\rangle]$$

*Definition of Steady:*

When the device is steady, the status bit and outputs are blocked by the appropriate inputs:

$$Steady(A) \quad \equiv_{\mathbf{def}}$$

$$beg(Status = 1) \supset (V\ blk\ Status \ \wedge \ V\ blk^{lat}\, \langle S, R\rangle)$$

where $V$ is based on the initial value of $E$:

| $E$ | $V$ |
|---|---|
| 0 | $\langle E \rangle$ |
| 1 | $\langle E, D \rangle$ |

### Combining the interface with SR-latch

The following predicate shows how to connect the interface's outputs to the inputs of an SR-latch:

$$
\begin{aligned}
&\mathit{DLatchImplementation}(A, L) \quad \equiv_{\mathsf{def}} \\
&\quad \mathit{DLInterface}(A) \wedge \mathit{SRLatch}(L) \\
&\qquad \wedge (A.S \approx L.S) \wedge (A.R \approx L.R)
\end{aligned}
$$

The next property states that this implementation results in a D-latch:

$$\models \mathit{DLatchImplementation}(A, L) \supset \mathit{DLatch}(M)$$

where

$$
\begin{aligned}
M.E &\approx A.E \\
M.D &\approx A.D \\
M.Q &\approx L.Q \\
M.\overline{Q} &\approx L.\overline{Q} \\
M.\mathit{Status} &\approx A.\mathit{Status} \wedge L.\mathit{Status} \\
M.\mathit{lat} &= A.\mathit{lat} + L.\mathit{lat} \\
M.\mathit{prd} &= A.\mathit{prd} + L.\mathit{prd}
\end{aligned}
$$

The interface itself can be built from combinational gates based on the steady-state formula

$$S = (E \wedge D) \quad \wedge \quad R = (E \wedge \neg D).$$

We omit the details.

### Introducing a hold time

In practice, a D-latch's data input need not be held stable during the entire period when the D-latch is disabled and the enable signal drops. This can be formalized by adding a *hold-time* parameter *hld* and redefining *Disable* to incorporate it:

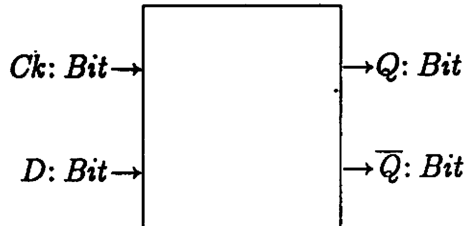
$$
\begin{aligned}
&\mathit{Disable}(L) \quad \equiv_{\mathrm{def}} \\
&\quad [\downarrow^{0,prd} E \ \wedge \ E \ \mathit{blk}^{hld} \ D \ \wedge \ \mathit{beg}(\mathit{Status} = 1)] \\
&\qquad\qquad \supset \ [\mathit{fin}(\mathit{Status} = 1) \ \wedge \ \mathit{stb}\langle Q, \overline{Q}\rangle]
\end{aligned}
$$

CHAPTER 8

# FLIP-FLOPS

## §8.1 Simple D-Flip-Flop

The simple D-flip-flop described here has as inputs a clock and a data signal. The overall structure is given by the following diagram:




$(c1, c2, c3, hld, lat)$: *time*

If we ignore the clock input *Ck* and assume unit delay, the flip-flop behavior can be described by the formula

$$[D \ del \ Q] \ \wedge \ [(\neg D) \ del \ \overline{Q}]$$

The predicate *SimpleDFlipFlop* given below takes a more detailed look at clocking and propagation.

*Definition of SimpleDFlipFlop:*

$$\begin{aligned} &\mathit{SimpleDFlipFlop}(F) \quad \equiv_{\mathrm{def}} \\ &\quad \mathit{SimpleDFFStructure}(F) \\ &\quad \wedge \boxdot\, \mathit{Store}(F, i), \quad \text{for } i \in \{0, 1\} \end{aligned}$$

*Definition of SimpleDFFStructure:*

$$\begin{aligned} &\mathit{SimpleDFFStructure}(F) \quad \equiv_{\mathrm{def}} \\ &\quad F\colon \mathit{struct}\Big[ \\ &\qquad (Ck, D)\colon \mathit{Bit} && \text{\%Inputs} \\ &\qquad (Q, \overline{Q})\colon \mathit{Bit} && \text{\%Outputs} \\ &\qquad (c1, c2, c3, \mathit{hld}, \mathit{lat})\colon \mathit{time} && \text{\%Parameters} \\ &\quad \Big] \end{aligned}$$

*Definition of Store:*

The predicate *Store* shows how to store a value in the flip-flop:

$$\begin{aligned} &\mathit{Store}(F, i) \quad \equiv_{\mathrm{def}} \\ &\quad [\uparrow\downarrow^{c1,c2,c3} Ck \;\wedge\; Ck\ \mathit{blk}^{\mathit{hld}}\ D \;\wedge\; \mathit{beg}(D = i)] \\ &\qquad \rightsquigarrow [\mathit{beg}(Q = i \;\wedge\; \overline{Q} = \neg i) \;\wedge\; Ck\ \mathit{blk}^{\mathit{lat}}\ \langle Q, \overline{Q}\rangle] \end{aligned}$$

The flip-flop specification can be generalized into a multi-bit register by representing the input data and the output as vectors of the appropriate length. If still more detail is desired, such a register can be viewed as a collection of one-bit flip-flops, each with its own status bit. Incidentally, it is easy to connect, say, the output of one device to the clock input of another. Here is an example:

$$\mathit{SimpleFlipFlop}(F) \;\wedge\; \mathit{SimpleFlipFlop}(G) \;\wedge\; (F.Q \approx G.Ck)$$

## §8.2 A Flip-Flop with More Timing Information

The predicate *DFlipFlop* presented below includes additional timing details. When the clock signal rises, the current value of the data line is stored in the device. Falling clock edges leave the stored value unchanged. This description also takes a more precise look at the process of setting up the input data prior to triggering. When the internal flag *Status* equals 1, as long as the clock is stable, the output bit $Q$ remains stable and is also available in complemented form as $\overline{Q}$.

*Definition of DFlipFlop:*

Here is the main predicate:

$$
\begin{aligned}
&DFlipFlop(F) \quad \equiv_{\mathrm{def}} \\
&\quad DFlipFlopStructure(F) \\
&\quad \land \boxdot\, Store(F) \\
&\quad \land \boxdot\, Nontrig(F) \\
&\quad \land \boxdot\, Steady(F)
\end{aligned}
$$

*Definition of DFlipFlopStructure:*

$$
\begin{aligned}
&DFlipFlopStructure(F) \quad \equiv_{\mathrm{def}} \\
&\quad F\colon struct\Big[ \\
&\qquad (Ck, D)\colon Bit && \%\text{Inputs} \\
&\qquad (Q, \overline{Q})\colon Bit && \%\text{Outputs} \\
&\qquad Status\colon Bit && \%\text{Internal} \\
&\qquad (stp, prd, hld, lat)\colon time && \%\text{Parameters} \\
&\quad \Big]
\end{aligned}
$$

*Definition of Store:*

The predicate *Store* shows how the clock *triggers* the flip-flop to accept a new value. The data must not change until after the clock goes high. Before the actual

triggering, the clock and data are *set up* by being initially stable for at least *stp* units of time. The actual clocking is given by the predicate *Trigger*.

$$\begin{aligned} &Store(F) \quad \equiv_{\text{def}} \\ &\qquad (stb\langle Ck, D\rangle \ \wedge \ [len \geq stp]) \leadsto Trigger(F) \end{aligned}$$

If desired, we can have separate set-up times for the clock and data inputs. For example, the value *stp.Ck* can give the time required to set up the clock. The following formula demonstrates one way to do this:

$$(tstb^{stp.Ck}\, Ck \ \wedge \ tstb^{stp.D}\, D) \leadsto Trigger(F)$$

Incidentally, an externally equivalent D-flip-flop specification can be given that includes an additional internal field *SetupStatus* equaling 1 whenever the inputs have been set up.

*Definition of Trigger:*

After the clock rises and triggers the device, the data input $D$ must remain stable for at least the *hold time* specified by the parameter *hld*. If this condition is fulfilled, the device ends up steady with *Status* equaling 1 and $Q$ receiving $D$'s initial value.

$$\begin{aligned} &Trigger(F) \quad \equiv_{\text{def}} \\ &\qquad (\uparrow^{0,prd} Ck \ \wedge \ Ck\ blk^{hld}\, D) \supset [fin(Status = 1) \ \wedge \ (D \rightarrow Q)] \end{aligned}$$

*Definition of Nontrig:*

If the clock has a falling or *non-triggering* edge and the device is initially steady then the device remains steady and outputs are stable.

$$\begin{aligned} &Nontrig(F) \quad \equiv_{\text{def}} \\ &\qquad [\downarrow^{0,prd} Ck \ \wedge \ beg(Status = 1)] \\ &\qquad\qquad \supset [fin(Status = 1) \ \wedge \ stb(Q, \overline{Q})] \end{aligned}$$

*Definition of Steady:*

Whenever the status bit equals 1, it and the outputs remain stable as long as the clock does, independent of the behavior of the data input. The outputs are complements.

$$\begin{aligned} &Steady(F) \quad \equiv_{\mathrm{def}} \\ &\quad beg(Status = 1) \\ &\qquad \supset \left[beg(\overline{Q} = \neg Q) \wedge Ck\ blk\ Status \wedge Ck\ blk^{lat} \langle Q, \overline{Q} \rangle\right] \end{aligned}$$

If desired, the latency factor can be a function of the initial value of the clock or even the currently stored value.

## Comparison of the predicates *SimpleDFlipFlop* and *DFlipFlop*

The next property shows how to reduce the predicate *DFlipFlop* to the predicate *SimpleDFlipFlop* presented earlier:

$$\models \quad DFlipFlop(F) \supset SimpleDFlipflop(G)$$

where $G$ is constructed from $F$ as follows:

$$\begin{aligned} G[field] &\approx F[field], \quad \text{for } field \in {}'\{Ck, D, Q, \overline{Q}\} \\ G.c1 &= F.stp \\ G.c2 &= F.prd \\ G.c3 &= F.prd \\ G.hld &= F.hld \\ G.lat &= F.lat \end{aligned}$$

## Simplifying the predicate *Store* in *DFlipFlop*

By merging the processes for setting up and triggering the flip-flop, we can eliminate the predicate *Trigger* and define *Store* as follows:

$$(\uparrow^{stp,prd} Ck \wedge Ck\ blk^{hld}\ D) \supset [fin(Status = 1) \wedge (D \rightarrow Q)]$$

Here the clock input is set up at least *stp* units of time. Because the clock blocks the data input $D$, $D$ is also set up.

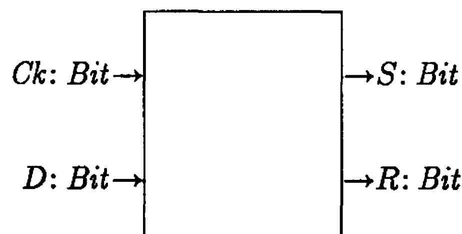
## §8.3 Implementation of D-Flip-Flip

A D-flip-flop can be constructed out of two components in a manner similar to building a D-latch. The first component, known as the *master latch*, serves as an interface between the clock and data inputs on one hand and the second component, the *slave latch*, on the other. The slave provides the flip-flop's outputs. There are four key time periods in the overall flip-flop operation: clock is 0, clock rises from 0 to 1, clock is 1, and clock drops from 1 to 0:

- When the clock is 0, the master latch disables the slave, which maintains whatever value was previously stored. At this time, the clock and data inputs can be set up for clocking in a new bit.
- Upon the clock transition from 0 to 1, the master latch itself stores the incoming data signal and actively propagates it to the slave. The slave in turn adjusts the outputs to reflect the new data.
- As long as the clock remains at 1, the master continues to transmit the stored value to the slave.
- When the clock drops from 1 to 0, the master disables the slave, leaving the stored value undisturbed. At this point, the cycle of clocking can be repeated.

### Specification of the master latch

The master latch has the following structure:




$(stp, hld, prd, lat)$: *time*

The timing parameters have the same form as in the flip-flop description since the master device has the clock and data signals as inputs.

$$
\begin{aligned}
&\mathit{Master}(M) \quad \equiv_{\text{def}} \\
&\quad \mathit{MasterStructure}(M) \\
&\quad \wedge \boxdot\, \mathit{Store}(M) \\
&\quad \wedge \boxdot\, \mathit{Nontrig}(M) \\
&\quad \wedge \boxdot\, \mathit{Steady}(M)
\end{aligned}
$$

*Definition of MasterStructure:*

$$
\begin{aligned}
&\mathit{MasterStructure}(M) \quad \equiv_{\text{def}} \\
&\quad M\!: \mathit{struct}[ \\
&\qquad (Ck, D)\!: \mathit{Bit} && \text{\%Inputs} \\
&\qquad (S, R)\!: \mathit{Bit} && \text{\%Outputs} \\
&\qquad \mathit{Status}\!: \mathit{Bit} && \text{\%Internal} \\
&\qquad (\mathit{stp}, \mathit{hld}, \mathit{prd}, \mathit{lat})\!: \mathit{time} && \text{\%Parameters} \\
&\quad ]
\end{aligned}
$$

*Definition of Store:*

The data value present when the clock rises determines the $S$ and $R$ outputs.

$$
\begin{aligned}
&\mathit{Store}(M) \quad \equiv_{\text{def}} \\
&\quad (\mathit{stb}\langle Ck, D\rangle \;\wedge\; \mathit{len} \ge \mathit{stp}) \leadsto \mathit{Trigger}(M)
\end{aligned}
$$

where the predicate *Trigger* is defined as follows:

$$
\begin{aligned}
&\mathit{Trigger}(M) \quad \equiv_{\text{def}} \\
&\quad ({\uparrow}^{0,\mathit{prd}} Ck \;\wedge\; Ck\ \mathit{blk}^{\mathit{hld}}\ D) \\
&\qquad\qquad \supset\ [\mathit{fin}(\mathit{Status} = 1) \;\wedge\; (D \to S) \;\wedge\; (\neg D \to R)]
\end{aligned}
$$

*Definition of Nontrig:*

If the master latch is initially steady, then after the clock drops, both $S$ and $R$ become smoothly disabled at **0**.

$$Nontrig(M) \equiv_{\text{def}}$$
$$[\downarrow^{0,prd} Ck \ \wedge \ beg(Status = 1)]$$
$$\supset \ [fin([S = 0] \ \wedge \ [R = 0] \ \wedge \ [Status = 1]) \ \wedge \ sm\langle S, R\rangle]$$

*Definition of Steady:*

When the master latch is steady, the status flag and the outputs are blocked by the clock.

$$Steady(M) \equiv_{\text{def}}$$
$$beg(Status = 1) \ \supset \ [Ck \ blk \ Status \ \wedge \ Ck \ blk^{lat} \langle S, R\rangle]$$

## Combining the latches

The next predicate shows how the master and slave latches are combined to implement a D-flip-flop. We use an SR-latch as the slave.

$$DFFImplementation(M, L) \equiv_{\text{def}}$$
$$Master(M) \ \wedge \ SRLatch(L)$$
$$\wedge \ (M.S \approx L.S) \ \wedge \ (M.R \approx L.R)$$

The mapping from the latches to the flip-flop takes the following form:

$$\models \ DFFImplementation(M, L) \ \supset \ DFlipFlop(F)$$

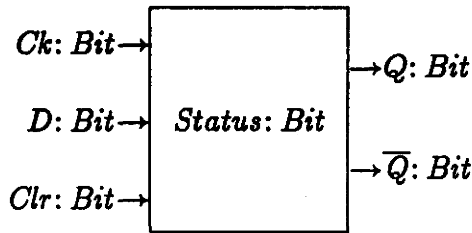
where the tuple $F$ is constructed as follows:

$$F.Ck \approx M.Ck$$
$$F.D \approx M.D$$

$$
\begin{aligned}
F.S &\approx L.S \\
F.R &\approx L.R \\
F.Status &\approx (M.Status \wedge L.Status) \\
F.stp &= M.stp \\
F.prd &= L.prd + M.prd \\
F.hld &= M.hld \\
F.lat &= L.lat + M.lat
\end{aligned}
$$

## §8.4 D-Flip-Flops with Asynchronous Initialization Signals

Integrated circuits such as the TTL 7474 chip [48] contain D-flip-flops with extra inputs for initialization. Since these pins are used more or less independently of the clock, they are called *asynchronous*. The device considered here has a single asynchronous input *Clr*:




*stp, prd, hld, lat*

*Definition of AsynchDFlipFlop:*

The specification has predicates for operating the clock and clear signals:

$$
\begin{aligned}
&AsynchDFlipFlop(F) \quad \equiv_{\text{def}} \\
&\quad AsynchDFFStructure(F) \\
&\quad \wedge \boxdot\ UseClock(F) \\
&\quad \wedge \boxdot\ UseClear(F) \\
&\quad \wedge \boxdot\ Steady(F)
\end{aligned}
$$

*Definition of AsynchDFFStructure:*

$$AsynchDFFStructure(F) \quad \equiv_{\mathrm{def}}$$

$F$: *struct*[
- $(Ck, D, Clr)$: *Bit* %Inputs
- $(Q, \overline{Q})$: *Bit* %Outputs
- *Status*: *Bit* %Internal
- $(stp, prd, hld, lat)$: *time* %Parameters

]

*Definition of UseClock:*

During periods when the input $Clr$ equals 0, the device acts according to the earlier specification *DFlipFlop*:

$$UseClock(F) \quad \equiv_{\mathrm{def}}$$

$$(Clr \approx 0) \supset DFlipFlop(G)$$

where $G$ contains exactly the following fields of $F$:

$$Ck, D, Q, \overline{Q}, Status, stp, prd, hld, lat$$

*Definition of UseClear:*

When the clock is stable, the input $Clr$ can be used to initialize the flip-flop:

$$UseClear(F) \quad \equiv_{\mathrm{def}}$$

$$\mathbf{stb}\ Ck \supset [Clear(F) \wedge Disable(F)]$$

*Definition of Clear:*

If the input $Clr$ equals 1 long enough, the output $Q$ is zeroed and the device becomes steady:

$$Clear(F) \quad \equiv_{\mathrm{def}}$$

$$(Clr \approx 1 \wedge \mathbf{len} \geq prd) \supset \mathbf{fin}[(Status = 1) \wedge (Q = 0)]$$

*Definition of Disable:*

When the device is steady and the input *Clr* drops to 0, the device remains steady:

$$Disable(F) \quad \equiv_{\text{def}}$$
$$[\downarrow^{0,prd} Clr \ \wedge \ beg(Status = 1)] \ \supset \ fin[(Status = 1) \ \wedge \ stb\langle Q, \overline{Q}\rangle]$$

*Definition of Steady:*

When the flip-flop is steady, the inputs *Ck* and *Clr* together block the signals *Status*, $Q$ and $\overline{Q}$:

$$Steady(F) \quad \equiv_{\text{def}}$$
$$beg(Status = 1) \ \supset \ [beg(\overline{Q} = \neg Q) \ \wedge \ \langle Ck, Clr\rangle \ blk \ \langle Status, Q, \overline{Q}\rangle]$$

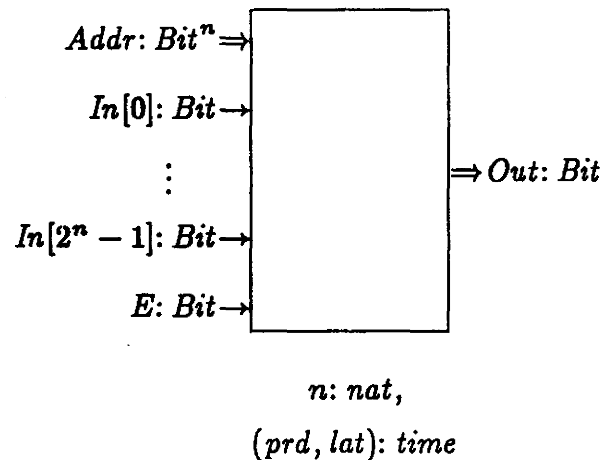
CHAPTER 9

# MORE DIGITAL DEVICES

We now consider techniques for describing and reasoning about multiplexers, random-access memories, counters and shift registers.

## §9.1 Multiplexer

A multiplexer has a number of addressible inputs and can selectively output any one of them. The device considered below can be optionally disabled, in which case it outputs a zero. The general structure is as follows:




$n$: *nat*,

$(prd, lat)$: *time*

The device operates roughly according the table below:

| operation | $E$ | *Out* |
|---|---|---|
| select | 1 | $In[loc]$ |
| disable | 0 | 0 |

where $loc = nval(Addr)$. If we ignore propagation delay, the multiplexer behaves according to the formula

$$Out \approx \big(if\ [E = 1]\ then\ In[nval(Addr)]\ else\ 0\big)$$

During periods when the device is enabled with $E = 1$, the formula reduces to

$$Out \approx In[nval(Addr)]$$

*Definition of Multiplexer:*

The multiplexer's main predicate is as follows:

$$\begin{aligned}
&Multiplexer(X) \quad \equiv_{\mathbf{def}}\\
&\quad MultiplexerStructure(X)\\
&\quad \wedge\ \boxdot\ Select(X, loc), \quad \text{for } loc \in [0, n-1]\\
&\quad \wedge\ \boxdot\ Disable(X)
\end{aligned}$$

*Definition of MultiplexerStructure:*

The device has an $n$-bit vector $Addr$ for selecting one of $2^n$ possible incoming bits of the vector $In$.

$$\begin{aligned}
&MultiplexerStructure(X) \quad \equiv_{\mathbf{def}}\\
&\quad X\colon struct\big[\\
&\qquad Addr\colon Bit^n, In\colon Bit^{(2^n)}, E\colon Bit \qquad \text{\%Inputs}\\
&\qquad Out\colon Bit \qquad \text{\%Outputs}\\
&\qquad n\colon nat, (prd, lat)\colon time \qquad \text{\%Parameters}\\
&\quad \big]
\end{aligned}$$

*Definition of Select:*

If the enable signal $E$ is held at 1 and the address line and its associated input are stable, the output ends up equal to the input line indicated by the static variable

*loc*.

$$Select(X, loc) \equiv_{\text{def}}$$
$$\big([E \approx 1] \land stb\, In[loc] \land [nval(Addr) \approx loc] \land len \geq prd\big)$$
$$\leadsto \big[beg(Out = In[loc]) \land \langle E, Addr, In[loc]\rangle\, blk^{lat}\, Out\big]$$

*Definition of Disable:*

Holding the signal $E$ at 0 clears the output.

$$Disable(X) \equiv_{\text{def}}$$
$$(E \approx 0 \land len \geq prd) \leadsto \big[beg(Out = 0) \land E\, blk^{lat}\, Out\big]$$

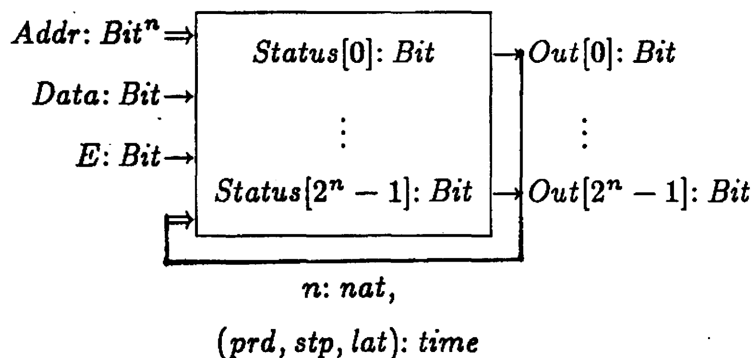
**Alternative specifications**

Like the adder discussed earlier, the predicate *Multiplexer* can be equivalently specified with an internal status bit and predicate *Steady*.

The timing parameters could be made more detailed so that, for example, the parameter *select.prd* would give the propagation time when using the predicate *Select*.

## §9.2 Memory

The memory described here has the following form




There is a series of cells, each associated with status and output bits. At any time, at most one cell can be selected and modified. During this period the remaining

cells are left untouched. When the enable signal is inactive at 0, no cell can be altered.

If we assume unit delay, the memory behaves as follows:

$$\left[\mathit{if}\ (E = 1)\ \mathit{then}\ \mathit{alter}(\mathit{Out}, \mathit{nval}(\mathit{Addr}), \mathit{Data})\ \mathit{else}\ \mathit{Out}\right]\ \mathit{del}\ \mathit{Out}$$

where the function $alter(Out, i, a)$ equals a vector whose $i$-th element equals $a$ and whose remaining elements equal those in $Out$. The behavior can also be expressed using iteration and an in-place variant of *alter*:

$$\left(\mathit{skip}\ \wedge\ [\mathit{if}\ (E = 1)\ \mathit{then}\ \mathit{Alter}(\mathit{Out}, \mathit{nval}(\mathit{Addr}), D)\ \mathit{else}\ (\mathit{stb}\ \mathit{Out})]\right)^*$$

where $Alter(Out, i, a)$ sets the $i$-th element of $Out$ to $a$ and leaves the others unchanged:

$$\mathit{Alter}(\mathit{Out}, i, a) \quad \equiv_{\text{def}} \quad [\mathit{alter}(\mathit{Out}, i, a) \rightarrow \mathit{Out}]$$

In practice, a memory has a multiplexer connected to the outputs so that at any time at most a single cell can be read. This technique permits one cell to be written while another is being retrieved. We do not include such multiplexers here.

*Definition of Memory:*

$$\begin{array}{l}
\mathit{Memory}(M) \quad \equiv_{\text{def}} \\
\quad \mathit{MemoryStructure}(M) \\
\quad \wedge\ \forall \mathit{loc} \in [0, 2^n - 1]. \\
\qquad \boxdot\ \mathit{Enable}(M, \mathit{loc}) \\
\qquad \wedge\ \boxdot\ \mathit{Write}(M, \mathit{loc}) \\
\qquad \wedge\ \boxdot\ \mathit{Disable}(M, \mathit{loc}, \mathit{mode}), \quad \text{for } \mathit{mode} \in {}'\{\mathit{selected}, \mathit{not_selected}\} \\
\qquad \wedge\ \boxdot\ \mathit{Steady}(M, \mathit{loc}, \mathit{mode}), \\
\qquad\qquad \text{for } \mathit{mode} \in {}'\{\mathit{disabled}, \mathit{selected}, \mathit{not_selected}\}
\end{array}$$

*Definition of MemoryStructure:*

$$MemoryStructure(M) \quad \equiv_{\text{def}}$$

$$M\text{: } struct\Big[$$

| | |
|---|---|
| $Addr\text{: } Bit^n, Data\text{: } Bit, E\text{: } Bit$ | %Inputs |
| $Out\text{: } Bit^{(2^n)}$ | %Outputs |
| $Status\text{: } Bit^{(2^n)}$ | %Internal |
| $n\text{: } nat, (prd, stp, lat)\text{: } time$ | %Parameters |

$$\Big]$$

*Definition of Enable:*

When the memory becomes enabled with $E$ rising from 0 to 1 and a cell does not have the address selected by $Addr$, the cell's output remains stable.

$$Enable(M, loc) \quad \equiv_{\text{def}}$$

$$\big(\uparrow^{stp,prd}E \ \wedge \ stb\, Addr \ \wedge \ beg[nval(Addr) \neq loc \ \wedge \ Status[loc] = 1]\big)$$
$$\supset \ fin\big[(Status[loc] = 1) \ \wedge \ stb\ Out[loc]\big]$$

*Definition of Write:*

When the device is enabled, the cell addressed by $Addr$ can be written with the value of the data input.

$$Write(M, loc) \quad \equiv_{\text{def}}$$

$$\big(len \geq prd \ \wedge \ [E \approx 1] \ \wedge \ stb\, Data \ \wedge \ nval(Addr) \approx loc\big)$$
$$\supset \ fin\big[(Status[loc] = 1) \ \wedge \ (Out[loc] = Data)\big]$$

*Definition of Disable:*

Disabling the memory does not affect a steady cell's output. If the cell is currently addressed, both $Addr$ and $Data$ must remain stable until after $E$ drops.

Otherwise only *Addr* need hold. The predicate *check*, defined below, ensures that the particular location is in the indicated mode.

$$Disable(M, loc, mode) \equiv_{\text{def}}$$
$$\left(\downarrow^{0,prd} E \wedge stb\, U \wedge beg[check(M, loc, mode) \wedge (Status[loc] = 1)]\right) \supset \left[fin(Status[loc] = 1) \wedge stb\; Out[loc]\right]$$

where $U$ is as follows:

| *mode* | $U$ |
|---|---|
| *selected* | $\langle Addr, Data \rangle$ |
| *not_selected* | $\langle Addr \rangle$ |

*Definition of Steady:*

$$Steady(M, loc, mode) \equiv_{\text{def}}$$
$$beg[(Status[loc] = 1) \wedge check(M, loc, mode)] \supset (V\ blk\ Status[loc] \wedge V\ blk^{lat}\ Out[loc])$$

where the table below gives $V$ as a function of the indicated mode:

| *mode* | $V$ |
|---|---|
| *disabled* | $\langle E \rangle$ |
| *selected* | $\langle E, Addr, Data \rangle$ |
| *not_selected* | $\langle E, Addr \rangle$ |

If a cell is steady, its output is blocked by the signal $E$ and other appropriate inputs based on whether the device is enabled and whether the cell is the one selected. If the entire memory is *disabled*, only $E$ blocks the cells. If the memory is enabled and the particular cell is the one *selected*, the cell's output is blocked by the inputs $E$, *Addr* and *Data*. If however the cell is currently *not selected*, it is blocked only by $E$ and *Addr*. This is summarized in the table shown after the definition of *Steady*. The predicate *check*, defined below, makes certain that the particular memory location is indeed in the chosen mode of operation.

*Definition of check:*

The predicate *check* verifies that the given location is in the specified mode of operation:

| *mode* | $check(M, loc, mode)$ |
|---|---|
| *disabled* | $E = 0$ |
| *selected* | $(E = 1) \wedge [nval(Addr) = loc]$ |
| *not_selected* | $(E = 1) \wedge [nval(Addr) \neq loc]$ |

## §9.3 Counters

We can model a simple counter by means of addition and unit-delay:

$$(I + 1)\ del\ I$$

The next formula shows a way to handle initialization:

$$[if\ (Clr = 1)\ then\ 0\ else\ (I + 1)]\ del\ I$$

If it is only necessary that the counter is initially equal to 0, the formula below suffices:

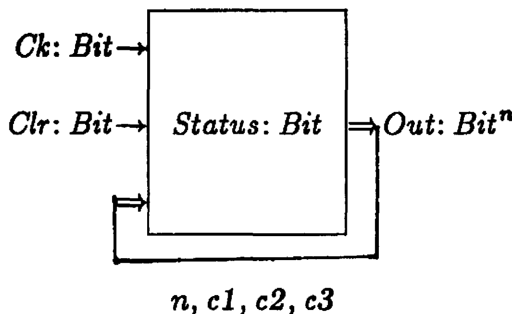
$$beg(I = 0) \wedge [(I + 1)\ del\ I]$$

The following example takes finite precision into account:

$$[(I + 1) \bmod 2^n]\ del\ I$$

### Clocked counter

A clocked counter stores a number that can by incremented by 1 modulo some base when the device is triggered. Here is the physical structure:




$n, c1, c2, c3$

The *binary* counter considered here has an $n$-bit output vector and cycles through the numbers 0 to $2^n - 1$. Not all counters are binary. For example, a *decade* counter has a 4-bit output and cycles through the numbers 0 to 9. The values 10 to 15 are never computed.

*Definition of Counter:*

The predicates *Clear* and *Increment* specify how to clear and increment the counter's output.

$$
\begin{aligned}
&Counter(C) \quad \equiv_{\text{def}} \\
&\quad CounterStructure(C) \\
&\quad \wedge\ \boxdot\ Clear(C) \\
&\quad \wedge\ \boxdot\ Increment(C) \\
&\quad \wedge\ \boxdot\ Steady(C)
\end{aligned}
$$

*Definition of CounterStructure:*

The device's structure is given below. The internal bit signal *Status* indicates when the device is in a steady state.

$$
\begin{aligned}
&CounterStructure(C) \quad \equiv_{\text{def}} \\
&\quad C\colon struct[ \\
&\qquad (Ck,\ Clr)\colon Bit && \%\text{Inputs} \\
&\qquad Out\colon Bit^{n} && \%\text{Outputs} \\
&\qquad Status\colon Bit && \%\text{Internal} \\
&\qquad n\colon nat, (c1,\ c2,\ c3)\colon time && \%\text{Parameters} \\
&\quad ]
\end{aligned}
$$

*Definition of Clear:*

When the clock has a positive pulse and the input *Clr* equals 1, the device is cleared and ends up steady with *Status* equaling 1:

$$
\begin{aligned}
&Clear(C) \quad \equiv_{\text{def}} \\
&\quad [\uparrow\downarrow^{c1,c2,c3} Ck\ \wedge\ beg(Clr = 1)\ \wedge\ Ck\ blk\ Clr] \\
&\qquad\qquad \supset\ fin[nval(Out) = 0\ \wedge\ Status = 1]
\end{aligned}
$$

*Definition of Increment:*

If the device is initially steady and the clock is pulsed and *Clr* equals 0, then the output vector's numerical value is incremented by 1 modulo $2^n$. The device ends up steady.

$$Increment(C) \quad \equiv_{\text{def}}$$

$$[\uparrow\downarrow^{c1,c2,c3} Ck \;\land\; beg(Status = 1 \;\land\; Clr = 0) \;\land\; Ck\ blk\ Clr]$$
$$\supset \; \big([nval(Out) + 1] \bmod 2^n \rightarrow nval(Out) \;\land\; fin[Status = 1]\big)$$

*Definition of Steady:*

When the bit signal *Status* equals 1, the clock input blocks both *Status* and *Out*. The blocking factor *lat* is associated with *Out*.

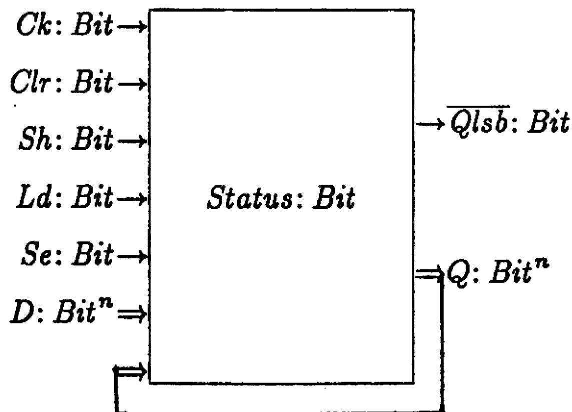
$$Steady(C) \quad \equiv_{\text{def}}$$

$$beg(Status = 1) \;\supset\; [Ck\ blk\ Status \;\land\; Ck\ blk^{lat}\ Out]$$

## §9.4 Shift Register

A shift register stores a bit vector that can be selectively initialized, shifted or left untouched. Some shift registers are bidirectional or can shift more than one place in a single operation. Others recirculate the bits or have special provisions for signed arithmetic. The output of a shift register may reflect the entire state or only part of it.

The TTL device discussed here stores $n$ bits that, when triggered, can be cleared, loaded with some data, shifted right by one place or maintained unchanged. The general form is given below. We omit the timing parameters from the diagram.



The register has a capacity of $n$ bits that are output by the vector $Q$. The least significant bit $Q[n-1]$ is also output in complemented form by $\overline{Qlsb}$. When clocking takes place, the fields *Clr*, *Sh* and *Ld* determine which operation occurs. The following table describes the general behavior upon clocking:

| operation | *Clr* | *Sh* | *Ld* | $Q$ |
|---|---|---|---|---|
| clear | 1 | – | – | $\langle 0 \rangle^n$ |
| shift | 0 | 1 | – | $\langle Se \rangle \parallel Q[0 \text{ to } n-2]$ |
| load | 0 | 0 | 1 | $D$ |
| nop | 0 | 0 | 0 | $Q$ |

The expression $\langle 0 \rangle^n$ stands for a list of $n$ 0's. Depending on the operation, only certain inputs are needed. For example, when *Clr* is 0, *Sh* is 1 and and a shift is to take place, the device ignores the inputs *Ld* and $D$.

*Definition of ShiftRegister:*

As with the counter described earlier, the shift register specification has predicates for clocking and steadiness.

$$
\begin{aligned}
&\mathit{ShiftRegister}(H) \quad \equiv_{\mathrm{def}} \\
&\quad \mathit{ShiftRegStructure}(H) \\
&\quad \wedge\ \boxdot\ \mathit{Trigger}(H, op), \quad \text{for } op \in \text{'}\{\mathit{clear}, \mathit{shift}, \mathit{load}, \mathit{nop}\} \\
&\quad \wedge\ \boxdot\ \mathit{Nontrig}(H) \\
&\quad \wedge\ \boxdot\ \mathit{Steady}(H)
\end{aligned}
$$

*Definition of ShiftRegStructure:*

$$
\begin{aligned}
&\mathit{ShiftRegStructure}(H) \quad \equiv_{\text{def}}\\
&\quad H\colon \mathit{struct}\Big[\\
&\qquad (Ck, Clr, Sh, Ld, Se)\colon Bit, D\colon Bit^n && \text{\%Inputs}\\
&\qquad Q\colon Bit^n,\ \overline{Qlsb}\colon Bit && \text{\%Outputs}\\
&\qquad Status\colon Bit && \text{\%Internal}\\
&\qquad n\colon positive, && \text{\%Parameters}\\
&\qquad (lat, prd)\colon time,\\
&\qquad stp\colon (Ck, Clr, Sh, Ld, Se, D, Q)\colon time,\\
&\qquad hld\colon (Clr, Sh, Ld, Se, D, Q)\colon time\\
&\quad \Big]
\end{aligned}
$$

The register's length $n$ must be at least 1.

*Definition of Steady:*

When the status bit equals 1, the output $\overline{Qlsb}$ equals the complement of $Q$'s least significant bit $Q[n-1]$.

$$
\begin{aligned}
&\mathit{Steady}(H) \quad \equiv_{\text{def}}\\
&\quad beg(Status = 1)\\
&\qquad\quad \supset \big[beg(\overline{Qlsb} = \neg Q[n-1])\\
&\qquad\qquad\qquad \wedge\ Ck\ blk\ Status\ \wedge\ Ck\ blk^{lat}\langle Q, \overline{Qlsb}\rangle\big]
\end{aligned}
$$

*Definition of Trigger:*

The value of *op* determines the particular operation to be undertaken. For example, the field name *'load* is used as a parameter to *Trigger* for performing a load operation.

$$
\begin{aligned}
&\mathit{Trigger}(H, op) \quad \equiv_{\text{def}}\\
&\quad SetUp(H, op) \rightsquigarrow Compute(H, op)
\end{aligned}
$$

*Definition of SetUp:*

The predicate *SetUp* ensures that the appropriate input signals have the proper values and are stable long enough prior to the actual operation. The predicates *check* and *inpset* used here are defined later.

$$SetUp(H, op) \quad \equiv_{\text{def}}$$
$$fin[check(H, op)]$$
$$\wedge\ \forall field \in [inpset(op) \cup \{'Ck\}].\ (tstb^{stp[field]}\ H[field])$$

*Definition of Compute:*

The text of *Compute* overviews the clocking involved in performing an operation. The predicates *Hold* describes how inputs must be held as the clock rises. The function *result* indicates the new value of the output $Q$.

$$Compute(H, op) \quad \equiv_{\text{def}}$$
$$[\uparrow^{0,prd} Ck\ \wedge\ Hold(H, op)]$$
$$\supset\ (fin[Status = 1]\ \wedge\ [result(H, op) \rightarrow Q])$$

After clocking, the status bit ends up equal to 1 and the output vector $Q$ receives the selected function of the inputs.

*Definition of check:*

The predicate *check* gives the values of the control bits *Clr*, *Sh* and *Ld* necessary for the desired operation.

| *op* | *check*(*H*, *op*) |
|---|---|
| *clear* | $Clr = 1$ |
| *shift* | $(Clr = 0) \wedge (Sh = 1)$ |
| *load* | $(Clr = 0) \wedge (Sh = 0) \wedge (Ld = 1)$ |
| *nop* | $(Clr = 0) \wedge (Sh = 0) \wedge (Ld = 0)$ |

*Definition of inpset:*

The function *inpset* specifies the set of inputs needed in performing the particular operation. For example, during shifting, the *Ld* control signal is ignored and is therefore not listed.

| *op* | *inpset(op)* |
|---|---|
| *clear* | $'\{Clr\}$ |
| *shift* | $'\{Clr, Sh, Se, Q\}$ |
| *load* | $'\{Clr, Sh, Ld, D\}$ |
| *nop* | $'\{Clr, Sh, Ld, Q\}$ |

*Definition of Hold:*

Each operation's required input signals must be held stable beyond the clock transition for the time given in the corresponding subfield of *hld*.

$$Hold(H, op) \quad \equiv_{\mathrm{def}}$$
$$\forall field \in inpset(op). \big(Ck\ blk^{hld[field]}\ H[field]\big)$$

*Definition of result:*

For each of the three clocked operations, the function *result* specifies the output $Q$'s new value.

| *op* | *result(H, op)* |
|---|---|
| *clear* | $\langle 0 \rangle^n$ |
| *shift* | $\langle Se \rangle \parallel Q[0 \text{ to } n-2]$ |
| *load* | $D$ |
| *nop* | $Q$ |

*Definition of Nontrig:*

If the counter is steady, a falling clock edge preserves the status bit and leaves the outputs $Q$ and $\overline{Qlsb}$ stable.

$$Nontrig(H) \quad \equiv_{\mathrm{def}}$$
$$[\downarrow^{0,prd} Ck \ \wedge\ beg(Status = 1)] \supset [fin(Status = 1) \ \wedge\ stb\langle Q, \overline{Qlsb}\rangle]$$

### Variant specifications

A more detailed description can be given with separate timing information for the operations *clear*, *shift*, *load* and *nop*. In addition, the times for rising and falling clock edges need not be the same.

Alternatively, we can combine the control inputs into a signal called $Op$ and ignore the details of clocking. The signal $Op$ ranges over the values *'clear*, *'shift*, *'load* and *'nop*. The next formula describes the corresponding behavior using unit delay and a case construct:

$$\begin{pmatrix} \textit{case Op of} \\ \quad \textit{clear}: \langle 0 \rangle^n \\ \quad \textit{shift}: \langle Se \rangle \,\|\, Q[0 \text{ to } n-2] \\ \quad \textit{load}: D \\ \quad \textit{nop}: Q \end{pmatrix} \textit{del } Q$$

The case expression uses as its value the entry selected by $Op$. For example, when $Op$ equals *'load*, the case expression equals $D$. The expression $\langle 0 \rangle^n$ equals an $n$-element list of 0's.

### Combining shift registers

Two shift registers can be connected to form a larger one. The following property reflects this with the shift register $H$ containing the most significant bits and $I$ containing the least significant bits:

$$\begin{aligned} &[\textit{ShiftRegister}(H) \land \textit{ShiftRegister}(I) \\ &\quad \land (H.Ck \approx I.Ck) \land (H.Clr \approx I.Clr) \land (H.Sh \approx I.Sh) \\ &\quad \land (H.Ld \approx I.Ld) \land (H.Q[n-1] \approx I.Se) \land (H.lat \geq I.hld.Se)] \\ &\qquad\qquad \supset \textit{ShiftRegister}(J) \end{aligned}$$

where

$$\begin{aligned} J[\textit{field}] &\approx H[\textit{field}], \quad \text{for } \textit{field} \in \text{'}\{Ck, Clr, Sh, Ld\} \\ J.D &\approx H.D \,\|\, I.D \\ J.Se &\approx H.Se \\ J.Q &\approx H.Q \,\|\, I.Q \\ J.Status &\approx H.Status \land I.Status \\ J.n &= H.n + I.n \end{aligned}$$

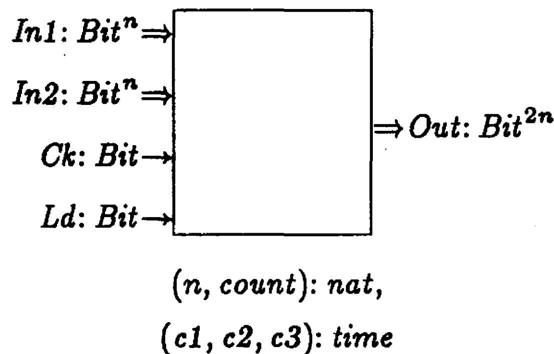
$$
\begin{aligned}
J.prd &= \max(H.prd, I.prd) \\
J.stp[field] &= \max(H.stp[field], I.stp[field]), \quad \text{for } field \in {}'\{Ck, Clr, Sh, Ld, D\} \\
J.stp.Se &= H.stp.Se \\
J.stp.Q &= \max(H.stp.Q, I.stp.Q, I.stp.Se) \\
J.hld[field] &= \max(H.hld[field], I.hld[field]), \quad \text{for } field \in {}'\{Clr, Sh, Ld, D\} \\
J.hld.Se &= H.hld.Se \\
J.hld.Q &= \max(H.hld.Q, I.hld.Q, I.hld.Se) \\
J.lat &= \min(H.lat, I.lat)
\end{aligned}
$$

An abbreviated form of this property can be expressed for combining two unit-delay shift registers.

CHAPTER 10

# MULTIPLICATION CIRCUIT

The hardware multiplier considered here is motivated by one discussed in Wagner's work on hardware verification [49]. The desired device behavior is first described followed by a look at implementation techniques. The multiplier has the following general structure:




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

## §10.1 Specification of Multiplier

The multiplier is first specified by means of the predicate *Multiplier*($M$). We then develop an iterative, timing-independent multiplication algorithm that com-

putes a product by a series of successive additions. Later, the predicate *Implementation*($H$) characterizes a device that computes sums and in fact has the algorithm's steps embedded within it. A logical implication is then given, showing how *Implementation*($H$) realizes *Multiplier*($M$).

*Definition of Multiplier:*

Here is the main predicate:

$$
\begin{aligned}
&\mathit{Multiplier}(M) \quad \equiv_{\mathrm{def}} \\
&\quad \mathit{MultStructure}(M) \\
&\quad\quad \wedge \boxdot\ \mathit{Calculate}(M)
\end{aligned}
$$

*Definition of MultStructure:*

The multiplier has the following structure:

$$
\begin{aligned}
&\mathit{MultStructure}(M) \quad \equiv_{\mathrm{def}} \\
&\quad M\colon \mathbf{struct}\big[ \\
&\quad\quad (Ck, Ld)\colon \mathit{Bit}, && \%\text{Inputs} \\
&\quad\quad (In1, In2)\colon \mathit{Bit}^n \\
&\quad\quad Out\colon \mathit{Bit}^{2n} && \%\text{Outputs} \\
&\quad\quad (n, count)\colon \mathit{nat}, && \%\text{Parameters} \\
&\quad\quad c1, c2, c3\colon \mathit{time} \\
&\quad \big]
\end{aligned}
$$

*Definition of Calculate:*

If the inputs behave as specified by the predicate *Control*, the output *Out* ends up with the product of the initial values of *In1* and *In2*.

$$
\begin{aligned}
&\mathit{Calculate}(M) \quad \equiv_{\mathrm{def}} \\
&\quad \mathit{Control}(M) \supset \\
&\quad\quad \big[\mathit{nval}(In1) \cdot \mathit{nval}(In2)\big] \rightarrow \mathit{nval}(Out)
\end{aligned}
$$

*Definition of Control:*

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

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

*Definition of Load:*

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

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

*Definition of SingleCycle:*

An individual clock cycle consists of a negative pulse:

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

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

*Definition of Cycling:*

The overall cycling of the clock is as follows:

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

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

**Variants of the specification**

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

## §10.2 Development of Multiplication Algorithm

The specification predicate *Multiplier* intentionally makes no reference to any particular technique for multiplying. Since the process of multiplication does not generally depend on any specific circuit timing, it is natural to separate algorithmic issues from other implementation details. We now use ITL as a basis for deriving a suitable circuit-independent algorithm for determining the product and in the next section as a means for describing hardware that realizes this method. The synthesis process can be viewed as a proof in reverse, starting with the goal and ending with the necessary assumptions to achieve it.

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

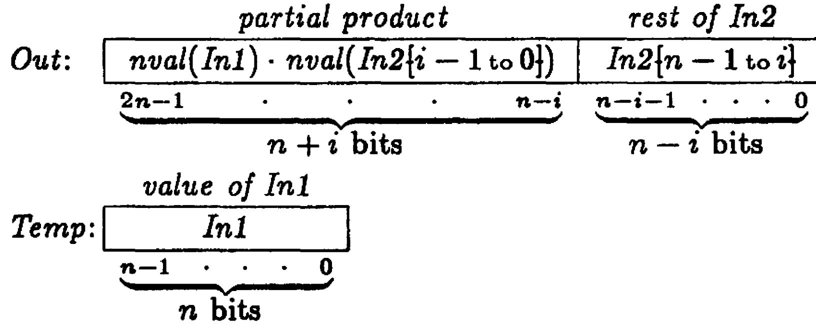
$$Goal(A) \quad \equiv_{\text{def}}$$
$$\big[nval(In1) \cdot nval(In2)\big] \to nval(Out)$$

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

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

is computed and available in the upper $n+i$ bits of *Out*. Recall that the subscripting brackets {} index a vector from the right. Although neither *In1* nor *In2* is guaranteed

to remain stable, their initial values must be used throughout the calculation. The lower $n-i$ bits of *Out* hold the unexamined bits of *In2* (i.e., $In2\{n-1 \text{ to } i\}$). In addition, an extra $n$-bit variable *Temp* is introduced in order to remember the original value of *In1*. The following figure informally depicts the situation after $i$ steps:




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

The predicate *Assert* below precisely specifies this behavior over $i$ iterations for $i \le n$.

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

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

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

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

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

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

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

Since this formula is a goal and not yet a property, we omit the validity symbol $\models$. The formulas $(*)$ and $(**)$ together ensure that $n$ iterations of the loop calculate the product:

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

### Deriving the predicate *Init*

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

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

Simplification of *Assert* yields the constraint

$$\begin{aligned} &Init(A) \supset \\ &\quad 0 \to nval(Out\{2n-1 \text{ to } n\}) \\ &\quad\quad \wedge\ In2 \to Out\{n-1 \text{ to } 0\} \\ &\quad\quad \wedge\ In1 \to Temp \end{aligned}$$

This can be achieved by the definition

$$\begin{aligned} &Init(A) \quad \equiv_{\text{def}} \\ &\quad \langle 0\rangle^n \to Out\{2n-1 \text{ to } n\} \\ &\quad\quad \wedge\ In2 \to Out\{n-1 \text{ to } 0\} \\ &\quad\quad \wedge\ In1 \to Temp \end{aligned}$$

where $\langle 0\rangle^n$ equals an $n$-element list of 0's.

### Deriving the predicate *Step*

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

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

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

$$Step(A) \supset$$
$$Temp \to Temp$$

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

The constraint for *Out* is

$$Step(A) \supset$$
$$\big[nval(Out\{2n-1 \text{ to } n\}) + Out\{0\} \cdot nval(Temp)\big]$$
$$\to nval(Out\{2n-1 \text{ to } n-1\})$$
$$\land \; Out\{n-1 \text{ to } 1\} \to Out\{n-2 \text{ to } 0\}$$

Thus the overall incremental step can be realized by the definition

$$Step(A) \quad \equiv_{\text{def}}$$
$$\big[nval(Out\{2n-1 \text{ to } n\}) + Out\{0\} \cdot nval(Temp)\big]$$
$$\to nval(Out\{2n-1 \text{ to } n-1\})$$
$$\land \; Out\{n-1 \text{ to } 1\} \to Out\{n-2 \text{ to } 0\}$$
$$\land \; Temp \to Temp$$

## §10.3 Description of Implementation

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

$$Implementation(H) \quad \equiv_{\text{def}}$$
$$ImpStructure(H)$$
$$\land \; \boxdot(LoadPhase(H) \land MultPhase(H))$$

*Definition of ImpStructure:*

The structure of the implementation differs from that of the original specification by the addition of the internal states *Temp* and *Status* and by the omission of a *count* field giving the required number of clock cycles for computing a product. The vector *Temp* maintains the value of *In1*. The bit signal *Status* equals 1 when the device is in a steady state. The specification given below shows how to set *Status* to 1 and keep it at this value.

$$
\begin{array}{lll}
ImpStructure(H) \quad \equiv_{\text{def}} & & \\
\quad H\colon struct\Big[ & & \\
\qquad (Ck, Ld)\colon Bit, & & \%\text{Inputs} \\
\qquad (In1, In2)\colon Bit^n & & \\
\qquad Out\colon Bit^{2n} & & \%\text{Outputs} \\
\qquad Temp\colon Bit^n, & & \%\text{Internal} \\
\qquad Status\colon Bit & & \\
\qquad n\colon nat, & & \%\text{Parameters} \\
\qquad c1, c2, c3\colon time & & \\
\quad \Big] & &
\end{array}
$$

An external form of the complete specification would in effect existentially quantify over the fields *Temp* and *Status*.

*Definition of LoadPhase:*

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

$$
\begin{array}{l}
LoadPhase(H) \quad \equiv_{\text{def}} \\
\quad Load(H) \supset \big[Init(H) \wedge fin(Status = 1)\big]
\end{array}
$$

The predicate *Load* gives the required loading sequence for the circuit inputs. The predicate *Init* refers to the algorithm's initialization predicate. Once loading is complete, the field *Status* is set to 1, indicating that the device is ready to proceed with the multiplication. The definition of *Load* is identical to that of its namesake

in *Multiplier*:

$$\begin{aligned} &Load(H) \quad \equiv_{\text{def}} \\ &\qquad SingleCycle(H) \wedge beg(Ld = 1) \wedge Ck\ blk\ \langle Ld, In1, In2 \rangle \end{aligned}$$

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

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

*Definition of MultPhase:*

When the load signal is inactive at 0 and the device is steady (i.e., *Status*=1), the circuit can be clocked to perform a single iteration. The algorithm's predicate *Step* takes place over two clock cycles. Afterwards, the device is again steady with *Status* equaling 1.

$$\begin{aligned} &MultPhase(H) \quad \equiv_{\text{def}} \\ &\qquad [Ld \approx 0 \wedge (SingleCycle(H))^2 \wedge beg(Status = 1)] \\ &\qquad\qquad \supset [Step(H) \wedge fin(Status = 1)] \end{aligned}$$

## Implementation theorem

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

$$\models \quad Implementation(H) \supset Multiplier(M)$$

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

$$\begin{aligned} M[field] &\approx H[field], \qquad \text{for } field \in {}'\{In1, In2, Out\} \\ M.n &= H.n \\ M.count &= 2H.n \end{aligned}$$

$$M[\mathit{field}] \quad = \quad H[\mathit{field}], \qquad \text{for } \mathit{field} \in \text{'}\{c1, c2, c3\}$$

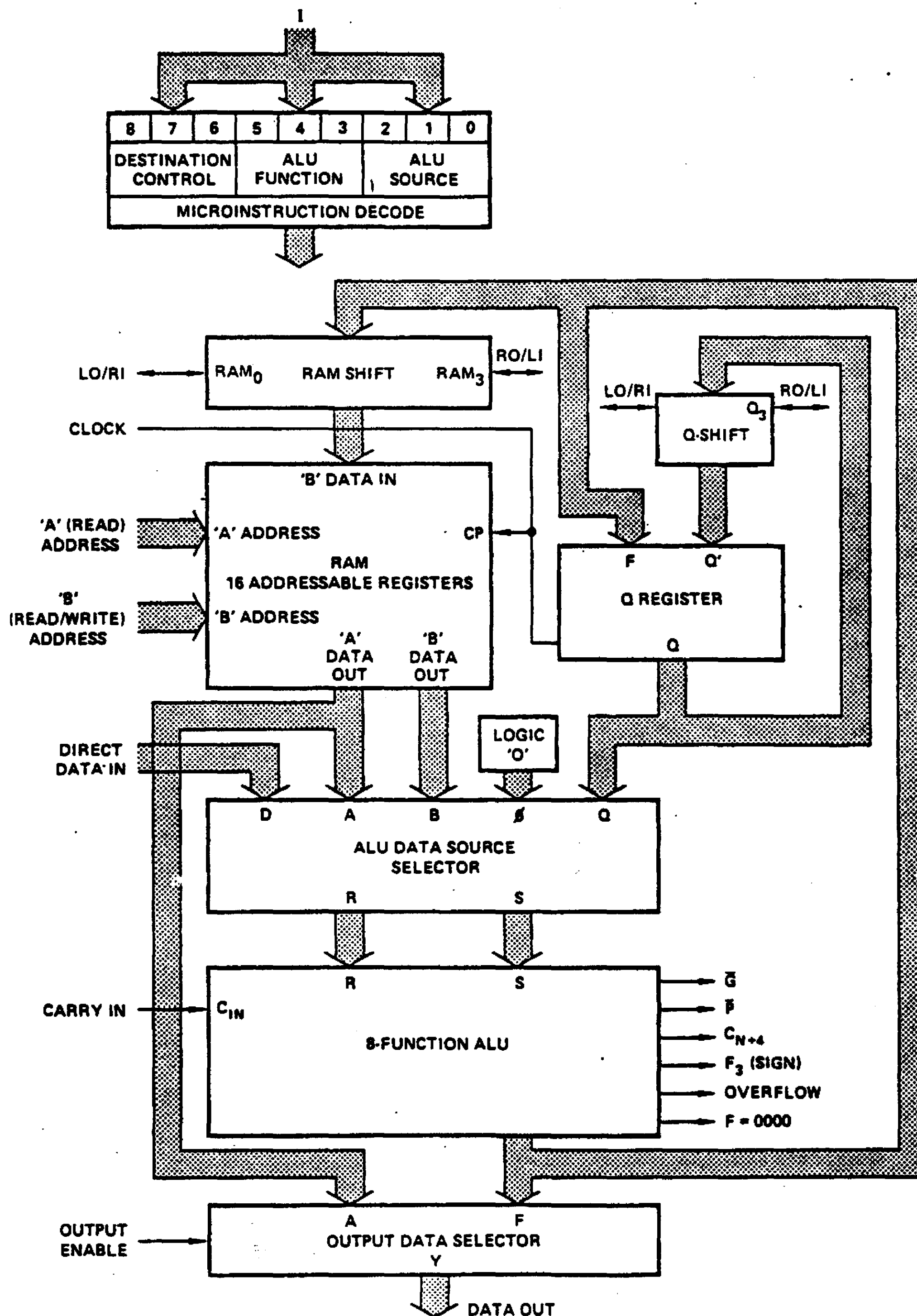
The value of *M.count* corresponds to the $2n$ clock cycles needed for doing the iterative computation.

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

CHAPTER 11

# THE AM2901 BIT SLICE

The Am2901 bit slice is a member of a popular family of integrated circuits developed by Advanced Micro Devices, Inc. for building processors and controllers. The next page contains a block diagram of the device. An individual Am2901 chip consists of four-bit *slices* of an arithmetic logic unit, memory, bus interface and other elements. These internal devices are connected together so as to provide various ways for computing and storing values. The next page contains a block diagram. A group of $m$ Am2901 chips can be connected to form circuits of bit length $4m$. We give a functional description of the Am2901 based on information contained in the Am2900 series' data book [1]. The temporal description is almost operational enough to be used as input to a suitable simulator. The reader desiring a detailed introduction to the Am2900 circuit family and its applications should consult the Am2900 data book [1], Mick and Brick [34] or Siewiorek et al. [43].




THE Am2901 4-BIT MICROPROCESSOR SLICE

Copyright © 1981 Advanced Micro Devices, Inc.

Reproduced with permission of copyright owner.

All rights reserved.

*Definition of BitSliceStructure:*

Here are the various signals and parameters used in our description of a generalized $n$-bit bit slice:

$$BitSliceStructure(N) \quad \equiv_{\text{def}}$$

$N$: *struct*[

| | |
|---|---|
| *Source*: *sig*(*sourceset*), | %Inputs |
| *Func*: *sig*(*funcset*), | |
| *Dest*: *sig*(*destset*), | |
| $D$: $Bit^n$, | |
| (*AAddr*, *BAddr*): *sig*([0 to 15]), | |
| (*QLsb*, *QMsb*): *Bit*, | |
| (*RamLsb*, *RamMsb*): *Bit*, | |
| (*CarryIn*, $\overline{OE}$): *Bit*, | |
| $Y$: $Bit^n$, | %Outputs |
| (*CarryOut*, $\overline{Gen}$, $\overline{Prop}$): *Bit*, | |
| (*FZero*, *FMsb*): *Bit*, | |
| *Ram*: $(Bit^n)^{16}$, | %Internal |
| $(Q, F, R, S)$: $Bit^n$ | |
| $n$: *positive* | %Parameters |

]

In the description of the bit slice, we represent the control input *Source* as a signal ranging over the elements of the set *sourceset*:

$$sourceset \quad =_{\text{def}} \quad '\{AQ, AB, ZQ, ZB, ZA, DA, DQ, DZ\}$$

The inputs *Func* and *Dest* range over similar sets:

$$funcset \quad =_{\text{def}} \quad '\{add, subr, subs, or, and, notrs, exor, exnor\}$$

$$destset \quad =_{\text{def}} \quad '\{qreg, nop, rama, ramf, ramqd, ramd, ramqu, ramu\}$$

The mnemonics are those used in the Am2901's data book description. A lower-level specification of the circuit can represent these fields as bit vectors. Similarly,

the approach taken here has the address fields *AAddr* and *BAddr* range over the integers $0, \ldots, 15$; a more detailed description can instead use bit vectors of length 4.

*Please note*: Throughout this description we refer to a vector $V$'s most significant bit as $V[0]$. The least significant bit is $V[n-1]$, where $n = |V|$. This is the *opposite* of the style used in the Am2901 data book but is consistent with the general convention taken elsewhere in this thesis.

*Definition of BitSlice:*

The slice's behavior can be broken down into separate parts for the random-access memory, Q-register, arithmetic unit and bus interface:

$$
\begin{aligned}
&\mathit{BitSlice}(N) \quad \equiv_{\mathrm{def}} \\
&\qquad \mathit{BitSliceStructure}(N) \\
&\qquad \wedge\ \mathit{RamPart}(N) \\
&\qquad \wedge\ \mathit{QRegPart}(N) \\
&\qquad \wedge\ \mathit{AluPart}(N) \\
&\qquad \wedge\ \mathit{BusPart}(N)
\end{aligned}
$$

## §11.1 Behavior of Random-Access Memory

The memory section has individual predicates for modifying the memory, the memory's end-bits *RamLsb* and *RamMsb* and the two output latches $A$ and $B$.

$$
\begin{aligned}
&\mathit{RamPart}(N) \quad \equiv_{\mathrm{def}} \\
&\qquad \mathit{SetRam}(\mathit{Ram}, \mathit{Dest}, \mathit{BAddr}, F, \mathit{RamLsb}, \mathit{RamMsb}, n) \\
&\qquad \wedge\ \mathit{SetRamLsbMsb}(\mathit{RamLsb}, \mathit{RamMsb}, \mathit{Dest}, F, n) \\
&\qquad \wedge\ \mathit{SetAB}(A, B, \mathit{Ram}, \mathit{AAddr}, \mathit{BAddr})
\end{aligned}
$$

*Definition of SetRam:*

In the description of the memory, we use the predicate *rdel* to refer to the unit-delay predicate *del* but with the operands reversed:

$$U \; rdel \; V \quad \equiv_{\text{def}} \quad V \; del \; U$$

Here is the predicate *SetRam* itself:

$$SetRam(Ram, Dest, BAddr, F, RamLsb, RamMsb, n) \quad \equiv_{\text{def}}$$
$$Ram \; rdel \begin{pmatrix} \textit{case Dest of} & \\ qreg: & Ram \\ nop: & Ram \\ rama: & alter(Ram, BAddr, F) \\ ramf: & alter(Ram, BAddr, F) \\ ramqd: & alter(Ram, BAddr, \langle RamMsb \rangle \,\|\, F[0 \text{ to } n-2]) \\ ramd: & alter(Ram, BAddr, \langle RamMsb \rangle \,\|\, F[0 \text{ to } n-2]) \\ ramqu: & alter(Ram, BAddr, F[1 \text{ to } n-1] \,\|\, \langle RamLsb \rangle) \\ ramu: & alter(Ram, BAddr, F[1 \text{ to } n-1] \,\|\, \langle RamLsb \rangle) \end{pmatrix}$$

Most of the operations alter the element of *Ram* selected by the input *BAddr*.

*Definition of SetRamLsbMsb:*

The predicate *SetRamLsbMsb* takes into account the high-impedance aspects (see section §4.12) of both end-bits *RamLsb* and *RamMsb*:

$$SetRamLsbMsb(RamLsb, RamMsb, Dest, F, n) \quad \equiv_{\text{def}}$$
$$\begin{pmatrix} \textit{case Dest of} & \\ qreg: & true \\ nop: & true \\ rama: & true \\ ramf: & true \\ ramqd: & RamLsb = F[n-1] \\ ramd: & RamLsb = F[n-1] \\ ramqu: & RamMsb = F[0] \\ ramu: & RamMsb = F[0] \end{pmatrix}$$

***Definition of SetAB:***

**The latch $A$ always equals the memory word addressed by $AAddr$. A similar relation holds between $B$ and $BAddr$.**

$$SetAB(A, B, Ram, AAddr, BAddr) \quad \equiv_{\text{def}}$$
$$(A \approx Ram[AAddr]) \wedge (B \approx Ram[BAddr])$$

## §11.2 Behavior of Q-Register

**The description of the Q-register has a predicate $SetQ$ for $Q$ and another predicate $QLsbMsb$ for using the end-bits $QLsb$ and $QMsb$.**

$$QRegPart(N) \quad \equiv_{\text{def}}$$
$$SetQ(Q, Dest, F, QLsb, QMsb, n)$$
$$\wedge \; SetQLsbMsb(QLsb, QMsb, Dest, Q, n)$$

*Definition of SetQ:*

$$SetQ(Q, Dest, F, QLsb, QMsb, n) \quad \equiv_{\text{def}}$$
$$Q \; rdel \begin{pmatrix} \text{case } Dest \text{ of} & \\ \quad qreg: & F \\ \quad nop: & Q \\ \quad rama: & Q \\ \quad ramf: & Q \\ \quad ramqd: & \langle QMsb \rangle \,\|\, Q[0 \text{ to } n-2] \\ \quad ramd: & Q \\ \quad ramqu: & Q[1 \text{ to } n-1] \,\|\, \langle QLsb \rangle \\ \quad ramu: & Q \end{pmatrix}$$

*Definition of SetQLsbMsb:*

Both end-bits $QLsb$ and $QMsb$ can float in a state of high impedance (see section §4.12). This is taken care of in the following predicate:

$$SetQLsbMsb(QLsb, QMsb, Dest, Q, n) \equiv_{\text{def}}$$

$$\boxdot\begin{pmatrix} \textit{case Dest of} & \\ \quad qreg: & true \\ \quad nop: & true \\ \quad rama: & true \\ \quad ramf: & true \\ \quad ramqd: & QLsb = Q[n-1] \\ \quad ramd: & QLsb = Q[n-1] \\ \quad ramqu: & QMsb = Q[0] \\ \quad ramu: & QMsb = Q[0] \end{pmatrix}$$

## §11.3 Behavior of Arithmetic Logic Unit

The arithmetic logic unit's specification has predicates associated with the many signals originating in this part of the slice.

$$AluPart(N) \equiv_{\text{def}}$$

$$SetRS(R, S, Source, A, B, D, Q, n)$$

$$\wedge\ SetF(F, Func, R, S, CarryIn, n)$$

$$\wedge\ SetCarryOut(CarryOut, Func, R, S, CarryIn, n)$$

$$\wedge\ SetOverflow(Overflow, Func, R, S, CarryIn, n)$$

$$\wedge\ Set\overline{Gen}(\overline{Gen}, Func, R, S, n)$$

$$\wedge\ Set\overline{Prop}(\overline{Prop}, Func, R, S, n)$$

$$\wedge\ SetFZeroFMsb(FZero, FMsb, F, n)$$

*Definition of SetRS:*

$$SetRS(R, S, Source, A, B, D, Q, n) \quad \equiv_{\mathrm{def}}$$
$$\langle R, S\rangle \approx \begin{pmatrix} \textit{case Dest of} \\ \quad AQ\colon \langle A, Q\rangle \\ \quad AB\colon \langle A, B\rangle \\ \quad ZQ\colon \langle zero, Q\rangle \\ \quad ZB\colon \langle zero, B\rangle \\ \quad ZA\colon \langle zero, A\rangle \\ \quad DA\colon \langle D, A\rangle \\ \quad DQ\colon \langle D, Q\rangle \\ \quad DZ\colon \langle D, zero\rangle \end{pmatrix}$$

where $zero = \langle 0\rangle^n$, that is, a sequence consisting of $n$ repetitions of 0.

*Definition of SetF:*

The following predicate shows arithmetic behavior for bit-vectors representing unsigned numbers:

$$SetF(F, Func, R, S, CarryIn, n) \quad \equiv_{\mathrm{def}}$$
$$\boxdot \begin{pmatrix} \textit{case Func of} & \\ \quad add\colon & nval(F) = [nval(R) + nval(S) + CarryIn] \bmod 2^n \\ \quad subr\colon & nval(F) = [nval(\neg R) + nval(S) + CarryIn] \bmod 2^n \\ \quad subs\colon & nval(F) = [nval(R) + nval(\neg S) + CarryIn] \bmod 2^n \\ \quad or\colon & F = (R \vee S) \\ \quad and\colon & F = (R \wedge S) \\ \quad notrs\colon & F = ([\neg R] \wedge S) \\ \quad exor\colon & F = (R \oplus S) \\ \quad exnor\colon & F = \neg(R \oplus S) \end{pmatrix}$$

Here the operator $\oplus$ represents exclusive-or. The Boolean operations such as $R \wedge S$ are applied bitwise to the vectors. The table can be augmented with information about arithmetic operations using one's and two's-complement representations.

*Definition of SetCarryOut:*

In the case-expression given below, hyphens indicate unspecified entries and are not partial values; a more detailed description could fill them in. The function *carry* determines the resulting carry output and is defined in section §6.4 in the discussion of carry look-ahead adders.

$$SetCarryOut(CarryOut, Func, R, S, CarryIn, n) \quad \equiv_{\text{def}}$$

$$CarryOut \approx \begin{pmatrix} \textit{case Func of} & \\ add: & carry(n, nval(R), nval(S), CarryIn) \\ subr: & carry(n, nval(\neg R), nval(S), CarryIn) \\ subs: & carry(n, nval(R), nval(\neg S), CarryIn) \\ or: & - \\ and: & - \\ notrs: & - \\ exor: & - \\ exnor: & - \end{pmatrix}$$

*Definition of SetOverflow:*

In determining the overflow bit's value, the two's-complement interpretations of the incoming bit vectors $R$ and $S$ are used. The function $tcval(\vec{X})$ takes a bit vector $\vec{X}$ and computes its numerical value based on representation by two's complement:

$$tcval(\vec{X}) \quad =_{\text{def}} \quad \textit{if } \vec{X}[0] = 0 \textit{ then } nval(\vec{X}) \textit{ else } -[2^{|\vec{X}|} - nval(\vec{X})]$$

$$SetOverflow(Overflow, Func, R, S, CarryIn, n) \quad \equiv_{\text{def}}$$

$$Overflow \approx \begin{pmatrix} \textit{case Func of} & \\ add: & overflow(n, tcval(R), tcval(S), CarryIn) \\ subr: & overflow(n, tcval(\neg R), tcval(S), CarryIn) \\ subs: & overflow(n, tcval(R), tcval(\neg S), CarryIn) \\ or: & - \\ and: & - \\ notrs: & - \\ exor: & - \\ exnor: & - \end{pmatrix}$$

**Here the function *overflow* equals 1 iff two's-complement overflow is occuring and is defined as follows:**

$$overflow(n, i, j, ci) \quad =_{\text{def}} \quad \textit{if } -2^{n-1} \leq (i + j + ci) \leq 2^{n-1} - 1 \textit{ then } 0 \textit{ else } 1$$

**Both parameters $i$ and $j$ can range over negative and nonnegative integers.**

*Definition of $Set\overline{Gen}$ and $Set\overline{Prop}$:*

The predicates $Set\overline{Gen}$ and $Set\overline{Prop}$ describe the bit slice's carry-lookahead signals. The functions *carrygen* and *carryprop* are defined in section §6.4.

$$Set\overline{Gen}(\overline{Gen}, Func, R, S, n) \quad \equiv_{\text{def}}$$

$$\overline{Gen} \approx \begin{pmatrix} \textit{case Func of} & \\ add: & \neg carrygen(n, nval(R), nval(S)) \\ subr: & \neg carrygen(n, nval(\neg R), nval(S)) \\ subs: & \neg carrygen(n, nval(R), nval(\neg S)) \\ or: & - \\ and: & - \\ notrs: & - \\ exor: & - \\ exnor: & - \end{pmatrix}$$

$$Set\overline{Prop}(\overline{Prop}, Func, R, S, n) \quad \equiv_{\text{def}}$$

$$\overline{Prop} \approx \begin{pmatrix} \textit{case Func of} & \\ add: & \neg carryprop(n, nval(R), nval(S)) \\ subr: & \neg carryprop(n, nval(\neg R), nval(S)) \\ subs: & \neg carryprop(n, nval(R), nval(\neg S)) \\ or: & - \\ and: & - \\ notrs: & - \\ exor: & - \\ exnor: & - \end{pmatrix}$$

*Definition of $SetFZeroFMsb$:*

**The values of the bit signals $FZero$ and $FMsb$ are derived from $F$:**

$$SetFZeroFMsb(FZero, FMsb, F, n) \quad \equiv_{\text{def}}$$

$$(FZero \approx \textit{if } [F = \langle 0 \rangle^n] \textit{ then } 1 \textit{ else } 0) \wedge (FMsb \approx F[0])$$

## §11.4 Behavior of Bus Interface

$$BusPart(N) \quad \equiv_{\text{def}}$$
$$SetY(Y, Dest, F, A, \overline{OE})$$

*Definition of SetY:*

When the signal $\overline{OE}$ equals 0, the bus interface $Y$ is enabled and receives a value according to the case formula. When the bus interface is disabled with $\overline{OE}$ equaling 1, $Y$'s behavior is left unspecified, thus modeling the effects of high impedance.

$$SetY(Y, Dest, F, A, \overline{OE}) \quad \equiv_{\text{def}}$$
$$\Box[(\overline{OE} = 0) \supset (Y = \begin{pmatrix} \textit{case Dest of} \\ qreg: \; F \\ nop: \; F \\ rama: \; A \\ ramf: \; F \\ ramqd: F \\ ramd: \; F \\ ramqu: F \\ ramu: \; F \end{pmatrix})]$$

## §11.5 Composition of Two Bit Slices

The predicate *CombineTwoBitSlices* describes how to combine two bit-slices in parallel to form a larger one. The bit slice $M$ contains the more significant bits and $L$ contains the less significant ones.

$$CombineTwoBitSlices(M, L) \quad \equiv_{\text{def}}$$
$$BitSlice(M) \wedge BitSlice(L)$$
$$\wedge\; M[field] \approx L[field],$$
$$\text{for } field \in \text{'}\{Source, Func, Dest, AAddr, BAddr, \overline{OE}\}$$
$$\wedge\; (M.RamLsb \approx L.RamMsb) \wedge (M.QLsb \approx L.QMsb)$$
$$\wedge\; (M.CarryIn \approx L.CarryOut)$$

The next property expresses how the implementation's various signals are mapped to the overall bit slice:

$$\models \quad \mathit{CombineTwoBitSlices}(M, L) \supset \mathit{BitSlice}(N)$$

where the tuple $N$ is constructed as follows:

$$
\begin{aligned}
N[\mathit{field}] &\approx M[\mathit{field}], \\
&\qquad \text{for } \mathit{field} \in \text{'}\{\mathit{Source}, \mathit{Func}, \mathit{Dest}, \mathit{AAddr}, \mathit{BAddr}, \overline{\mathit{OE}}\} \\
N[\mathit{field}] &\approx M[\mathit{field}], \quad \text{for } \mathit{field} \in \text{'}\{\mathit{QMsb}, \mathit{RamMsb}, \mathit{CarryOut}, \mathit{FMsb}\} \\
N[\mathit{field}] &\approx L[\mathit{field}], \quad \text{for } \mathit{field} \in \text{'}\{\mathit{QLsb}, \mathit{RamLsb}, \mathit{CarryIn}\} \\
N[\mathit{field}] &\approx M[\mathit{field}] \,||\, L[\mathit{field}], \\
&\qquad \text{for } \mathit{field} \in \text{'}\{D, Y, Q, F, R, S\} \\
N.\mathit{Ram}[i] &\approx M.\mathit{Ram}[i] \,||\, L.\mathit{Ram}[i], \quad \text{for } 0 \le i \le 15 \\
N.\overline{\mathit{Gen}} &\approx [M.\overline{\mathit{Gen}} \wedge (M.\overline{\mathit{Prop}} \vee L.\overline{\mathit{Gen}})] \\
N.\overline{\mathit{Prop}} &\approx (M.\overline{\mathit{Prop}} \vee L.\overline{\mathit{Prop}}) \\
N.\mathit{FZero} &\approx (M.\mathit{FZero} \wedge L.\mathit{FZero}) \\
N.n &= M.n + L.n
\end{aligned}
$$

## §11.6 Timing Details

The predicate *BitSlice* presented here contains little quantitative information about timing. For example, the bit slice's clock input is not mentioned. One way to include timing details is by giving behavioral descriptions at a level similar to those discussed in previous chapters. For example, the arithmetic unit can be specified in a manner similar to that used in the predicates *BasicAdder*, *DetailedAdder* and *CarryLookAheadAdder*. A predicate such as *ShiftRegister* can be modified to capture the behavior of the Q-register.

CHAPTER 12

# DISCUSSION

## §12.1 Related Work

We now mention some related research on the semantics of hardware. Gordon's work [15,16] on register-transfer systems uses a denotational semantics with partial values to provide a concise means for reasoning about clocking, feedback, instruction-set implementation and bus communication. Talantsev [47] as well as Betancourt and McCluskey [7] examine qualitative signal transition concepts corresponding to $\uparrow X$ and $\downarrow X$. Wagner [49] also uses such constructs as $\uparrow X$ in a semi-automated proof development system for reasoning about signal transitions and register transfer behavior. Malachi and Owicki [28] utilize a temporal logic to model self-timed digital systems by giving a set of axioms. Bochmann [9] uses a linear-time temporal logic to describe and verify properties of an arbiter, a device for regulating access to shared resources. The presentation reveals some tricky aspects in reasoning about such components.

Leinwand and Lamdan [26] present a type of Boolean algebra for modeling signal transitions. Applications include systems with feedback and critical timing constraints. Patterson [36] examines the verification of firmware from the standpoint of sequential programming. Meinen [33] discusses a semantics of register transfer behavior. McWilliams [27] develops computational techniques for determining timing constraints in hardware. Eveking [13] uses predicate calculus with explicit time variables to explore verification in the hardware specification language Conlan.

# CHAPTER 12—DISCUSSION

A number of people have used temporal logics to describe computer communication protocols [18,25,40]. Bernstein and Harter [6] augment linear-time temporal logic with a construct for expressing that one event is followed by another within some specified time range. This facilitates the treatment of various quantitative timing issues. Recently Schwartz et al. [41] have introduced a temporal logic for reasoning about intervals. They distinguish intervals from propositions.

The research mentioned above has made large strides in developing a semantics of digital systems. However, for our purposes much of this work either has difficulties in treating quantitative timing, lacks rigor, is unintuitive or does not easily generalize. This seems unavoidable due to the magnitude of the problem area. We note that the computational models used in works on temporal logic generally interleave the executions of different processes. In the treatment of digital circuits, this approach seems inappropriate. We have chosen instead to model true parallelism. The semantics of the connective *logical-and* ( $\wedge$ ) appear to directly correspond to this.

It might seem that temporal logic is simply a subset of dynamic logic [19,37]. However, once interval-dependent constructs are added, this is no longer the case. Operators such as *semicolon* and *yields* are not directly expressible in dynamic logic. Furthermore, the descriptive styles used in dynamic logic and temporal logic differ rather greatly. Dynamic logic and process logics [11,20,38] stress the interaction between programs and propositions. ITL is expressive enough to conveniently and directly specify a variety of programs containing such constructs as assignments, while-loops and procedures. Our current view is that the addition of program variables would be redundant.

Lamport [25] feels that temporal logic is a valuable tool but advocates against the use of the operator *next* by claiming that this introduces unnecessary granularity into the reasoning process. We do not agree and believe that explicit access to discrete state transitions is invaluable when dealing with such concepts as iteration and feedback. Furthermore, temporal logic appears to be flexible enough to facilitate projecting out critical points in a computation so as to ignore intermediate states. Thus, specifications and theorems that assume a certain degree of atomicity can be

generalized. If temporal logic is itself used as a programming language, constructs such as *del* that are based on $\bigcirc$ occupy a snug and secure place in the overall formalism.

## §12.2 Future Research Directions

There are many aspects of interval temporal logic that require more investigation. We now point out a few.

### Proof theory

All the valid properties presented in this thesis have been justified on the basis of ITL's semantics. Work should be done on suitably axiomatizing various parts of the logic and automating some of the proof process. For example, if bit signals are represented as truth values, simple versions of temporal constructs such as stability (*stb*) and unit delay (*del*) can be expressed and reasoned about using existing propositional linear-time temporal logics [14] and their axiomatizations and decision procedures. Using a program written by Frank Yellin, we have already automatically established properties such as the following:

$$\models \quad [\uparrow X \land \uparrow Y] \supset \uparrow(X \land Y)$$

$$\models \quad (X \; del \; X) \equiv stb \, X$$

### Some variants of temporal logic

There are a variety of operators and concepts that can be added to temporal logic. We discuss some here.

*Ignoring intervals*

Many of the concepts presented here can generally be expressed in linear-time temporal logic [31] with $\bigcirc$, $\Box$, $\Diamond$ and $\mathcal{U}$. In section §2.4 we gave a linear translation from local propositional ITL to linear-time temporal logic with quantification.

However, the clarity and modularity provided by *semicolon* and other interval-dependent constructs is often lost. A more detailed understanding of the various tradeoffs involved and the proper roles of different temporal logics should be developed.

*Infinite intervals*

In the semantics already given, all intervals are restricted to being finite. It can however be advantageous to consider infinite intervals arising out of nonterminating computations. As we mentioned in section §2.4, the inclusion of such intervals does not alter the complexity of satisfiability.

*Traces*

The *trace* of a signal $A$ in an interval $s_0 \ldots s_n$ can be defined as the sequence of values that $A$ assumes:

$$trace(A) = \langle (\bigcirc^i A): 0 \leq i \leq len \rangle,$$

that is,

$$trace(A) = \langle \bigcirc^0 A, \bigcirc^1 A, \ldots, \bigcirc^{len} A \rangle$$

In an interval of length $n$, the trace of a variable has length $n+1$.

The following property shows how to express unit delay by comparing the traces of the input and output:

$$\models \quad (A \; del \; B) \;\equiv\; \big[trace(A)[0 \text{ to } len-1] \;=\; trace(B)[1 \text{ to } len]\big]$$

It would be interesting to compare the use of traces with other styles of specification.

*Projection*

Sometimes it is desirable to examine the behavior of a device at certain points in time and ignore all intermediate states. This can be done using the idea of *temporal projection*. The formula $w_1 \, \Pi \, w_2$ in an interval forms a subinterval consisting of

those states where $w_1$ is true and then determines the value of $w_2$ in this subinterval:

$$\mathcal{M}_{s_0 \ldots s_n}[\![w_1 \, \Pi \, w_2]\!] = \mathcal{M}_{t_0 \ldots t_m}[\![w_2]\!],$$

where $t_0 \ldots t_m$ is the sequence of the states in $s_0 \ldots s_n$ that satisfy $w_1$:

$$\mathcal{M}_{t_i}[\![w_1]\!] = \mathit{true}, \quad \text{for } 0 \le i \le m$$

Note that $t_0 \ldots t_m$ need not be a contiguous subsequence of $s_0 \ldots s_n$. If no states can be found, the projection is vacuously *true*. In the semantics given here, the formula $w_1$ examines states, not intervals. For example, the formula

$$(X = 1) \, \Pi \, \mathit{stb}\, A$$

is true if $A$ has a constant value throughout the states where $X$ equals 1. Variables like $X$ act as markers for measuring time and facilitate different levels of atomicity. If two parts of a system are active at different times or are running at different rates, markers can be constructed to project away the asynchrony. Other styles of projection are also possible. For example, a "synchronous" form of projection might require the marker to be true in the initial and final states of an interval.

In section §2.3 we showed how to express the iterative construct $w^*$ by means of a marker $P$:

$$w^* \quad \equiv_{\mathrm{def}} \quad \exists P.\big(\mathit{beg}\, P \ \wedge\ \boxdot\big[\mathit{beg}\, P \supset (\mathit{empty} \ \vee\ \diamondsuit[w \ \wedge\ \bigcirc\, \mathit{halt}\; \mathit{beg}\, P])\big]\big)$$

This provides a general means for identifying the end points of the iteration steps and extracting them using projection. It is even desirable to have variants of the iteration constructs for making markers explicit. For example, the extended while-loop

$$\mathit{while}_P\ Q\ \mathit{do}\ R$$

indicates that $P$ marks off individual steps. Other constructs such as *next* and *trace* can have marker-oriented variants.

We feel that low-level clocking and propagation details in digital circuits can be more effectively decoupled from high-level functional behavior through the introduction of markers and projection. The Am2901 bit slice discussed in chapter 11 might be a good test of this hypothesis.

*Additional modifications*

Further possible extensions include interval temporal logics based on branching or probabilistic models of time. Operators for reversing or expanding an interval may also turn out to be useful.

## Temporal types and higher-order temporal objects

A theory of temporal types needs to be developed. This should provide various ways of constructing and comparing types. For example, the predicate $p^*$ is true for vectors of arbitrary, possibly null length whose elements all satisfy $p$. Thus, the type $bit^*$ is true for all bit-vectors. The type $sig(bit^*)$ is true for any bit vector signal with a possibly varying length. The temporal type $Bit^*$ requires that the signal's length be fixed over time:

$$\models \quad A\colon Bit^* \equiv [A\colon sig(bit^*) \wedge stb|A|]$$

We hope to permit parameterized types such as $sig(s \times t)$, where $s$ and $t$ are type-valued variables. Operators for such things as unioning or recursively defining types also need to be developed. Perhaps the techniques needed here can be made general enough so that any unary predicate can be viewed as a type.

It would be interesting to have a semantics of higher-order temporal objects such as time-dependent functionals. Perhaps a suitable variant of proposition ITL can facilitate some sort of Gödelization by representing all values as temporal formulas. Alternatively, an encoding like that used by Scott [42,45] in developing a model of the typeless lambda calculus might work. However, we wish to strongly resist the introduction of partial values. One concession we make in this direction is to not require that every function have a fixed point.

## Temporal logic as a programming language

Temporal logic can be used directly as a programming language. For example, the formula

$$beg(I = 0) \wedge [(I + 1)\ del\ I] \wedge halt(I = 5)$$

can be viewed operationally as initializing $I$ to 0, and then incrementing $I$ by 1 over each computation step until $I$ equals 5. At that instant, the computation halts. This style of temporal programming is similar to the language Lucid [2,4] developed by Ashcroft and Wadge. Note that the formula given above has the same semantics has the following:

$$beg(I = 0) \wedge while\,(I \neq 5)\; do\,(skip \wedge [I + 1 \rightarrow I])$$

This illustrates how by using ITL we can compare different ways of expressing the same computation.

In general, if $w_1$ and $w_2$ are temporal formulas, the combined form $w_1 \wedge w_2$ operationally specifies that $w_1$ and $w_2$ be run in parallel. Note that $w_1$ and $w_2$ are implicitly synchronized to start and finish at the same time. Similarly, the formula $w_1; w_2$ involves running $w_1$ and then $w_2$. For example, the formula

$$([0 \rightarrow I] \wedge [0 \rightarrow J]); while\,(I \neq n)\; do\,([I + 1 \rightarrow I] \wedge [J + I \rightarrow J])$$

clears $I$ and $J$ and then repeatedly increments $I$ and simultaneously sums $I$ into $J$. Asynchronous operations can also be handled. For instance, the formula

$$(stb\, I \wedge halt[X = 1]); [(I + 1)\; del\; I]$$

leaves $I$ stable until the flag $X$ equals 1 and then keeps increasing $I$ by 1.

Manna and Moszkowski [29,30] describe how to reason about programming concepts in ITL and also present a prototype programming language called *Tempura* that is based on the ideas just given. Along with the programming languages Lucid and Prolog [24], Tempura has the property of having a semantics based on logic. Much work remains ahead in exploring this temporal approach to language design and developing practical techniques for specifying, executing, transforming, synthesizing and verifying Tempura programs. We strongly feel that there is a large potential for the cross-fertilization of ideas arising from simultaneously using temporal logic as a hardware specification tool and as a basis for general-purpose programming languages. It also appears worthwhile to examine interpreters and

other systems that transmit and manipulate commands and programs. Perhaps the state sequences of temporal logic can also be used as a convenient basis for logics of, say, formal languages, typesetting and music. More generally, temporal logic may provide a semantics of both time and space.

### Hardware

The largest device considered in this thesis is the Am2901 bit slice; there is clearly no reason to stop at that. Future work will explore microprocessors, pipelines, buses and protocols, DMA, firmware and instruction sets, as well as the combined semantics of hardware and software. The treatment of specific areas such as fault-analysis also seems worthwhile. It would be interesting to see how suitable ITL is as a tool for teaching the basic operation of digital circuits covered in such textbooks as Gschwind and McCluskey [17] and Hill and Peterson [21]. The feasibility of hardware-oriented simulation languages based on subsets of ITL should certainly be investigated. For example, propositional ITL can be used for bit-valued signals.

## §12.3 Conclusion

Standard temporal logics and other such notations are not designed to concisely handle the kinds of quantitative timing properties, signal transitions and structural information occurring in the examples considered. Temporal intervals provide a unifying means for presenting a wide range of digital devices and concepts. Interval temporal logic can be used for both specifying and reasoning about circuits and their properties. The same formalism that handles devices with clock signals, set-up constraints and hold times can also deal with high-level algorithms. The omission of partial values does not appear to restrict the generality of specifications; even high-impedance can be treated.

The future seems bright. Let us therefore conclude this thesis with the conjecture that temporal logics will be around for a long interval to come.

# Bibliography


1. Advanced Micro Devices, Inc. *Bipolar Microprocessor Logic and Interface Data Book.* Sunnyvale, California, 1981.

2. E. A. Ashcroft and W. W. Wadge. "Lucid: A formal system for writing and proving programs." *SIAM Journal of Computing 5,* 3 (September 1976), 336–354.

3. E. A. Ashcroft and W. W. Wadge. "Lucid, a nonprocedural language with iteration." *Communications of the ACM 20,* 7 (July 1977), 519–526.

4. E. A. Ashcroft and W. W. Wadge. *Lucid, the Data Flow Programming Language.* To be published.

5. M. R. Barbacci. "Instruction Set Processor Specifications (ISPS): The notation and its applications." *IEEE Transactions on Computers C-30,* 1 (January 1981), 24–40.

6. A. Bernstein and P. Harter. "Proving real-time properties of programs with temporal logic." Proceedings of the Eighth Symposium on Operating Systems Principles, Pacific Grove, California, December, 1981.

7. R. Betancourt and E. J. McCluskey. Analysis of sequential circuits using clocked flip-flops. Technical Report 82, Digital Systems Laboratory, Stanford University, August, 1975.

8. D. F. Blunden, A. H. Boyce, and G. Taylor. "Logic simulation, parts I and II." *The Marconi Review 40,* 206 (1977), 157–171 and 236–254.

9. G. V. Bochmann. "Hardware specification with temporal logic: An example." *IEEE Transactions on Computers C-31,* 3 (March 1982), 223–231.

10. M. A. Breuer and A. D. Friedman. *Diagnosis and Reliable Design of Digital Systems.* Computer Science Press, Inc., Woodland Hills, California, 1976.

11. A. Chandra, J. Halpern, A. Meyer, and R. Parikh. Equations between regular terms and an application to process logic. Proceedings of the 13-th Annual ACM Symposium on Theory of Computing, Milwaukee, Wisconsin, May, 1981, pages 384–390.

12. H. B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, New York, 1972.

13. H. Eveking. The application of Conlan assertions to the correct description of hardware. Proceedings of the IFIP TC–10 Fifth International Conference on Computer Hardware Description Languages and their Applications, Kaiserslautern, West Germany, September, 1981, pages 37–50.

14. D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, January, 1980, pages 163–173.

15. M. Gordon. Register transfer systems and their behavior. Proceedings of the IFIP TC–10 Fifth International Conference on Computer Hardware Description Languages and their Applications, Kaiserslautern, West Germany, September, 1981, pages 23–36.

16. M. Gordon. A model of register transfer systems with applications to microcode and VLSI correctness. Department of Computer Science, University of Edinburgh, 1981.

17. H. W. Gschwind and E. J. McCluskey. *Design of Digital Computers.* Springer-Verlag, New York, 1975.

18. B. T. Hailpern and S. Owicki. Verifying network protocols using temporal logic. Technical Report 192, Computer Systems Laboratory, Stanford University, June, 1980.

19. D. Harel. *First-Order Dynamic Logic.* Springer-Verlag, Berlin, 1979. Number 68 in the series *Lecture Notes in Computer Science.*

20. D. Harel, D. Kozen, and R. Parikh. "Process logic: Expressiveness, decidability,

completeness." *Journal of Computer and System Sciences*, 25, 2 (October 1982), 144–170.

21. F. J. Hill and G. R. Peterson. *Introduction to Switching Theory and Logical Design*. John Wiley and Sons, New York, 1981.

22. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, Reading, Mass., 1979.

23. G. E. Hughes and M. J. Cresswell. *An Introduction to Modal Logic*. Methuen and Co., Ltd., London, 1968.

24. R. Kowalski. *Logic for Problem Solving*. Elsevier North Holland, Inc., New York, 1979.

25. L. Lamport. "Specifying concurrent program modules." *ACM Transactions on Programming Languages and Systems 5*, 2 (April 1983), 190–222.

26. S. Leinwand and T. Lamdan. Algebraic analysis of nondeterministic behavior. Proceedings of the 17-th Design Automation Conference, Minneapolis, June, 1980, pages 483–493.

27. T. M. McWilliams. Verification of timing constraints on large digital systems. Proceedings of the 17-th Design Automation Conference, Minneapolis, June, 1980, pages 139–147.

28. Y. Malachi and S. S. Owicki. Temporal specifications of self-timed systems. In H.T. Kung, B. Sproul, and G. Steele, editors, *VLSI Systems and Computations*, pages 203–212, Computer Science Press, Inc., Rockville, Maryland, 1981.

29. Z. Manna and B. Moszkowski. Reasoning in interval temporal logic. To appear in Proceedings of ACM/NSF/ONR Workshop on Logics of Programs, June, 1983.

30. Z. Manna and B. Moszkowski. Temporal logic as a programming language. In preparation.

31. Z. Manna and A. Pnueli. Verification of concurrent programs: The temporal framework. In R. S. Boyer and J. S. Moore, editors, *The Correctness Problem in Computer Science*, pages 215–273, Academic Press, New York, 1981.

32. C. Mead and L. Conway. *Introduction to VLSI Systems.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1980.

33. P. Meinen. Formal semantic description of register transfer language elements and mechanized simulator construction. Proceedings of the 4-th International Symposium on Computer Hardware Description Languages, Palo Alto, California, October, 1979, pages 69–74.

34. J. Mick and J. Brick. *Bit-Slice Microprocessor Design.* McGraw Hill, New York, 1980.

35. A. C. Parker and J. J. Wallace. "SLIDE: An I/O hardware description language." *IEEE Transactions on Computers C–30,* 6 (June 1981), 423–439.

36. D. A. Patterson. "Strum: Structured microprogram development system for correct firmware." *IEEE Transactions on Computers C–25,* 10 (October 1976), 974– 985.

37. V. R. Pratt. Semantical considerations on Floyd-Hoare logic. 17-th Annual IEEE Symposium on Foundations of Computer Science, Houston, Texas, October, 1976, pages 109–121.

38. V. R. Pratt. Process Logic. Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, January, 1979, pages 93–100.

39. N. Rescher and A. Urquart. *Temporal Logic.* Springer-Verlag, New York, 1971.

40. R. L. Schwartz and P. M. Melliar-Smith. Temporal logic specification of distributed systems. Proceedings of the Second International Conference on Distributed Computing Systems, Paris, France, April, 1981, pages 446–454.

41. R. L. Schwartz, P. M. Melliar-Smith, and F. H. Vogt. An interval logic for higher-level temporal reasoning: Language definition and examples. Technical

Report CSL–138, Computer Science Laboratory, SRI International, February, 1983.

42. D. Scott. "Data types as lattices." *SIAM Journal of Computing 5*, 3 (September 1976), 522–587.

43. D. P. Siewiorek, C. G. Bell and A. Newell. *Computer Structures: Principles and Examples.* McGraw-Hill Book Company, New York, 1982.

44. L. J. Stockmeyer. *The Complexity of Decision Problems in Automata Theory and Logic.* PhD Thesis, MIT, July, 1974. Available as Project MAC Technical Report 133.

45. J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, Masachusetts, 1977.

46. S. Y. H. Su, C. Huang, and P. Y. K. Fu. A new multi-level hardware design language (LALSD II) and translator. Proceedings of the IFIP TC–10 Fifth International Conference on Computer Hardware Description Languages and their Applications, Kaiserslautern, West Germany, September, 1981, pages 155–169.

47. A. D. Talantsev. "On the analysis and synthesis of certain electrical circuits by means of special logical operators." *Automation and Remote Control 20*, 7 (July 1959), 874–883.

48. Texas Instruments, Inc. *The TTL Data Book for Design Engineers.* Second edition, Dallas, Texas, 1976.

49. T. Wagner. *Hardware Verification.* PhD Thesis, Department of Computer Science, Stanford University, 1977.

50. P. L. Wolper. *Specification and Synthesis of Communicating Processes Using an Extended Temporal Logic.* PhD Thesis, Department of Computer Science, Stanford University, 1982.

51. P. L. Wolper, M. Y. Vardi, A. P. Sistla. Reasoning about infinite computation paths. In preparation.