Performance Analysis and Optimization of Asynchronous Circuits

Thesis by

Steven Morgan Burns

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California USA

1991

(Defended December 5, 1990)

© 1991

Steven Morgan Burns

All rights reserved

Acknowledgments

I would first like to thank the members of my thesis defense committee, Chuck Seitz, Mani Chandy, Joel Franklin, Jan van de Snepscheut, and, my thesis advisor, Alain Martin, for the time and effort they expended in reading and criticizing my work. I thankfully acknowledge the funding support of DARPA and IBM.

I had the great pleasure of working with Alain, in an exciting, new field during my years at Caltech. I wish to thank the members of his research group that overlapped with me: Peggy Li, Kevin Van Horn, Blake Lewis, Pieter Hazewindus, Tony Lee, Dražen Borković, Marcel van der Goot, and Jose Tierno. Their constructive criticism of some early ideas greatly improved the quality of this thesis. I wish to thank the entire group, but especially, Marcel, Pieter, and Alain, for the time they spent carefully reading this thesis as well as other manuscripts. I will miss the weekly three-hour group meetings.

My graduate residence at Caltech would not have been as pleasant (although probably shorter) without the close friendship of: Nan and Andy Boden, Steve DeWeerth and Valerie, Emily, and Erik Patterson, Mass Sivilotti and Ruth Erlanson, Mary Ann Maher, Zoya Popović, Michael Emerling and Ruth Ballinger, John, Linda and Melissa Tanner, John and Julia Snyder, Pieter Hazewindus, and Andy Fyfe. "Walleyball", volleyball, basketball, and softball provided an excellent means to vent anger and have fun. I thank the Tanner's and the Snyder's for the time and effort required to organize those activities.

To Phil, Jean, Richard, Nancy, and Cathy, with whom I share a half,

and to Helen, Owen, and Gracie, with whom I share a quarter.

Abstract

Analytical techniques are developed to determine the performance of asynchronous digital circuits. These techniques can be used to guide the designer during the synthesis of such a circuit, leading to a high-performance, efficient implementation. Optimization techniques are also developed that further improve this implementation by determining the optimal sizes of the low-level devices (CMOS transistors) that compose the circuit.

Contents

1 Introduction 1

1.1 Synthesis, Analysis, and Optimization 4

1.2 Contributions 6

1.3 Thesis Overview 8

2 Event-Rule Systems 9

2.1 General Event-Rule Systems 9

2.2 Repetitive Systems 16

2.2.1 Linear Timing Functions 20

2.2.2 Strongly Connected Systems 21

2.2.3 The Cycle Period As a Performance Metric 22

2.3 Pseudorepetitive Systems 24

2.3.1 Definitions 24

2.3.2 Approximating the Timing Simulation 26

2.4 Minimum-Period Linear Timing Functions 29

2.4.1 Objective Function 32

2.4.2 Cycle Vectors of a Graph 34

2.4.3 Approximating the Timing Simulation 39

2.5 Fast Algorithms 42

2.5.1 Graph Transformations 42
2.5.2 Primal–Dual Method 44
2.6 Summary and Related Work 53

3 The Synthesis Method **57**
3.1 Notation and Intermediate Forms 57
3.1.1 Communicating Sequential Processes 58
3.1.2 Handshaking Expansions 59
3.1.3 Production Rules 60
3.1.4 Quasi-Delay-Insensitive Circuits 61
3.2 Transformations 64
3.2.1 Process Decomposition 64
3.2.2 Reshuffling 65
3.2.3 Decomposition into Control and Data Parts 65
3.2.4 Multiple-Bit Datapaths and Completion Trees 67
3.2.5 Strengthening the Production Rules 70
3.2.6 Introducing State Variables 70
3.2.7 Bubble Shuffling 70
3.2.8 Decomposition of Large Elements 71
3.2.9 Resetting to the Initial State 71

4 Modeling Programs with *ER* Systems **72**
4.1 Modeling Delays 73
4.2 Modeling Handshaking Expansions 74
4.2.1 Straight-Line Handshaking Expansions 77
4.2.2 Communication Between Processes 85
4.2.3 Conversion From a General *ER* System to a Pseudorepetitive *ER* System 85

4.2.4 Vacuous Firings 88
4.2.5 Other Initial States 89
4.2.6 Multiple Assignments 89
4.2.7 General Parallel Composition Statements 90
4.2.8 Conversion to Simple Repetitive Systems 90
4.3 Modeling Production Rule Sets 91
4.4 Data-Dependent Computations 96
4.5 Inherently Disjunctive Computations 98

5 Linear Arrays of Processes **102**
5.1 Regular Systems 102
5.1.1 Linear Timing Function 105
5.1.2 Linear Program 105
5.2 Minimum-Period/Minimum-Latency Linear Timing Functions 107
5.3 Boundary Processes 119

6 Case Studies **122**
6.1 FIFO Queues 122
6.1.1 CRT Constraint 123
6.1.2 Passive/Active Data Constraints 123
6.1.3 Interleavings of Passive/Active Protocols 125
6.1.4 Interleavings of Active/Passive Protocols 135
6.2 Microprocessor 142
6.3 Other Asynchronous Pipeline Circuits 150
6.3.1 Meng 150
6.3.2 Greenstreet, Williams, and Staunstrup 154

7 Performance Optimization **165**

7.1 Tau Model 165

7.2 Convex Objective Function 170

7.2.1 Power Constraint 172

7.2.2 Minimum Transistor Widths 174

7.3 Subgradient Algorithm 176

7.3.1 The Subgradient 176

7.3.2 Basic Algorithm 176

7.3.3 Convex Constraints 177

7.3.4 Heuristics and Space Dilation 178

8 Summary and Concluding Remarks **183**

8.1 Loose Ends 184

Bibliography **186**

A Accuracy of Tau Model **191**

A.1 Tied Transistors in the Pull-up Chain 198

A.2 Maximum Approximation 199

A.3 Transistor Models 203

B Detailed Example **204**

List of Figures

1.1 Execution of fib.asm on 2.0μm processor. 2

2.1 Collapsed-constraint graph of Example 2.9. 54
2.2 Complete graph at iteration 0. 54
2.3 Critical-arc graph at iteration 0. 54
2.4 Complete graph at iteration 1. 55
2.5 Critical-arc graph at iteration 1. 55
2.6 Complete graph at iteration 2. 55
2.7 Critical-arc graph at iteration 2. 55
2.8 Complete graph at iteration 3. 55
2.9 Critical-arc graph at iteration 3. 55
2.10 Complete graph at iteration 4. 55
2.11 Cyclic critical-arc graph at iteration 4. 55

3.1 Schematic symbols for various generalized C-elements. 63
3.2 Weak-feedback CMOS implementation of a generalized C-element . . 64
3.3 Fully-static CMOS implementation of a generalized C-element 64
3.4 Implementation of an output unit. 68
3.5 Alternative implementation of an output unit. 68
3.6 Implementation of an active-input unit. 68

3.7 Implementation of a passive-input unit. 68

3.8 Standard implementation of a passive-input/active-output datapath. . 69

3.9 Alternative implementation of a passive-input/active-output datapath. 69

5.1 Regular linear array of n *lap* processes. 104

5.2 Latency/period constraint graph for the lazy-active/passive buffer of Example 5.1. 106

5.3 Four types of constraints imposed by (5.6). 109

5.4 Six types of constraint intersections imposed by (5.6). 110

5.5 Graphical depiction of constraints imposed in Example 5.3 111

5.6 Case 2 critical cycle in three instances of the lazy-active/passive buffer 115

5.7 Intersection of Case 1 and Case 2 critical cycle in the lazy-active/passive buffer. 116

6.1 Possible datapath implementations for a passive/active *FIFO*. 124

6.2 Diagram to enumerate the passive/active interleavings. 125

6.3 Diagram to enumerate the passive/active interleavings that contain the vacuous wait $[\neg ri]$. 126

6.4 Diagram to enumerate the passive/active interleavings that contain the vacuous firing $ro\downarrow$. 126

6.5 Latency/period constraint graph for buffer *pa*. 129

6.6 Latency/period constraint graph for buffer *a1*. 130

6.7 Latency/period constraint graph for buffer *c8*. 130

6.8 Latency/period constraint graph for buffer *c9*. 130

6.9 Latency/period constraint graph for buffer *pla*. 130

6.10 Latency/period constraint graph for buffer *b1*. 130

6.11 Latency/period constraint graph for buffer *b2*. 130

6.12 Latency/period constraint graph for buffer *c3*. 131

6.13 Latency/period constraint graph for buffer *c4*. 131

6.14 Latency/period constraint graph for buffer *c5*. 131

6.15 Latency/period constraint graph for buffer *c0*. 131

6.16 Latency/period constraint graph for buffer *c1*. 131

6.17 Latency/period constraint graph for buffer *c2*. 131

6.18 Possible datapath implementations for an active/passive *FIFO*. 135

6.19 Diagram to enumerate the active/passive interleavings. 137

6.20 Diagram to enumerate the active/passive interleavings that contain the vacuous action $ro\downarrow$. 139

6.21 Latency/period constraint graph for buffer *ap*. 140

6.22 Latency/period constraint graph for buffer *d1*. 140

6.23 Latency/period constraint graph for buffer *lap*. 140

6.24 Latency/period constraint graph for buffer *d3*. 140

6.25 Latency/period constraint graph for buffer *d4*. 140

6.26 Latency/period constraint graph for buffer *d5*. 140

6.27 Latency/period constraint graph for buffer *e0*. 141

6.28 Latency/period constraint graph for buffer *e1*. 141

6.29 Latency/period constraint graph for buffer *e2*. 141

6.30 Latency/period constraint graph for buffer *e3*. 141

6.31 Latency/period constraint graph for buffer *e4*. 141

6.32 Interconnections between processes of the *AM*. 144

6.33 First part of the cycle period graph for *AM*. 145

6.34 Second part of the cycle period graph for *AM*. 146

6.35 Simplified cycle-period graph corresponding to Figure 6.34. 147

6.36 Meng's *FIFO* implementation 150

6.37 Latency/period constraint graph corresponding to Meng's *FIFO* implementation. 151

6.38 Latency/period constraint graph of the improved implementation. . . 153
6.39 Implementation of the pipeline stage used by Greenstreet, Williams, and Staunstrup. 155
6.40 Latency/period constraint graph of the GWS pipeline stage. 156
6.41 Three GWS pipeline stages connected to form a ring. 157
6.42 Latency/period constraint graph of three GWS pipeline stages connected to form a ring.. 158
6.43 Latency/period constraint graph for the modified GWS pipeline stage. 161
6.44 Three modified GWS pipeline stages connected in a ring. 162
6.45 Latency/period constraint graph of three modified GWS pipeline stages connected in a ring. 163

7.1 RC approximation of a CMOS pulldown. 166
7.2 CMOS circuit for C-element and the trivial environment. 168
7.3 Cross section of f with respect to w_1. 171
7.4 Cross section of f with respect to w_2. 171
7.5 Cross section of f with respect to w_3. 171
7.6 Cross section of f with respect to w_4. 171

A.1 *RC* pulldown circuit with dimensionless variables. 191
A.2 Time evolution of internal and external node voltages.. 193
A.3 Plot comparing tau-model estimated and actual RC delay. 200
A.4 Cross section of smooth f with respect to w_1. 202
A.5 Cross section of smooth f with respect to w_2. 202
A.6 Cross section of smooth f with respect to w_3. 202
A.7 Cross section of smooth f with respect to w_4. 202

B.1 Input representation of a single lazy-active/passive buffer. 205

B.2 Optimization equations for the lazy-active/passive buffer. 206
B.3 Optimization equations for the lazy-active/passive buffer (cont.). . . . 207
B.4 Output of optimization tool for the lazy-active/passive buffer. 208
B.5 Output of optimization tool for the lazy-active/passive buffer (cont.). 209
B.6 Circuit for lazy-active/passive buffer. 210
B.7 Period constraint graph with optimized transistors. 211
B.8 Period constraint graph with nonoptimized transistors. 212
B.9 SPICE output of optimized lazy-active/passive buffer. 213
B.10 SPICE output of nonoptimized lazy-active/passive buffer. 214
B.11 SPICE output of optimized lazy-active/passive buffer using the switch model. 215
B.12 SPICE output of nonoptimized lazy-active/passive buffer using the switch model. 216
B.13 SPICE output of optimized lazy-active/passive buffer. 217
B.14 SPICE output of nonoptimized lazy-active/passive buffer. 218
B.15 SPICE output of optimized lazy-active/passive buffer using the switch model. 219
B.16 SPICE output of nonoptimized lazy-active/passive buffer using the switch model. 220
B.17 Graph of simulated vs. predicted delays. Transistor simulation. Simple model. 223
B.18 Graph of simulated vs. predicted delays. Transistor simulation. Simple model. 224
B.19 Graph of simulated vs. predicted delays. Transistor simulation. Tied model. 227
B.20 Graph of simulated vs. predicted delays. Switch-level simulation. Tied model. 228

B.21 SPICE output for three-stage optimized lazy-active/passive pipeline. . 229
B.22 SPICE output for three-stage nonoptimized lazy-active/passive pipeline. 230
B.23 Switch-level SPICE output for three-stage optimized lazy-active/passive pipeline. 231
B.24 Switch-level SPICE output for three-stage nonoptimized lazy-active/passive pipeline. 232

List of Tables

3.1 Basic constructs of CSP used to describe processes. 58

3.2 Syntax of boolean expressions used to guard commands. 59

4.1 Correspondence between occurrence indices of input and output variables. 81

4.2 Assignment and wait patterns of a *SLHE* in standard form 82

4.3 Subsumption of rules under the target-name convention for delays. . . 82

6.1 Table of external delays in the cycle period and latency of each passive/active *CRT* buffer. 132

6.2 Table of external delays in the cycle period and latency of each passive/active *CRT* buffer. 138

7.1 Performance of optimization tool. 180

B.1 Table of predicted vs. simulated delays. Transistor simulation. Simple model. 221

B.2 Table of predicted vs. simulated delays. Switch-level simulation. Simple model. 222

B.3 Table of predicted vs. simulated delays. Transistor simulation. Tied model. 225

B.4 Table of predicted vs. simulated delays. Switch-level simulation. Tied model. 226

List of Algorithms

2.1 Transformations that potentially reduce the number of cycles in a constraint graph. 43
2.2 Polynomial-complexity algorithm to find the cycle period. 45
4.1 Algorithm to produce an equivalent set of processes with no repeated assignments. 86
5.1 Algorithm to find the optimal solution to (5.3). 112
7.1 Subgradient method with space dilation along the gradient. 179
7.2 The r-algorithm, modified to decrease the number of function evaluations. 181

Chapter 1

Introduction

Fundamental to the traditional techniques used to design digital circuits is the notion of a *global clock*. The computation to be performed by the system is divided into discrete pieces, and each individual piece is performed between ticks of this clock. Such circuits are called *synchronous*.

Other types of digital circuits do not use clocks. Instead of a global signal controlling the rhythmic pulsing of the computation, the mechanism that performs a single step of the computation determines when it has completed, and then triggers the next step of the computation to be performed. Figure 1.1 contains measured data from the *Asynchronous Microprocessor* [23], an example of such an asynchronous digital circuit.

Techniques to design these *asynchronous* circuits have existed for nearly as long as there have been techniques for their clock-based counterparts. However, for various reasons, the vast majority of the existing digital circuits are synchronous. Probably the most compelling reason for this wide disparity in number is the perceived difficulty of designing asynchronous circuits.

The work described in this thesis is an extension to, and relies heavily on, a

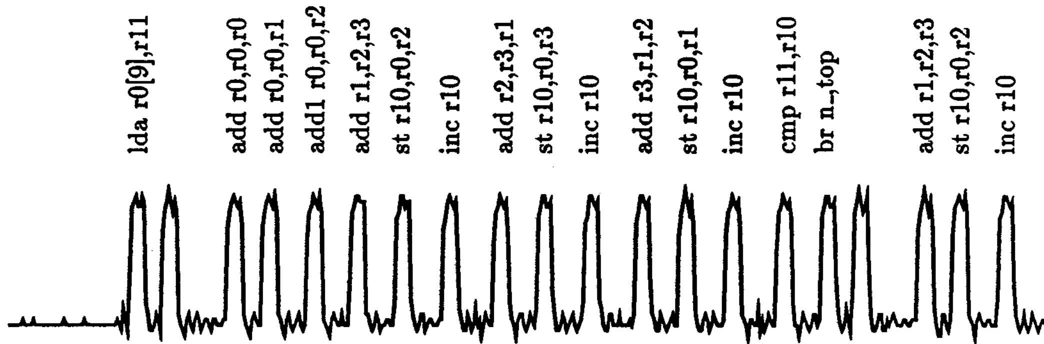


Figure 1.1: Measured data from an execution of a program that computes Fibonacci numbers on the 2.0μm *AM* processor. The oscilloscope trace shown is the request-for-instruction signal generated by the processor. Notice the irregularity of this request.

particular method for synthesizing asynchronous digital circuits. To understand the thesis, it is necessary first to appreciate this synthesis method, especially the reasons why it exists and how it addresses the difficult problems associated with the design of digital circuits.

The synthesis method developed by Alain J. Martin has made the design of asynchronous circuits a simple, methodical activity. This synthesis method is based on a series of semantics-preserving transformations: Computations described in a high-level concurrent programming language are refined, through successively lower-level internal representations, until an asynchronous circuit that performs the original computation has been constructed. Low-level design activities that were once difficult and time-consuming are now automatic. The circuit designer can concentrate on the design at a high level, where most radical improvements in performance can be achieved.

The synthesis method produces circuits that contain no races or hazards; in fact, the circuits have a property known as *quasi-delay-insensitivity* [22]. Such a circuit functions correctly regardless of delays within its elements, except for some necessary assumptions about the relative delays along a few local wires. This view of delays,

except for a trivial difference in the way wires with delays are named, is equivalent to the *speed-independent* delay model [27, 7], originally introduced by David Muller in the 1950s.

There are many advantages of asynchronous circuits over synchronous circuits. These advantages include design modularity, interchangeability of subcomponents, tolerance to variation in power-supply voltages, tolerance to variation in temperatures, lower power consumption, and higher performance. The reader is referred to the literature for a more complete list of these advantages (especially [35, 8, 37, 25]). Here we will concentrate on the possible higher performance of an implementation of a computation as an asynchronous circuit rather than as a synchronous circuit. It is not always the case that an asynchronous circuit will outperform its synchronous counterpart. Which implementation is faster may depend on the relative importance of the many factors that influence the performance of the designs, and can only be determined by comparing actual asynchronous and synchronous implementations. We do not perform this type of analysis here. Instead, we provide techniques to evaluate and improve the performance of an asynchronous implementation.

The most obvious performance advantage is apparent even in the introductory definitions of synchronous and asynchronous given above. Because all steps of a computation performed by a synchronous circuit occur between clock ticks, the clock rate is determined by the slowest step, or worst-case delay. In an asynchronous circuit, the time needed to perform a particular step is determined only locally; thus, the computation proceeds at a rate that corresponds to the actual delay. However, in the asynchronous case, extra circuitry is required to perform the sequencing between two steps, and this extra delay may outweigh the potential speedup. Probably more important to the performance of a circuit are the numerous synchronizations that are necessary in complex concurrent computations. These activities might be synchronized in such a way that the rate at which the computation proceeds might again be

determined by the slowest step in the entire computation.

The traditional technique for determining the effect on performance of the many synchronizations that occur in an asynchronous circuit is (event-driven) simulation. However, simulation has two drawbacks: It is time-consuming (all concurrent events must be executed, even those that are not rate-determining), and it only determines the overall performance. Knowledge of the performance of a particular design is crucial, but it is even more important to know what is limiting the performance of the circuit. Without this information, the designer, when confronted with a variety of design decisions that affect performance, will be unable to make an informed choice. In this thesis we develop techniques that expose the effects on performance of the various interdependencies within large, highly concurrent circuits. We do this without simulating the circuit, but rather by using analytic techniques. The designer can use the results of this *performance analysis*, which includes not only the performance of the particular design, but also information about what is limiting its performance, to synthesize efficient asynchronous circuits.

1.1 Synthesis, Analysis, and Optimization

We espouse a methodology where synthesis, analysis, and optimization work in concert. The synthesis method allows the designer to transform a concurrent program into an asynchronous circuit through various intermediate representations. At any of these levels, the current refinement can be analyzed for performance. Decisions of which transformation to apply can be made at any level by comparing the performance of several competing alternatives. This technique does not necessarily find the optimal circuit, since there is some inevitable error associated with estimating performance at a high level. However, only a few of the best alternatives need be considered at the next synthesis level, thus significantly reducing the vast search space

of the synthesis problem. Instead of exploiting performance information to prune the search space, existing solutions to the synthesis problem [3, 4] seek to use the same sub-circuit template to implement each language construct.

We do not use the term *optimization* to refer to this searching activity, but instead reserve that term for the final adjustment of low-level parameters, in particular, the sizing of transistors. When designing a computer system, a good designer attempts to optimize some combination of the speed of the system, the size of the system, and the power consumption of the system. In a synchronous system, speed is typically constrained, i.e., the clock frequency is fixed for the entire system, and then changes are made to the system in order to reduce the area and power consumption while still maintaining this speed. This strategy makes less sense for an asynchronous circuit since there is no global clock. Instead, we constrain the area and power consumption, and optimize for speed.

In this thesis, we provide techniques to analyze the performance of an asynchronous circuit constructed using our synthesis method. At each level we can analyze a candidate implementation. The result of the analysis is in the form of a performance metric, an indication of the speed of the circuit. One possible performance metric is the time between the start and the finish of a particular computation (if the computation terminates). Other possibilities include the time between consecutive occurrences of a transition, or the time between identically-numbered occurrences of a transition in adjacent processes. These last two metrics are called the *cycle-period* and the *latency* of a system, and are particularly useful, because, for a large class of systems, they provide simple, accurate measures of a circuit's performance.

Frequently, the analysis to determine the performance metric can be performed without explicit knowledge of the delays of the components that compose the circuit. In such a case, the performance metric is expressed as a function of these individual delays.

In this thesis, we also provide techniques for the low-level optimization of a performance metric. By using a simple resistance-capacitance (RC) timing model, the component delays can be approximated based on the connectivity of the components and the sizes of the transistors within components. Composing the performance metric in terms of component delays with the delay approximation of the components in terms of transistor sizes, we get an expression for the performance of the system in terms of transistor sizes. This expression is optimized, producing optimal sizes for the transistors.

1.2 Contributions

The contributions made in this thesis are of three types: theory, algorithms and applications.

The theory contribution is a series of definitions, theorems, and elementary proofs that develop precise statements about the rate at which a concurrent computation proceeds. This is done in two parts: A representation of a concurrent program at one of the synthesis levels is transformed into another intermediate representation, called an *event-rule* system, that was designed especially for performance analysis. This intermediate form is then analyzed, producing a performance metric. Simplicity was a major goal in developing the intermediate representation. The transformation to event-rule systems is conceptually simple, but is actually quite involved, if it is to be performed in a mechanical manner. We describe mechanical techniques to perform this transformation for a certain class of computations.

The techniques we use to analyze the performance of an *event-rule* system are similar to techniques advocated by other researchers for other timing analysis problems. One such alternative methodology uses timed Petri nets [31] as the underlying description of the system. While both approaches are based on linear programming,

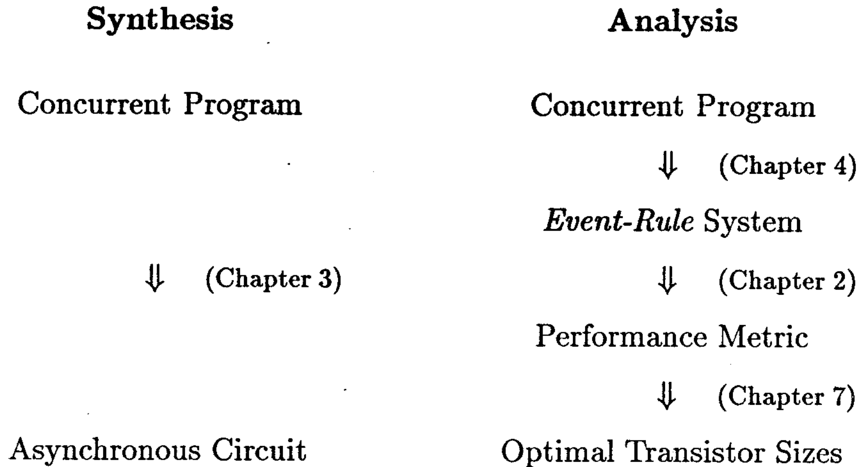
they are developed differently. Our development, we believe, is more elegant, and is easily extended to include linear arrays of identical processes. We provide new proofs, from basic principles, of the accuracy of the generated performance metrics.

The algorithms contribution includes a new technique to determine efficiently the cycle-period of an event-rule system, and a new tool to size optimally the transistors of an asynchronous circuit. A new low-order polynomial-time algorithm for determining the cycle-period of a computation is developed using the primal–dual technique for solving special-case linear programs. A non-trivial termination proof is provided. As a means to calculate efficiently the objective function, this algorithm has been incorporated into an effective tool for globally optimizing the sizes of the transistors in an asynchronous circuit. This tool implements a general algorithm for minimizing a non-linear, non-differentiable, convex function subject to any number of inequality constraints. For many circuits, a 20–30 percent improvement in performance can be achieved by using this tool.

The applications contribution consists of new analyses of the performance of several asynchronous circuits. It includes a taxonomy of the performance of various implementations of first-in, first-out (FIFO) buffer processes. A large design, the *Asynchronous Microprocessor*, is briefly described and analyzed. Two designs by other researchers which were thought to be optimal are analyzed and improved.

1.3 Thesis Overview

The synthesis and analysis procedures are logically divided as follows:



The transformations between the representations are described in the annotated chapters. The synthesis method transforms a concurrent program into an asynchronous circuit. The analysis method transforms a concurrent program, described in any of the representations used during the synthesis process, into an abstract system designed explicitly for performance analysis. This event-rule system is analyzed, resulting in a metric that describes the performance of the original program. This performance metric can then be optimized as a function of transistor sizes.

The two remaining chapters describe the applicability of the synthesis and analysis procedures. Chapter 5 describes methods for analyzing the performance of large concurrent programs that have a regular structure. Chapter 6 is a collection of applications of the various analysis procedures to concrete examples.

Chapter 2

Event-Rule Systems

The analysis method that we use to determine the timing performance of an asynchronous circuit is based on an abstract notion of events, and constraints between events, called an *event-rule* or *ER* system. Most of the results are developed within this abstract framework without referring in any way to circuits. The purpose of this chapter is to develop these results. Later, in Chapter 4, we show how to transform various high-level representations of an asynchronous circuit into an *ER* system. Together, these techniques form a complete method for determining the performance of asynchronous circuits.

2.1 General Event-Rule Systems

A (general) *event-rule system* is a pair $\langle E, R \rangle$, where:

E is a set of events, and

R is a set of rules defining timed constraints between the events. Each $r \in R$ is written $e \overset{\alpha}{\mapsto} f$, where

$e \in E$ is the *source* of r,

$f \in E$ is the *target* of r, and

$\alpha \in [0, +\infty)$ is the *delay* of r.

Neither E nor R need be finite. When R is infinite, we require that no event depends on an infinite number of other events. That is, the set of predecessors (immediate or otherwise) of an event must be finite. The set of sources of an event $f \in E$ is denoted

$$sources(f) = \{e \mid e \overset{\alpha}{\mapsto} f \in R\}.$$

Similarly, the set of targets of an event $e \in E$ is denoted

$$targets(e) = \{f \mid e \overset{\alpha}{\mapsto} f \in R\}.$$

Associated with each $\langle E, R\rangle$ is a *constraint graph* G, which is a directed, labeled graph (multiple arcs and self-loops allowed) that contains one node per event and one arc per rule, in which each arc is labeled with the associated delay α. For a given $\langle E, R\rangle$, there is a set of functions T, that satisfies:

T is a subset of the functions from E to $[0, +\infty)$;

$t \in T$ if and only if

$$t(f) \geq t(e) + \alpha \text{ for every } e \overset{\alpha}{\mapsto} f \in R \ . \tag{2.1}$$

We call a function t in the set T a *timing function* of $\langle E, R\rangle$. Each t represents a possible or consistent timing specification for the events of the system. If the set T is empty, the constraints (2.1) cannot be satisfied by any such function t. In this case, $\langle E, R\rangle$ is called *infeasible*; otherwise, it is called *feasible*.

Example 2.1 Consider $\langle E, R \rangle$ with

$$\begin{aligned} E &= \{a, b, c\} \\ R &= \{a \overset{\alpha_b}{\mapsto} b, b \overset{\alpha_a}{\mapsto} a, b \overset{\alpha_c}{\mapsto} c\}. \end{aligned}$$

This *ER* system is feasible if and only if $\alpha_a = 0$ and $\alpha_b = 0$. □

The smallest timing function corresponds to the earliest time at which the events of E can execute. We call this smallest timing function the *timing simulation* because it represents the time values that are determined by event-driven simulation of the *ER* system. The timing simulation represents the most detailed performance metric for an *ER* system. We now show that such a function exists and is unique in the special case of an acyclic system. We then prove a preliminary lemma and show existence and uniqueness in the general case.

Lemma 2.1 If the constraint graph G of an event-rule system $\langle E, R \rangle$ is acyclic, then there exists a unique function $\hat{t} \in T$ such that for every $t \in T$,

$$\hat{t}(e) \leq t(e) \text{ for every } e \in E. \tag{2.2}$$

We call $\hat{t}$ the *timing simulation* of $\langle E, R \rangle$.

Proof: We propose the following recursive definition for $\hat{t}$:

$$\hat{t}(f) = \begin{cases} 0 & \text{if } \mathit{sources}(f) = \emptyset \\ \max\{\hat{t}(e) + \alpha \mid e \overset{\alpha}{\mapsto} f \in R\} & \text{otherwise.} \end{cases} \tag{2.3}$$

Such a function is well-defined because, by hypothesis, there are no cycles in G and thus no circular dependencies between the events in E. This $\hat{t}$ is by construction a timing function and thus an element of T. We show, by contradiction, that this $\hat{t}$

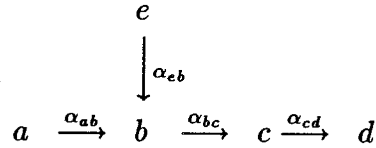
satisfies (2.2). Let t be a candidate smaller timing function such that the set F of events e that satisfy $t(e) < \hat{t}(e)$ is non-empty. Let $f \in F$ be such that there exists no $e \overset{\alpha}{\mapsto} f \in R$ with $e \in F$. Thus the element f is a minimal element of F under the partial order imposed by the rule set R. Either $sources(f) = \emptyset$ and we have $0 \leq t(f) < \hat{t}(f) = 0$, or there exists a $e \overset{\alpha}{\mapsto} f \in R$ that achieves the maximum in (2.3) and we have

$$t(f) < \hat{t}(f) = \hat{t}(e) + \alpha \leq t(e) + \alpha \leq t(f) .$$

In both cases we have a contradiction. The inequality $\hat{t}(e) + \alpha \leq t(e) + \alpha$ follows since $e \notin F$. The last inequality holds since $t \in T$. Therefore F must be empty, and $\hat{t}$ satisfies (2.2).

The timing simulation $\hat{t}$ is unique because if a smallest element of a partially ordered set (in this case, the partially ordered set of timing functions) exists, it is unique. ∎

Example 2.2 The *ER* system defined by the constraint graph:



has the timing simulation:

$$\begin{aligned} \hat{t}(a) &= 0 \\ \hat{t}(e) &= 0 \\ \hat{t}(b) &= \max(\alpha_{ab}, \alpha_{eb}) \\ \hat{t}(c) &= \max(\alpha_{ab}, \alpha_{eb}) + \alpha_{bc} \\ \hat{t}(d) &= \max(\alpha_{ab}, \alpha_{eb}) + \alpha_{bc} + \alpha_{cd} . \end{aligned}$$

□

Lemma 2.2 $\langle E, R\rangle$ is feasible if and only if $\alpha = 0$ for all rules in all cycles of G.

Proof: $\Rightarrow$ Let

$$c = (e_0 \overset{\alpha_0}{\mapsto} e_1, e_1 \overset{\alpha_1}{\mapsto} e_2, \ldots, e_{n-2} \overset{\alpha_{n-2}}{\mapsto} e_{n-1}, e_{n-1} \overset{\alpha_{n-1}}{\mapsto} e_0)$$

be a cycle in G of length $n > 0$. For any $t \in T$,

$$\begin{array}{lcl} t(e_1) & \geq & t(e_0) + \alpha_0\,, \\ t(e_2) & \geq & t(e_1) + \alpha_1\,, \\ \vdots & & \vdots \\ t(e_{n-1}) & \geq & t(e_{n-2}) + \alpha_{n-2}\,, \\ t(e_0) & \geq & t(e_{n-1}) + \alpha_{n-1}\,. \end{array}$$

By summing the inequalities produced by each edge of the cycle, we get,

$$0 \geq \alpha_0 + \alpha_1 + \ldots + \alpha_{n-2} + \alpha_{n-1}\,,$$

and, thus, $\alpha_i = 0$ for all $0 \leq i < n$.

$\Leftarrow$ Let $SC(f)$ denote the set of events in the same strongly connected component of G as event f. Let ι be an injection of E mapping each event in a strongly connected component to a unique representative. Thus $\iota(e) = \iota(f)$ for all $e, f \in E$ such that $e \in SC(f)$. Define the new event-rule system $\langle E_a, R_a\rangle$ as

$$\begin{array}{lcl} E_a & = & \{\iota(e) | e \in E\} \\ R_a & = & \{\iota(e) \overset{\alpha}{\mapsto} \iota(f) \mid e \overset{\alpha}{\mapsto} f \in R \wedge \iota(e) \neq \iota(f)\} \end{array}$$

This system is acyclic and, thus, by Lemma 2.1, it is feasible. Let t_a be a timing function of $\langle E_a, R_a \rangle$. It now remains to show that $t_a(\iota(e))$ is a timing function of $\langle E, R \rangle$ implying that $\langle E, R \rangle$ is feasible. By construction, $t_a(\iota(f)) \geq t_a(\iota(e)) + \alpha$ for all rules $e \overset{\alpha}{\mapsto} f \in R$ such that $\iota(e) \neq \iota(f)$. If $\iota(e) = \iota(f)$, then both e and f are in the same strongly connected component and, thus, $e \overset{\alpha}{\mapsto} f$ is in a cycle. By hypothesis, $\alpha = 0$ so

$$t_a(\iota(e)) = t_a(\iota(f)) \;\; \geq \;\; t_a(\iota(e)) + 0$$

and $t_a(\iota(e))$ is a timing function of $\langle E, R \rangle$. ∎

Theorem 2.3 If the event-rule system $\langle E, R \rangle$ is feasible, then there exists a unique smallest timing function $\hat{t}$.

Proof: By Lemma 2.1, we need only consider cyclic systems. Construct the corresponding acyclic system $\langle E_a, R_a \rangle$ as shown in the proof of Lemma 2.2. Let ι be the unique-representative function that was used to generate $\langle E_a, R_a \rangle$. Now, for any timing function t, it is the case that $t(e) = t(f)$ for all e, f such that $\iota(e) = \iota(f)$. This follows because if e and f are in the same strongly connected component, then there is a cycle with only zero-delay arcs that contains both e and f.

Let $\hat{t}_a$ be the timing simulation of $\langle E_a, R_a \rangle$. We now propose $\hat{t}(e) = \hat{t}_a(\iota(e))$ for all $e \in E$ as the smallest timing function of $\langle E, R \rangle$. For proof, suppose that t is a timing function with $t(\iota(e)) = t(e) < \hat{t}(e) = \hat{t}_a(\iota(e))$ for some $e \in E$. But then, t restricted to the domain E_a is also a timing function of $\langle E_a, R_a \rangle$, contradicting the minimality of $\hat{t}_a$. Corresponding to the particular rule $e' \overset{\alpha}{\mapsto} f' \in R_a$ is the rule $e \overset{\alpha}{\mapsto} f \in R$ with $e' = \iota(e)$ and $f' = \iota(f)$. Thus,

$$t(f') = t(\iota(f)) = t(f) \geq t(e) + \alpha = t(\iota(e)) + \alpha = t(e') + \alpha$$

and t restricted to E_a is a timing function of $\langle E_a, R_a \rangle$. ∎

The proof of this theorem provides a means to transform any cyclic *ER* system into an acyclic system, without altering the timing simulations. For the remainder of the thesis, unless explicitly stated, we assume that an *ER* system is acyclic.

2.2 Repetitive Systems

Many *ER* systems of unbounded size can be generated from bounded structures. Consider the event set E generated from a finite set E' by

$$E \;=\; E' \times \mathbb{N}\,.$$

The elements of E' are called ***transitions***. An event $\langle u, i\rangle \in E$ is the indexed occurrence of the transition $u \in E'$. The nonnegative integer i is called the occurrence index.

The rule set R is also generated from a finite set R'. The elements of R' are quadruples

$$r' = \langle u, v, \alpha, \varepsilon\rangle \in R', \text{ where } R' \subseteq E' \times E' \times [0, +\infty) \times \mathbb{Z},$$

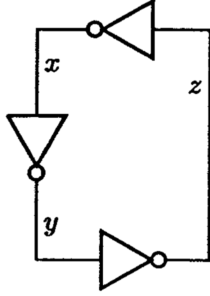
which we will write as

$$r' \;=\; \langle u, i-\varepsilon\rangle \overset{\alpha}{\mapsto} \langle v, i\rangle.$$

The integer ε is called the *occurrence-index offset* of r'. The dummy variable i is replaced by a nonnegative integer no less than ε when r' is instantiated (an infinite number of times) to form the generated rule set R. We require $i \geq \max(0, \varepsilon)$ so that the occurrence indices of both the source and the target events of r are both nonnegative and thus in E. We call $\langle E, R\rangle$ the (general) *ER* system generated from the *repetitive ER* system $\langle E', R'\rangle$.

Example 2.3 Ring Oscillator

Consider the repetitive *ER* system



$$
\begin{array}{rcl}
E' & = & \{x\uparrow, y\uparrow, z\uparrow, x\downarrow, y\downarrow, z\downarrow\} \\
R' & = & \{ \ \langle z\downarrow, i-1\rangle \overset{\alpha_{x\uparrow}}{\mapsto} \langle x\uparrow, i\rangle, \\
 & & \quad \langle x\uparrow, i\rangle \overset{\alpha_{y\downarrow}}{\mapsto} \langle y\downarrow, i\rangle, \\
 & & \quad \langle y\downarrow, i\rangle \overset{\alpha_{z\uparrow}}{\mapsto} \langle z\uparrow, i\rangle, \\
 & & \quad \langle z\uparrow, i\rangle \overset{\alpha_{x\downarrow}}{\mapsto} \langle x\downarrow, i\rangle, \\
 & & \quad \langle x\downarrow, i\rangle \overset{\alpha_{y\uparrow}}{\mapsto} \langle y\uparrow, i\rangle, \\
 & & \quad \langle y\uparrow, i\rangle \overset{\alpha_{z\downarrow}}{\mapsto} \langle z\downarrow, i\rangle \\
 & & \} \ .
\end{array}
$$

Initially, x and z are **false** and y is **true**. The events of the system, those generated from E', represent transitions of circuit variables. Event $\langle x\uparrow, i\rangle$ represents the ith occurrence of a transition from $x =$ **false** to $x =$ **true**. Similarly, $\langle x\downarrow, i\rangle$ represents the ith occurrence of a transition from $x =$ **true** to $x =$ **false**. The repeated rules correspond to dependencies introduced by the inverters connecting the circuit variables x and y, y and z, and z and x. We can represent the generated infinite sets E and R graphically.

$$
\begin{array}{ccccc}
\langle x\uparrow, 0\rangle & \xrightarrow{\alpha_{y\downarrow}} & \langle y\downarrow, 0\rangle & \xrightarrow{\alpha_{z\uparrow}} & \langle z\uparrow, 0\rangle \\
 & & & & \downarrow \alpha_{x\downarrow} \\
\langle z\downarrow, 0\rangle & \xleftarrow{\alpha_{z\downarrow}} & \langle y\uparrow, 0\rangle & \xleftarrow{\alpha_{y\uparrow}} & \langle x\downarrow, 0\rangle \\
\downarrow \alpha_{x\uparrow} & & & & \\
\langle x\uparrow, 1\rangle & \xrightarrow{\alpha_{y\downarrow}} & \langle y\downarrow, 1\rangle & \xrightarrow{\alpha_{z\uparrow}} & \langle z\uparrow, 1\rangle \\
 & & & & \downarrow \alpha_{x\downarrow} \\
\langle z\downarrow, 1\rangle & \xleftarrow{\alpha_{z\downarrow}} & \langle y\uparrow, 1\rangle & \xleftarrow{\alpha_{y\uparrow}} & \langle x\downarrow, 1\rangle \\
\vdots & & & &
\end{array}
$$

In this diagram, nodes represent elements of E and arcs represent elements of R. Notice that event $\langle x\uparrow, 0\rangle$ has no predecessors. In a timing simulation $\hat{t}(\langle x\uparrow, 0\rangle)$ is set to 0. The entire timing simulation $\hat{t}$ can be constructed by inspection. (For ease of notation, we write one-parameter functions of an instantiated repeated event as two-parameter functions of the event and its occurrence index. For example, $\hat{t}(\langle x\uparrow, 0\rangle)$ will be written $\hat{t}(x\uparrow, 0)$.)

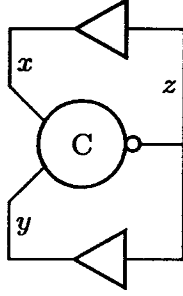
$$
\begin{aligned}
\hat{t}(x\uparrow, i) &= p\,i \\
\hat{t}(y\downarrow, i) &= \alpha_{y\downarrow} + p\,i \\
\hat{t}(z\uparrow, i) &= \alpha_{y\downarrow} + \alpha_{z\uparrow} + p\,i \\
\hat{t}(x\downarrow, i) &= \alpha_{y\downarrow} + \alpha_{z\uparrow} + \alpha_{x\downarrow} + p\,i \\
\hat{t}(y\uparrow, i) &= \alpha_{y\downarrow} + \alpha_{z\uparrow} + \alpha_{x\downarrow} + \alpha_{y\uparrow} + p\,i \\
\hat{t}(z\downarrow, i) &= \alpha_{y\downarrow} + \alpha_{z\uparrow} + \alpha_{x\downarrow} + \alpha_{y\uparrow} + \alpha_{z\downarrow} + p\,i
\end{aligned}
$$

where $p = \alpha_{y\downarrow} + \alpha_{z\uparrow} + \alpha_{x\downarrow} + \alpha_{y\uparrow} + \alpha_{z\downarrow} + \alpha_{x\uparrow}$. □

Example 2.4 C-element

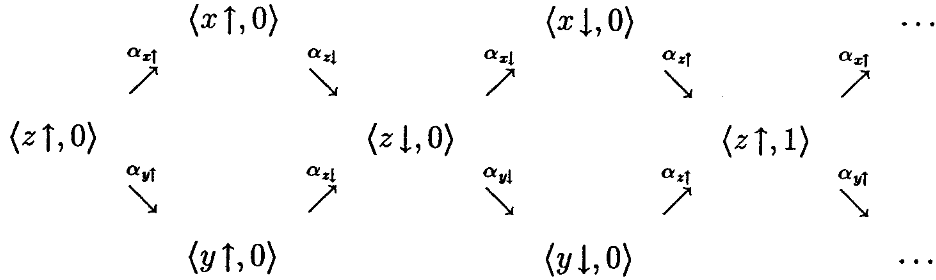
Consider the repetitive *ER* system constructed from a circuit containing an inverting

Muller C-element and two noninverting drivers.



$$
\begin{array}{rcl}
E' & = & \{x\uparrow, y\uparrow, z\uparrow, x\downarrow, y\downarrow, z\downarrow\} \\
R' & = & \{\ \langle x\downarrow, i-1\rangle \overset{\alpha_{z\uparrow}}{\mapsto} \langle z\uparrow, i\rangle, \\
 & & \quad \langle y\downarrow, i-1\rangle \overset{\alpha_{z\uparrow}}{\mapsto} \langle z\uparrow, i\rangle, \\
 & & \quad \langle z\uparrow, i\rangle \overset{\alpha_{x\uparrow}}{\mapsto} \langle x\uparrow, i\rangle, \\
 & & \quad \langle z\uparrow, i\rangle \overset{\alpha_{y\uparrow}}{\mapsto} \langle y\uparrow, i\rangle, \\
 & & \quad \langle x\uparrow, i\rangle \overset{\alpha_{z\downarrow}}{\mapsto} \langle z\downarrow, i\rangle, \\
 & & \quad \langle y\uparrow, i\rangle \overset{\alpha_{z\downarrow}}{\mapsto} \langle z\downarrow, i\rangle, \\
 & & \quad \langle z\downarrow, i\rangle \overset{\alpha_{x\downarrow}}{\mapsto} \langle x\downarrow, i\rangle, \\
 & & \quad \langle z\downarrow, i\rangle \overset{\alpha_{y\downarrow}}{\mapsto} \langle y\downarrow, i\rangle \\
 & & \}\,.
\end{array}
$$

Initially x, y and z are **false**. We can represent the infinite sets E and R graphically as



In this case, the event $\langle z\uparrow, 0\rangle$ has no predecessors, and, thus $\hat{t}(z\uparrow, 0)$ is 0. The entire timing simulation $\hat{t}$, constructed by inspection from the constraint graph, is:

$$
\begin{array}{rcl}
\hat{t}(z\uparrow, i) & = & p\,i \\
\hat{t}(x\uparrow, i) & = & \alpha_{x\uparrow} + p\,i \\
\hat{t}(y\uparrow, i) & = & \alpha_{y\uparrow} + p\,i \\
\hat{t}(z\downarrow, i) & = & \max(\alpha_{x\uparrow}, \alpha_{y\uparrow}) + \alpha_{z\downarrow} + p\,i
\end{array}
$$

$$\hat{t}(x\downarrow, i) = \max(\alpha_{x\uparrow}, \alpha_{y\uparrow}) + \alpha_{z\downarrow} + \alpha_{x\downarrow} + p\, i$$
$$\hat{t}(y\downarrow, i) = \max(\alpha_{x\uparrow}, \alpha_{y\uparrow}) + \alpha_{z\downarrow} + \alpha_{y\downarrow} + p\, i$$

where $p = \max(\alpha_{x\uparrow}, \alpha_{y\uparrow}) + \alpha_{z\downarrow} + \max(\alpha_{x\downarrow}, \alpha_{y\downarrow}) + \alpha_{z\uparrow}$. □

2.2.1 Linear Timing Functions

In the previous examples, we saw that the timing simulations of two repetitive *ER* systems took on a simple form that is linear in the occurrence index i. This is not the case for all repetitive *ER* systems. However, as we later show (Theorems 2.9 and 2.10), a *linear timing function* exists whenever the timing simulation exists, and the "best" such function is a very good approximation of the timing simulation.

We call $\bar{t} \in T$ a *linear timing function* (of $\langle E', R' \rangle$), if

$$\bar{t}(v, i) = x_v + p_v i \text{ for every } v \in E' \text{ and } i \in \mathbb{N} \ . \tag{2.4}$$

Each x_v and p_v is independent of i. For each $v \in E'$, x_v and p_v are called, respectively, the *offset* and *cycle period* of the transition v.

Because of the linear form of $\bar{t}$, the timing function constraints (2.1) reduce to linear inequalities in the offsets and cycle periods of the system. All dependence on the occurrence index i can be eliminated. For each rule $r = \langle u, i - \varepsilon \rangle \overset{\alpha}{\mapsto} \langle v, i \rangle \in R'$, we have the infinite set of constraints

$$\bar{t}(v, i) \ \geq \ \bar{t}(u, i - \varepsilon) + \alpha, \ \text{ for each } i \geq \max(0, \varepsilon)\,.$$

Replacing $\bar{t}$ by its definition, we get

$$x_v + p_v i \ \geq \ x_u + p_u(i - \varepsilon) + \alpha \tag{2.5}$$

$$x_v \;\geq\; x_u - p_u\varepsilon + \alpha + (p_u - p_v)i\,. \qquad (2.6)$$

If $p_u > p_v$ then equation (2.6) can never be satisfied for all i, since i can be arbitrarily large. Thus, the infinite set of constraints generated by r can be replaced by the two inequalities,

$$x_v \;\geq\; x_u - p_u\varepsilon + \alpha + (p_u - p_v)\max(0,\varepsilon)\,, \text{ and} \qquad (2.7)$$

$$p_v \;\geq\; p_u\,. \qquad (2.8)$$

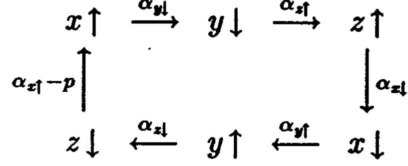
2.2.2 Strongly Connected Systems

The *collapsed-constraint graph* G' of $\langle E', R'\rangle$ is the directed graph with nodes from E' and arcs from R'. From (2.8) we see that for a linear timing function to exist, a partial ordering between the p_v's must be satisfied. If two nodes, u and v, are in the same cycle of the collapsed-constraint graph G', then p_u must equal p_v. Thus, all transitions in the same strongly-connected component of G' have the same cycle period. In the following, unless stated otherwise, G' is assumed to be strongly connected, and we use p to denote the cycle period of every element in E'. Thus (2.7) reduces to

$$x_v \;\geq\; x_u + \alpha - \varepsilon\, p\,. \qquad (2.9)$$

Analogously to the constraint graphs of general *ER* systems, the arcs of the collapsed-constraint graph are labeled so as to specify the timing constraints. Instead of just α, in this case the label is $\alpha - \varepsilon\, p$.

Example 2.5 Consider the repetitive *ER* system of Example 2.3. The labeled collapsed-constraint graph G' is:



Notice the use of $\alpha_{x\uparrow} - p$ as the label on the arc from $z\downarrow$ to $x\uparrow$. This arc corresponds to the rule

$$\langle z\downarrow, i-1\rangle \overset{\alpha_{x\uparrow}}{\mapsto} \langle x\uparrow, i\rangle\,,$$

which imposes the constraint inequality

$$x_{x\uparrow} \;\geq\; x_{z\downarrow} + \alpha_{x\uparrow} - p\,.$$

□

2.2.3 The Cycle Period As a Performance Metric

A cycle period p found in a linear timing function $\bar{t}$ is an upper-bound on the asymptotic performance of a repetitive *ER* system. The time at which occurrence i of a particular transition u fires in an actual execution of the system is $\hat{t}(u)$, where $\hat{t}$ is the timing simulation. Since $\bar{t}(u,i) \geq \hat{t}(u,i)$ (because $\hat{t}$ is the smallest timing function), it must be the case that for all $i > 0$,

$$p \;\geq\; \frac{\hat{t}(u,i)}{i} - \frac{x_u}{i}$$

Thus for any u and large enough i, p is not less than a number arbitrarily close to the average time between consecutive occurrences of u. In Sections 2.4 and 2.4.3, we

show that the minimum p is not only not less than, but is actually equal to a number arbitrarily close to the average time between consecutive occurrences of u. Thus, p is a good, single-valued performance metric for the system.

2.3 Pseudorepetitive Systems

While repetitive *ER* systems are simple and easy to analyze, a broader class of systems can be analyzed with a slightly more complex model. In such a model, events are of two types, either initial or repeated. There are only finitely many initial events, and thus after a finite period, only the repeated events occur.

2.3.1 Definitions

Pseudorepetitive ER systems consist of a finite set of *initial events*, an infinite set of *repeated events*, a finite set of *initial transitions*, a finite set of *repeated transitions*, a finite set of *initial rules*, and a finite set of *repeated rules*. In this order, a pseudorepetitive *ER* system is a six-tuple $\langle E_0, E_1, E'_0, E'_1, R_0, R'_1 \rangle$. As in the case of repetitive systems, a pseudorepetitive system generates a general *ER* system. The corresponding system $\langle E, R \rangle$ consists of the event set

$$E = E_0 \cup E_1 ,$$

where E_0 is a subset of $E'_0 \times \mathbb{N}$ and E_1 is a subset of $E'_1 \times \mathbb{N}$.

The elements of the initial rule set R_0 are of the same form as the rules in a general *ER* system. The source of an initial rule must be an initial event. The elements of the repeated rule set R'_1 are of the same form as the rules in a repetitive *ER* system. Both the source and the target transitions of a repeated rule must be a repeated transition. Furthermore, the elements of R'_1 are only instantiated for those occurrence indices such that both the source and target events are members of E_1, the set of repeated events.

Example 2.6 Consider the pseudorepetitive *ER* system:

$$
\begin{aligned}
E'_0 &= \{a, b, c\} \\
E'_1 &= \{a, b, c\} \\
E_0 &= \{\langle a, 0\rangle, \langle b, 0\rangle, \langle b, 1\rangle, \langle c, 0\rangle\} \\
E_1 &= \{\langle a, i\rangle \mid i > 0\} \cup \{\langle b, i\rangle \mid i > 1\} \cup \{\langle c, i\rangle \mid i > 0\} \\
R_0 &= \{\ \langle b, 0\rangle \overset{\alpha_{bc}}{\mapsto} \langle a, 0\rangle, \\
&\quad \langle a, 0\rangle \overset{\alpha_{ac}}{\mapsto} \langle c, 0\rangle, \\
&\quad \langle c, 0\rangle \overset{\alpha_{cb}}{\mapsto} \langle b, 1\rangle, \\
&\quad \langle b, 1\rangle \overset{\alpha_{ba}}{\mapsto} \langle a, 1\rangle \\
&\quad \} \\
R'_1 &= \{\ \langle a, i-1\rangle \overset{\alpha_{ab}}{\mapsto} \langle b, i\rangle, \\
&\quad \langle b, i+1\rangle \overset{\alpha_{bc}}{\mapsto} \langle c, i\rangle, \\
&\quad \langle c, i-1\rangle \overset{\alpha_{ca}}{\mapsto} \langle a, i\rangle \\
&\quad \}.
\end{aligned}
$$

The generated general system is the pair $\langle E, R\rangle$, where

$$
\begin{aligned}
E &= E_0 \cup E_1 \\
R &= R_0 \cup \{\langle a, i-1\rangle \overset{\alpha_{ab}}{\mapsto} \langle b, i\rangle \mid i > 1\} \\
&\quad \cup \{\langle b, i+1\rangle \overset{\alpha_{bc}}{\mapsto} \langle c, i\rangle \mid i > 0\} \\
&\quad \cup \{\langle c, i-1\rangle \overset{\alpha_{ca}}{\mapsto} \langle a, i\rangle \mid i > 1\}.
\end{aligned}
$$

□

2.3.2 Approximating the Timing Simulation

While pseudorepetitive systems are more general than repetitive systems, we show now that we can approximate the steady-state timing behavior of a pseudorepetitive system by the timing simulation of a simpler repetitive system.

Theorem 2.4 Let $P = \langle E_0, E_1, E_0', E_1', R_0, R_1' \rangle$ be a pseudorepetitive *ER* system. Let S be the general (possibly cyclic) *ER* system generated from P, and let S' be the general (possibly cyclic) *ER* system generated from just the repetitive part of P, that is the repetitive *ER* system $\langle E_1', R_1' \rangle$. If S is feasible, then S' is feasible, and the timing simulation of S differs by no more than a constant from the timing simulation of S'.

Proof: We first show that S' is feasible. Assume S' is not feasible. Then by Lemma 2.2, there exists a cycle of non-zero cost. Let

$$\begin{array}{ccc}\langle u_0, i_0 \rangle & \overset{\alpha_1}{\mapsto} & \langle u_1, i_1 \rangle \\ \langle u_1, i_1 \rangle & \overset{\alpha_2}{\mapsto} & \langle u_2, i_2 \rangle \\ & \vdots & \\ \langle u_{\ell-1}, i_{\ell-1} \rangle & \overset{\alpha_0}{\mapsto} & \langle u_0, i_0 \rangle \end{array}$$

be the edges of such a cycle with at least one $\alpha_j > 0$. Since S' is repetitive, we can add an arbitrary n to each i_j in the above rules. For large enough n, each $\langle u_j, i_j + n \rangle$ is a member of E_1, the repeated events of the pseudorepetitive system. Thus, there exists a cycle of non-zero cost in the general *ER* system generated from S. By Lemma 2.2, S is not feasible, contradicting the hypothesis.

We go on to prove the approximation part of the theorem. We start by transforming both S and S' into acyclic *ER* systems with the technique used in Lemma 2.2. To

avoid cluttered notation, we denote the new acyclic systems with the same names, S and S'. This transformation allows the simpler definition of the timing simulation (2.3) to be used. Consider the timing simulations $\hat{t}$ of S and $\hat{t}'$ of S' restricted to the events $\langle v, i \rangle \in E_1$, such that $\langle v, i \rangle$ is the target of a rule with a source in E_1 and not a target of a rule with a source in E_0. All but a finite number of events are of this type. Since S and S' are both feasible, each excluded event that exist in both S and S' has a finite $\hat{t}$ and $\hat{t}'$, and thus there exists a finite B such that

$$|\hat{t}(v,i) - \hat{t}'(v,i)| \leq B \text{ for all excluded } \langle v, i \rangle \text{ with } v \in E_1' \ .$$

We show that the two timing simulations can differ by no more on a target event than they do on a source event. By the definition of a timing simulation on the restricted set of events, we have

$$\begin{aligned} \hat{t}(v,i) &= \max\{\hat{t}(u, i-\varepsilon) + \alpha \mid \langle u, v, \alpha, \varepsilon \rangle \in R_1'\} \\ \hat{t}'(v,i) &= \max\{\hat{t}'(u, i-\varepsilon) + \alpha \mid \langle u, v, \alpha, \varepsilon \rangle \in R_1'\} \end{aligned}$$

Notice that for each target event the same rules apply for the two timing simulations. We must show that

$$|\hat{t}(v,i) - \hat{t}'(v,i)| \ \leq \ \max\{|\hat{t}(u, i-\varepsilon) - \hat{t}'(u, i-\varepsilon)| \mid \langle u, v, \alpha, \varepsilon \rangle \in R_1'\} \, .$$

Let $r = \langle u, v, \alpha, \varepsilon \rangle$ be the rule such that $\hat{t}(v,i) = \hat{t}(u, i-\varepsilon) + \alpha$ and let $r' = \langle u', v, \alpha', \varepsilon' \rangle$ be the rule such that $\hat{t}'(v,i) = \hat{t}'(u', i-\varepsilon') + \alpha'$. If $\hat{t}(v,i) \geq \hat{t}'(v,i)$, then

$$\begin{aligned} \hat{t}(v,i) - \hat{t}'(v,i) &= \hat{t}(u, i-\varepsilon) + \alpha - \hat{t}'(u', i-\varepsilon') - \alpha' \\ &\leq \hat{t}(u, i-\varepsilon) + \alpha - \hat{t}'(u, i-\varepsilon) - \alpha \end{aligned}$$

$$= \hat{t}(u, i-\varepsilon) - \hat{t}'(u, i-\varepsilon) .$$

The inequality above follows since, by construction,

$$\hat{t}'(u', i-\varepsilon') + \alpha' \geq \hat{t}'(u, i-\varepsilon) + \alpha .$$

Similarly, if $\hat{t}'(v,i) \geq \hat{t}(v,i)$, then

$$\hat{t}'(v,i) - \hat{t}(v,i) \leq \hat{t}'(u', i-\varepsilon') - \hat{t}(u', i-\varepsilon') .$$

Since the absolute difference between $\hat{t}'(v,i)$ and $\hat{t}(v,i)$ is no larger for each subsequent event,

$$|\hat{t}'(v,i) - \hat{t}(v,i)| \leq B \text{ for all } \langle v,i \rangle \in E_1' \times \mathbb{N} . \quad \blacksquare$$

2.4 Minimum-Period Linear Timing Functions

Among the possible linear timing functions, there are those that minimize the cycle period p. The techniques of ***linear programming*** [13, 29] can be used to find such a minimum-period linear timing function (MPLTF). To fit our needs, we will use linear programs in *standard* form:

$$Ax \geq b, x \geq 0, z = \min c^T x \ , \tag{2.10}$$

where A is an m-by-n matrix and x, b and c are column vectors of lengths n, m and n, respectively. The program has a *feasible* solution x when both vector constraints can be satisfied. The program has an *optimal* solution x when x is feasible and when for all other feasible x', $c^T x \leq c^T x'$.

A fundamental result from linear programming relates the primal program (2.10) to the dual program

$$y^T A \leq c^T, y \geq 0, w = \max y^T b \ . \tag{2.11}$$

Theorem 2.5 (Duality Theorem) Exactly one the these four cases occurs:

1. Both the primal and the dual have optimal solutions, and $z = w$.

2. The primal has no feasible solution, but the dual has feasible solutions y with $y^T b = +\infty$.

3. The dual has no feasible solution, but the primal has feasible solutions x with $c^T x = -\infty$.

4. Neither the primal nor the dual has a feasible solution.

Proof: See [13]. ∎

The constraints of a linear timing function (2.9) are simple linear inequalities in the offsets x_v and cycle period p. By ordering the sets E' and R' of a repetitive ER system, we can construct a linear program in matrix form.

$$\min \begin{pmatrix} c_0^T & c_1^T \end{pmatrix} \begin{pmatrix} x \\ p \end{pmatrix} = z \tag{2.12}$$

$$\begin{pmatrix} A' & \varepsilon \end{pmatrix} \begin{pmatrix} x \\ p \end{pmatrix} \geq \alpha \tag{2.13}$$

$$x, p \geq 0 \tag{2.14}$$

We will discuss the coefficients of the objective function, c_0 and c_1, later. The elements of the vector x correspond to the offsets (from the linear timing function) of the transitions that make up the set E'. The matrix A' is the arc-node incidence matrix of the collapsed-constraint graph G'. If row j of A' represents the constraint $r_j \in R'$ and column k of A' represents the transition $u_k \in E'$, then

$$a'_{jk} = \begin{cases} -1 & \text{if } u_k \text{ is the source transition of } r_j \\ 1 & \text{if } u_k \text{ is the target transition of } r_j \\ 0 & \text{otherwise} \end{cases} \tag{2.15}$$

The jth elements of the column vectors ε and α are the occurrence-index offset and the delay of the constraint r_j, respectively.

The dual of (2.13) is

$$\max y^T \alpha = w \tag{2.16}$$

$$y^T \begin{pmatrix} A' & \varepsilon \end{pmatrix} \leq \begin{pmatrix} c_0^T & c_1^T \end{pmatrix} \tag{2.17}$$

$$y \geq 0 \ . \tag{2.18}$$

Example 2.7 Lazy-Active/Passive Buffer

Consider the repetitive *ER* system

$$
\begin{array}{rcll}
E' & = & \{lo\uparrow, li\uparrow, ro\uparrow, ri\uparrow, lo\downarrow, li\downarrow, ro\downarrow, ri\downarrow\} & \\
R' & = & \{\ \langle li\downarrow, i-1\rangle & \overset{\alpha_{lo\uparrow}}{\mapsto} \ \langle lo\uparrow, i\rangle, \\
 & & \quad \langle ro\downarrow, i-1\rangle & \overset{\alpha_{lo\uparrow}}{\mapsto} \ \langle lo\uparrow, i\rangle, \\
 & & \quad \langle li\uparrow, i\rangle & \overset{\alpha_{lo\downarrow}}{\mapsto} \ \langle lo\downarrow, i\rangle, \\
 & & \quad \langle ri\uparrow, i\rangle & \overset{\alpha_{ro\uparrow}}{\mapsto} \ \langle ro\uparrow, i\rangle, \\
 & & \quad \langle lo\downarrow, i\rangle & \overset{\alpha_{ro\uparrow}}{\mapsto} \ \langle ro\uparrow, i\rangle, \\
 & & \quad \langle ri\downarrow, i\rangle & \overset{\alpha_{ro\downarrow}}{\mapsto} \ \langle ro\downarrow, i\rangle, \\
 & & \quad \langle lo\uparrow, i\rangle & \overset{\alpha_{li\uparrow}}{\mapsto} \ \langle li\uparrow, i\rangle, \\
 & & \quad \langle lo\downarrow, i\rangle & \overset{\alpha_{li\downarrow}}{\mapsto} \ \langle li\downarrow, i\rangle, \\
 & & \quad \langle ro\downarrow, i-1\rangle & \overset{\alpha_{ri\uparrow}}{\mapsto} \ \langle ri\uparrow, i\rangle, \\
 & & \quad \langle ro\uparrow, i\rangle & \overset{\alpha_{ri\downarrow}}{\mapsto} \ \langle ri\downarrow, i\rangle \\
 & & \}\,. &
\end{array}
$$

The corresponding primal program is:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_{lo\uparrow} \\ x_{li\uparrow} \\ x_{ro\uparrow} \\ x_{ri\uparrow} \\ x_{lo\downarrow} \\ x_{li\downarrow} \\ x_{ro\downarrow} \\ x_{ri\downarrow} \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} p \geq \begin{pmatrix} \alpha_{lo\uparrow} \\ \alpha_{lo\uparrow} \\ \alpha_{lo\downarrow} \\ \alpha_{ro\uparrow} \\ \alpha_{ro\uparrow} \\ \alpha_{ro\downarrow} \\ \alpha_{li\uparrow} \\ \alpha_{li\downarrow} \\ \alpha_{ri\uparrow} \\ \alpha_{ri\downarrow} \end{pmatrix}$$

□

2.4.1 Objective Function

So far we have left the objective coefficients, c_0 and c_1, unspecified. Our goal is to find the minimum cycle time, so one choice for the objective function, in the case of a connected system, is $\min z = 0^T x + 1p$. Thus, for a connected repetitive *ER* system we have the simplified linear program:

$$z = \min p \qquad (2.19)$$

$$A'x + \varepsilon p \geq \alpha \qquad (2.20)$$

$$x, p \geq 0 \;, \qquad (2.21)$$

and corresponding dual

$$w = \max y^T \alpha \tag{2.22}$$

$$y^T A' \leq 0 \tag{2.23}$$

$$y^T \varepsilon \leq 1 \tag{2.24}$$

$$y \geq 0 \; . \tag{2.25}$$

The above linear program can be further simplified since A' is an incidence matrix. For each row, there is exactly one column with entry -1 and exactly one column with entry 1. Thus $A'1 = 0$. In order to satisfy (2.23), we claim that $y^T A' = 0$. This follows since $0 = y^T(A'1) = (y^T A')1$ and each element of $y^T A'$ is non-positive. The dual then becomes

$$\left.\begin{array}{lll} w & = & \max y^T \alpha \\ y^T A' & = & 0 \\ y^T \varepsilon & \leq & 1 \\ y & \geq & 0 \; , \end{array}\right\} \tag{2.26}$$

simplifying the primal to

$$\left.\begin{array}{lll} z & = & \min p \\ A'x + \varepsilon p & \geq & \alpha \\ x & \text{is} & \text{unconstrained} \\ p & \geq & 0 \; . \end{array}\right\} \tag{2.27}$$

While the linear programming formulation shows that x need not be constrained in order to find the minimum period, each element of x must be non-negative if it is to be used as an offset in a linear timing function. To form an offset vector, after an optimal solution x,p is found, we subtract from each element of x the smallest

element of x.

2.4.2 Cycle Vectors of a Graph

A *cycle* c of length ℓ in a directed (multi-)graph $G = \langle \mathcal{N}, \mathcal{A} \rangle$ is an ordered subset $(a_0, a_1, \ldots, a_{\ell-1})$ of the arcs $\mathcal{A}$ such that $target(a_{k-1}) = source(a_k)$ for all $0 < k < \ell$ and $target(a_{\ell-1}) = source(a_0)$. The cycle c can be represented by a *cycle* vector u, a $\{0,1\}$-vector of length $|\mathcal{A}|$ where $u_j = 1$ if and only if the jth arc of $\mathcal{A}$ is in the set c. For each cycle vector u, $u^T A' = 0$, where A' is the arc-node incidence matrix of the graph G. An interesting fact is that any $y \geq 0$ which satisfies $y^T A' = 0$ can be formed by the non-negative linear combination of the cycle vectors of G.

Lemma 2.6 If $y \geq 0$ is such that $y^T A' = 0$, then there exists $\theta_i \geq 0$ and cycle vectors U_i such that

$$y = \theta_0 U_0 + \theta_1 U_1 + \ldots + \theta_{q-1} U_{q-1} \ . \tag{2.28}$$

Proof: By induction on the number of cycles in the graph $G = \langle \mathcal{N}, \mathcal{A} \rangle$.

Base Case: G is acyclic. By Lemma 2.7 (following), y must be identically zero.

General Case: If G is cyclic, let $\mathcal{A}' = \{r_j \in \mathcal{A} \mid y_j > 0\}$. Form the graph $G' = \langle \mathcal{N}, \mathcal{A}' \rangle$. Any cycle in the graph G' is also a cycle in G since arcs are removed, but never added, to form G'. If G' has no arcs, then by definition of $\mathcal{A}'$, y must be identically zero. Otherwise, a scalar multiple of any cycle vector of G' can be subtracted from y, resulting in a y^* with more zero elements than y; that is for any cycle c of G' with corresponding cycle vector u,

$$y^* = y - \theta u$$

where $\theta = \min\{y_j \mid r_j \in c\} > 0$. Now let $\mathcal{A}'' = \{r_j \in \mathcal{A} | y_j^* > 0\}$. Also form the vector y'' of length $|\mathcal{A}''|$ from y^* by removing the zero elements. The

graph $G'' = \langle \mathcal{N}, \mathcal{A}'' \rangle$ has strictly fewer cycles than G' (and G) since the cycle corresponding to u cannot be a member of G''. By the induction hypothesis applied to G'' and y'', we get

$$y'' = \sum_{k=0}^{q''-1} \theta''_k U''_k$$

where each $\theta''_k \geq 0$ and each U''_k is a cycle vector of G''. Now each U''_k can be expanded to a vector U_k of length $\mathcal{A}$ by filling in zeroes for the removed arcs. Similarly, u can be expanded to U. Thus,

$$y = \theta U + \sum_{k=0}^{q''-1} \theta''_k U_k$$

which is the required form for y. ∎

Lemma 2.7 Let G be an acyclic graph with non-empty arc set. If $y \geq 0 \wedge y^T A' = 0$ then $y = 0$.

Proof: By induction on the number of arcs.

Base case: If G consists of a single arc, then the single row of A' contains two non-zero entries, one -1 and one 1. For $y^T A' = 0$, $y = 0$.

General case: If G has more than one arc, pick a single arc r_j with source e such that $targets(e) = \emptyset$. Now y_j must be 0. By the induction hypothesis applied to the graph formed from G by removing r_j from the arc set, the rest of y is identically 0. ∎

These lemmas provide a straightforward means of determining the minimum cycle period p. By enumerating every cycle in the collapsed constraint graph of a repetitive *ER* system, and by computing the sum of the delays and the sum of the occurrence-index offsets around each cycle, we can find the minimum cycle period p.

Theorem 2.8 The minimum cycle period p, that is the optimal value z of the primal program (2.27) is

$$\max\left\{\frac{U_k^T\alpha}{U_k^T\varepsilon}\;\middle|\; U_k \text{ is a cycle vector}\right\} \tag{2.29}$$

if $U_k^T\varepsilon > 0$ for all cycle vectors U_k.

Proof: Let U be the cycle matrix constructed by concatenating the (column) cycle vectors $U_0, U_1, \ldots, U_{q-1}$. By construction, $U^T A' = 0$. By Lemma 2.6, any $y \geq 0$ with $y^T A' = 0$ can be represented as the product $U\Theta$, where the vector Θ has non-negative elements. The dual program (2.26) reduces to

$$z = \max \Theta^T(U^T\alpha) \tag{2.30}$$

$$\Theta^T(U^T\varepsilon) \leq 1 \tag{2.31}$$

$$\Theta \geq 0. \tag{2.32}$$

The primal program corresponding to the reduced dual program is easily solved.

$$z = \min p \tag{2.33}$$

$$(U^T\varepsilon)p \geq (U^T\alpha) \tag{2.34}$$

$$p \geq 0. \tag{2.35}$$

The smallest scalar p that satisfies the vector inequality (2.34) yields the desired minimum cycle period. ∎

Example 2.8 We compute the minimum cycle period of Example 2.7. Two views of the collapsed constraint graph are:

The view on the left uses the standard labeling; on the right, the labels denote the numbered arcs. The three cycles through the graph can be represented by the cycle-vector matrix:

$$U^T = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \end{pmatrix}$$

The simplified primal program $(U^T\varepsilon)\, p \geq (U^T\alpha)$ becomes:

$$\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} p \geq \begin{pmatrix} \alpha_{lo\uparrow} + \alpha_{lo\downarrow} + \alpha_{ro\uparrow} + \alpha_{ro\downarrow} + \alpha_{li\uparrow} + \alpha_{ri\downarrow} \\ \alpha_{lo\uparrow} + \alpha_{lo\downarrow} + \alpha_{li\uparrow} + \alpha_{li\downarrow} \\ \alpha_{ro\uparrow} + \alpha_{ro\downarrow} + \alpha_{ri\uparrow} + \alpha_{ri\downarrow} \end{pmatrix} = \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \end{pmatrix}$$

Thus, the minimum p is $\max(\alpha_0, \alpha_1, \alpha_2)$. □

We finish this section by showing that there exists a MPLTF for every feasible, repetitive *ER* system.

Theorem 2.9 Let $\langle E, R\rangle$ be a general *ER* system with constraint graph G. Furthermore, let $\langle E, R\rangle$ be generated from the repetitive system $\langle E', R'\rangle$ with a collapsed constraint graph G'. There exists a minimum period linear timing function for $\langle E', R'\rangle$.

Proof: By Theorem 2.8, such a MPLTF exists except possibly when $U_k^T \varepsilon \leq 0$ for some cycle vector U_k. If $U_k^T \varepsilon = 0$ for some cycle vector U_k, then there is a cycle in G and by Lemma 2.2 this cycle has zero cost. But then p is not constrained in (2.34) and the proof of Theorem 2.8 is not violated.

If $U_k^T \varepsilon < 0$ for some cycle vector U_k, then there is an infinite set of events in G that precede an arbitrary event in this cycle and, by definition, $\langle E, R\rangle$ is not an ER system. ∎

2.4.3 Approximating the Timing Simulation

We now show that a MPLTF provides an accurate approximation to the timing simulation.

Theorem 2.10 Let $\bar{t}$ and $\hat{t}$ be a minimum-period linear timing function and the timing simulation, respectively, of the connected repetitive system $\langle E', R' \rangle$. There exists a finite B such that for all $u \in E'$ and all $i \geq 0$

$$s_{u,i} = \bar{t}(u,i) - \hat{t}(u,i) \leq B \quad .$$

Proof: By definition for each u and i

$$\begin{aligned} \bar{t}(u,i) &= x_u + pi \\ \hat{t}(u,i) &= x_u + pi - s_{u,i} \quad . \end{aligned}$$

Each $s_{u,i}$ is nonnegative because $\hat{t}$ is the smallest timing function. For the constraints generated from $r = \langle u, i-\varepsilon \rangle \overset{\alpha}{\mapsto} \langle v, i \rangle \in R'$, we define the non-negative slack variables, $\hat{z}_{r,i}$ and $\bar{z}_r$, thus transforming inequalities into equalities:

$$x_u - p\varepsilon + \alpha + \bar{z}_r = x_v \tag{2.36}$$

$$x_u - p\varepsilon - s_{u,i-\varepsilon} + \alpha + \hat{z}_{r,i} = x_v - s_{v,i} \tag{2.37}$$

By subtracting these equations and simplifying, we get

$$\bar{z}_r - \hat{z}_{r,i} = s_{v,i} - s_{u,i-\varepsilon} \quad . \tag{2.38}$$

From Theorem 2.8, $p\sum_{r\in c}\varepsilon_r = \sum_{r\in c}\alpha_r$ for at least one cycle c. Adding the constraints on $\bar{t}$, (2.36), for each $r \in c$, we see that

$$\sum_{r\in c} x_{u_r} - p\sum_{r\in c}\varepsilon_r + \sum_{r\in c}\alpha_r + \sum_{r\in c}\bar{z}_r = \sum_{r\in c} x_{v_r} \quad .$$

Since along any cycle $\sum_{r\in c} x_{u_r} = \sum_{r\in c} x_{v_r}$, we have for all $r \in c$,

$$\bar{z}_r = 0 \quad .$$

By (2.38), $s_{u,i-\varepsilon} \geq s_{v,i}$ for all $i \geq \max(0,\varepsilon)$ and all u, v on cycle c. By summing along the cycle c, we see that for each $u \in c$ and $i' \geq 0$

$$s_{u,i'} \geq s_{u,i} \text{ where } i = i' + \sum_{r\in c}\varepsilon_r \quad .$$

Therefore, we can bound $s_{u,i}$, for every $u \in c$, by

$$B' = \max\left\{ s_{u,i'} \,\middle|\, u \in c \wedge i' < \sum_{r\in c}\varepsilon_r \right\} \quad .$$

For any event, v, not on cycle, c, we find a path, P_v, to this event from an event u on c. Because G' is strongly connected, such a path must exist and be independent of i. Then, by summing (2.38) along that path, we get for all $i, i' \geq 0$

$$s_{u,i'} + \sum_{r\in P_v} \bar{z}_r \geq s_{v,i} \quad ,$$

where $i = i' + \sum_{r\in P_v}\varepsilon_r$. But $\sum_{r\in P_v}\bar{z}_r$ is independent of i; thus, $s_{v,i}$ is bounded by a quantity that does not increase with successive occurrences. Thus, every $s_{v,i}$ with

$v \notin c$ is bounded by B where

$$
\begin{aligned}
B &\geq \max\left\{ s_{v,i} \middle| v \notin c \wedge i < \sum_{r \in P_v} \varepsilon_r \right\}, \text{ and} \\
B &\geq B' + \max\left\{ \sum_{r \in P_v} \bar{z}_r \middle| v \notin c \right\}. \blacksquare
\end{aligned}
$$

2.5 Fast Algorithms

The method proposed by Theorem 2.8 may require exponential time to compute the cycle period as the number of cycles in an arbitrary graph can be exponential in the number of its arcs. Simple transformations on the graphs will often dramatically reduce the number of cycles in the graph. We describe two such transformations in Section 2.5.1. Theorem 2.8 does not provide the only means for determining the minimum cycle period p. The linear program (2.27) can be solved directly by one of the new, polynomial-time algorithms for linear programming. However, since the linear program has such a special form, it can be solved in low-order polynomial time by using a specialized algorithm. This algorithm is described in Section 2.5.2.

2.5.1 Graph Transformations

The two transformations described in Algorithm 2.1 can be applied to any collapsed constraint graph even if the value of the delays are unknown. The purpose of these transformations is to reduce the complexity of the repetitive ER system and thus reduce the number of cycles in the system. The transformations do not change the minimum cycle period of the system.

The first transformation eliminates an arbitrary transition (node of the graph) from the system. It should always be applied if $|P| = |S|\,|T| \leq |S| + |T|$, since, when this condition holds, the rule set R' is not increased in size by applying the transformation. It may be desirable to perform this transformation even if R' is increased in size since the second transformation may then be able to be applied. However, blindly applying the first transformation will exponentially increase the size of R' for certain graphs. (Remember, multiple arcs—with different ε values—are allowed between the same two nodes.)

The second transformation eliminates rules (arcs of the graph) from the system.

1. To remove the node u from the transition set E':

   (a) Form the sets

$$\begin{aligned} T &= \{\langle u, i-\varepsilon\rangle \overset{\alpha}{\mapsto} \langle t, i\rangle \in R' \mid u \neq t\} \\ S &= \{\langle s, i-\varepsilon\rangle \overset{\alpha}{\mapsto} \langle u, i\rangle \in R' \mid s \neq u\} \\ L &= \{\langle u, i-\varepsilon\rangle \overset{\alpha}{\mapsto} \langle u, i\rangle \in R'\} \end{aligned}$$

   (b) Combine each rule in S with each rule in T by summing the α and ε values and remove the intermediate node u:

$$\begin{aligned} P = \{\langle s, i-\varepsilon-\varepsilon'\rangle \overset{\alpha+\alpha'}{\mapsto} \langle t, i\rangle \mid & \langle s, i-\varepsilon\rangle \overset{\alpha}{\mapsto} \langle u, i\rangle \in S \wedge \\ & \langle u, i-\varepsilon'\rangle \overset{\alpha'}{\mapsto} \langle t, i\rangle \in T\} \end{aligned}$$

   (c) Set $R' := (R' \setminus (S \cup T)) \cup P$

   (d) If $L = \emptyset$ then set $E' := E' \setminus \{u\}$

2. If there are two arcs with the same source and same target, and the same occurrence offset ε, then replace the two arcs with a single arc combining the α values. That is, if

$$\begin{aligned} \langle s, i-\varepsilon\rangle &\overset{\alpha'}{\mapsto} \langle t, i\rangle \\ \langle s, i-\varepsilon\rangle &\overset{\alpha''}{\mapsto} \langle t, i\rangle \end{aligned}$$

   are both members of the rule set R', then remove both rules from R' and add the rule

$$\langle s, i-\varepsilon\rangle \overset{\alpha' \max \alpha''}{\mapsto} \langle t, i\rangle$$

Algorithm 2.1: Transformations that potentially reduce the number of cycles in a constraint graph. The first transformation may increase the number of arcs as it decreases the number of nodes while the second transformation always reduces the size of the graph.

It should be applied whenever possible as it decreases, by as much as a factor of two, the number of cycles that visit both nodes s and t. See Section 6.2 for an example of these transformations.

2.5.2 Primal–Dual Method

In this section, we develop an algorithm which solves the primal program (2.27) and the dual program (2.26), simultaneously. This iterative algorithm is constructed by applying the primal–dual method [29], a general technique for constructing special-case algorithms for solving linear programs. The constructed algorithm is of the following form: Starting with an initial feasible solution $x^{(0)}, p^{(0)}$ to the primal program

$$A'x + \varepsilon p \geq \alpha, \; p \geq 0, \; z = \min p \ , \tag{2.39}$$

the algorithm iteratively produces new feasible solutions $x^{(k)}, p^{(k)}$ to the primal program and eventually produces a feasible solution y to the dual program

$$y^T A' = 0, \; y^T \varepsilon \leq 1, \; y \geq 0, \; w = \max y^T \alpha \, , \tag{2.40}$$

such that the objective values of both the primal and dual solution are equal. Thus $x^{(k)}, p^{(k)}$ is an optimal solution, and the iteration terminates.

The complete procedure is shown in Algorithm 2.2. At several points in the algorithm, the α values are compared and thus must be known. Furthermore, we assume that every cycle c through the collapsed constraint graph G' has a positive sum of ε values. We denote this by $\varepsilon(c) > 0$.

We now describe how this algorithm is derived and give a proof of its correctness. The algorithm is based on the two equilibrium conditions of the following lemma.

Lemma 2.11 Let x, p be a feasible solution to (2.39) and let y be a feasible solution

1. Form the initial feasible solution $x^{(0)}$, $p^{(0)}$

   (a) Temporarily remove all arcs with $\varepsilon > 0$ in G'.

   (b) Topologically sort the resulting acyclic graph.

   (c) Using the topological order, set

$$x_v{}^{(0)} = \begin{cases} 0 & \text{if } v \text{ is a root} \\ \max\left\{x_u{}^{(0)} + \alpha \mid \langle u, i-\varepsilon\rangle \overset{\alpha}{\mapsto} \langle v, i\rangle \in R' \wedge \varepsilon \leq 0\right\} & \text{otherwise} \end{cases}$$

   (d) Set

$$p^{(0)} = \max\left\{\frac{x_v{}^{(0)} - x_u{}^{(0)} - \alpha}{-\varepsilon} \;\middle|\; \langle u, i-\varepsilon\rangle \overset{\alpha}{\mapsto} \langle v, i\rangle \in R' \wedge \varepsilon > 0\right\}$$

2. Decrease $x_v{}^{(k)}$ by $x_u{}^{(k)}$ and compare with $\alpha - \varepsilon p^{(k)}$ for all arcs $\langle u, i-\varepsilon\rangle \overset{\alpha}{\mapsto} \langle v, i\rangle$. If equal, the arc is called a *critical* arc.

3. If the graph of critical arcs is cyclic or $p^{(k)} = 0$, then set $\hat{p}^{(k)} = 0$ and exit with the optimal solution $x^{(k)}$, $p^{(k)}$. Else,

   (a) Topologically sort the graph of critical arcs.

   (b) Set $\hat{p}^{(k)} = 1$.

   (c) Using the topological order, set

$$\hat{x}_v^{(k)} = \begin{cases} 0 & \text{if } v \text{ is a root} \\ \min\left\{\hat{x}_u^{(k)} - \varepsilon\hat{p}^{(k)} \mid \langle u, i-\varepsilon\rangle \overset{\alpha}{\mapsto} \langle v, i\rangle \text{ is critical}\right\} & \text{otherwise} \end{cases}$$

   (d) Set

$$\theta^{(k)} = \min\left\{\frac{x_v{}^{(k)} - x_u{}^{(k)} + \varepsilon p^{(k)} - \alpha}{\hat{x}_v^{(k)} - \hat{x}_u^{(k)} + \varepsilon\hat{p}^{(k)}} \;\middle|\; \hat{x}_v^{(k)} - \hat{x}_u^{(k)} + \varepsilon\hat{p}^{(k)} > 0\right\}$$

   (e) Set

$$x^{(k+1)} = x^{(k)} - \theta^{(k)}\hat{x}^{(k)}, \qquad p^{(k+1)} = p^{(k)} - \theta^{(k)}\hat{p}^{(k)}$$

4. Increment k and go to step two.

Algorithm 2.2: Polynomial-complexity algorithm to find the cycle period of the repetitive system $\langle E', R'\rangle$ if the delay values are known in advance and $\varepsilon(c) > 0$ for every cycle c in $\langle E', R'\rangle$.

to (2.40). If the equilibrium conditions

$$y^T\varepsilon = 1 \quad \vee \quad p = 0 \tag{2.41}$$

$$(A'x + \varepsilon p)_i = \alpha_i \quad \vee \quad y_i = 0, \text{ for all } i \tag{2.42}$$

are satisfied, then x, p and y are optimal solutions, and $p = y^T\alpha$.

Proof: (This result and its converse follow from the equilibrium theorem [29], a direct consequence of the duality theorem. Here, we prove the result directly.) By (2.42),

$$y^T(A'x + \varepsilon p - \alpha) \quad = \quad 0\,.$$

Thus, $y^T\varepsilon p = y^T\alpha$. By (2.41), $y^T\varepsilon p = p$. So, the objective values at the two feasible solutions are equal, and thus the feasible solutions are optimal. ∎

At each iteration, we use these equilibrium conditions to construct an auxiliary linear program that is easier to solve than the original program. Let

$$I^{(k)} \quad = \quad \{i \mid (A'x^{(k)} + \varepsilon p^{(k)})_i = \alpha_i\}\,,$$

that is, let $I^{(k)}$ be the set of critical arcs at the kth iteration. We search for a solution to the dual by considering only those y where $y_i = 0$ for every $i \notin I^{(k)}$. Suppose at the kth iteration we find the optimal solution to the auxiliary linear program

$$\xi = \min y',\ y^TA' = 0,\ y^T\varepsilon + y' = 1,\ y, y' \geq 0,\ y_i = 0 \text{ if } i \notin I^{(k)}\,. \tag{2.43}$$

If $\xi = 0$, then by Lemma 2.11, y is an optimal solution to (2.40) and $x^{(k)}, p^{(k)}$ is an optimal solution to (2.39).

We now prove two lemmas that provide the machinery to solve this auxiliary linear program (2.43).

Lemma 2.12 A dual program to (2.43) is

$$\max \hat{p}^{(k)},\ (A'\hat{x}^{(k)} + \varepsilon\hat{p}^{(k)})_i \le 0 \text{ for all } i \in I^{(k)},\ \hat{p}^{(k)} \le 1\,. \qquad (2.44)$$

Proof: The linear program (2.43), excluding the extra constraint

$$y_i \;=\; 0 \text{ if } i \notin I^{(k)}\,,$$

can be expressed in standard dual form (2.11) as follows:

$$\begin{aligned}
\xi &= \max \begin{pmatrix} y^T & y' \end{pmatrix} \begin{pmatrix} 0 \\ -1 \end{pmatrix} \\
\begin{pmatrix} y^T & y' \end{pmatrix} \begin{pmatrix} A' & -A' & \varepsilon & -\varepsilon \\ 0 & 0 & 1 & -1 \end{pmatrix} &\le \begin{pmatrix} 0^T & 0^T & 1 & -1 \end{pmatrix} \\
y, y' &\ge 0\,.
\end{aligned}$$

By definition, the primal program corresponding to this dual program is:

$$\begin{aligned}
\eta &= \min \begin{pmatrix} 0^T & 0^T & 1 & -1 \end{pmatrix} \begin{pmatrix} \hat{x}_0 \\ \hat{x}_1 \\ \hat{p}_0 \\ \hat{p}_1 \end{pmatrix} \\
\begin{pmatrix} A' & -A' & \varepsilon & -\varepsilon \\ 0 & 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} \hat{x}_0 \\ \hat{x}_1 \\ \hat{p}_0 \\ \hat{p}_1 \end{pmatrix} &\ge \begin{pmatrix} 0 \\ -1 \end{pmatrix} \\
\hat{x}_0, \hat{x}_1, \hat{p}_0, \hat{p}_1 &\ge 0\,,
\end{aligned}$$

with the new variables $\hat{x}_0, \hat{x}_1, \hat{p}_0, \hat{p}_1$. Setting $\hat{x} = \hat{x}_1 - \hat{x}_0$ and $\hat{p} = \hat{p}_1 - \hat{p}_0$ yields

$$\eta = \max \hat{p},\ A'\hat{x} + \varepsilon\hat{p} \le 0,\ \hat{p} \le 1\,.$$

If for some i, y_i is forced to be 0, then row i of A' can be eliminated without changing the optimality or feasibility of the solutions to the two linear programs. Thus, if $y_i = 0$ for all $i \notin I^{(k)}$, then only the constraints $\hat{p} \le 1$ and

$$(A'\hat{x} + \varepsilon\hat{p})_i \quad \le \quad 0 \quad \text{for all } i \in I^{(k)}$$

need to be satisfied in the primal program. ∎

Lemma 2.13 The linear program (2.44) has an optimal solution with $\hat{p}^{(k)} = 0$ if the subgraph of G' containing only critical arcs is cyclic. If this subgraph is acyclic, then this program has an optimal solution with $\hat{p}^{(k)} = 1$.

Proof: Let u be the cycle vector of a cycle c in the critical subgraph. Since c is also in G', we know that $u^T\varepsilon = \varepsilon(c) > 0$. But from (2.44), we know that

$$\begin{aligned} u^T(A'\hat{x}^{(k)} + \varepsilon\hat{p}^{(k)}) &\le 0\,, \quad \text{which implies} \\ u^T\varepsilon\hat{p}^{(k)} &\le 0\,, \quad \text{which implies} \\ \hat{p}^{(k)} &\le 0\,. \end{aligned}$$

The optimal value is 0 since $\hat{x}^{(k)} = 0$ and $\hat{p}^{(k)} = 0$ is a feasible solution.

If the critical subgraph is acyclic, we can always build up a feasible solution to (2.44) with $\hat{p}^{(k)} = 1$. See Steps 3(a),(b), and (c) of Algorithm 2.2. ∎

From these lemmas, we know that if the critical subgraph is cyclic, then $x^{(k)}, p^{(k)}$ is an optimal solution to the original program (2.39). Also, if $p^{(k)} = 0$ in a feasible solution, then the solution must be optimal. This proves correctness given termination.

We now show that by construction, each $x^{(k)}, p^{(k)}$ is feasible. If the critical subgraph is acyclic, then $\hat{p}^{(k)} = 1$ and we can generate a new feasible solution to the original primal problem (2.39) by

$$x^{(k+1)} = x^{(k)} - \theta^{(k)}\hat{x}^{(k)},\ p^{(k+1)} = p^{(k)} - \theta^{(k)}\hat{p}^{(k)} \tag{2.45}$$

where $\theta^{(k)}$ is chosen to be as large as possible while still retaining the feasibility of $x^{(k+1)}, p^{(k+1)}$.

Lemma 2.14 The feasibility conditions

$$A'x^{(k)} + \varepsilon p^{(k)} - \theta^{(k)}(A'\hat{x}^{(k)} + \varepsilon\hat{p}^{(k)}) \ \geq \ \alpha \tag{2.46}$$

$$p^{(k)} - \theta^{(k)}\hat{p}^{(k)} \ \geq \ 0 \tag{2.47}$$

are satisfied if $\theta^{(k)}$ is chosen such that

$$\theta^{(k)} \ = \ \min\left(\left\{\frac{(A'x^{(k)} + \varepsilon p^{(k)})_i - \alpha_i}{(A'\hat{x}^{(k)} + \varepsilon\hat{p}^{(k)})_i} \,\middle|\, (A'\hat{x}^{(k)} + \varepsilon\hat{p}^{(k)})_i > 0\right\} \bigcup \left\{p^{(k)}\right\}\right).$$

Furthermore, $\theta^{(k)} > 0 \vee p^{(k)} = 0$ and $\theta^{(k)}$ is as large as possible.

Proof: By the definition of $\theta^{(k)}$, (2.46) is satisfied for all arcs with

$$(A'\hat{x}^{(k)} + \varepsilon\hat{p}^{(k)})_i \ > \ 0\,. \tag{2.48}$$

For the remainder of the arcs,

$$-\theta^{(k)}(A'\hat{x}^{(k)} + \varepsilon\hat{p}^{(k)})_i$$

is non-negative, and only relaxes the previously true constraint

$$A'x^{(k)} + \varepsilon p^{(k)} \geq \alpha .$$

If $p^{(k)} > 0$, then to prove $\theta^{(k)}$ is positive we must show that each element in the large set is positive. Suppose that 2.48 holds for some i. But then, by (2.44), $i \notin I^{(k)}$, and both the numerator and the denominator are positive.

To prove that $\theta^{(k)}$ is as large as possible, we need only show that increasing it will violate a constraint. If $\theta^{(k)} = p^{(k)}$, then any increase will violate (2.47). Otherwise, for some i, (2.48) holds and

$$\theta^{(k)}(A'\hat{x}^{(k)} + \varepsilon \hat{p}^{(k)})_i \quad = \quad (A'x^{(k)} + \varepsilon p^{(k)} - \alpha)_i .$$

Increasing $\theta^{(k)}$ violates (2.46) for this i. ∎

We proceed by proving that the algorithm terminates. Again some preliminary lemmas are needed.

Lemma 2.15 Let

$$I_c{}^{(k)} \quad = \quad \{i \mid i \in I^{(k)} \wedge (A'\hat{x}^{(k)} + \varepsilon \hat{p}^{(k)})_i = 0\} . \tag{2.49}$$

At each iteration $k > 0$ of the algorithm,

$$I^{(k)} \quad \supseteq \quad I_c{}^{(k-1)} . \tag{2.50}$$

Proof: If $i \in I_c{}^{(k-1)}$ then

$$(A'x^{(k-1)} + \varepsilon p^{(k-1)})_i = \alpha_i \quad \wedge \quad (A'\hat{x}^{(k-1)} + \varepsilon \hat{p}^{(k-1)})_i = 0 ,$$

and, thus,

$$(A'x^{(k)} + \varepsilon p^{(k)})_i \;=\; (A'x^{(k-1)} + \varepsilon p^{(k-1)})_i - \theta^{(k-1)}(A'\hat{x}^{(k-1)} + \varepsilon \hat{p}^{(k-1)})_i = \alpha_i\,,$$

which is equivalent to $i \in I^{(k)}$. ∎

Lemma 2.16 At each iteration $k > 0$ of the algorithm,

$$\hat{x}^{(k-1)} \geq \hat{x}^{(k)} \quad \vee \quad \hat{p}^{(k)} = 0 \tag{2.51}$$

$$\hat{x}^{(k-1)} \neq \hat{x}^{(k)} \quad \vee \quad \hat{p}^{(k)} = 0 \tag{2.52}$$

Proof: To show (2.52), we argue that each arc that determined the value of an element of $\hat{x}^{(k-1)}$ in step $3c$ of the algorithm is still in $I^{(k)}$, since each such arc is in ${I_c}^{(k-1)}$ and by Lemma 2.15, $I^{(k)} \supseteq {I_c}^{(k-1)}$. Now either $\hat{p}^{(k)} = 0$ and we terminate the algorithm, or we execute step $3c$ again, but with a larger set over which to take the minimum. Thus each element of $\hat{x}^{(k)}$ can be no greater than its corresponding element in $\hat{x}^{(k-1)}$.

To show (2.52), we argue that if the algorithm does not terminate at iteration k (in which case $\hat{p}^{(k)} = 0$), then if $\hat{x}^{(k-1)} = \hat{x}^{(k)}$ we could have been increased $\theta^{(k-1)}$ by $\theta^{(k)}$ without violating (2.46). But by Lemma 2.14, $\theta^{(k)}$ is chosen to be as large as possible. ∎

We can now show the main termination result.

Theorem 2.17 There exists a B such that the variant function

$$v^{(k)} \;=\; B + \sum_i \hat{x}_i^{(k)}$$

is a monotonically decreasing, integer valued, and non-negative.

Proof: By (2.51) and (2.52) at least one element of $\hat{x}^{(k)}$ decreases at each iteration or the algorithm terminates. Thus the sum of the elements decreases at each iteration. Each element of $\hat{x}^{(k)}$ is integral valued by construction. A possible bound B in the variant function is

$$B = \sum_{u \in E'} \max\left\{\varepsilon(p) \,\middle|\, p \in P_u\right\},$$

where P_u is the set of paths (not cycles) ending at node u, and $\varepsilon(p)$ is the sum of the ε values along the path. ∎

Typically, the bound B is small, on the order of the number of transitions in the system, since most paths p through the graph have a small $\varepsilon(p)$.

Example 2.9 Figures 2.2 through 2.11 show the execution of Algorithm 2.2 on a simple example shown in Figure 2.1. The initial feasible solution, Figure 2.2, was produced by Step 1 of the algorithm. In this figure and all other figures representing the complete graph, the solid lines denote critical arcs and the dashed lines denote non-critical arcs. In Figure 2.3 and all other figures representing the critical-arc graph (the reduced primal graph), the solid lines denote critical arcs and the dashed lines (both light and heavy) denote non-critical arcs. The light dashed arcs do not contribute the θ computation since $\hat{x}_v^{(k)} - \hat{x}_u^{(k)} + \varepsilon\hat{p}^{(k)} \leq 0$.

By examining the complete and critical-arc graphs, we compute the θ values as follows:

$$\begin{aligned}\theta^{(0)} &= \min\left\{ \tfrac{0-6+11-1}{0-0+1}, \tfrac{0-6+11-2}{0-0+1}, \tfrac{0-6+11-3}{0-0+1}, \tfrac{0-6+11-4}{0-0+1}, \right. \\ &\qquad \left. \tfrac{5-0+0-1}{0+1+0} \right\} \\ &= \min\{4,3,2,1,4\} = 1\end{aligned}$$

$$\begin{aligned}
\theta^{(1)} &= \min\left\{ \tfrac{0-6+10-1}{0-0+1}, \tfrac{0-6+10-2}{0-0+1}, \tfrac{0-6+10-3}{0-0+1}, \right. \\
&\qquad \left. \tfrac{4-0+0-1}{0+1+0}, \tfrac{5-1+0-1}{0+1+0} \right\} \\
&= \min\{3,2,1,3,3\} = 1 \\
\theta^{(2)} &= \min\left\{ \tfrac{0-6+9-1}{0-0+1}, \tfrac{0-6+9-2}{0-0+1}, \right. \\
&\qquad \left. \tfrac{3-0+0-1}{0+1+0}, \tfrac{4-1+0-1}{0+1+0}, \tfrac{5-2+0-1}{0+1+0} \right\} \\
&= \min\{2,1,2,2,2\} = 1 \\
\theta^{(3)} &= \min\left\{ \tfrac{0-6+8-1}{0-0+1}, \right. \\
&\qquad \left. \tfrac{2-0+0-1}{0+1+0}, \tfrac{3-1+0-1}{0+1+0}, \tfrac{4-2+0-1}{0+1+0}, \tfrac{5-3+0-1}{0+1+0} \right\} \\
&= \min\{1,1,1,1,1\} = 1
\end{aligned}$$

□

2.6 Summary and Related Work

The theory developed in this chapter provides the necessary tools for analyzing the performance of a computation described in the abstract formalism of an event-rule system. Timing simulations and their approximations, minimum-period linear timing functions, provide convenient representations of the times at which the events of the system occur. We have shown that the timing simulation exists whenever the underlying general *ER* system is feasible. We have shown that a MPLTF exists whenever a repetitive *ER* system is feasible. We have provided two techniques for determining the cycle period of a repetitive *ER* system. In the first technique, we enumerate all the cycles in a graph to compute p. The second technique is less intuitive, but provides a solution in low-order polynomial time.

Timed Petri nets could also be used as the underlying formalism. Similar results to

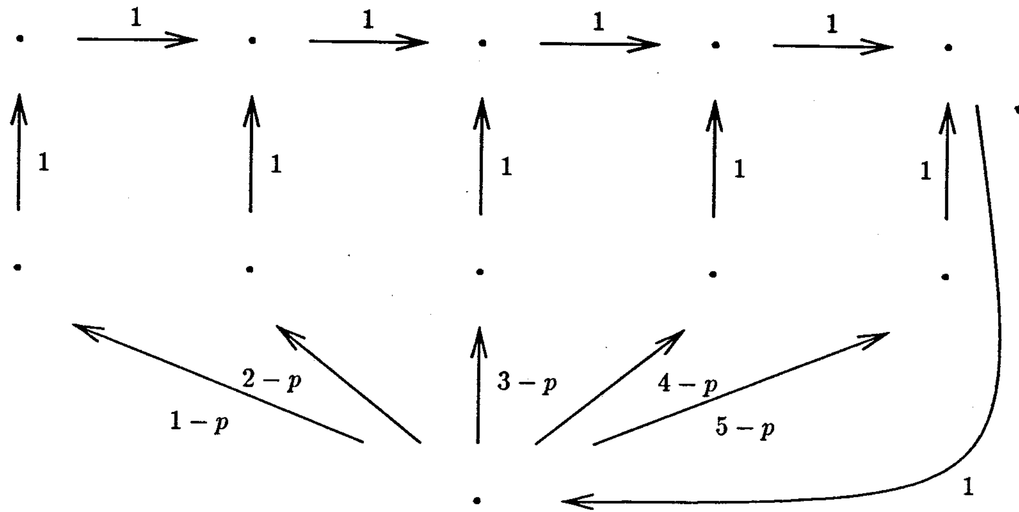


Figure 2.1: Collapsed-constraint graph of Example 2.9.

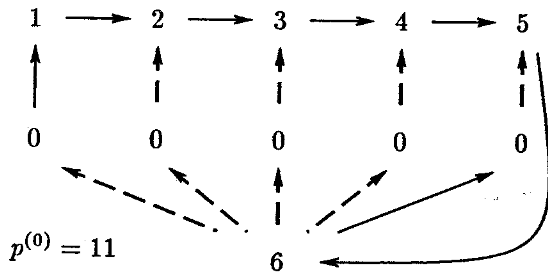


Figure 2.2: Complete graph at iteration 0.

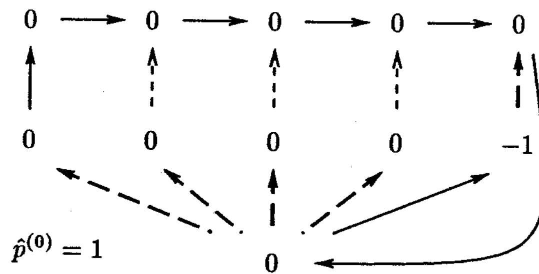


Figure 2.3: Critical-arc graph at iteration 0.

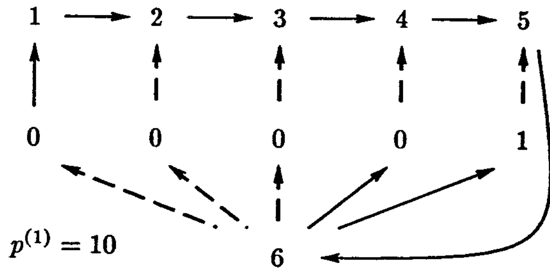


Figure 2.4: Complete graph at iteration 1.

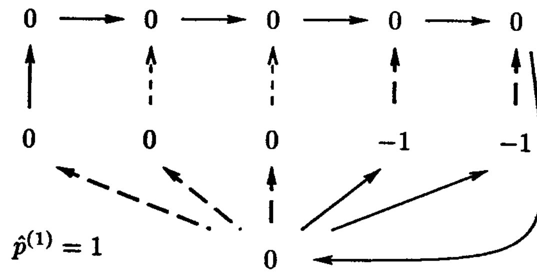


Figure 2.5: Critical-arc graph at iteration 1.

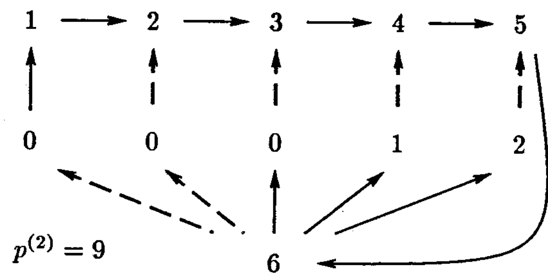


Figure 2.6: Complete graph at iteration 2.

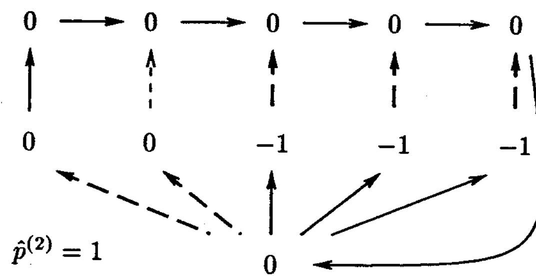


Figure 2.7: Critical-arc graph at iteration 2.

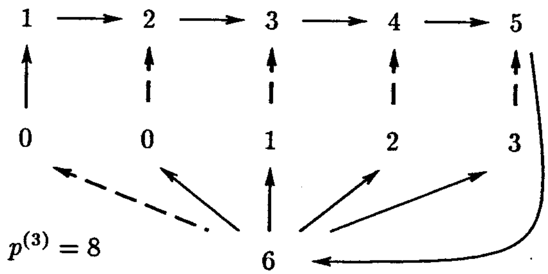


Figure 2.8: Complete graph at iteration 3.

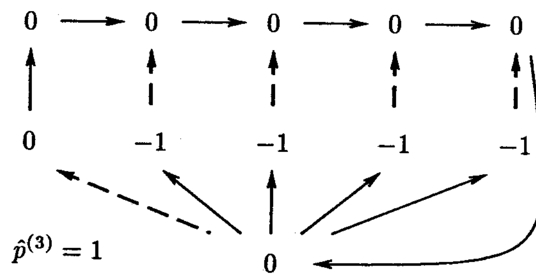


Figure 2.9: Critical-arc graph at iteration 3.

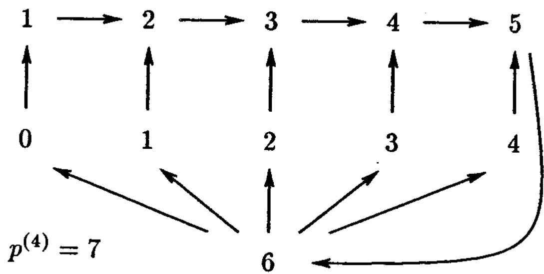


Figure 2.10: Complete graph at iteration 4.

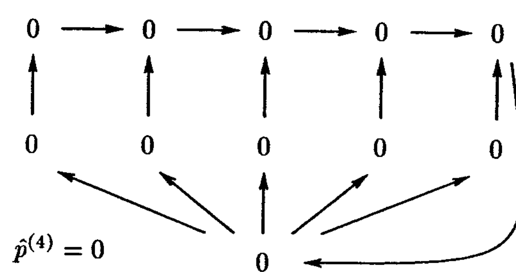


Figure 2.11: Cyclic critical-arc graph at iteration 4.

Theorem 2.8 and Theorem 2.10 were shown by Ramamoorthy and Ho [31] for decision-free timed Petri nets. The connection between this problem and linear programming was established by Magott [20]. Neither work utilized a result similar to Lemma 2.6. Magott briefly mentions that the resulting linear programming problem can be solved by a general-purpose polynomial-time algorithm. Algorithm 2.2 represents a new technique for efficiently determining the cycle period of a repetitive *ER* system (or, if desired, a decision-free timed Petri net).

One reason for using *ER* systems and timing functions instead of timed Petri nets, is that they extend easily to systems with regular arrays of processes. This advantage is described in detail in Chapter 5.

Chapter 3

The Synthesis Method

In this chapter, we briefly review the synthesis method developed by Martin [25] for systematically transforming concurrent programs into self-timed circuits.

3.1 Notation and Intermediate Forms

The synthesis method is based on semantics-preserving transformations between intermediate representations of a concurrent program. The highest-level representations are in the form of *Communication Sequential Processes* (*CSP*) programs based on Hoare's original programming notation [16]. These representations are transformed into *handshaking expansions*, a refined form of *CSP* programs where all communication actions are replaced by explicit manipulations of boolean variables. Handshaking expansions are further transformed into sets of *production rules* where all explicit sequencing has been removed. There is a direct transformation from production rule sets into *quasi-delay-insensitive circuits*, circuits that function correctly regardless of delays in all gates and most wires [22].

Syntax	Operational Description
skip	Do nothing.
$x\uparrow, x\downarrow$	Assign **true** or **false** to the variable x.
$X, X?x, X!x$	Perform a communication across the port X.
$S;T$	Perform statement S, then perform statement T.
S,T	Perform statements S and T concurrently.
$X \bullet Y$	Perform the two communication actions, X and Y, simultaneously.
$[B_0 \longrightarrow S_0$ $\vert \ldots$ $\vert B_{n-1} \longrightarrow S_{n-1}]$	Wait until at least one guard evaluates to true. Perform one of the commands with a true guard. Continue with the next command.
$[B]$	Abbreviation for $[B \longrightarrow$ **skip**$]$.
$*[B_0 \longrightarrow S_0$ $\vert \ldots$ $\vert B_{n-1} \longrightarrow S_{n-1}]$	Evaluate all the guards. If no guards are true, then continue with the next command. Perform one of the commands with a true guard. Repeat the previous steps.
$*[S]$	Abbreviation for $*[$**true** $\longrightarrow S]$.

Table 3.1: Basic constructs of CSP used to describe processes.

3.1.1 Communicating Sequential Processes

The language constructs of our variant of CSP are shown in Table 3.1. Both synchronization and distributed assignment are performed by communication actions. The execution of the statement X, where X is the name of a communication port, performs a zero-slack [24] synchronization with a communication action on port Y in a second process. The control flow of the first process cannot pass the statement X without the control flow of the second process reaching (and being assured of passing) the statement Y. Distributed assignment is performed during the synchronization. As a side-effect of the *output communication action* $X!x$ in the first process and the corresponding *input communication action* $Y?y$ in the second process, the value of the variable x is assigned to the variable y.

The control structures are sequential composition (;), parallel composition (,), synchronized communication ($\bullet$), selection among guarded commands ($[\ldots]$) and rep-

Syntax	Meaning
true, false	Constant values.
$\neg B$	Negation of expression B.
$B_0 \wedge B_1$	Conjunction of expressions B_0 and B_1.
$B_0 \vee B_1$	Disjunction of expressions B_0 and B_1.
x	Value of variable x.
$\overline{X}$	Probe of X. Communication action on port X pending.

Table 3.2: Syntax of boolean expressions used to guard commands.

etition of a guarded command set ($*[\ldots]$). Table 3.2 shows the syntax of the boolean expression used as the guard for a guarded command.

3.1.2 Handshaking Expansions

Communication actions are implemented by a sequence of assignments and waits on boolean variables. The communication *channel* is the physical mechanism that connects the two processes. In the case of a synchronization communication between a process with the port X and a process with the port Y, the physical mechanism consists of two wires, one connecting the output variable xo to the input variable yi, and the other connecting the output variable yo to the input variable xi. Given this interconnection structure, we could use the following sequence of assignments and waits on the variables xo, xi, yo and yi to implement the synchronization:

$$
\begin{aligned}
P_1 &\equiv \ldots; xo\uparrow; [xi]; \ldots \\
P_2 &\equiv \ldots; [yi]; yo\uparrow; \ldots
\end{aligned}
$$

Initially all variables are **false**. A communication action is said to be *passive* if its handshaking expansion begins with a wait and *active* if its handshaking expansion

begins with an assignment. In this case, the communication action in process P_1 is active and the communication action in process P_2 is passive. This implementation of the synchronization communication is called a two-phase handshake.

In order to issue subsequent communications on these ports, it is convenient to reset the variables back to **false** immediately after a communication. The handshaking sequences

$$\begin{aligned} P_1 &\equiv \ldots; xo\uparrow; [xi]; xo\downarrow; [\neg xi]; \ldots \\ P_2 &\equiv \ldots; [yi]; yo\uparrow; [\neg yi]; yo\downarrow; \ldots \end{aligned}$$

can also be used to implement a communication action. A second synchronization is performed during the down-going assignments of this sequence. This implementation is called a four-phase handshake.

Multiple wires in one direction can be used to implement distributed assignments. Implementations of this type of communication are described in Section 3.2.3.

3.1.3 Production Rules

A production rule consists of an assignment guarded by a boolean expression.

$$B \rightarrow z\uparrow$$

The execution of a set of production rules has the following operational semantics: repeatedly evaluate the guard of a production rule and if it evaluates to true, perform (fire) the corresponding assignment. This execution is weakly-fair. If a guard evaluates to true and remains true, the corresponding assignment will eventually fire. If firing the assignment does not change the value of the output variable, in this case z is already true, then the firing is called *vacuous.*

A set of production rules must satisfy the properties called *stability* and *non-interference.* A set of production rules is stable if, during execution, it is never the case that a guard is falsified before the assignment fires. A set of production rules is non-interfering if, during execution, it is never the case that the guards of two production rules—with the same output variable but assigning the opposite value—evaluate to true.

Requiring *stability* and *non-interference* means that computations requiring arbitration cannot be described completely as production rule sets. In such cases, special arbiter and synchronizer processes (describable in handshaking expansion form) are transformed directly to operators.

3.1.4 Quasi-Delay-Insensitive Circuits

A *circuit* is a collection of *operators* connected by wires. A circuit is *delay-insensitive* (DI) if it functions correctly regardless of delays in both its operators and its wires. The class of computations that can be performed without making any assumptions about wire delays is very limited [22].

A circuit is *speed-independent* (SI) if it functions correctly regardless of delays in its operators. The wires in a SI circuit are assumed to be instantaneous or *isochronic.* A circuit is *quasi-delay-insensitive* (QDI) if it functions correctly regardless of delays in its operators and in all its wires except those labeled as *isochronic forks.* A circuit is SI if and only if it is QDI, since every wire can be labeled as isochronic. A non-isochronic wire can be introduced into a SI circuit by an explicit wire operator. In the circuits we synthesize, there are fewer isochronic forks than non-isochronic forks, and thus it is more convenient to label a small number of wires than it is to introduce a large number of extra operators. For this reason, we adopt the QDI model.

Operator Sets

An operator is a group of production rules with the same output variable. Often it is convenient to describe collections and interconnections of operators in schematic form. If an operator can be viewed as a combinational logic gate, such as an *AND* or an *XOR* gate, or as an inverter, we use the traditional logic symbol. Bubbles are used to indicate inversion.

Operators that are not combinational are called state-holding. Many operators are a slight modification of the Muller C-element and we use a special convention for naming these operators. The symbols for a C-element and other *generalized* C-elements are shown in Figure 3.1. The two production rules corresponding to these elements both have simple conjunctive guards. If an input x to one of these operators is not annotated with either a "+" or a "–", then the literal x appears in the guard of the production rule for $z\uparrow$ and $\neg x$ appears in the guard of the production rule for $z\downarrow$. If the input x is annotated with "+", then x appears in the guard of $z\uparrow$ but $\neg x$ does not appear in the guard of $z\downarrow$. If the input x is annotated with "–", then $\neg x$ appears in the guard of $z\downarrow$ but x does not appear in the guard of $z\uparrow$.

CMOS Implementation

Figures 3.2 and 3.3 show two alternative CMOS implementations of the production rules:

$$\begin{aligned} w \wedge x \wedge y &\rightarrow z\uparrow \\ \neg x \wedge \neg y &\rightarrow z\downarrow \end{aligned}$$

In both implementations, series chains of transistors are used to implement the conjunctive guards of the production rule. The implementations differ in how the output

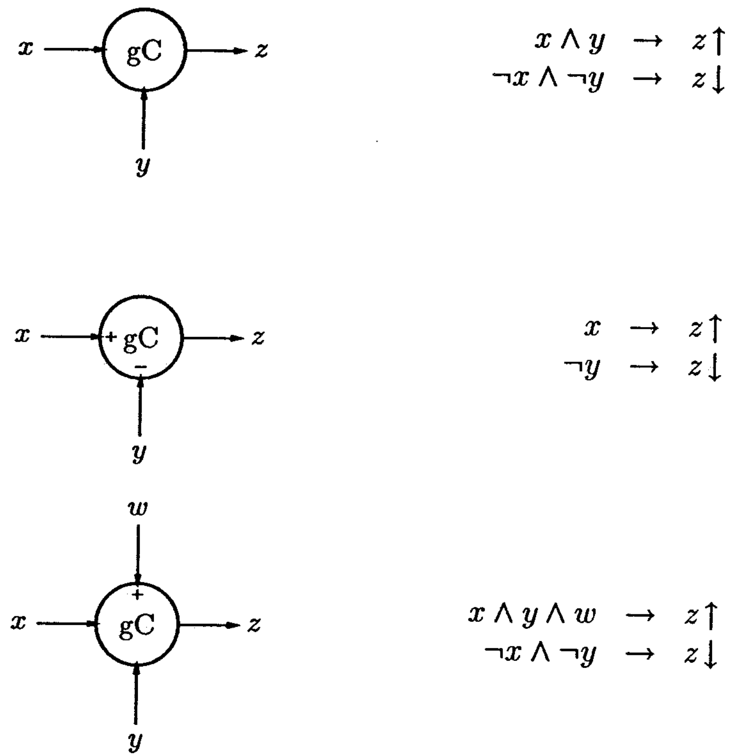


Figure 3.1: Schematic symbols for various generalized C-elements. The first state-holding element is the Muller C-element. The second element is a set/reset flip-flop.

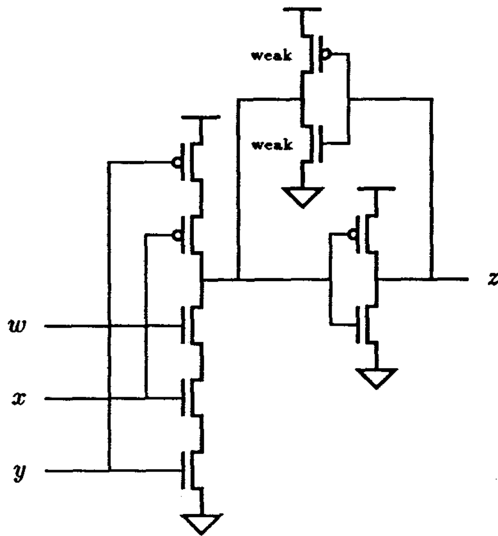


Figure 3.2: Weak-feedback CMOS implementation.

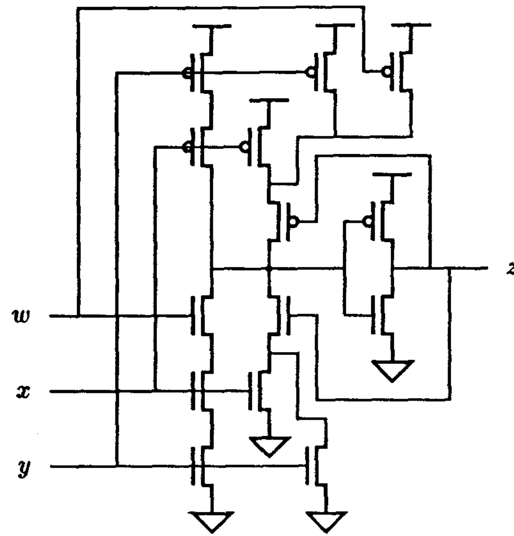


Figure 3.3: Fully-static CMOS implementation.

value is retained when the guards of both production rules evaluate to false. In Figure 3.2, a weak inverter is used to feed the output signal back to the storage node. The correct functioning of the circuit shown in Figure 3.3 is not dependent on the ratio of transistor strengths.

3.2 Transformations

We now briefly describe the transformations of the synthesis procedure and mention the possible effect of the transformations on the performance of the resulting circuit.

3.2.1 Process Decomposition

At the CSP level, any single program statement S can be replaced by: i) a pair of communication actions on a single port C and ii) a new concurrent process implementing just that statement surrounded by a pair of communication actions on the single port D. Ports C and D are connected to form a channel. Variables and ports

must be shared between the original process and the newly created process. The decomposition can be written as

$$S_0; S; S_1 \quad \triangleright \quad (S_0; C; C; S_1 \parallel *[D; S; D]) \ \textbf{channel} \ (C, D) \,.$$

The symbol $\triangleright$ should be read as "compiles into" [5]. (In previous work [4, 5], process decomposition has been performed by replacing S with a single communication and surrounding S in the new process by a single probed communication. The techniques are identical if C is active and D is passive. However, the probed communication cannot be implemented if D is active, so we will exclusively use the more general technique.)

3.2.2 Reshuffling

One of the two synchronizations done during each four-phase communication is not necessary. The term *reshuffling* is used to describe the procedure of interleaving the unnecessary synchronization (usually the down-going one) with other activity performed by the process. Reshuffling has a profound effect on the performance of the resulting implementation, usually resulting in a trade-off between area efficiency and time efficiency. One of the major uses of the performance analysis techniques of Chapter 2 is to determine when reshuffling can be performed—thus producing a smaller circuit—without sacrificing performance.

3.2.3 Decomposition into Control and Data Parts

Communications involving data are implemented using process decomposition. It is convenient, though not necessary, to implement the pair of communication actions $C; C$ with a single four-phase handshaking of the same type (active or passive) as

the port X. What we describe here is just one possible scheme for inserting data processes into the control communications. The reshuffled handshaking expansions given represent possible, although not necessarily optimal, implementations of a data communication. Operator-level implementations of these processes are shown in Figures 3.4 through 3.7.

Passive $X!x$

$$*[[\overline{X}]; D; X!x; D] \quad \triangleright \quad *[[xi]; do\uparrow; [di]; [x \rightarrow x1o\uparrow \ | \neg x \rightarrow x0o\uparrow]; \\ [\neg xi]; x1o\downarrow, x0o\downarrow; do\downarrow; [\neg di]]$$

$$\triangleright \quad *[[xi]; do\uparrow; [di]; [x \rightarrow x1o\uparrow \ | \neg x \rightarrow x0o\uparrow]; \\ [\neg xi]; do\downarrow; [\neg di]; x1o\downarrow, x0o\downarrow]$$

Active $X!x$

$$*[D; X!x; D] \quad \triangleright \quad *[[di]; do\uparrow; [x \rightarrow x1o\uparrow \ | \neg x \rightarrow x0o\uparrow]; \\ [xi]; x1o\downarrow, x0o\downarrow; [\neg xi]; [\neg di]; do\downarrow]$$

$$\triangleright \quad *[[di]; [x \rightarrow x1o\uparrow \ | \neg x \rightarrow x0o\uparrow]; \\ [xi]; do\uparrow; [\neg di]; x1o\downarrow, x0o\downarrow; [\neg xi]; do\downarrow]$$

Active $X?x$

$$*[D; X?x; D] \quad \triangleright \quad *[[di]; do\uparrow; xo\uparrow; [x1i \rightarrow x\uparrow \ | x0i \rightarrow x\downarrow]; \\ xo\downarrow; [\neg x1i \wedge \neg x0i]; [\neg di]; do\downarrow]$$

$$\triangleright \quad *[[di]; xo\uparrow; [x1i \rightarrow x\uparrow \ | x0i \rightarrow x\downarrow]; \\ do\uparrow; [\neg di]; xo\downarrow; [\neg x1i \wedge \neg x0i]; do\downarrow]$$

Passive $X?x$

$$
\begin{aligned}
*[[\overline{X}]; D; X?x; D] \;\triangleright\; & *[[x1i \vee x0i]; do\uparrow; [di]; [x1i \rightarrow x\uparrow \,|\, x0i \rightarrow x\downarrow]; \\
& xo\uparrow; [\neg x1i \wedge \neg x0i]; xo\downarrow; do\downarrow; [\neg di]] \\
\triangleright\; & *[[x1i \vee x0i]; do\uparrow; [di]; [x1i \rightarrow x\uparrow \,|\, x0i \rightarrow x\downarrow]; \\
& xo\uparrow; [\neg x1i \wedge \neg x0i]; do\downarrow; [\neg di]; xo\downarrow]
\end{aligned}
$$

Figure 3.5 shows an alternative implementation of the handshaking expansion for an output unit. (The circuit is the same in both the active and the passive cases.) In this implementation, the value of the data variable x can be changed at any moment after one of the output wires becomes high. This implementation avoids an isochronic fork between the control and data parts when sophisticated sequencing is employed in the control part. See Chapter 6.

Figure 3.9 shows an alternative to the standard implementation (Figure 3.8) of a one-bit datapath using a passive-input and an active-output. Because of its simplicity, we will use this implementation whenever a passive-input/active-output unit is required.

3.2.4 Multiple-Bit Datapaths and Completion Trees

The above handshaking expansions and circuits implement a single-bit datapath. Multiple-bit datapaths can be derived through another application of process decomposition. The resulting implementation is an instance of a single-bit datapath process for each of the multiple-bits, and a completion detection process that is used to collect the acknowledgment signals returning to the control part. A completion-detection process consists of a tree of C-elements.

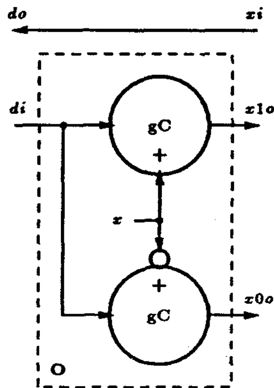


Figure 3.4: Implementation of an output unit. The circuit is the same for both the active and passive cases.

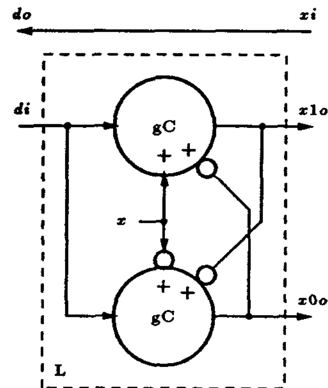


Figure 3.5: Alternative implementation of an output unit. The data variable x can safely change as soon as $x1o$ or $x0o$ rises.

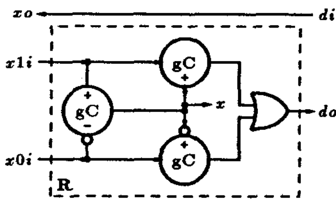


Figure 3.6: Implementation of an active-input unit.

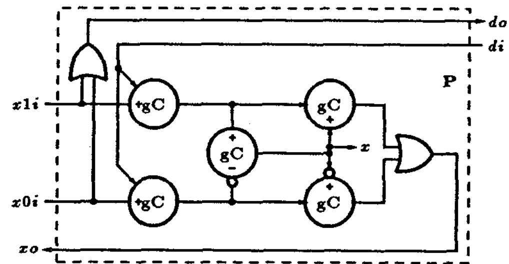


Figure 3.7: Implementation of a passive-input unit.

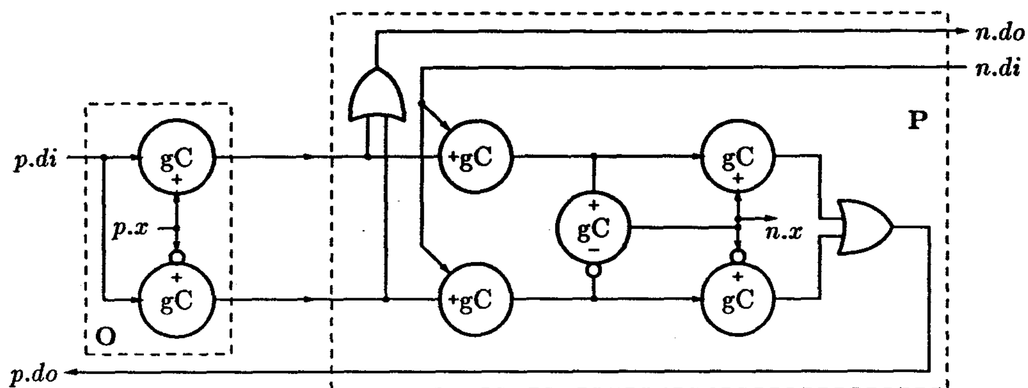


Figure 3.8: Standard implementation of a passive-input/active-output datapath.

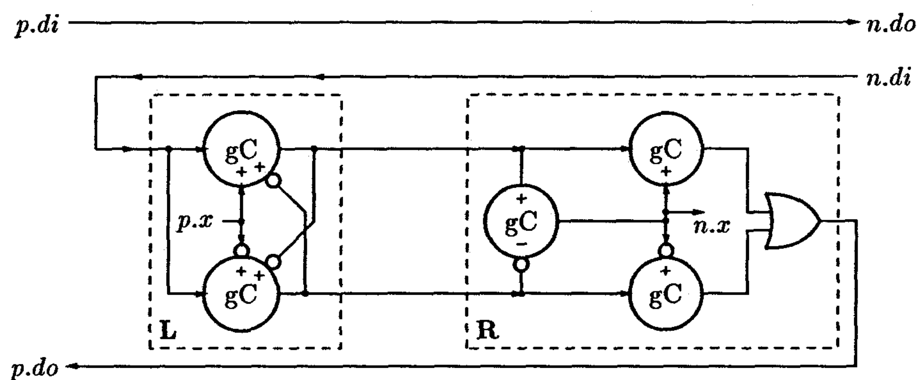


Figure 3.9: Alternative implementation of a passive-input/active-output datapath. This implementation eliminates one OR gate and two state-holding elements per bit, and eliminates a completion tree.

3.2.5 Strengthening the Production Rules

Handshaking expansions are transformed into production rules by restricting the firing of an assignment until after the previous assignment in the sequence has fired and the boolean expression in the intervening wait evaluates to true. To ensure that the assignments are not performed out of order, the current position in the handshaking sequence is also needed to restrict the firing. Sometimes, the existing variables of the process can be used to distinguish between the positions in the handshaking sequence.

3.2.6 Introducing State Variables

Sometimes there are insufficient variables in the process to distinguish the states of the handshaking expansion. New variables called *state variables* are introduced. The position in the handshaking expansion where these variables are introduced can have an effect on the performance of the system. From the point of view of performance, a good place to add a state variable assignment is after the raising or lowering of an output variable of a communication action. Then, the delay in changing the value of the variable is concurrent with the delay of the communication action.

3.2.7 Bubble Shuffling

Before production rules are mapped into operators, it is desirable to change the sense of the variables in the production rules so that all inputs constraining a downward transition on the output variable are negated and all inputs constraining an upward transition are non-negated. This facilitates the translation to a CMOS circuit. This transformation is called *bubble shuffling*, because the "bubbles" representing the negation of a signal are "shuffled" back and forth from the input of one operator to the output of another. The quality of this transformation also has a large impact on the performance of the system since extra operators and thus delays are required to

implement the inversion of signals.

3.2.8 Decomposition of Large Elements

The particular technology used limits the allowable size of the boolean expression that guards the firing of a production rule. For a CMOS implementation, the maximum number of literals in a conjunction is from four to six literals. In some GaAs logic families, the limit is one or two literals in a conjunction. The existing techniques for decomposing large elements consist of introducing new state variables into the handshaking expansion and then rederiving the production rules.

3.2.9 Resetting to the Initial State

Extra circuitry must be added to force the circuit into the correct initial state. Production rules for raising or lowering a circuit variable must be added. To ensure noninterference, some production rules must be prevented from firing while the circuit is resetting. Which production rules are cut can have an influence on the performance of the resulting circuit.

A third class of programs cannot be transformed into repetitive *ER* systems. Programs in this class describe ***inherently disjunctive computations***, and are explained in Section 4.5.

4.1 Modeling Delays

In the remainder of this chapter, we restrict the program representations to handshaking expansions and production rule sets. We do this so that the primitive actions of the programs are assignments to boolean variables. The act of performing an assignment that changes a boolean variable, that is performing a *non-vacuous* assignment, is called a *transition.* A transition on a boolean variable v is denoted by $v\uparrow$ if the transition is from **false** to **true** and $v\downarrow$ if the transition is from **true** to **false**. This use of the notation $v\uparrow$ should not be confused with the assignment $v :=$ **true**. A transition defined this way corresponds directly to a transition of a repetitive *ER* system. An occurrence of a transition corresponds directly to an event in the (general) *ER* system generated from a repetitive *ER* system. For example, occurrence 7 of an up-going transition on variable v is denoted $\langle v\uparrow, 7\rangle$.

A handshaking expansion or a production rule set imposes an ordering on these indexed occurrences of transitions. Ordinarily, only an ordering is imposed, but because we are interested in the performance of the program, we introduce delays in addition to the ordering. We typically do not know exact values for the delays we introduce. Therefore, we denote the delay value with a parameter. We have two conventions for naming the delay parameters. Consider the rule

$$\langle u\uparrow, i-\varepsilon\rangle \overset{\alpha}{\mapsto} \langle v\uparrow, i\rangle$$

In the *target-name* convention, we use $\alpha_{v\uparrow}$ as the name of the delay. In the *full-name*

convention, we use $\alpha_{u\uparrow v\uparrow}$ as the name of the delay.

If the full-name convention is used, rule $e \overset{\alpha_{ef}}{\mapsto} f$ cannot always be removed from the collection:

$$\{e \overset{\alpha_{ef}}{\mapsto} f,\ e \overset{\alpha_{eg}}{\mapsto} g,\ g \overset{\alpha_{gf}}{\mapsto} f\}.$$

This is because if $\alpha_{ef} > \alpha_{eg} + \alpha_{gf}$, event f will not be properly constrained if $e \overset{\alpha_{ef}}{\mapsto} f$ is eliminated. However, using the target-name convention, $\alpha_{ef} = \alpha_{gf} = \alpha_f$, rule $e \overset{\alpha_{ef}}{\mapsto} f$ can be safely removed. The target-name convention is used when modeling handshaking expansions and the full-name convention is used when modeling production rule sets.

4.2 Modeling Handshaking Expansions

We now transform two example handshaking expansion into repetitive *ER* systems using ad hoc techniques. In Section 4.2.1, we develop a general algorithm.

As a simple example, we will transform a program composed of three concurrent handshaking expansions that behave in a completely sequential manner, into a repetitive *ER* system.

Example 4.1 The handshaking expansion for a sequencing process (D element) is as follows:

$$D \equiv *[[li]; lo\uparrow; [\neg li]; ro\uparrow; [ri]; ro\downarrow; [\neg ri]; lo\downarrow]\ .$$

In order to make a closed-system, we add the two processes

$$E_l \equiv *[li\uparrow; [lo]; li\downarrow; [\neg lo]]$$
$$E_r \equiv *[[ro]; ri\uparrow; [\neg ro]; ri\downarrow]$$

that describe the environment connecting to the left and right ports of the D-element, respectively. Initially, all program variables are **false.** We will execute this closed system of handshaking expansions in order to construct the corresponding *ER* system. The first event that can fire is $\langle li\uparrow, 0\rangle$ in process E_l, to be followed by $\langle lo\uparrow, 0\rangle$ in process D. Continuing, we see that the execution of this closed system is completely sequential:

$$\ldots\langle li\downarrow, 0\rangle; \langle ro\uparrow, 0\rangle; \langle ri\uparrow, 0\rangle; \langle ro\downarrow, 0\rangle; \langle ri\downarrow, 0\rangle; \langle lo\downarrow, 0\rangle; \langle li\uparrow, 1\rangle; \langle lo\uparrow, 1\rangle\ldots$$

In such a simple example, the repetitive nature can be immediately recognized, and we get

$$\begin{aligned}
\langle lo\downarrow, i-1\rangle &\overset{\alpha_{li\uparrow}}{\mapsto} \langle li\uparrow, i\rangle, i > 0\\
\langle li\uparrow, i\rangle &\overset{\alpha_{lo\uparrow}}{\mapsto} \langle lo\uparrow, i\rangle\\
\langle lo\uparrow, i\rangle &\overset{\alpha_{li\downarrow}}{\mapsto} \langle li\downarrow, i\rangle\\
\langle li\downarrow, i\rangle &\overset{\alpha_{ro\uparrow}}{\mapsto} \langle ro\uparrow, i\rangle\\
\langle ro\uparrow, i\rangle &\overset{\alpha_{ri\uparrow}}{\mapsto} \langle ri\uparrow, i\rangle\\
\langle ri\uparrow, i\rangle &\overset{\alpha_{ro\downarrow}}{\mapsto} \langle ro\downarrow, i\rangle\\
\langle ro\downarrow, i\rangle &\overset{\alpha_{ri\downarrow}}{\mapsto} \langle ri\downarrow, i\rangle\\
\langle ri\downarrow, i\rangle &\overset{\alpha_{lo\downarrow}}{\mapsto} \langle lo\downarrow, i\rangle
\end{aligned}$$

as the rule set of a repetitive *ER* system. □

In the next example, we consider a more concurrent computation. Although each process is sequential, the activity of the closed system is highly concurrent.

Example 4.2 Consider the closed system

$$\begin{aligned} BD &\equiv *[[li]; lo\uparrow; [\neg ri]; ro\uparrow; [\neg li]; lo\downarrow; [ri]; ro\downarrow] \\ E_l &\equiv *[li\uparrow; [lo]; li\downarrow; [\neg lo]] \\ E_r &\equiv *[[ro]; ri\uparrow; [\neg ro]; ri\downarrow] \end{aligned}$$

where initially all the variables are **false**. We cannot write down a sequential execution of this closed system as we could in Example 4.1. However, we can generate the constraints separately for the process BD and for the two environment processes. By examining this handshaking expansion, we see that $\langle lo\uparrow, 0\rangle$ is constrained to fire after $\langle li\uparrow, 0\rangle$. Then $\langle ro\uparrow, 0\rangle$ can only fire after $\langle lo\uparrow, 0\rangle$. An occurrence of $ri\downarrow$ does not constrain $\langle ro\uparrow, 0\rangle$ because $\neg ri$ is true in the initial state. The next assignment $\langle lo\downarrow, 0\rangle$ is constrained by the two events $\langle ro\uparrow, 0\rangle$ and $\langle li\downarrow, 0\rangle$, and similarly $\langle ro\downarrow, 0\rangle$ occurs only after $\langle lo\downarrow, 0\rangle$ and $\langle ri\uparrow, 0\rangle$. Starting over at the beginning of the repetition, the assignment $\langle lo\uparrow, 1\rangle$ depends on both $\langle ro\downarrow, 0\rangle$ and $\langle li\uparrow, 1\rangle$. In this case there is a dependence between a 0th occurrence of transition $ro\downarrow$ and the 1st occurrence of transition $lo\uparrow$. And finally, we see that $\langle ro\uparrow, 1\rangle$ depends on both $\langle lo\uparrow, 1\rangle$ and $\langle ri\downarrow, 0\rangle$. Continuing the execution adds no new dependencies. The environmental processes

are easily transformed, resulting in the complete rule set

$$
\begin{aligned}
\langle lo\uparrow, i\rangle &\overset{\alpha_{ro\uparrow}}{\mapsto} \langle ro\uparrow, i\rangle \\
\langle ro\uparrow, i\rangle &\overset{\alpha_{lo\downarrow}}{\mapsto} \langle lo\downarrow, i\rangle \\
\langle li\downarrow, i\rangle &\overset{\alpha_{lo\downarrow}}{\mapsto} \langle lo\downarrow, i\rangle \\
\langle lo\downarrow, i\rangle &\overset{\alpha_{ro\downarrow}}{\mapsto} \langle ro\downarrow, i\rangle \\
\langle ri\uparrow, i\rangle &\overset{\alpha_{ro\downarrow}}{\mapsto} \langle ro\downarrow, i\rangle \\
\langle ro\downarrow, i-1\rangle &\overset{\alpha_{lo\uparrow}}{\mapsto} \langle lo\uparrow, i\rangle, i>0 \\
\langle li\uparrow, i\rangle &\overset{\alpha_{lo\uparrow}}{\mapsto} \langle lo\uparrow, i\rangle \\
\langle ri\downarrow, i-1\rangle &\overset{\alpha_{ro\uparrow}}{\mapsto} \langle ro\uparrow, i\rangle, i>0
\end{aligned}
\qquad
\begin{aligned}
\langle lo\downarrow, i-1\rangle &\overset{\alpha_{li\uparrow}}{\mapsto} \langle li\uparrow, i\rangle, i>0 \\
\langle lo\uparrow, i\rangle &\overset{\alpha_{li\downarrow}}{\mapsto} \langle li\downarrow, i\rangle \\
\langle ro\uparrow, i\rangle &\overset{\alpha_{ri\uparrow}}{\mapsto} \langle ri\uparrow, i\rangle \\
\langle ro\downarrow, i\rangle &\overset{\alpha_{ri\downarrow}}{\mapsto} \langle ri\downarrow, i\rangle
\end{aligned}
$$

for the repetitive *ER* system corresponding to the closed system. □

4.2.1 Straight-Line Handshaking Expansions

A *straight-line handshaking expansion* (*SLHE*) is a handshaking expansion where each select statement is of the form $[C \longrightarrow \textbf{skip}]$, with C being a conjunction of literals, and each repetition statement is of the form $*[\textbf{true} \longrightarrow S]$. A *SLHE* has a *vacuous wait on the literal* ℓ if ℓ is **true** in the initial state, and if as the process is executed starting from the initial state, the control flow of the process crosses a wait $[\ell \wedge C']$ without ℓ ever having been falsified. Similarly, a *SLHE* has a *vacuous assignment* if during the execution of the process there is an assignment that does not change the program state. A *SLHE* has a *repeated* assignment if within the same repetition statement, there are two assignments of the same value to the same variable.

Example 4.3 The buffer process

$$delayedC \;\equiv\; *[lo\uparrow; [li \wedge \neg ri]; ro\downarrow; lo\downarrow; [\neg li \wedge ri]; ro\uparrow]$$

has one vacuous wait on literal $\neg ri$ and one vacuous assignment ($ro\downarrow$). The toggle process

$$toggle \equiv *[[li]; lo\uparrow; [\neg li]; lo\downarrow; [li]; lo\uparrow; [\neg li]; lo\downarrow; ro\uparrow; [ri]; ro\downarrow; [\neg ri]]$$

has two repeated assignments ($lo\uparrow$ and $lo\downarrow$). □

To ease the translation to an *ER* system, we restrict the class of *SLHE* collections to those that satisfy the following conditions:

1. The collection is deadlock-free.

2. Each process P is of the form $S; *[T]$ and contains no vacuous waits (on any literal) or assignments.

3. S and T are sequences of alternating waits and assignments. (If $a_0; a_1$ and $[c_0]; [c_1]$, then replace with $a_0; [\mathbf{true}]; a_1$ and $[c_0 \wedge c_1]$, respectively.)

4. S begins with a wait and ends with a wait (could be just $[\mathbf{true}]$).

5. T begins with an assignment and ends with a wait.

6. Each variable appears in one process only and is either a local variable or a variable of a communication action.

7. If a variable v appears in a wait of one process, then v is either local to this process or else not assigned to in any process and is the input variable of a communication action with its corresponding output variable assigned to only in this process.

8. All variables are initially **false**.

A collection of straight-line handshaking expansions satisfying these restrictions is said to be in *standard form.*

Example 4.4 The collections of handshaking expansions in Examples 4.1 and 4.2 are not in standard form. In particular, they both violate Rules 6 and 7 because input variables of a communication action are directly assigned to in the environment processes. We now put the closed system of Example 4.1 in standard form.

$$\begin{aligned} D &\equiv [li]; *[lo\uparrow; [\neg li]; ro\uparrow; [ri]; ro\downarrow; [\neg ri]; lo\downarrow; [li]] \\ E_l &\equiv [\mathbf{true}]; *[xo\uparrow; [xi]; xo\downarrow; [\neg xi]] \\ E_r &\equiv [yi]; *[yo\uparrow; [\neg yi]; yo\downarrow; [yi]] \end{aligned}$$

The handshake variables li and lo are connected to xo and xi, respectively. Similarly, ro and ri are connected to yi and yo. □

Example 4.5 The handshaking expansions from Example 4.3 correspond to these standard form expansions:

$$\begin{aligned} \mathit{delayedC} &\equiv [\mathbf{true}]; lo\uparrow; [li]; *[lo\downarrow; [\neg li \wedge ri]; ro\uparrow; [\mathbf{true}]; lo\uparrow; [li \wedge \neg ri]; ro\downarrow; [\mathbf{true}]] \\ \mathit{toggle} &\equiv [li]; *[lo\uparrow; [\neg li]; lo\downarrow; [li]; lo\uparrow; [\neg li]; lo\downarrow; [\mathbf{true}]; ro\uparrow; [ri]; ro\downarrow; [\neg ri \wedge li]] \end{aligned}$$

□

The events and rules of the *ER* system are generated individually for each process directly from the handshaking expansion. Events and rules are also added to implement the constraints corresponding to the communication actions.

We begin by generating the events and rules contributed by a single handshaking process $P = S; *[T]$. We attach to each assignment a of the handshaking expansion the occurrence index i_a of that assignment; that is, the number of times an identical

assignment (on the same variable and to the same value) precedes a in the sequence $S;T$. For the first assignment a of the loop we will attach a second occurrence index corresponding to the number of times a occurs in $S;T$. This second index represents the occurrence of assignment a on the second trip through the loop.

Example 4.6 Continuing with the previous example:

$$\begin{aligned} \mathit{delayedC} \;\equiv\;& [\mathbf{true}]; lo^{(0)}\uparrow; [li]; *[lo^{(0),(1)}\downarrow; [\neg li \wedge ri]; ro^{(0)}\uparrow; [\mathbf{true}]; \\ & \qquad lo^{(1)}\uparrow; [li \wedge \neg ri]; ro^{(0)}\downarrow; [\mathbf{true}]] \\ \mathit{toggle} \;\equiv\;& [li]; *[lo^{(0),(2)}\uparrow; [\neg li]; lo^{(0)}\downarrow; \\ & \quad [li]; lo^{(1)}\uparrow; [\neg li]; lo^{(1)}\downarrow; \\ & \quad [\mathbf{true}]; ro^{(0)}\uparrow; [ri]; ro^{(0)}\downarrow; [\neg ri \wedge li]] \end{aligned}$$

□

We now determine the occurrence indices of the literals in the waits. For each literal ℓ of wait w, we define i_ℓ in terms of the occurrence indices of the assignments. If the variable v of ℓ is local, then we look backwards from w in the sequence $S;T$ until we find the assignment $v\uparrow$ if $\ell = v$ or the assignment $v\downarrow$ if $\ell = \neg v$, and then use the occurrence index previously found for that assignment. If vi is the variable of ℓ and is the input variable of a communication action V, then we can use the occurrence indices of the assignments to the output variable vo of V in order to determine the occurrence index i_{vi}. Now there are four cases to consider, as shown in Table 4.1, which correspond to whether the communication action is passive or active, and to whether the input variable is non-negated or negated. In every case except the first, we use the occurrence index of the last assignment on vo. In the first case (passive, non-negated), we use one plus this number, because the transition on vi is the first transition of the communication action. In this case as well, it is possible that no previous assignment to vo exists. If this is so, we use 0 for the occurrence index of vi.

Type	Fragment of Handshake Sequence
Passive	$\ldots; vo^{(k)}\downarrow; \ldots; [vi^{(k+1)} \wedge \ldots]; \ldots$
Passive	$\ldots; vo^{(k)}\uparrow; \ldots; [\neg vi^{(k)} \wedge \ldots]; \ldots$
Active	$\ldots; vo^{(k)}\uparrow; \ldots; [vi^{(k)} \wedge \ldots]; \ldots$
Active	$\ldots; vo^{(k)}\downarrow; \ldots; [\neg vi^{(k)} \wedge \ldots]; \ldots$

Table 4.1: Correspondence between occurrence indices of the input and output variables of a communication action. This correspondence is only valid if all handshaking variables are false in the initial state.

Example 4.7 Continuing with the previous example:

$$\begin{aligned} \mathit{delayedC} \;\equiv\;& [\mathbf{true}]; lo^{(0)}\uparrow; [li^{(0)}]; *[lo^{(0),(1)}\downarrow; [\neg li^{(0)} \wedge ri^{(0)}]; ro^{(0)}\uparrow; [\mathbf{true}];\\ & lo^{(1)}\uparrow; [li^{(1)} \wedge \neg ri^{(0)}]; ro^{(0)}\downarrow; [\mathbf{true}]]\\ \mathit{toggle} \;\equiv\;& [li^{(0)}]; *[lo^{(0),(2)}\uparrow; [\neg li^{(0)}]; lo^{(0)}\downarrow;\\ & [li^{(1)}]; lo^{(1)}\uparrow; [\neg li^{(1)}]; lo^{(1)}\downarrow;\\ & [\mathbf{true}]; ro^{(0)}\uparrow; [ri^{(0)}]; ro^{(0)}\downarrow; [\neg ri^{(0)} \wedge li^{(2)}]] \end{aligned}$$

□

We are now in a position to determine the contribution to the event set E and rule set R that corresponds to each assignment a in $S; *[T]$. To determine the assignment a' and the wait w that constrain the firing of a, we refer to Table 4.2. We form the sets E^* and R^* for each assignment a in the handshaking expansion:

$$\begin{aligned} E^* &= E_w \cup E_{a'} \cup \{\langle a, i_a \rangle\}\\ R^* &= \{e \overset{\alpha_q}{\mapsto} \langle a, i_a \rangle \mid e \in E_w \cup E_{a'}\} \end{aligned}$$

Type of Assignment	Matches With$S;*[T]$
first assignment of S	$w;a;S'';*[T]$
subsequent assignments of S	$S';a';w;a;S'';*[T]$
first assignment of T (first trip)	$w;*[a;T']$
	$S';a';w;*[a;T']$
subsequent assignments of T	$S;*[T';a';w;a;T'']$
first assignment of T (second trip)	$S;*[a;T';a';w]$

Table 4.2: Possible assignment and wait patterns of a *SLHE* in standard form. We use this table to determine the wait w and previous assignment a' that constrain a particular occurrence of assignment a.

Type	Literal ℓ	Event $\langle u,i\rangle$	Subsumes
Passive	vi	$\langle vi\uparrow,i\rangle$	$\langle vo\downarrow,i-1\rangle$ and $\langle vo\uparrow,i-1\rangle$
Passive	$\neg vi$	$\langle vi\downarrow,i\rangle$	$\langle vo\uparrow,i\rangle$ and $\langle vo\downarrow,i-1\rangle$
Active	vi	$\langle vi\uparrow,i\rangle$	$\langle vo\uparrow,i\rangle$ and $\langle vo\downarrow,i-1\rangle$
Active	$\neg vi$	$\langle vi\downarrow,i\rangle$	$\langle vo\downarrow,i\rangle$ and $\langle vo\uparrow,i\rangle$

Table 4.3: Subsumption of rules under the target-name convention for delays. The subsumes column shows the latest assignments on vo that are subsumed by $\langle u,i\rangle$ under this convention. Under the full-name convention for delays, no subsumption is possible.

where

$$E_w = \{\langle u,i_u\rangle \mid \ell \text{ is a literal of } w \text{ and transition } u \text{ corresponds to } \ell\}$$

$$E_{a'} = \begin{cases} \{\langle a',i'_a\rangle\} & \text{if } a' \text{ exists and is not subsumed by } e \in E_w \\ \emptyset & \text{otherwise} \end{cases}$$

Event $\langle a',i_{a'}\rangle$ is included in $E_{a'}$ if it is not subsumed by one of the events in the wait. The subsumption rules are summarized in Table 4.3. The contribution to sets E and R from the current assignment a can be easily generated from sets E^* and R^*. For the

initial assignments—those in S or the first assignment of T (first trip)—sets E^* and R^* are included into sets E and R by the set-union operation. For the other (repeated) assignments, sets E^* and R^* are instantiated for all trips through the loop and are then included into sets E and R. So if $\langle a, i_a \rangle$ is a member of E^*, then we include in E all the events $\langle a, i_a + n_a i \rangle$ for all $i \geq 0$, where n_a is the number of assignments identical to a that occur in the loop body T. Furthermore, if $\langle u, i_u \rangle \overset{\alpha_a}{\mapsto} \langle a, i_a \rangle$ is a member of R^*, then we include in R all the rules $\langle u, i_u + n_u i \rangle \overset{\alpha_a}{\mapsto} \langle a, i_a + n_a i \rangle$, again for all $i \geq 0$.

Example 4.8 Continuing with just the toggle process:

$$\begin{aligned} toggle \;\equiv\; & [li^{(0)}]; *[lo^{(0),(2)}\uparrow; [\neg li^{(0)}]; lo^{(0)}\downarrow; \\ & \quad [li^{(1)}]; lo^{(1)}\uparrow; [\neg li^{(1)}]; lo^{(1)}\downarrow; \\ & \quad [\mathbf{true}]; ro^{(0)}\uparrow; [ri^{(0)}]; ro^{(0)}\downarrow; [\neg ri^{(0)} \wedge li^{(2)}]] \end{aligned}$$

The first assignment, $lo^{(0)}\uparrow$, is constrained by the last wait of the head:

$$\begin{aligned} E_w &= \{\langle li\uparrow, 0\rangle\} \\ E_{a'} &= \emptyset \\ E^* &= \{\langle li\uparrow, 0\rangle, \langle lo\uparrow, 0\rangle\} \\ R^* &= \{\langle li\uparrow, 0\rangle \overset{\alpha_{lo\uparrow}}{\mapsto} \langle lo\uparrow, 0\rangle\} \end{aligned}$$

The second assignment, $lo^{(0)}\downarrow$ produces:

$$\begin{aligned} E_w &= \{\langle li\downarrow, 0\rangle\} \\ E_{a'} &= \emptyset, \text{ event } \langle lo\uparrow, 0\rangle \text{ subsumed.} \\ E^* &= \{\langle li\downarrow, 0\rangle, \langle lo\downarrow, 0\rangle\} \end{aligned}$$

$$R^* = \{\langle li\downarrow, 0\rangle \overset{\alpha_{lo\downarrow}}{\mapsto} \langle lo\downarrow, 0\rangle\}$$

Continuing, we see that the subsequent assignments, $lo^{(1)}\uparrow$, $lo^{(1)}\downarrow$, $ro^{(0)}\uparrow$, $ro^{(0)}\downarrow$ and $lo^{(2)}\uparrow$, generate the following R^* sets:

$$\begin{array}{ll}
\text{for } lo^{(1)}\uparrow, & R^* = \{\langle li\uparrow, 1\rangle \overset{\alpha_{lo\uparrow}}{\mapsto} \langle lo\uparrow, 1\rangle\} \\
\text{for } lo^{(1)}\downarrow, & R^* = \{\langle li\downarrow, 1\rangle \overset{\alpha_{lo\downarrow}}{\mapsto} \langle lo\downarrow, 1\rangle\} \\
\text{for } ro^{(0)}\uparrow, & R^* = \{\langle lo\downarrow, 1\rangle \overset{\alpha_{ro\uparrow}}{\mapsto} \langle ro\uparrow, 0\rangle\} \\
\text{for } ro^{(0)}\downarrow, & R^* = \{\langle ri\uparrow, 0\rangle \overset{\alpha_{ro\downarrow}}{\mapsto} \langle ro\downarrow, 0\rangle\} \\
\text{for } lo^{(2)}\uparrow, & R^* = \{\langle ri\downarrow, 0\rangle \overset{\alpha_{lo\uparrow}}{\mapsto} \langle lo\uparrow, 2\rangle, \langle li\uparrow, 2\rangle \overset{\alpha_{lo\uparrow}}{\mapsto} \langle lo\uparrow, 2\rangle\}
\end{array}$$

The rules for all subsequent trips through the loop are formed from the rules generated during the first trip through the loop. All the rules except the rule using constraining events from the head S are replicated for each $i \geq 0$. The repetition factor, n_a, is multiplied by the loop index i and added to occurrence indices determined previously. All transitions on the handshaking variables li and lo have repetition factor 2. The transitions on ri and ro have repetition factor 1. Thus, the complete rule system contributed by this process is composed of the single rule

$$\langle li\uparrow, 0\rangle \overset{\alpha_{lo\uparrow}}{\mapsto} \langle lo\uparrow, 0\rangle$$

and the rules

$$\begin{array}{lcl}
\langle li\downarrow, 2i+0\rangle & \overset{\alpha_{lo\downarrow}}{\mapsto} & \langle lo\downarrow, 2i+0\rangle \\
\langle li\uparrow, 2i+1\rangle & \overset{\alpha_{lo\uparrow}}{\mapsto} & \langle lo\uparrow, 2i+1\rangle \\
\langle li\downarrow, 2i+1\rangle & \overset{\alpha_{lo\downarrow}}{\mapsto} & \langle lo\downarrow, 2i+1\rangle \\
\langle lo\downarrow, 2i+1\rangle & \overset{\alpha_{ro\uparrow}}{\mapsto} & \langle ro\uparrow, i+0\rangle
\end{array}$$

$$\langle ri\uparrow, i+0\rangle \overset{\alpha_{ro\downarrow}}{\mapsto} \langle ro\downarrow, i+0\rangle$$
$$\langle ri\downarrow, i+0\rangle \overset{\alpha_{lo\uparrow}}{\mapsto} \langle lo\uparrow, 2i+2\rangle$$
$$\langle li\uparrow, 2i+2\rangle \overset{\alpha_{lo\uparrow}}{\mapsto} \langle lo\uparrow, 2i+2\rangle$$

instantiated for all $i \geq 0$. □

4.2.2 Communication Between Processes

We must now add the events and rules corresponding to the synchronization between processes. If *li* and *lo* represent the handshaking variables of a communication action in one process and, if *ri* and *ro* represent the corresponding handshaking variables in a second process, then for all $i \geq 0$, the following should be added to the rule set, regardless of which port is active and which is passive:

$$\langle lo\uparrow, i\rangle \overset{\alpha_{ri\uparrow}}{\mapsto} \langle ri\uparrow, i\rangle$$
$$\langle lo\downarrow, i\rangle \overset{\alpha_{ri\downarrow}}{\mapsto} \langle ri\downarrow, i\rangle$$
$$\langle ro\uparrow, i\rangle \overset{\alpha_{li\uparrow}}{\mapsto} \langle li\uparrow, i\rangle$$
$$\langle ro\downarrow, i\rangle \overset{\alpha_{li\downarrow}}{\mapsto} \langle li\downarrow, i\rangle\ .$$

4.2.3 Conversion From a General *ER* System to a Pseudorepetitive *ER* System

The constructions up to this point produce general *ER* system. Most of the time the transformation into pseudorepetitive form is trivial. However, if there are repeated assignments in some processes but not in others, we cannot always find a simple correspondence between the events of processes connected by a communication channel. To ensure the existence of this correspondence, we must replicate the repeated

1. **Partition the q processes $P_0, P_1, \ldots, P_{q-1}$ into singleton partition blocks.**
2. **For each communication action between a port L in process P_i and port R in process P_j,**
   (a) **If P_i and P_j are in the same partition block and $n_{lo\uparrow} \neq n_{ro\uparrow}$, then abort. (The system is not deadlock-free.)**
   (b) **Let $n = \text{lcm}(n_{lo\uparrow}, n_{ro\uparrow})$.**
   (c) **Unroll $\frac{n}{n_{lo\uparrow}}$ times the loop of each process in the partition block containing P_i**
   (d) **Unroll $\frac{n}{n_{ro\uparrow}}$ times the loop of each process in the partition block containing P_j**
   (e) **Merge the partition blocks containing P_i and P_j into a single block.**

Algorithm 4.1: Algorithm to produce an equivalent set of processes with no repeated assignments.

parts of the processes—the T sequences—until the number of occurrences in T of a communication action C is the same for both of the processes paired by C.

Algorithm 4.1 can be applied to any collection of straight-line handshaking expansions in standard form. Please note, however, that the size of the collection of unrolled processes may be exponentially larger (in the number of communication channels) than the original collection.

Example 4.9 Consider the *SLHE* collection in standard form:

$$P_0 \equiv [\mathbf{true}]; *[r0o\uparrow; [r0i]; r0o\downarrow; [\neg r0i]]$$

$$P_1 \equiv [l0i]; *[l0o\uparrow; [\neg l0i]; l0o\downarrow; [l0i]; l0o\uparrow; [\neg l0i]; l0o\downarrow; [\mathbf{true}]; r1o\uparrow; [r1i]; r1o\downarrow; [\neg r1i \wedge l0i]]$$

$$P_2 \equiv [l1i]; *[l1o\uparrow; [\neg l1i]; l1o\downarrow; [l1i]; l1o\uparrow; [\neg l1i]; l1o\downarrow; [\mathbf{true}]; r2o\uparrow; [r2i]; r2o\downarrow; [\neg r2i \wedge l1i]]$$

$$P_3 \equiv [l2i]; *[l2o\uparrow; [\neg l2i]; l2o\downarrow; [l2i]]$$

where $R0$ connects to $L0$, $R1$ connects to $L1$ and $R2$ connects to $L2$. For the pair of ports $R0$ and $L0$, we see that

$$n = \mathrm{lcm}(n_{r0o\uparrow}, n_{l0o\uparrow}) = \mathrm{lcm}(1,2) = 2\,.$$

Thus the loop of P_0 should be duplicated.

$$P_0 \;\equiv\; [\mathbf{true}]; *[r0o\uparrow; [r0i]; r0o\downarrow; [\neg r0i]; r0o\uparrow; [r0i]; r0o\downarrow; [\neg r0i]]$$

For the pair of ports $R1$ and $L1$,

$$n = \mathrm{lcm}(n_{r1o\uparrow}, n_{l1o\uparrow}) = \mathrm{lcm}(1,2) = 2\,.$$

The loops of P_0 and P_1 are now duplicated.

$$\begin{aligned}
P_0 \;\equiv\;& [\mathbf{true}]; *[r0o\uparrow; [r0i]; r0o\downarrow; [\neg r0i]; r0o\uparrow; [r0i]; r0o\downarrow; [\neg r0i];\\
&\qquad r0o\uparrow; [r0i]; r0o\downarrow; [\neg r0i]; r0o\uparrow; [r0i]; r0o\downarrow; [\neg r0i]]\\
P_1 \;\equiv\;& [l0i]; *[l0o\uparrow; [\neg l0i]; l0o\downarrow; [l0i]; l0o\uparrow; [\neg l0i]; l0o\downarrow; [\mathbf{true}];\\
&\quad r1o\uparrow; [r1i]; r1o\downarrow; [\neg r1i \wedge l0i];\\
&\quad l0o\uparrow; [\neg l0i]; l0o\downarrow; [l0i]; l0o\uparrow; [\neg l0i]; l0o\downarrow; [\mathbf{true}];\\
&\quad r1o\uparrow; [r1i]; r1o\downarrow; [\neg r1i \wedge l0i]]
\end{aligned}$$

For the final pair of ports $R2$ and $L2$,

$$n = \mathrm{lcm}(n_{r2o\uparrow}, n_{l2o\uparrow}) = \mathrm{lcm}(1,1) = 1$$

and no loops need further unrolling. Now all the assignments with $n_a > 1$ are made

unique by attaching a distinguishing subscript:

$$
\begin{aligned}
P_0 \;\equiv\;& [\mathbf{true}]; *[r0o_0\uparrow; [r0i_0]; r0o_0\downarrow; [\neg r0i_0]; \\
& \quad r0o_1\uparrow; [r0i_1]; r0o_1\downarrow; [\neg r0i_1]; \\
& \quad r0o_2\uparrow; [r0i_2]; r0o_2\downarrow; [\neg r0i_2]; \\
& \quad r0o_3\uparrow; [r0i_3]; r0o_3\downarrow; [\neg r0i_3]] \\
P_1 \;\equiv\;& [l0i_0]; *[l0o_0\uparrow; [\neg l0i_0]; l0o_0\downarrow; [l0i_1]; l0o_1\uparrow; [\neg l0i_1]; l0o_1\downarrow; \\
& \quad [\mathbf{true}]; r1o_0\uparrow; [r1i_0]; r1o_0\downarrow; [\neg r1i_0 \wedge l0i_2]; \\
& \quad l0o_2\uparrow; [\neg l0i_2]; l0o_2\downarrow; [l0i_3]; l0o_3\uparrow; [\neg l0i_3]; l0o_3\downarrow; \\
& \quad [\mathbf{true}]; r1o_1\uparrow; [r1i_1]; r1o_1\downarrow; [\neg r1i_1 \wedge l0i_0]] \\
P_2 \;\equiv\;& [l1i_0]; *[l1o_0\uparrow; [\neg l1i_0]; l1o_0\downarrow; [l1i_1]; l1o_1\uparrow; [\neg l1i_1]; l1o_1\downarrow; \\
& \quad [\mathbf{true}]; r2o\uparrow; [r2i]; r2o\downarrow; [\neg r2i \wedge l1i_0]] \\
P_3 \;\equiv\;& [l2i]; *[l2o\uparrow; [\neg l2i]; l2o\downarrow; [l2i]]
\end{aligned}
$$

Subscripts are eliminated when generating the names for the delays. Thus the rule for the first assignment in P_0 (after the first trip through the loop) is:

$$\langle r0i_3\downarrow, i-1\rangle \overset{\alpha_{r0o\uparrow}}{\mapsto} \langle r0o_0\uparrow, i\rangle$$

□

4.2.4 Vacuous Firings

The restriction to *SLHE* collections without vacuous waits or vacuous assignments is artificial and is done only to eliminate special cases from the transformation rules. For each vacuous wait or assignment, we can use an occurrence index of -1 and proceed as before. When generating the sets E and R, we do as before but do not include any events (or rules containing events) having negative indices.

Example 4.10 The original *SLHE* for the *delayedC* process (with vacuous waits and assignments) has the following occurrence numbering:

$$delayedC \equiv *[lo^{(0),(1)}\uparrow; [li^{(0)} \wedge \neg ri^{(-1)}]; ro^{(-1)}\downarrow; lo^{(0)}\downarrow; [\neg li^{(0)} \wedge ri^{(0)}]; ro^{(0)}\uparrow]$$

□

4.2.5 Other Initial States

The only thing that must change if the variables of the handshaking expansion are not all initially **false** is the definition of the active and passive handshaking sequences for a communication action. For example, if vo is initially **true**, then the handshaking sequence

$$\ldots; vo\uparrow; [vi]; vo\downarrow; [\neg vi]; \ldots$$

becomes a passive sequence with the first assignment ($vo\uparrow$) being vacuous. The roles of $vo\uparrow$ and $vo\downarrow$ must also be interchanged in Tables 4.1 and 4.3.

4.2.6 Multiple Assignments

The single assignments considered previously could just as well be multiple assignments. Consider the fragment of the handshaking expansion

$$\ldots; A'; w; A; \ldots$$

where A' and A are sets of primitive assignments instead of the single primitive assignments a' and a as before. In this case the set of events constraining each

primitive assignment in A is $E_w \cup E_{A'}$, where

$$E_{A'} = \{\langle a', i_{a'} \rangle \mid a' \in A' \text{ not subsumed by } e \in E_w\} \ .$$

Thus the events and rules contributed by primitive assignments in A are:

$$\begin{aligned} E^* &= E_w \cup E_{A'} \cup \{\langle a, i_a \rangle \mid a \in A\} \\ R^* &= \{e \overset{\alpha_a}{\mapsto} \langle a, i_a \rangle \mid e \in E_w \cup E_{A'} \wedge a \in A\} \end{aligned}$$

4.2.7 General Parallel Composition Statements

If two general statements (other than simple assignments) are joined by a comma, we can still generate an *ER* system. Consider the handshaking expansion fragment

$$\ldots; ((\ldots; a'; w'), (\ldots; a''; w'')); a; \ldots$$

where a', a'' and a are simple assignments. The set of events constraining a is $E_w \cup E_{w'} \cup E_{a'} \cup E_{a''}$, where E_w and $E_{w'}$ are generated in the usual manner, as are $E_{a'}$ and $E_{a''}$, except that the events in $E_{a'}$ (and similarly for $E_{a''}$) can be subsumed by elements in either $E_{w'}$ or $E_{w''}$ or $E_{a''}$.

In the case of the first assignment in one of the branch statements, we use exactly the same rules as for sequential composition.

4.2.8 Conversion to Simple Repetitive Systems

From Theorem 2.4 we know that to determine the steady-state timing of a pseudorepetitive *ER* system we need only consider the repetitive part of the system. Thus if only steady-state timing is desired, the head sequence of a *SLHE* can be ignored except when determining the occurrence-index offsets.

4.3 Modeling Production Rule Sets

The techniques of the previous section are sufficient to translate all collections of straight-line handshaking expansions into *ER* systems. The following example illustrates how to translate production rule sets derived from straight-line handshaking expansions into *ER* systems.

Example 4.11 Consider the lazy-active/lazy-active buffer:

$$LALA \equiv *[[\neg li]; lo\uparrow; x\uparrow; [li]; lo\downarrow; [\neg ri]; ro\uparrow; x\downarrow; [ri]; ro\downarrow] \ .$$

By applying the Martin synthesis procedure, the following production rules for process *LALA* are derived:

$$\begin{aligned} \neg ro \wedge \neg li \wedge \neg x &\rightarrow lo\uparrow \\ lo &\rightarrow x\uparrow \\ li \wedge x &\rightarrow lo\downarrow \\ \neg lo \wedge \neg ri \wedge x &\rightarrow ro\uparrow \\ ro &\rightarrow x\downarrow \\ ri \wedge \neg x &\rightarrow ro\downarrow \end{aligned}$$

These production rules immediately provide the structure of the corresponding *ER* system. There is no need to determine which assignments and waits influence a given firing through examination of the handshaking expansion. However, to determine the correct ε values for the various rules, information is required about the order in which the production rules fire. Since the rules were synthesized from a sequential specification (the handshaking expansion), we can use this specification to determine the between-occurrence-number dependencies. We can assign the following consistent

occurrence numbering to the variables of the handshaking expansion:

$$\begin{aligned} LALA \;\equiv\; & *[[\neg li^{(-1)}]; lo^{(0)}\uparrow; x^{(0)}\uparrow; [li^{(0)}]; lo^{(0)}\downarrow; \\ & [\neg ri^{(-1)}]; ro^{(0)}\uparrow; x^{(0)}\downarrow; [ri^{(0)}]; ro^{(0)}\downarrow] \;. \end{aligned}$$

The first event $lo\uparrow$ is constrained by $ro\downarrow$, $li\downarrow$ and $x\downarrow$. To determine the occurrence indices, we use the consistent occurrence numbering from above modified by subtracting one if the constraint crosses a loop boundary. Thus we get the three rules constraining $lo\uparrow$ all with $\varepsilon = 1$. All the other rules, with the exception of $ri\downarrow$ constraining $ro\uparrow$, have $\varepsilon = 0$, resulting in the following *ER* system:

$$\begin{aligned}
\langle ro\downarrow, i-1\rangle &\overset{\alpha_{ro\downarrow lo\uparrow}}{\mapsto} \langle lo\uparrow, i\rangle, i > 0 \\
\langle li\downarrow, i-1\rangle &\overset{\alpha_{li\downarrow lo\uparrow}}{\mapsto} \langle lo\uparrow, i\rangle, i > 0 \\
\langle x\downarrow, i-1\rangle &\overset{\alpha_{x\downarrow lo\uparrow}}{\mapsto} \langle lo\uparrow, i\rangle, i > 0 \\
\\
\langle lo\uparrow, i\rangle &\overset{\alpha_{lo\uparrow x\uparrow}}{\mapsto} \langle x\uparrow, i\rangle \\
\\
\langle li\uparrow, i\rangle &\overset{\alpha_{li\uparrow lo\downarrow}}{\mapsto} \langle lo\downarrow, i\rangle \\
\langle x\uparrow, i\rangle &\overset{\alpha_{x\uparrow lo\downarrow}}{\mapsto} \langle lo\downarrow, i\rangle \\
\\
\langle lo\downarrow, i\rangle &\overset{\alpha_{lo\downarrow ro\uparrow}}{\mapsto} \langle ro\uparrow, i\rangle \\
\langle ri\downarrow, i-1\rangle &\overset{\alpha_{ri\downarrow ro\uparrow}}{\mapsto} \langle ro\uparrow, i\rangle, i > 0 \\
\langle x\uparrow, i\rangle &\overset{\alpha_{x\uparrow ro\uparrow}}{\mapsto} \langle ro\uparrow, i\rangle \\
\\
\langle ro\uparrow, i\rangle &\overset{\alpha_{ro\uparrow x\downarrow}}{\mapsto} \langle x\downarrow, i\rangle \\
\\
\langle ri\uparrow, i\rangle &\overset{\alpha_{ri\uparrow ro\downarrow}}{\mapsto} \langle ro\downarrow, i\rangle \\
\langle x\downarrow, i\rangle &\overset{\alpha_{x\downarrow ro\downarrow}}{\mapsto} \langle ro\downarrow, i\rangle
\end{aligned}$$

□

The important thing to notice in this example is that the rules $\langle ro{\downarrow}, i-1\rangle \overset{\alpha_{ro\downarrow lo\uparrow}}{\mapsto} \langle lo{\uparrow}, i\rangle, i > 0$ and $\langle lo{\downarrow}, i\rangle \overset{\alpha_{lo\downarrow ro\uparrow}}{\mapsto} \langle ro{\uparrow}, i\rangle$ have different values of ε because $lo{\uparrow}$ fires before $ro{\uparrow}$ in all executions allowed by the handshaking expansion and thus the production rule set. Without independent knowledge of this execution order, determining the occurrence-index offsets from a production rule set requires simulation of the system. As a second example, we will consider the case of the toggle process.

Example 4.12 Applying Martin synthesis to the handshaking expansion

$$\begin{aligned} toggle \equiv\ & *[[li]; lo{\uparrow}; [\neg li]; v{\uparrow}; lo{\downarrow}; [li]; u{\uparrow}; lo{\uparrow}; [\neg li]; v{\downarrow}; lo{\downarrow}; \\ & ro{\uparrow}; [ri]; u{\downarrow}; ro{\downarrow}; [\neg ri]] \end{aligned}$$

yields the production rule set:

$$\begin{aligned} li \wedge \neg v \wedge \neg u \wedge \neg ri &\rightarrow lo{\uparrow} \\ \neg li \wedge lo \wedge \neg u &\rightarrow v{\uparrow} \\ v \wedge \neg u &\rightarrow lo{\downarrow} \\ li \wedge v &\rightarrow u{\uparrow} \\ v \wedge u &\rightarrow lo{\uparrow} \\ \neg li \wedge u &\rightarrow v{\downarrow} \\ \neg v \wedge u &\rightarrow lo{\downarrow} \\ \neg lo \wedge \neg v \wedge u &\rightarrow ro{\uparrow} \\ ri &\rightarrow u{\downarrow} \\ \neg u &\rightarrow ro{\downarrow} \end{aligned}$$

From the production rule set and the handshaking expansions we generate the following rules for all $i \geq 0$:

$$
\begin{aligned}
\langle li\uparrow, 2i\rangle &\overset{\alpha_{li\uparrow\, lo\uparrow}}{\longmapsto} \langle lo\uparrow, 2i\rangle \\
\langle v\downarrow, i-1\rangle &\overset{\alpha_{v\downarrow\, lo\uparrow}}{\longmapsto} \langle lo\uparrow, 2i\rangle, i > 0 \\
\langle u\downarrow, i-1\rangle &\overset{\alpha_{u\downarrow\, lo\uparrow}}{\longmapsto} \langle lo\uparrow, 2i\rangle, i > 0 \\
\langle ri\downarrow, i-1\rangle &\overset{\alpha_{ri\downarrow\, lo\uparrow}}{\longmapsto} \langle lo\uparrow, 2i\rangle, i > 0
\end{aligned}
$$

$$
\begin{aligned}
\langle li\downarrow, 2i\rangle &\overset{\alpha_{li\downarrow\, v\uparrow}}{\longmapsto} \langle v\uparrow, i\rangle \\
\langle lo\uparrow, 2i\rangle &\overset{\alpha_{lo\uparrow\, v\uparrow}}{\longmapsto} \langle v\uparrow, i\rangle \\
\langle u\downarrow, i-1\rangle &\overset{\alpha_{u\downarrow\, v\uparrow}}{\longmapsto} \langle v\uparrow, i\rangle, i > 0
\end{aligned}
$$

$$
\begin{aligned}
\langle v\uparrow, i\rangle &\overset{\alpha_{v\uparrow\, lo\downarrow}}{\longmapsto} \langle lo\downarrow, 2i\rangle \\
\langle u\downarrow, i-1\rangle &\overset{\alpha_{u\downarrow\, lo\downarrow}}{\longmapsto} \langle lo\downarrow, 2i\rangle, i > 0
\end{aligned}
$$

$$
\begin{aligned}
\langle li\uparrow, 2i+1\rangle &\overset{\alpha_{li\uparrow\, u\uparrow}}{\longmapsto} \langle u\uparrow, i\rangle \\
\langle v\uparrow, i\rangle &\overset{\alpha_{v\uparrow\, u\uparrow}}{\longmapsto} \langle u\uparrow, i\rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle v\uparrow, i\rangle &\overset{\alpha_{v\uparrow\, lo\uparrow}}{\longmapsto} \langle lo\uparrow, 2i+1\rangle \\
\langle u\uparrow, i\rangle &\overset{\alpha_{u\uparrow\, lo\uparrow}}{\longmapsto} \langle lo\uparrow, 2i+1\rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle li\downarrow, 2i+1\rangle &\overset{\alpha_{li\downarrow\, v\downarrow}}{\longmapsto} \langle v\downarrow, i\rangle \\
\langle u\uparrow, i\rangle &\overset{\alpha_{u\uparrow\, v\downarrow}}{\longmapsto} \langle v\downarrow, i\rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle v\downarrow, i\rangle &\overset{\alpha_{v\downarrow\, lo\downarrow}}{\longmapsto} \langle lo\downarrow, i\rangle \\
\langle u\uparrow, i\rangle &\overset{\alpha_{u\uparrow\, lo\downarrow}}{\longmapsto} \langle lo\downarrow, i\rangle
\end{aligned}
$$

$$
\langle lo\downarrow, 2i+1\rangle \overset{\alpha_{lo\downarrow\, ro\uparrow}}{\longmapsto} \langle ro\uparrow, i\rangle
$$

$$\langle v\downarrow, i\rangle \overset{\alpha_{v\downarrow ro\uparrow}}{\longmapsto} \langle ro\uparrow, i\rangle$$

$$\langle u\uparrow, i\rangle \overset{\alpha_{u\uparrow ro\uparrow}}{\longmapsto} \langle ro\uparrow, i\rangle$$

$$\langle ri\uparrow, i\rangle \overset{\alpha_{ri\uparrow u\downarrow}}{\longmapsto} \langle u\downarrow, i\rangle$$

$$\langle u\downarrow, i\rangle \overset{\alpha_{u\downarrow ro\downarrow}}{\longmapsto} \langle ro\downarrow, i\rangle$$

4.4 Data-Dependent Computations

Many non-straight-line programs can be transformed into straight-line programs, given a limited amount of extra information. This extra information is a trace of the control flow through each process for a particular execution of the computation. It is used to determine which events of a disjunction occur first (or occur at all), and to decide which rules constraining these events should be added to the *ER* system.

Data-dependent computations expressed as handshaking expansions can be transformed into *ER* systems by unrolling the loops of the processes until the data dependence has been eliminated.

Example 4.13 Consider the asynchronous kill–propagate–generate (*KPG*) adder process:

$$\begin{aligned} adder \;=\; & *[[k \rightarrow d0\uparrow, [c0 \rightarrow s0\uparrow \,|c1 \rightarrow s1\uparrow] \\ & |g \rightarrow d1\uparrow, [c0 \rightarrow s0\uparrow \,|c1 \rightarrow s1\uparrow] \\ & |p \wedge c1 \rightarrow d1\uparrow, s0\uparrow \\ & |p \wedge c0 \rightarrow d0\uparrow, s1\uparrow \\ &]; ([\neg k \wedge \neg g \wedge \neg p]; d1\downarrow, d0\downarrow), ([\neg c0 \wedge \neg c1]; s0\downarrow, s1\downarrow)] \;. \end{aligned}$$

The inputs k, p and g are a triple-rail encoding of the kill, propagate and generate values. The inputs $c1$ and $c0$ are a dual-rail encoding of the carry-in value. The outputs $s1$ and $s0$, and $d1$ and $d0$ are dual-rail encodings of the sum and carry-out,

respectively. Assume this adder process is connected to a trivial environment:

$$
\begin{aligned}
\mathit{env} \equiv\ & *[[\mathbf{true} \rightarrow k\uparrow \,|\mathbf{true} \rightarrow g\uparrow \,|\mathbf{true} \rightarrow p\uparrow],\\
& [\mathbf{true} \rightarrow c0\uparrow \,|\mathbf{true} \rightarrow c1\uparrow];\\
& [(s0 \lor s1) \land (d0 \lor d1)];\\
& k\downarrow, p\downarrow, g\downarrow, c0\downarrow, c1\downarrow;\\
& [\neg s0 \land \neg s1 \land \neg d0 \land \neg d1]]
\end{aligned}
$$

and assume that the environment repeatedly chooses the guarded commands $k\uparrow, c0\uparrow$ followed by the guarded commands $p\uparrow, c1\uparrow$. Then by unrolling both processes we get the straight-line handshaking expansions:

$$
\begin{aligned}
\mathit{env} \equiv\ & *[k\uparrow, c0\uparrow; [s0 \land d0]; k\downarrow, c0\downarrow; [\neg s0 \land \neg d0];\\
& p\uparrow, c1\uparrow; [s0 \land d1]; p\downarrow, c1\downarrow; [\neg s0 \land \neg d1]]\\
\mathit{adder} \equiv\ & *[[k]; (d0\uparrow, ([c0]; s0\uparrow)); (([\neg k]; d0\downarrow), ([\neg c0]; s0\downarrow));\\
& [p \land c1]; s0\uparrow, d1\uparrow; (([\neg p]; d1\downarrow), ([\neg c1]; s0\downarrow))]
\end{aligned}
$$

After removing the repeated assignments, we can transform this pair of expansions into a repetitive *ER* system upon which we can apply performance analysis. □

Given which guarded commands are selected during each trip through a process, we can generate the *ER* system. It is the timing of this particular execution of the program that is being determined. No attempt is made to determine the timing of an arbitrary execution or worst-case execution of the program. This is because the timing of different executions can vary tremendously, and a loose upper bound on the execution time is not an interesting metric for the performance of the computation. Tight timing bounds are necessary to compare competing designs.

4.5 Inherently Disjunctive Computations

Up to this point, the translations from program to *ER* system have been independent of the actual values of the delays. Some computations cannot be translated into *ER* systems unless the actual values of the delays are known.

Example 4.14 Consider the process P connected to a standard environment.

$$P \;\equiv\; *[[ai \vee bi]; co\uparrow; [ci]; [ai \wedge bi]; ao\uparrow, bo\uparrow; [\neg ai \wedge \neg bi]; co\downarrow; [\neg ci]; ao\downarrow, bo\downarrow]$$

The rule constraining the firing of $\langle co\uparrow, i\rangle$ is either:

$$\langle ai\uparrow, i\rangle \;\overset{\alpha_{co\uparrow}}{\mapsto}\; \langle co\uparrow, i\rangle\,, \qquad (4.1)$$

or

$$\langle bi\uparrow, i\rangle \;\overset{\alpha_{co\uparrow}}{\mapsto}\; \langle co\uparrow, i\rangle \qquad (4.2)$$

depending on whether

$$\alpha_{ao\downarrow} + \alpha_{ai\uparrow} \;>\; \alpha_{bo\downarrow} + \alpha_{bi\uparrow}\,.$$

□

Inherently disjunctive processes have not been encountered in any of the circuits fabricated by Martin's group at Caltech, including the Caltech Asynchronous Microprocessor [23]. If they are needed, we suggest two techniques for modeling such processes.

First, in the previous example, we can simply choose one of the two rules (4.1) and (4.2) as the constraining rule for $co\uparrow$. Now we have an upper bound on the timing

performance of the actual system. However, tight bounds are needed to compare competing designs, and, thus, this technique is not attractive.

Second, we could redefine *ER* systems to allow disjunctive rules. Thus, for each target event f, there is a collection of sets of events,

$$C_f = \{S_0, S_1, \ldots, S_{q-1}\}$$

such that event f is constrained by the events in one of the sets S_i.

Example 4.15 Consider the process from Example 4.14. The collections of source event sets are:

$$\begin{aligned}
C_{\langle co\uparrow,i\rangle} &= \{\{\langle ai\uparrow,i\rangle\},\{\langle bi\uparrow,i\rangle\}\} \\
C_{\langle ao\uparrow,i\rangle} = C_{\langle bo\uparrow,i\rangle} &= \{\{\langle ci\uparrow,i\rangle,\langle ai\uparrow,i\rangle,\langle bi\uparrow,i\rangle\}\} \\
C_{\langle co\downarrow,i\rangle} &= \{\{\langle ai\downarrow,i\rangle,\langle bi\downarrow,i\rangle\}\} \\
C_{\langle ao\downarrow,i\rangle} = C_{\langle bo\downarrow,i\rangle} &= \{\{\langle ci\downarrow,i\rangle\}\}
\end{aligned}$$

□

In a disjunctive model, a timing function must satisfy:

$$\begin{aligned}
t(f) &\geq 0 \\
t(f) &\geq t(e) + \alpha_{ef} \text{ for some } S \in C_f \text{ and for all } e \in S\ .
\end{aligned}$$

A timing simulation is the smallest such timing function, and, if the dependencies are acyclic, satisfies:

$$\hat{t}(f) = 0 \text{ if } \emptyset \in C_f \tag{4.3}$$

$$\hat{t}(f) = \min\{\max\{\hat{t}(e) + \alpha_{ef} \mid e \in S\} \mid S \in C_f\} \tag{4.4}$$

Example 4.16 Continuing Example 4.14 and using the target-name convention for naming delays, we get:

$$
\begin{aligned}
\hat{t}(ai\uparrow,0) &= 0\\
\hat{t}(bi\uparrow,0) &= 0\\
\hat{t}(co\uparrow,i) &= \min\{\hat{t}(ai\uparrow,i)+\alpha_{co\uparrow},\hat{t}(bi\uparrow,i)+\alpha_{co\uparrow}\}\\
\hat{t}(ci\uparrow,i) &= \hat{t}(co\uparrow,i)+\alpha_{ci\uparrow}\\
\hat{t}(ao\uparrow,i) &= \max\{\hat{t}(ci\uparrow,i)+\alpha_{ao\uparrow},\hat{t}(ai\uparrow,i)+\alpha_{ao\uparrow},\hat{t}(bi\uparrow,i)+\alpha_{ao\uparrow}\}\\
\hat{t}(bo\uparrow,i) &= \max\{\hat{t}(ci\uparrow,i)+\alpha_{bo\uparrow},\hat{t}(ai\uparrow,i)+\alpha_{bo\uparrow},\hat{t}(bi\uparrow,i)+\alpha_{bo\uparrow}\}\\
\hat{t}(ai\downarrow,i) &= \hat{t}(ao\uparrow,i)+\alpha_{ai\downarrow}\\
\hat{t}(bi\downarrow,i) &= \hat{t}(bo\uparrow,i)+\alpha_{bi\downarrow}\\
\hat{t}(co\downarrow,i) &= \max\{\hat{t}(ai\downarrow,i)+\alpha_{co\downarrow},\hat{t}(bi\downarrow,i)+\alpha_{co\downarrow}\}\\
\hat{t}(ci\downarrow,i) &= \hat{t}(co\downarrow,i)+\alpha_{ci\downarrow}\\
\hat{t}(ao\downarrow,i) &= \hat{t}(ci\downarrow,i)+\alpha_{ao\downarrow}\\
\hat{t}(bo\downarrow,i) &= \hat{t}(ci\downarrow,i)+\alpha_{bo\downarrow}\\
\hat{t}(ai\uparrow,i) &= \hat{t}(ao\downarrow,i-1)+\alpha_{ai\uparrow},\, i>0\\
\hat{t}(bi\uparrow,i) &= \hat{t}(bo\downarrow,i-1)+\alpha_{bi\uparrow},\, i>0
\end{aligned}
$$

□

However, linear programming cannot be used effectively to find the minimum-period linear timing function in the disjunctive model. The constraint of any timing function contributed by the set

$$
\begin{aligned}
C_f \equiv\ & \{\{e_{00},e_{01},\ldots,e_{0,n_0-1}\},\{e_{10},e_{11},\ldots,e_{1,n_1-1}\},\ldots,\\
& \{e_{m-1,0},e_{m-1,1},\ldots,e_{m-1,n_{m-1}-1}\}\}
\end{aligned}
$$

is

$$\bigvee_{i=0}^{m-1} \bigwedge_{j=0}^{n_i-1} t(f) \ \geq \ t(e_{ij}) + \alpha_{e_{ij}f} \, .$$

The conjunction of these constraints for each event f is the complete constraint on t. If the system is repetitive, we can, using a linear formulation of the timing function, reduce the number of constraints from an infinite number to the number of transitions in the system. But the resulting constraint is in *AND-OR-AND* form, not the *AND* form used in linear programming. *OR-AND* constraints can be solved by finding the minimum of the optimal solutions to the linear programs expressed by each *AND* term individually. The *AND-OR-AND* form can be transformed into *OR-AND* form but the transformation is, in general, intractable since the resulting *OR-AND* form can be exponential in the number of transitions in the system.

Because of these computational difficulties, it is best not to use a linear programming formulation to analyze the performance of an arbitrary disjunctive *ER* system. Instead, direct construction of the timing simulation using (4.3) and (4.4) will more efficiently produce performance information. However, simulation must proceed until the circuit has reached a steady-state in order to determine a performance metric such as the cycle period.

Chapter 5

Linear Arrays of Processes

The power of the theory introduced in Chapter 2 emerges when we extend the techniques to include regular arrays of processes. Concise, accurate statements can be made about the performance of large systems that contain regular structures, by analyzing a structure whose size is comparable to a single process. Here, we extend the notion of a linear timing function to include a term that relates the time at which a particular occurrence of a transition in one process of the array occurs with the time at which the same occurrence of the same transition occurs in the adjacent process of the array. We call this time the *latency* of the system. In this chapter we show how to determine simultaneously the latency and cycle-period of a system.

5.1 Regular Systems

To begin, we consider an array of n identical processes. The array may be of arbitrary size. Special processes are connected to the two boundaries of the array. For now, we ignore the boundary processes. We treat them separately in Section 5.3. Such a

regular system is modeled as the pair

$$\langle E'', R'' \rangle$$

where the elements of E'' represent the transitions in the process form that is replicated, and where the elements of R'' are five-tuples

$$\langle g, h, \varepsilon, \rho, \alpha \rangle \in E'' \times E'' \times \mathbb{Z} \times \mathbb{Z} \times [0, +\infty) , \tag{5.1}$$

usually written,

$$\langle g, i - \varepsilon, j - \rho \rangle \overset{\alpha}{\mapsto} \langle h, i, j \rangle$$

that constrain the timing behavior of the system. As before, the dummy parameter i represents the occurrence index. The dummy parameter j represents the process index, denoting the individual process in the array to which this event corresponds. The ε, ρ and α above denote the occurrence-index offset, the process-index offset and the delay associated with that rule. For a given n, this regular system generates the repetitive system $\langle E', R' \rangle$ where

$$\begin{aligned} E' &= \{\langle g, j \rangle \mid g \in E'' \wedge 0 \le j < n\} \\ R' &= \Big\{ \langle \langle g, j - \rho \rangle, \langle h, j \rangle, \varepsilon, \alpha \rangle \,\Big|\, \langle g, h, \varepsilon, \rho, \alpha \rangle \in R'' \wedge \\ &\qquad\qquad \max(0, \rho) \le j < \min(n, n + \rho) \Big\} \end{aligned}$$

Example 5.1 The linear array of the lazy-active/passive buffers described in Figure 5.1 and by the handshaking expansions

$$\begin{aligned} lap &= *[lo\uparrow; [li]; lo\downarrow; [ri]; ro\uparrow; [\neg ri]; ro\downarrow; [\neg li]] \\ lenv &= *[[xi]; xo\uparrow; [\neg xi]; xo\downarrow] \end{aligned}$$

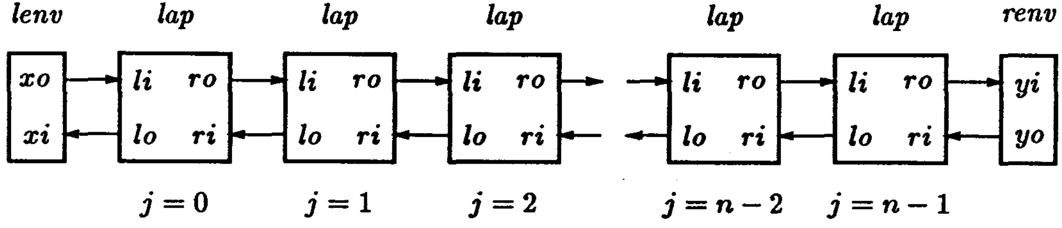


Figure 5.1: Regular linear array of n *lap* processes including a left boundary process *lenv* and a right boundary process *renv*.

$$renv \;=\; *[yo\!\uparrow;[yi];yo\!\downarrow;[\neg yi]]$$

has the regular system $\langle E'', R''\rangle$:

$$\begin{aligned}
E'' &= \{lo\!\uparrow, lo\!\downarrow, ro\!\uparrow, ro\!\downarrow, li\!\uparrow, li\!\downarrow, ri\!\uparrow, ri\!\downarrow\} \\
R'' &= \{\ \langle ro\!\downarrow, i-1, j\rangle \overset{\alpha_{lo\uparrow}}{\mapsto} \langle lo\!\uparrow, i, j\rangle, \\
&\quad \langle li\!\downarrow, i-1, j\rangle \overset{\alpha_{lo\uparrow}}{\mapsto} \langle lo\!\uparrow, i, j\rangle, \\
&\quad \langle li\!\uparrow, i, j\rangle \overset{\alpha_{lo\downarrow}}{\mapsto} \langle lo\!\downarrow, i, j\rangle, \\
&\quad \langle lo\!\downarrow, i, j\rangle \overset{\alpha_{ro\uparrow}}{\mapsto} \langle ro\!\uparrow, i, j\rangle, \\
&\quad \langle ri\!\uparrow, i, j\rangle \overset{\alpha_{ro\uparrow}}{\mapsto} \langle ro\!\uparrow, i, j\rangle, \\
&\quad \langle ri\!\downarrow, i, j\rangle \overset{\alpha_{ro\downarrow}}{\mapsto} \langle ro\!\downarrow, i, j\rangle, \\
&\quad \langle lo\!\uparrow, i, j+1\rangle \overset{\alpha_{ri\uparrow}}{\mapsto} \langle ri\!\uparrow, i, j\rangle, \\
&\quad \langle lo\!\downarrow, i, j+1\rangle \overset{\alpha_{ri\downarrow}}{\mapsto} \langle ri\!\downarrow, i, j\rangle, \\
&\quad \langle ro\!\uparrow, i, j-1\rangle \overset{\alpha_{li\uparrow}}{\mapsto} \langle li\!\uparrow, i, j\rangle, \\
&\quad \langle ro\!\downarrow, i, j-1\rangle \overset{\alpha_{li\downarrow}}{\mapsto} \langle li\!\downarrow, i, j\rangle\}
\end{aligned}$$

The first six rules come from the internal constraints within the process *lap*. The next four rules come from the interconnections between the processes. (The additional rules corresponding to the boundary processes are described in Example 5.5.) □

5.1.1 Linear Timing Function

In the context of regular systems, a ***linear timing function*** takes on the following form:

$$\begin{aligned} \bar{t}(\langle g,i,j\rangle) &= \\ \bar{t}(g,i,j) &= x_g + pi + \ell j \qquad \text{for } g \in E'', i \in \mathbb{N} \text{ and } 0 \leq j < n, \end{aligned}$$

where x_g is an offset independent of i and j; p is the cycle period independent of g, i and j; and ℓ is the latency independent of g, i and j. The existence of such a function is not ensured by the feasibility of the generated *ER* system as was the case for standard repetitive systems. Instead, a linear timing function exists for a regular system if the generated repetitive systems with an arbitrary number of processes have a property known as *constant response time* [18, 34]. A regular system has constant response time (CRT) if after sufficiently many occurrences of a transition, the time between consecutive occurrences is bounded by a constant regardless of the number of processes in the array.

5.1.2 Linear Program

The timing constraints (2.1) for the rules generated from R'' become

$$\begin{aligned} x_h + pi + \ell j &\geq x_g + p(i-\varepsilon) + \ell(j-\rho) + \alpha \\ x_h &\geq x_g - \varepsilon p - \rho\ell + \alpha \qquad (5.2) \end{aligned}$$

for each rule $\langle g, i-\varepsilon, j-\rho\rangle \overset{\alpha}{\mapsto} \langle h,i,j\rangle \in R''$.

The collapsed constraint graph for the regular system $\langle E'', R''\rangle$ is, again, the labeled directed graph with nodes from E'' and arcs from R''. To avoid confusion with the collapsed constraint graph of a repetitive system, we frequently refer to such

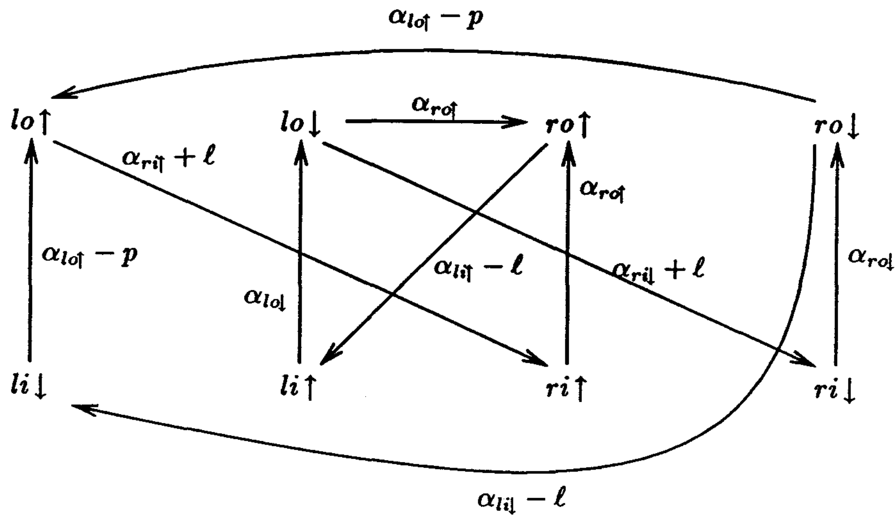


Figure 5.2: Latency/period constraint graph of a regular-array system for the lazy-active/passive buffer of Example 5.1.

graphs as *latency/period graphs.* The labels on the arcs correspond to the collapsed linear timing function constraints. The arc of the rule corresponding to (5.2) is labeled with $\alpha - \varepsilon p - \rho\ell$.

Example 5.2 The latency/period graph of the regular system of Example 5.1 is shown in Figure 5.2. □

The constraints of the linear timing function (5.2) can be rewritten in standard linear programming form:

$$\left.\begin{array}{rcl} \min c_0 p + c_1 \ell & = & z \\ A'x + \varepsilon p + \rho\ell & \geq & \alpha \\ x, p, \ell & \geq & 0 \end{array}\right\} \qquad (5.3)$$

In some cases, the process numbering may have to be reversed in order for $\ell \geq 0$.

The dual program to (5.3) is

$$\left.\begin{array}{rcl} \max y^T\alpha & = & w \\ y^T A' & \leq & 0 \\ y^T\varepsilon & \leq & c_0 \\ y^T\rho & \leq & c_1 \\ y & \geq & 0 \end{array}\right\} \qquad (5.4)$$

As in Section 2.4.1, $y^T A' = 0$ because A' is an incidence matrix of a directed graph. As a consequence, x can be unrestricted in the primal equation (5.3).

5.2 Minimum-Period/Minimum-Latency Linear Timing Functions

Our goal now is to find simultaneous optimal solutions for the objective functions $\min p$ and $\min \ell$. Optimization problems of this type are called multi-objective linear programming problems [13]. A simultaneous optimal solution exists if there exists a feasible x, p, ℓ that is optimal for every objective function $c_0 p + c_1 \ell$ with non-negative c_0 and c_1. We now construct an algorithm that produces this simultaneous optimal solution, or decides that no such solution exists.

The dual linear program (5.4) can be rewritten as

$$\left.\begin{array}{rcl} \max \Theta^T(U^T\alpha) & = & w \\ \Theta^T(U^T\varepsilon) & \leq & c_0 \\ \Theta^T(U^T\rho) & \leq & c_1 \\ \Theta & \geq & 0 \end{array}\right\} \qquad (5.5)$$

by introducing the cycle vector matrix U and applying Lemma 2.6. Converting back

to primal space, we get:

$$\left.\begin{aligned} \min c_0 p + c_1 \ell &= z \\ (U^T\varepsilon)p + (U^T\rho)\ell &\geq (U^T\alpha) \\ p, \ell &\geq 0 \end{aligned}\right\} \tag{5.6}$$

This linear program has two variables (p and ℓ) and as many constraints as there are cycles in the graph corresponding to A'. The constraints corresponding to cycle c (row i of U^T) can be classified into four categories depending on the values of $\varepsilon(c) = (U^T\varepsilon)_i$ and $\rho(c) = (U^T\rho)_i$:

Case 1. $\rho(c) > 0, \varepsilon(c) \leq 0$ Case 2. $\rho(c) \leq 0, \varepsilon(c) > 0$

Case 3. $\rho(c) > 0, \varepsilon(c) > 0$ Case 4. $\rho(c) \leq 0, \varepsilon(c) \leq 0$

Case 1 constraints are ℓ-determining and Case 2 constraints are p-determining. A Case 4 constraint cannot be satisfied unless $\alpha(c) = 0$. All non-degenerate optimal solutions to (5.6) must occur at the intersection of a pair of constraints. All other optimal solutions (degenerate) occur along a line connecting a pair of these intersections. However, since we are looking for solutions which minimize both p and ℓ, we need only consider the intersections of Case 1 with Case 2. (See Figure 5.4.) Furthermore, the slope of the line bounding the Case 1 constraint must be larger than the slope of the line bounding the Case 2 constraint. If this is not the case, the intersection is in the negative p and negative ℓ quadrant, and the linear program is infeasible.

If one of these intersection points is at least as large, in both p and ℓ, as all other intersection points, then we choose this point as the candidate optimal solution. If the candidate optimal solution is also a feasible solution, then we have found the optimal solution to the linear program. This can be done simply by checking that each constraint is satisfied at the candidate optimal solution. If this solution is not feasible, then no optimal solution minimizes both p and ℓ. This solution can be found

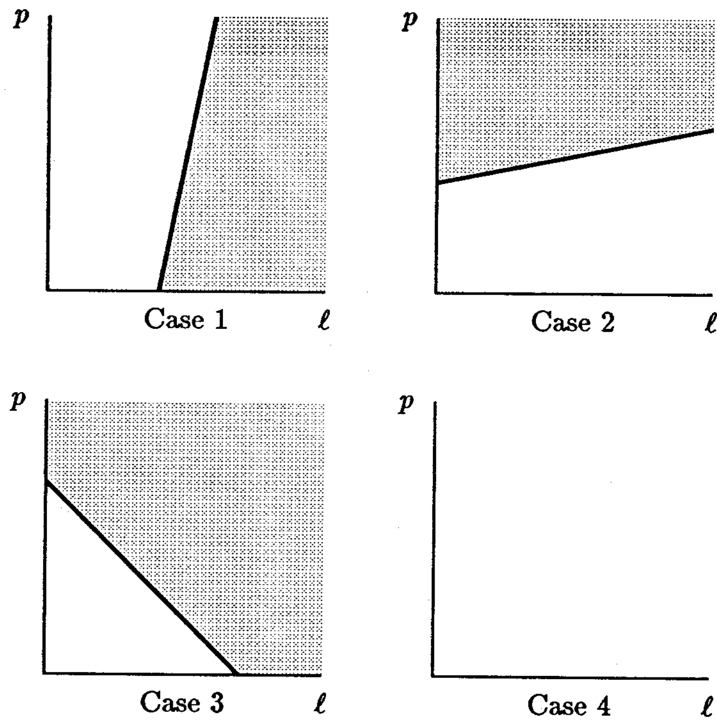


Figure 5.3: The four types of constraints imposed by the reduced primal linear program (5.6).

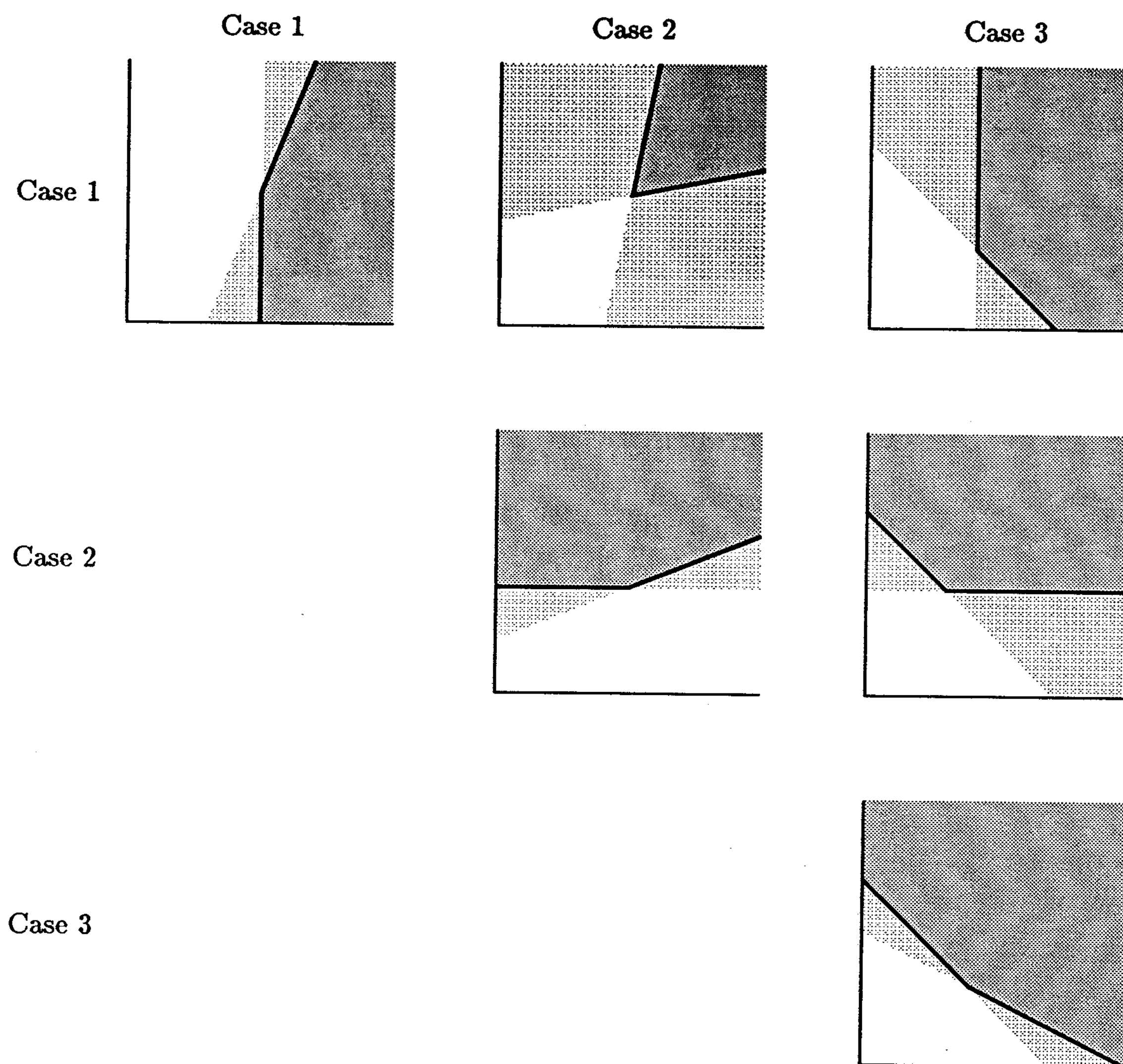


Figure 5.4: The six types of intersections of constraints imposed by the reduced primal linear program (5.6). Only the intersections of Case 1 with Case 2 can represent an optimal solution to both min p and min ℓ.

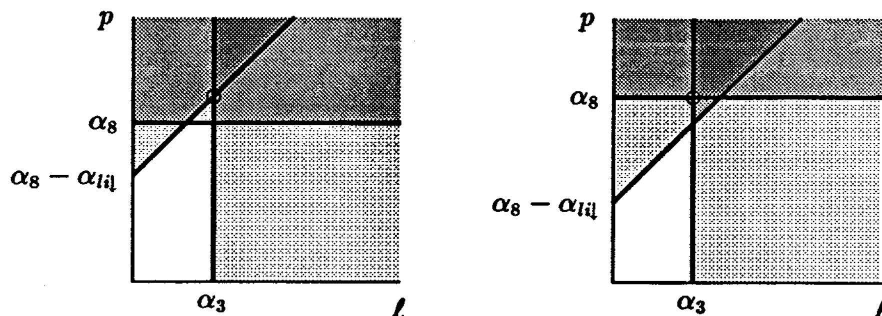


Figure 5.5: Graphical depiction of constraints imposed in Example 5.3. There are two possible optimal points depending on the relative values of α_3 and $\alpha_{li\downarrow}$.

by using Algorithm 5.1.

Example 5.3 We apply Algorithm 5.1 to the regular system of Example 5.1. There are three cycles through the corresponding latency/period graph (Figure 5.2). The Case 1 cycles are:

$$\ell \;\geq\; \alpha_{li\uparrow} + \alpha_{lo\downarrow} + \alpha_{ro\uparrow} \;=\; \alpha_3$$

The Case 2 cycles are:

$$\begin{aligned} p \;&\geq\; \alpha_{lo\uparrow} + \alpha_{ri\uparrow} + \alpha_{ro\uparrow} + \alpha_{li\uparrow} + \alpha_{lo\downarrow} + \alpha_{ri\downarrow} + \alpha_{ro\downarrow} + \alpha_{li\downarrow} \;=\; \alpha_8 \\ p - \ell \;&\geq\; \alpha_{lo\uparrow} + \alpha_{ri\uparrow} + \alpha_{ro\uparrow} + \alpha_{li\uparrow} + \alpha_{lo\downarrow} + \alpha_{ri\downarrow} + \alpha_{ro\downarrow} \;=\; \alpha_8 - \alpha_{li\downarrow} \end{aligned}$$

These constraints are shown graphically in Figure 5.5. The optimal solution occurs at either:

$$\langle \alpha_3, \alpha_8 \rangle \qquad \text{or} \qquad \langle \alpha_3, \alpha_8 + \alpha_3 - \alpha_{li\downarrow} \rangle$$

1. Construct the set C of cycles through the graph corresponding to A'. Compute for each cycle $c \in C$

$$\varepsilon(c) = \sum_{r \in c} \varepsilon_r, \qquad \rho(c) = \sum_{r \in c} \rho_r, \qquad \text{and } \alpha(c) = \sum_{r \in c} \alpha_r.$$

2. If $\rho(c)$ is negative for all cycles c with $\varepsilon(c) \leq 0$, then reverse the numbering of the processes and start over again. If some are positive and some are negative, no feasible solution with positive ℓ exists.

3. Compute the intersection points

$$I = \{\langle \ell(c_1,c_2), p(c_1,c_2)\rangle \mid c_1 \in C_1 \wedge c_2 \in C_2 \wedge d(c_1,c_2) > 0\}$$

   where

$$\begin{aligned}
\ell(c_1,c_2) &= \frac{\alpha(c_1)\varepsilon(c_2) - \varepsilon(c_1)\alpha(c_2)}{d(c_1,c_2)} \\
p(c_1,c_2) &= \frac{\alpha(c_2)\rho(c_1) - \rho(c_2)\alpha(c_1)}{d(c_1,c_2)} \\
d(c_1,c_2) &= \rho(c_1)\varepsilon(c_2) - \varepsilon(c_1)\rho(c_2) \\
C_1 &= \{c \mid c \in C \wedge \varepsilon(c) \leq 0 \wedge \rho(c) > 0\} \\
C_2 &= \{c \mid c \in C \wedge \varepsilon(c) > 0 \wedge \rho(c) \leq 0\}
\end{aligned}$$

   If $d(c_1,c_2) \leq 0$ for some pair c_1 and c_2, then no feasible solution with positive p (or ℓ) exists.

4. Find a point $\langle \ell, p\rangle \in I$ such that for all $\langle \ell', p'\rangle \in I$, $p \geq p'$ and $\ell \geq \ell'$. This point may not exist, in which case there is no optimal solution.

5. Check that

$$\varepsilon(c)p + \rho(c)\ell \geq \alpha(c)$$

   for each cycle c. If this is the case, then p and ℓ represent the optimal solution to the multiobjective program. If not, there exists no optimal solution to the multiobjective program.

Algorithm 5.1: Algorithm to find, if it exists, the optimal solution of the multiobjective linear programming problem (5.3).

Thus, the optimal values are

$$\ell = \alpha_{li\uparrow} + \alpha_{lo\downarrow} + \alpha_{ro\uparrow} \tag{5.7}$$

$$p = \alpha_{lo\uparrow} + \alpha_{ri\uparrow} + \alpha_{ro\uparrow} + \alpha_{li\uparrow} + \alpha_{lo\downarrow} + \alpha_{ri\downarrow} + \alpha_{ro\downarrow} + (\alpha_{li\downarrow} \max \alpha_{li\uparrow} + \alpha_{lo\downarrow} + \alpha_{ro\uparrow}) \tag{5.8}$$

□

The following lemma shows that the cycle period determined by Algorithm 5.1 provides an accurate approximation to the cycle period of the generated repetitive system.

Lemma 5.1 Let p be the minimum period and ℓ be the minimum latency corresponding to the optimal solution of the multiobjective linear program (5.3). Let c_1 be the critical Case 1 cycle generated by Algorithm 5.1. Let c_2 be the critical Case 2 cycle. Let S be the generated repetitive system with n processes. If n is large enough, and if either

1. $\rho(c_2) = 0$, or
2. cycles c_1 and c_2 have a node in common,

then the cycle period of S is equal to p.

Proof: We first argue that p can be no smaller than the cycle period of S. The timing function from which p is defined is more restrictive than that defining the cycle period of S. The offsets x_g must be related for all j though the latency. Thus the linear program has more constraints, and p is no smaller than the cycle period of S.

Now, we argue the reverse. We use the cycles of rules that constrain p and ℓ with equality, and from these cycles, we generate a cycle of the same period in S.

If $\rho(c_2) = 0$, then c_2 immediately produces a cycle of period p in S. Otherwise, $\rho(c_2) < 0$; and if cycles c_1 and c_2 have a node in common, then we can build up the necessary cycle. This is done by walking cycles c_1 and c_2

$$m_1 = \frac{\mathrm{lcm}(-\rho(c_2), \rho(c_1))}{-\rho(c_2)} \text{ and } m_2 = \frac{\mathrm{lcm}(-\rho(c_2), \rho(c_1))}{\rho(c_1)}$$

times, respectively. ∎

Example 5.4 Figures 5.6 and 5.7 show how to instantiate the critical cycles produced during the analysis of the regular array system in a corresponding repetitive system. Nodes $ro\uparrow$ and $ro\downarrow$ are common to both critical cycles, so by Lemma 5.1, (5.8) represents the actual cycle period of a large enough array. Arrays with at least three elements will have this cycle period regardless of the α values. If $\alpha_{li\downarrow} \geq \alpha_{li\uparrow} + \alpha_{lo\downarrow} + \alpha_{ro\uparrow}$, then arrays with at least two elements will also exhibit the cycle period of (5.8). □

The hypotheses of Lemma 5.1 are typically satisfied. In the case where there is no common node between the critical cycles, we must use a different argument. Since it occurs infrequently, we do not discuss it here. The restriction that n be large enough is needed so that when the critical cycles are replicated, there are enough processes in S so that the composite cycle can be constructed. The restriction that the optimal solution must minimize both p and ℓ is not necessary for the determination of the cycle period. This theory could be developed with the minimization of p only. A minimum latency is required if the linear timing function is to be used as an approximation of the timing simulation of the generated general *ER* system. The following lemma is a (slightly weaker) analog of Theorem 2.10, the approximation theorem for repetitive systems.

Lemma 5.2 Let $\bar{t}$ be a minimum-period/minimum-latency linear timing function of the regular-array repetitive *ER* system $\langle E'', R'' \rangle$. Let G'' be the strongly-connected

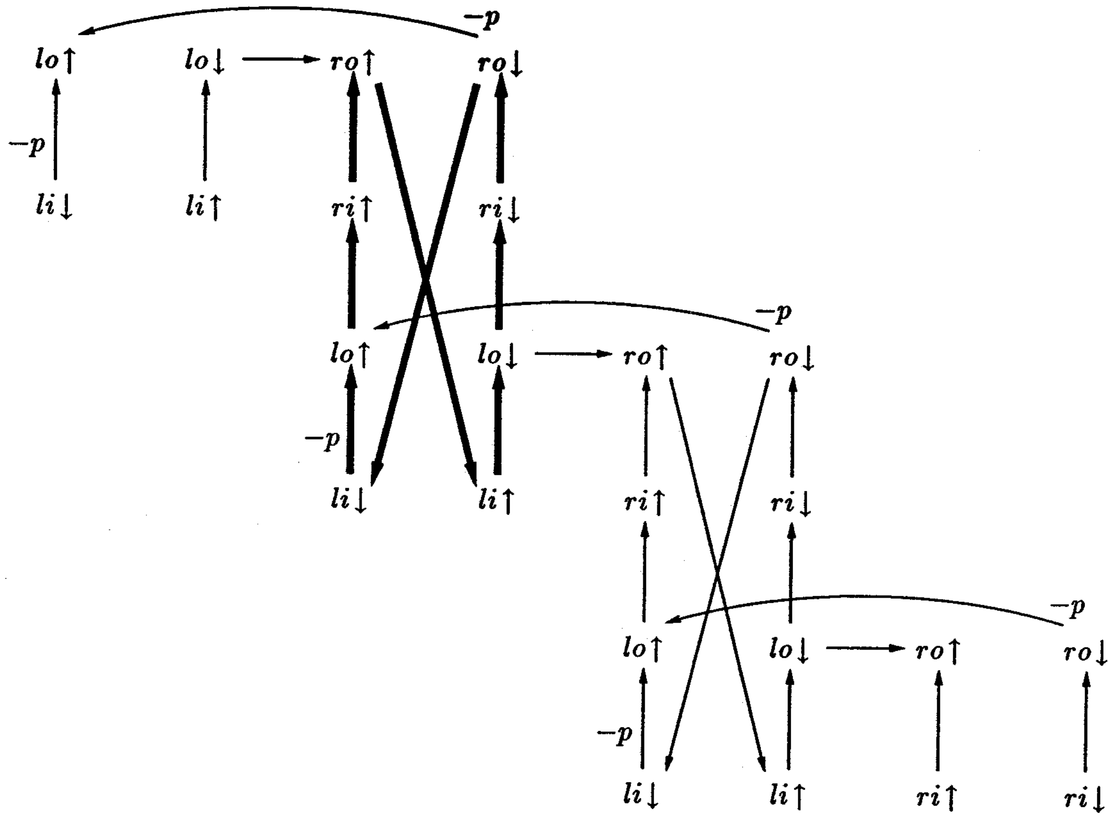


Figure 5.6: Three instances of processes for the lazy-active/passive buffer. The bold line represents the critical cycle determined by the Case 2 constraint alone. The cycle visits nodes in two processes of the array.

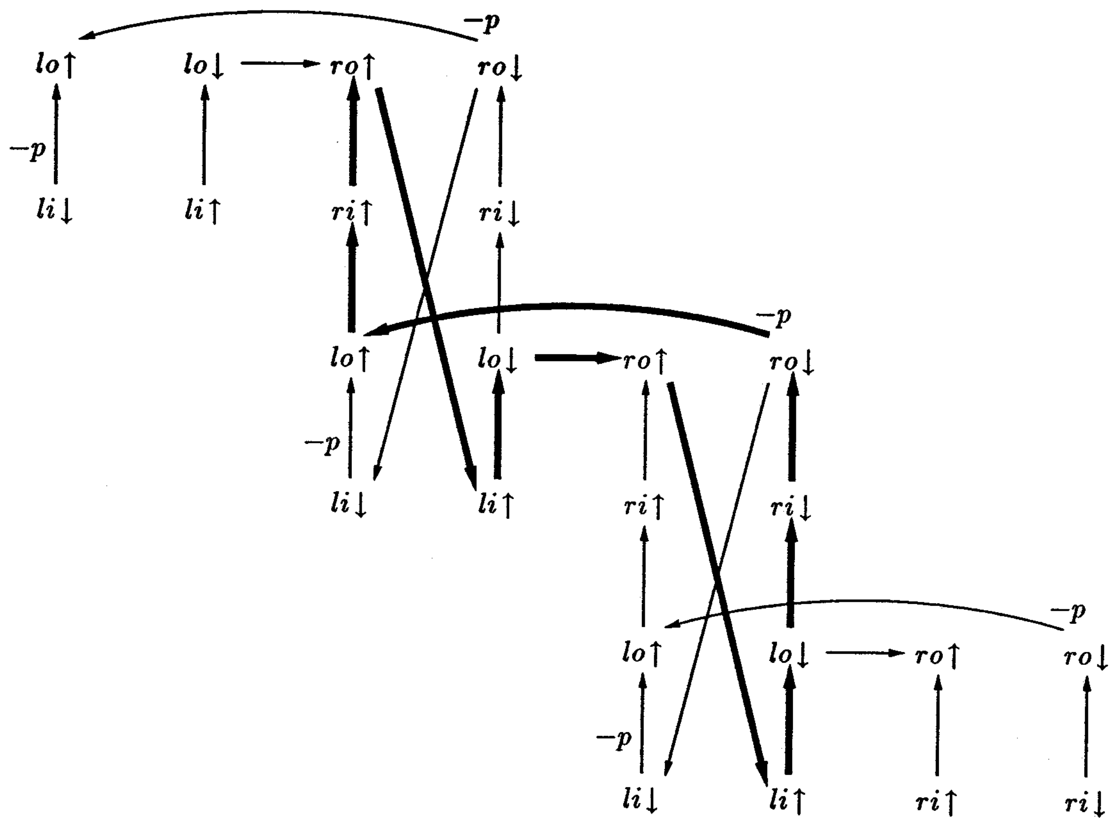


Figure 5.7: Three instances of processes for the lazy-active/passive buffer. The bold line represents the critical cycle determined by the Case 1 and the Case 2 constraints together. The cycle visits nodes in all three processes of the array.

latency/period graph of $\langle E'', R'' \rangle$. Let $\hat{t}$ be the timing simulation of the general *ER* system with n processes generated from $\langle E'', R'' \rangle$. If n is large enough, and Case 1 and Case 2 critical cycles exist (c_1 and c_2, respectively) and have a node g_0 in common, then the difference between $\bar{t}$ and $\hat{t}$ for all events generated from g_0 is bounded by a constant:

$$s_{g_0,i,j} = \bar{t}(g_0, i, j) - \hat{t}(g_0, i, j) \;\leq\; B$$

Proof: By definition for each $g \in E''$, and each i and j, we have

$$\begin{aligned} \bar{t}(g,i,j) &= x_g + pi + \ell j \\ \hat{t}(g,i,j) &= x_g + pi + \ell j - s_{g,i,j} \end{aligned}$$

where $s_{g,i,j}$ is non-negative. For the constraints generated from the rule

$$r = \langle g, i-\varepsilon, j-\rho \rangle \overset{\alpha}{\mapsto} \langle h, i, j \rangle \in R'',$$

we, as in Theorem 2.10, define the non-negative slack variables $\hat{z}_{r,i,j}$ and $\bar{z}_r$ and produce the constraint equalities

$$x_g - p\varepsilon - \ell\rho + \alpha + \bar{z}_r \;=\; x_h \tag{5.9}$$

$$x_g - p\varepsilon - \ell\rho - s_{g,i-\varepsilon,j-\rho} + \alpha + \hat{z}_{r,i,j} \;=\; x_h - s_{h,i,j}\,. \tag{5.10}$$

Subtracting equation (5.10) from (5.9) yields

$$\bar{z}_r - \hat{z}_{r,i,j} \;=\; s_{h,i,j} - s_{g,i-\varepsilon,j-\rho}\,. \tag{5.11}$$

For any critical cycle c, we know that

$$\varepsilon(c)p + \rho(c)\ell \;=\; \alpha(c)\,,$$

and thus, that (5.9) reduces to

$$\bar{z}_r \;=\; 0\,,$$

for all the rules r in c. For each rule r in a critical cycle, (5.11) reduces to

$$s_{g,i-\varepsilon,j-\rho} \;\geq\; s_{h,i,j}\,, \tag{5.12}$$

for all i and j such that $i \geq \max(0, \varepsilon)$ and $\min(n, n-\rho) > j \geq \max(0, \rho)$.

Concentrating on the node g_0 that is in common between the two critical cycles, we see that we can construct a no-slack path ($z_r = 0$ for each rule) through the generated (general) *ER* system by instantiating k_1 copies of c_1 and k_2 copies of c_2. Multiple copies of c_1 increase the j coordinate and possibly decrease the i coordinate. Multiple copies of c_2 increase the i coordinate and possibly decrease the j coordinate. Thus for each non-negative k_1 and k_2 we have

$$s_{g_0,i',j'} \;\geq\; s_{g_0,i,j}\,,$$

where

$$\begin{aligned} i &= i' + k_1\varepsilon(c_1) + k_2\varepsilon(c_2), \text{ and} \\ j &= j' + k_1\rho(c_1) + k_2\rho(c_2)\,. \end{aligned}$$

Only a finite number of the $s_{g_0,i,j}$ are unconstrained, and thus B—equal to the max-

imum unconstrained difference—is finite. ∎

We have just shown that there is at least one transition in each instantiated process for which the difference between the best linear timing function and the timing simulation is bounded. We would like to relate this result to *all* transitions.

Conjecture 5.3 Given the hypotheses of Lemma 5.2, we can find a bound B' independent of n such that for every g and j there exists an i' large enough so that $s_{g,i,j} \leq B'$ for all $i \geq i'$.

5.3 Boundary Processes

The regular array of n identical processes is terminated with two, presumably different, special processes. By ignoring these processes, we have constructed lower bounds on the cycle period and latency of the system since the additional rules contributed by boundary processes would only further constrain the generated repetitive system.

We now define the structure of these boundary processes, and the additions to the system $\langle E', R' \rangle$. The transitions in the two boundary processes should have distinct names with each other, and with the transitions in E''. Let E'_0 and E'_1 be transitions in the left and right boundary processes, respectively. Rules between the transitions in E'_0, or between a transition in E'_0 and a transition in a particular process in the array, are specified in the repeated rule set R'_0. And similarly for the rules in R'_1. There are no rules between E'_0 and E'_1 in either rule set, nor are there rules between two transitions in E''.

Example 5.5 Continuing Example 5.1, the boundary processes contribute the following additional transitions and rules.

$$E'_0 = \left\{ xo\uparrow, xo\downarrow, xi\uparrow, xi\downarrow \right\}$$

$$
\begin{aligned}
E'_1 &= \{yo\uparrow, yo\downarrow, yi\uparrow, yi\downarrow\} \\
R'_0 &= \{ \langle xi\downarrow, i-1\rangle \overset{\alpha_{xo\uparrow}}{\mapsto} \langle xo\uparrow, i\rangle, \\
&\qquad \langle xi\uparrow, i\rangle \overset{\alpha_{xo\downarrow}}{\mapsto} \langle xo\downarrow, i\rangle, \\
&\qquad \langle xo\uparrow, i\rangle \overset{\alpha_{li\uparrow}}{\mapsto} \langle li\uparrow, i, 0\rangle, \\
&\qquad \langle xo\downarrow, i\rangle \overset{\alpha_{li\downarrow}}{\mapsto} \langle li\downarrow, i, 0\rangle, \\
&\qquad \langle lo\uparrow, i, 0\rangle \overset{\alpha_{xi\uparrow}}{\mapsto} \langle xi\uparrow, i\rangle, \\
&\qquad \langle lo\downarrow, i, 0\rangle \overset{\alpha_{xi\downarrow}}{\mapsto} \langle xi\downarrow, i\rangle\} \\
R'_1 &= \{ \langle yi\uparrow, i\rangle \overset{\alpha_{yo\uparrow}}{\mapsto} \langle yo\uparrow, i\rangle, \\
&\qquad \langle yi\downarrow, i\rangle \overset{\alpha_{yo\downarrow}}{\mapsto} \langle yo\downarrow, i\rangle, \\
&\qquad \langle yo\uparrow, i\rangle \overset{\alpha_{ri\uparrow}}{\mapsto} \langle ri\uparrow, i, n-1\rangle, \\
&\qquad \langle yo\downarrow, i\rangle \overset{\alpha_{ri\downarrow}}{\mapsto} \langle ri\downarrow, i, n-1\rangle, \\
&\qquad \langle ro\uparrow, i, n-1\rangle \overset{\alpha_{yi\uparrow}}{\mapsto} \langle yi\uparrow, i\rangle, \\
&\qquad \langle ro\downarrow, i, n-1\rangle \overset{\alpha_{yi\downarrow}}{\mapsto} \langle yi\downarrow, i\rangle\}
\end{aligned}
$$

□

The timing constraints for the rules generated from R'_0 and R'_1 are just

$$
\begin{aligned}
x_v + pi &\geq x_u + p(i - \varepsilon) + \alpha \\
x_v &\geq x_u - \varepsilon p + \alpha
\end{aligned}
\tag{5.13}
$$

for each rule $\langle u, i-\varepsilon\rangle \overset{\alpha}{\mapsto} \langle v, i\rangle \in R'_{\text{bound}}$,

$$
\begin{aligned}
x_v + pi &\geq x_g + p(i - \varepsilon) + \ell j_0 + \alpha \\
x_v &\geq x_g - \varepsilon p + \ell j_0 + \alpha
\end{aligned}
\tag{5.14}
$$

for each rule $\langle g, i-\varepsilon, j_0\rangle \overset{\alpha}{\mapsto} \langle v, i\rangle \in R'_{\text{bound}}$, and

$$\begin{aligned} x_h + pi + \ell j_0 &\geq x_u + p(i-\varepsilon) + \alpha \\ x_h &\geq x_u - \varepsilon p - \ell j_0 + \alpha \end{aligned} \tag{5.15}$$

for each rule $\langle u, i-\varepsilon\rangle \overset{\alpha}{\mapsto} \langle h, i, j_0\rangle \in R'_{\text{bound}}$. All these inequalities are linear constraints and can be included in the linear program. Latency/period analysis can be used to determine optimal solutions to p and ℓ.

However, not every cycle in this system corresponds to a cycle in the generated repetitive system S. For example, a cycle cannot contain nodes in both boundary processes if $\rho = 0$ for every arc. In order to accurately determine the cycle period of S, we must not use such cycles when determining the optimal values of p and ℓ.

Example 5.6 Of the five cycles created by the addition of E'_0, E'_1, R'_0 and R'_1 to the system of Example 5.1, only the four cycles that produce the constraints

$$\begin{aligned} p &\geq \alpha_{xi\uparrow} + \alpha_{xo\uparrow} + \alpha_{li\uparrow} + \alpha_{lo\downarrow} + \alpha_{xi\downarrow} + \alpha_{xo\downarrow} + \alpha_{li\downarrow} + \alpha_{lo\uparrow} \\ p &\geq \alpha_{ri\uparrow} + \alpha_{ro\uparrow} + \alpha_{yi\uparrow} + \alpha_{yo\downarrow} + \alpha_{ri\downarrow} + \alpha_{ro\downarrow} + \alpha_{yi\downarrow} + \alpha_{yo\uparrow} \\ p &\geq \alpha_{xi\uparrow} + \alpha_{xo\uparrow} + \alpha_{li\uparrow} + \alpha_{lo\downarrow} + \alpha_{ri\downarrow} + \ell + \alpha_{ro\downarrow} + \alpha_{lo\uparrow} \\ p &\geq \alpha_{ri\uparrow} + \ell + \alpha_{ro\uparrow} + \alpha_{yi\uparrow} + \alpha_{yo\downarrow} + \alpha_{ri\downarrow} + \alpha_{ro\downarrow} + \alpha_{lo\uparrow} \end{aligned}$$

can be instantiated in S. The cycle (denote by a list of nodes):

$$(lo\uparrow, xi\uparrow, xo\uparrow, li\uparrow, lo\downarrow, ro\uparrow, yi\uparrow, yo\downarrow, ri\downarrow, ro\downarrow)$$

cannot exist in S. It includes nodes from both boundary processes, and all arcs have zero ρ. □

Chapter 6

Case Studies

6.1 FIFO Queues

The techniques for determining the performance of linear arrays of processes can be used to compare the cycle period and latency of various implementations of *FIFO* queues. Here, a *FIFO* queue is a linear array of identical processes each of the form:

$$\mathit{fifo} \;\equiv\; *[L?x; R!x]$$

We use the control and datapath decomposition methods from Chapter 3 to construct each *fifo* process. The control part is just the process $*[L; R]$, where L and R are now dataless synchronizations. The data part is inserted between the control parts of adjacent processes. To improve the standard implementation, we consider interleaving the waits and assignments implementing the handshaking protocols of L and R in the control process. However, such interleavings must retain the constant response time (CRT) property of the array of processes, and must also not violate any of the assumptions—about when data variables change—that were made during synthesis of the data part.

6.1.1 CRT Constraint

From the work of Lee [18], a simple necessary condition for such an interleaving to be *CRT* is

$$(ro\uparrow \prec [li]) \wedge (ro\downarrow \prec [\neg li]) \;\; \vee \;\; (lo\uparrow \prec [ri]) \wedge (lo\downarrow \prec [\neg ri])\,. \tag{6.1}$$

A statement $x \prec y$ should be read as "x precedes y in the handshaking expansion." This condition is not sufficient. In general, a complete cycle analysis is required to determine whether an interleaving is *CRT*. The necessary condition is used here only to prune the search space.

6.1.2 Passive/Active Data Constraints

Three possible implementations of the data part in the case where L is passive and R is active are shown in Figure 6.1. We now provide constraints on the indexed occurrences of the transitions in the two handshaking sequences

$$[li]; lo\uparrow; [\neg li]; lo\downarrow \qquad\qquad ro\uparrow; [ri]; ro\downarrow; [\neg ri]$$

that must be satisfied to maintain the correct flow of data.

To ensure that a datum has been received before it is sent, we must have:

$$[\neg li] \;\; \prec \;\; ro\uparrow \tag{6.2}$$

One of the following three constraints is needed to ensure that the current datum has been successfully received by the next stage before the next datum is received:

$$[\neg ri^{(-1)}] \;\; \prec \;\; lo\uparrow \qquad \textit{pa-no-iso} \tag{6.3}$$

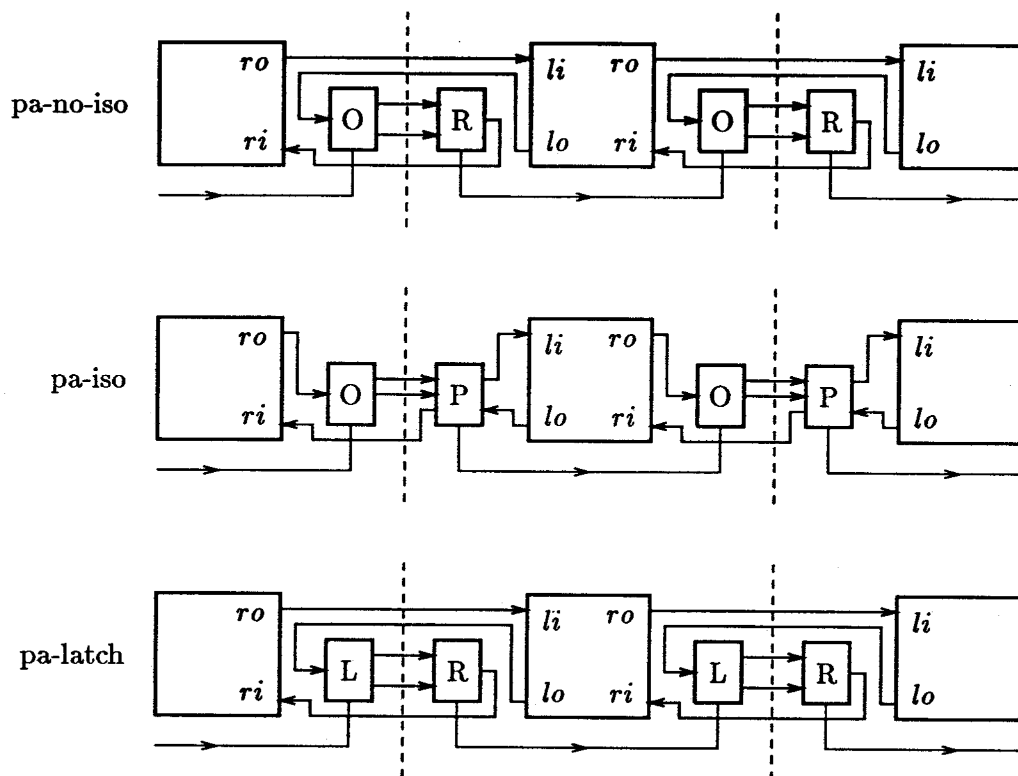


Figure 6.1: Possible datapath implementations for a passive/active *FIFO*. The blocks labeled O, L, P, and R denote the output and input stages described in Figure 3.4, Figure 3.5, Figure 3.7, and Figure 3.6, respectively.

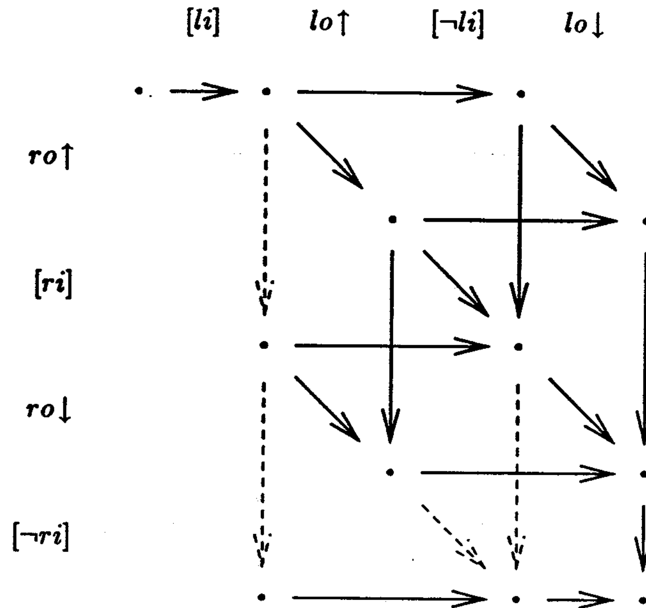


Figure 6.2: Diagram to enumerate the passive/active interleavings. Crossing a dashed arc violates the second term of (6.1).

$$ro^{(-1)}\downarrow \prec lo\uparrow \qquad \textit{pa-iso} \tag{6.4}$$

$$[ri^{(-1)}] \prec lo\uparrow \qquad \textit{pa-latch} \tag{6.5}$$

Note that (6.5) $\Rightarrow$ (6.4) $\Rightarrow$ (6.3). The three constraints correspond to different assumptions and different implementations of the output communication. In the first case, *pa-no-iso*, no isochronic fork is allowed between the control and datapath and an unlatched output implementation is used. In the second case, *pa-iso*, an isochronic fork is allowed between the control and datapath and an unlatched output implementation is used. In the third case, *pa-latch*, a latched output implementation is used.

6.1.3 Interleavings of Passive/Active Protocols

The lattice-path diagrams (Figures 6.2 through 6.4) provide a convenient means of organizing the various interleavings of the handshaking expansions for L and R. Each

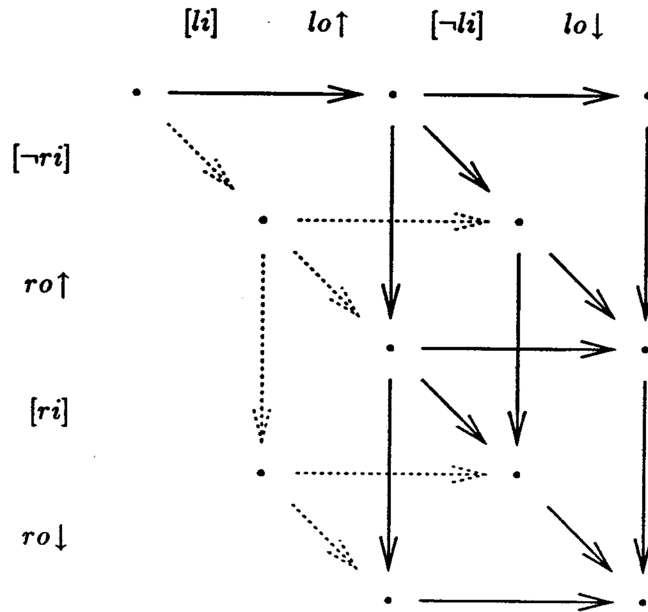


Figure 6.3: Diagram to enumerate the passive/active interleavings that contain the vacuous wait [¬*ri*]. Paths with dotted arcs appear in Figure 6.2.

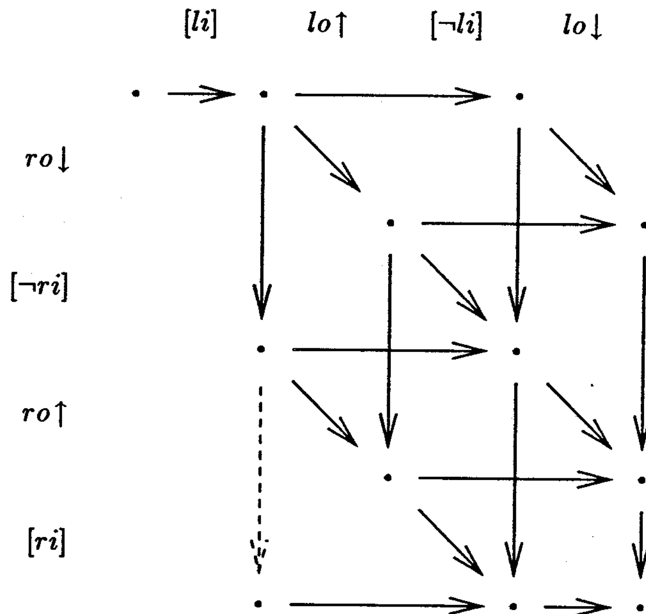


Figure 6.4: Diagram to enumerate the passive/active interleavings that contain the vacuous firing *ro*↓ and the vacuous wait [¬*ri*].

path through one of these diagrams corresponds to an interleaving of these two sequences. To avoid duplication, we always start the sequence with the wait $[li]$. Traveling along a right-pointing arc adds the next statement of the L handshake to the current sequence. Traveling along a down-pointing arc adds the next statement of the R handshake. Traveling along a diagonal arc adds both the next L statement and the next R statement, simultaneously.

We are interested in enumerating those constant response time interleavings of these two sequences that also satisfy the data constraints. Data constraint (6.2) contradicts the first term of the *CRT* constraint, so we now enumerate all the interleavings that satisfy the second term of (6.1). Interleavings *pa* through *a4* represent all the possible interleavings of a passive and active sequence (Figure 6.2):

$$
\begin{aligned}
pa &\equiv *[[li]; lo\uparrow; [\neg li]; lo\downarrow, ro\uparrow; [ri]; ro\downarrow; [\neg ri]] \\
a1 &\equiv *[[li]; lo\uparrow; [\neg li]; ro\uparrow; [ri]; lo\downarrow, ro\downarrow; [\neg ri]] \\
a2 &\equiv *[[li]; lo\uparrow, ro\uparrow; [\neg li]; lo\downarrow; [ri]; ro\downarrow; [\neg ri]] \\
a3 &\equiv *[[li]; lo\uparrow, ro\uparrow; [\neg li \wedge ri]; lo\downarrow, ro\downarrow; [\neg ri]] \\
a4 &\equiv *[[li]; lo\uparrow, ro\uparrow; [ri]; ro\downarrow; [\neg li]; lo\downarrow; [\neg ri]]
\end{aligned}
$$

Interleavings *pla* through *b5* represent all the possible interleavings of a passive and lazy-active sequence (Figure 6.3):

$$
\begin{aligned}
pla &\equiv *[[li]; lo\uparrow; [\neg li]; lo\downarrow; [\neg ri]; ro\uparrow; [ri]; ro\downarrow] \\
b1 &\equiv *[[li]; lo\uparrow; [\neg li \wedge \neg ri]; lo\downarrow, ro\uparrow; [ri]; ro\downarrow] \\
b2 &\equiv *[[li]; lo\uparrow; [\neg li \wedge \neg ri]; ro\uparrow; [ri]; lo\downarrow, ro\downarrow] \\
b3 &\equiv *[[li]; lo\uparrow; [\neg ri]; ro\uparrow; [\neg li]; lo\downarrow; [ri]; ro\downarrow] \\
b4 &\equiv *[[li]; lo\uparrow; [\neg ri]; ro\uparrow; [\neg li \wedge ri]; lo\downarrow, ro\downarrow]
\end{aligned}
$$

$$b5 \equiv *[[li]; lo\uparrow; [\neg ri]; ro\uparrow; [ri]; ro\downarrow; [\neg li]; lo\downarrow]$$

Interleavings *c0* through *c12* represent all the possible interleavings of the passive handshaking expansion for L and the handshaking sequence

$$ro^{(-1)}\downarrow; [\neg ri^{(-1)}]; ro\uparrow; [ri]$$

for R:

$$\begin{aligned}
c0 &\equiv *[[li]; lo\uparrow; [\neg li]; lo\downarrow, ro\downarrow; [\neg ri]; ro\uparrow; [ri]] \\
c1 &\equiv *[[li]; lo\uparrow; [\neg li]; ro\downarrow; [\neg ri]; lo\downarrow, ro\uparrow; [ri]] \\
c2 &\equiv *[[li]; lo\uparrow; [\neg li]; ro\downarrow; [\neg ri]; ro\uparrow; [ri]; lo\downarrow] \\
c3 &\equiv *[[li]; lo\uparrow, ro\downarrow; [\neg li]; lo\downarrow; [\neg ri]; ro\uparrow; [ri]] \\
c4 &\equiv *[[li]; lo\uparrow, ro\downarrow; [\neg li \wedge \neg ri]; lo\downarrow, ro\uparrow; [ri]] \\
c5 &\equiv *[[li]; lo\uparrow, ro\downarrow; [\neg li \wedge \neg ri]; ro\uparrow; [ri]; lo\downarrow] \\
c6 &\equiv *[[li]; lo\uparrow, ro\downarrow; [\neg ri]; ro\uparrow; [\neg li]; lo\downarrow; [ri]] \\
c7 &\equiv *[[li]; lo\uparrow, ro\downarrow; [\neg ri]; ro\uparrow; [\neg li \wedge ri]; lo\downarrow] \\
c8 &\equiv *[[li]; ro\downarrow; [\neg ri]; lo\uparrow; [\neg li]; lo\downarrow, ro\uparrow; [ri]] \\
c9 &\equiv *[[li]; ro\downarrow; [\neg ri]; lo\uparrow; [\neg li]; ro\uparrow; [ri]; lo\downarrow] \\
c10 &\equiv *[[li]; ro\downarrow; [\neg ri]; lo\uparrow, ro\uparrow; [\neg li]; lo\downarrow; [ri]] \\
c11 &\equiv *[[li]; ro\downarrow; [\neg ri]; lo\uparrow, ro\uparrow; [\neg li \wedge ri]; lo\downarrow]
\end{aligned}$$

Further unrollings of the R handshaking protocol imply that $lo\uparrow \prec ri^{(-1)}$ which violates the weakest data constraint (6.5).

We now eliminate interleavings that violate the data constraints. Assuming the

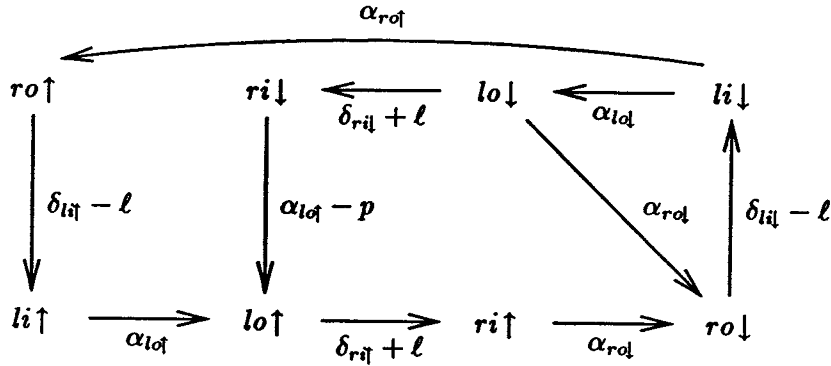


Figure 6.5: Latency/period constraint graph for buffer *pa*. The results of the analysis are $p = \delta_{ri\uparrow} + \delta_{li\downarrow} + \delta_{ri\downarrow} + \ell$ and $\ell = \delta_{li\uparrow} + \delta_{ri\uparrow} + \delta_{li\downarrow}$ if the δ delays are assumed to dominate the α delays. In the graphs that follow (Figures 6.6 through 6.17), we do not label arcs with their respective delays. Refer to this graph for the correct labels on the arcs between processes ($\rho \neq 0$). The labels on the other arcs are not important except for their ϵ values, which we denote by a single tick mark if $\epsilon = 1$, by a pair of tick marks if $\epsilon = -1$ and by no tick mark if $\epsilon = 0$.

most strict case *pa-no-iso*, the following interleavings satisfy the constraints:

pa, *a1*, *c8*, *c9*

Interleaving *c9* is not *CRT*. In the second case *pa-iso*, the following additional interleavings satisfy the constraints:

pla, *b1*, *b2*, *c3*, *c4*, *c5*

In the final case, *pa-latch*, the following additional interleavings satisfy the constraints:

c0, *c1*, *c2*

However, interleavings *c0*, *c1*, and *c2* are not *CRT*.

Figures 6.5 through 6.17 show the constraint graphs and corresponding cycle-

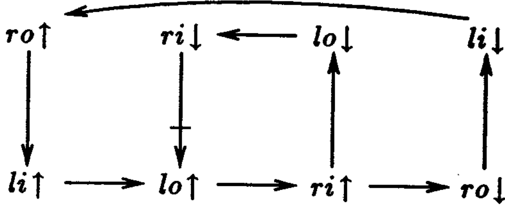


Figure 6.6: **Buffer** *a1*, $p = \delta_{ri\Uparrow} + \delta_{ri\Downarrow} + 2\ell$ and $\ell = \delta_{ri\Uparrow} + \delta_{li\Uparrow} + \delta_{li\Downarrow}$.

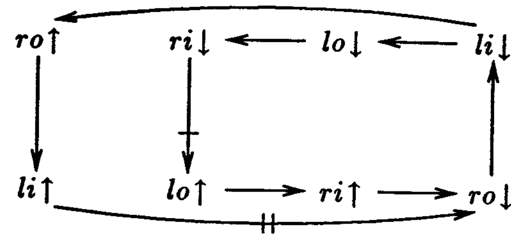


Figure 6.7: **Buffer** *c8*, $p = \delta_{ri\Uparrow} + \delta_{ri\Downarrow} + \delta_{li\Downarrow} + \ell$ and $\ell = \delta_{ri\Uparrow} + \delta_{ri\Downarrow} + \delta_{li\Uparrow} + 2\delta_{li\Downarrow}$.

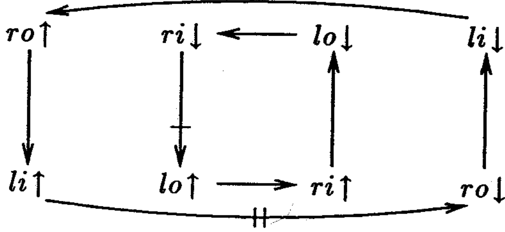


Figure 6.8: **Buffer** *c9*, **not** *CRT*.

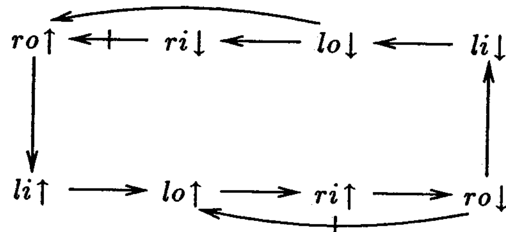


Figure 6.9: **Buffer** *pla*, $p = \delta_{ri\Uparrow} + (\ell \max \delta_{ri\Downarrow} + \delta_{li\Uparrow} + \delta_{li\Downarrow})$ and $\ell = \delta_{ri\Uparrow} + \delta_{li\Uparrow} + \delta_{li\Downarrow}$.

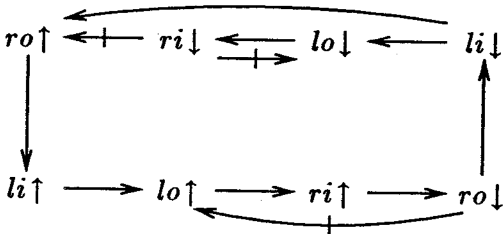


Figure 6.10: **Buffer** *b1*, $p = \ell + (\delta_{ri\Uparrow} \max \delta_{ri\Downarrow})$ and $\ell = \delta_{ri\Uparrow} + \delta_{li\Uparrow} + \delta_{li\Downarrow}$.

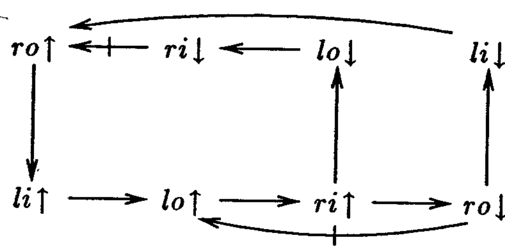


Figure 6.11: **Buffer** *b2*, $p = \ell + \delta_{ri\Uparrow} + \delta_{ri\Downarrow} + \delta_{li\Uparrow}$ and $\ell = \delta_{ri\Uparrow} + \delta_{li\Uparrow} + \delta_{li\Downarrow}$.

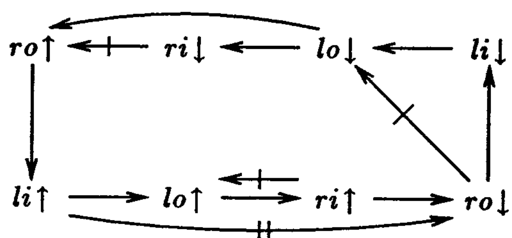


Figure 6.12: Buffer *c3*, $p = (\delta_{ri\Uparrow} \max \delta_{ri\Downarrow}) + \ell$ and $\ell = (\delta_{ri\Uparrow} \max \delta_{ri\Downarrow}) + \delta_{li\Uparrow} + \delta_{li\Downarrow}$.

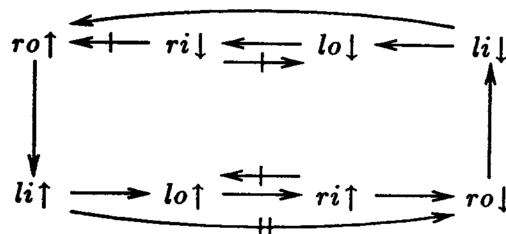


Figure 6.13: Buffer *c4*, $p = (\delta_{ri\Uparrow} \max \delta_{ri\Downarrow}) + \ell$ and $\ell = (\delta_{ri\Uparrow} \max \delta_{ri\Downarrow}) + \delta_{li\Uparrow} + \delta_{li\Downarrow}$.

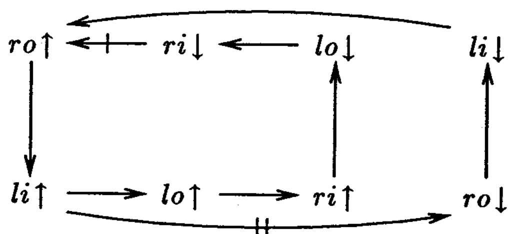


Figure 6.14: Buffer *c5*, $p = \ell + \delta_{ri\Uparrow} + \delta_{ri\Downarrow} + \delta_{li\Uparrow}$ and $\ell = \delta_{ri\Uparrow} + \delta_{ri\Downarrow} + 2\delta_{li\Uparrow} + \delta_{li\Downarrow}$.

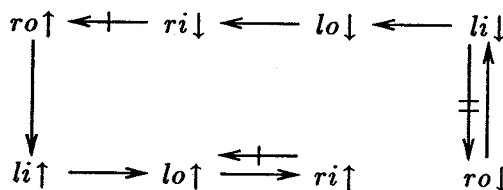


Figure 6.15: Buffer *c0*, not *CRT*.

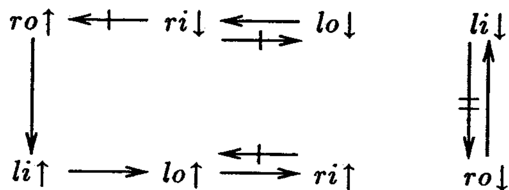


Figure 6.16: Buffer *c1*, not *CRT*.

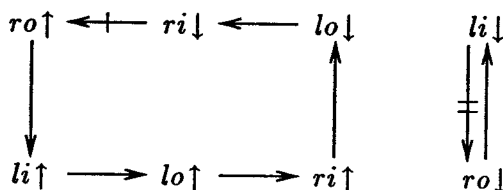


Figure 6.17: Buffer *c2*, not *CRT*.

	Period					Latency				
	$\delta_{ri\uparrow}$	$\delta_{ri\downarrow}$	$\delta_{li\uparrow}$	$\delta_{li\downarrow}$	α	$\delta_{ri\uparrow}$	$\delta_{ri\downarrow}$	$\delta_{li\uparrow}$	$\delta_{li\downarrow}$	α
pa	2	1	1	2	6	1	0	1	1	3
a1	3	1	2	2	8	1	0	1	1	3
c8	2	2	1	3	8	1	1	1	2	5
pla	2	0	1	1	6	1	0	1	1	4
	1	1	1	1	4	1	0	1	1	4
b1	2	0	1	1	5	1	0	1	1	3
	1	1	1	1	4	1	0	1	1	3
b2	2	1	1	2	6	1	0	1	1	3
c3	2	0	1	1	5	1	0	1	1	4
	1	1	1	1	4	0	1	1	1	3
c4	2	0	1	1	4	1	0	1	1	3
	0	2	1	1	4	0	1	1	1	3
c5	3	2	2	1	8	1	1	2	1	5

Table 6.1: Number of each external delay in the cycle period and latency for the *CRT* interleavings that can be used as a *FIFO* stage. If more than one coefficient vector is given, then the p or ℓ is the maximum of the set. The interleavings *pla*, *b1*, *c3*, and *c4* each have four external delays in series.

period analyses for each of the *CRT* interleavings that satisfy the data constraints. In these graphs, the arcs into nodes $li\uparrow$, $li\downarrow$, $ri\uparrow$ and $ri\downarrow$ span processes and thus have a non-zero process-number offset (ρ) value. These arcs represent the delays of the datapaths that intercept the handshake signals interconnecting the processes. For these analyses, we assume that the datapath delays, e.g., $\delta_{li\uparrow}$, are dominant and thus we set all other delays in the circuit, e.g., $\alpha_{lo\uparrow}$, to zero.

We conclude from these analyses that four interleavings, *pla*, *b1*, *c3* and *c4*, have at most four external delays in series. Furthermore, all the other solutions have a longer cycle period for all possible values of the external delays.

In Table 6.1, we tabulate the δ coefficient vector for the cycle-period and latency of

each legal interleaving. An extra column is included that specifies the number of non-datapath delays that occur in each cycle-period and latency expression. Assuming that $\delta_{li} = \delta_{li\Uparrow} = \delta_{li\Downarrow}$, $\delta_{ri} = \delta_{ri\Uparrow} = \delta_{ri\Downarrow}$ and $1 = \alpha_{lo\Uparrow} = \alpha_{lo\Downarrow} = \alpha_{ro\Uparrow} = \alpha_{ro\Downarrow}$, we have that the cycle period of *pla* is $2\delta_{li}+2\delta_{ri}+6$, of *b1* is $2\delta_{li}+2\delta_{ri}+5$, of *c3* is $2\delta_{li}+2\delta_{ri}+5$, and of *c4* is $2\delta_{li}+2\delta_{ri}+4$. Thus, *c4* represents the best design for a passive/active *FIFO* under this model. However, the initial vacuous wait for $[ri]$ makes this interleaving difficult to implement. A thorough analysis of implemented circuits, not handshaking expansions, is required to compare these designs. We do not perform this analysis here.

We can also use these results to compare the datapath implementations of Figure 6.1. None of the efficient interleavings satisfies the *pa-no-iso* constraints. In the *pa-iso* case, we can model the datapath delays more finely as

$$\begin{aligned} \delta_{li} &= \delta_o + \delta_c, \qquad \text{and} \\ \delta_{ri} &= \delta_r + \delta_c, \end{aligned}$$

where δ_o, δ_r and δ_c represent the propagation delays (up-going delay equals down-going delay) through an output unit, a register unit and a completion tree, respectively. In the *pa-latch* case, we have

$$\begin{aligned} \delta_{li} &= 0, \qquad \text{and} \\ \delta_{ri} &= \delta_o + \delta_r + \delta_c. \end{aligned}$$

Thus, we have:

$$\begin{aligned} p_{pa-iso} &= 2\delta_o + 2\delta_r + 4\delta_c, \qquad \text{and} \\ p_{pa-latch} &= 2\delta_o + 2\delta_r + 2\delta_c. \end{aligned}$$

The datapath configuration using *pa-latch* is faster than that using *pa-iso* by two completion-tree delays.

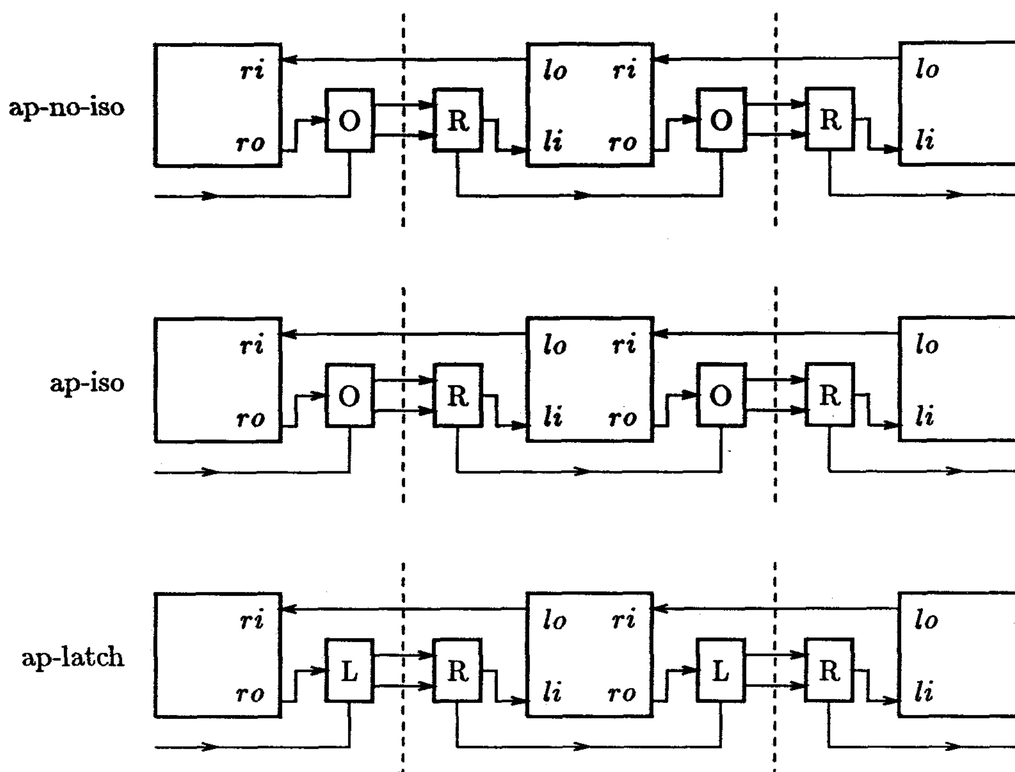


Figure 6.18: Possible datapath implementations for an active/passive *FIFO*. The blocks labeled O, L, and R denote the output and input stages described in Figure 3.4, Figure 3.5, and Figure 3.6, respectively.

6.1.4 Interleavings of Active/Passive Protocols

We now consider the *FIFO* queues where L is active and R is passive. A stage of the queue, complete with datapath, is shown in Figure 6.18, for three different implementations. The datapath constraints are

$$[li] \prec ro\uparrow \tag{6.6}$$

and one of

$$[ri^{(0)}] \prec lo\uparrow \qquad \textit{ap-no-iso} \tag{6.7}$$

$$ro^{(-1)}\downarrow \;\prec\; lo\uparrow \qquad \textit{ap-iso} \tag{6.8}$$

$$[\neg ri^{(-1)}] \;\prec\; lo\uparrow \qquad \textit{ap-latch} \tag{6.9}$$

depending of the implementation. Again, the first datapath-constraint contradicts the first term of the *CRT* disjunction (6.1). Constraints (6.7) and (6.1) contradict, so the first implementation, *ap-no-iso*, is not possible.

Interleavings *ap* through *d5* and *e0* through *e4* meet the necessary condition for constant-response-time and meet the weakest sequencing requirements of the above datapath implementations. (Each satisfies (6.1), (6.6) and (6.9).)

$$\begin{aligned}
ap &\equiv *[lo\uparrow; [li]; lo\downarrow; [\neg li \wedge ri]; ro\uparrow; [\neg ri]; ro\downarrow] \\
d1 &\equiv *[lo\uparrow; [li]; lo\downarrow; [ri]; ro\uparrow; [\neg li \wedge \neg ri]; ro\downarrow] \\
lap &\equiv *[lo\uparrow; [li]; lo\downarrow; [ri]; ro\uparrow; [\neg ri]; ro\downarrow; [\neg li]] \\
d3 &\equiv *[lo\uparrow; [li \wedge ri]; lo\downarrow; [\neg li]; ro\uparrow; [\neg ri]; ro\downarrow] \\
d4 &\equiv *[lo\uparrow; [li \wedge ri]; lo\downarrow, ro\uparrow; [\neg li \wedge \neg ri]; ro\downarrow] \\
d5 &\equiv *[lo\uparrow; [li \wedge ri]; lo\downarrow, ro\uparrow; [\neg ri]; ro\downarrow; [\neg li]] \\
e0 &\equiv *[lo\uparrow; [li]; lo\downarrow; [\neg li]; ro\downarrow; [ri]; ro\uparrow; [\neg ri]] \\
e1 &\equiv *[lo\uparrow; [li]; lo\downarrow, ro\downarrow; [\neg li \wedge ri]; ro\uparrow; [\neg ri]] \\
e2 &\equiv *[lo\uparrow; [li]; lo\downarrow, ro\downarrow; [ri]; ro\uparrow; [\neg li \wedge \neg ri]] \\
e3 &\equiv *[lo\uparrow; [li]; ro\downarrow; [ri]; lo\downarrow; [\neg li]; ro\uparrow; [\neg ri]] \\
e4 &\equiv *[lo\uparrow; [li]; ro\downarrow; [ri]; lo\downarrow, ro\uparrow; [\neg li \wedge \neg ri]]
\end{aligned}$$

Interleavings *ap* through *d5* use the standard passive expansion for R, and are enumerated using the lattice-path diagram shown in Figure 6.19. Interleavings *e0* through *e4* use a passive expansion for R with $ro\downarrow$ first and are enumerated using Figure 6.20.

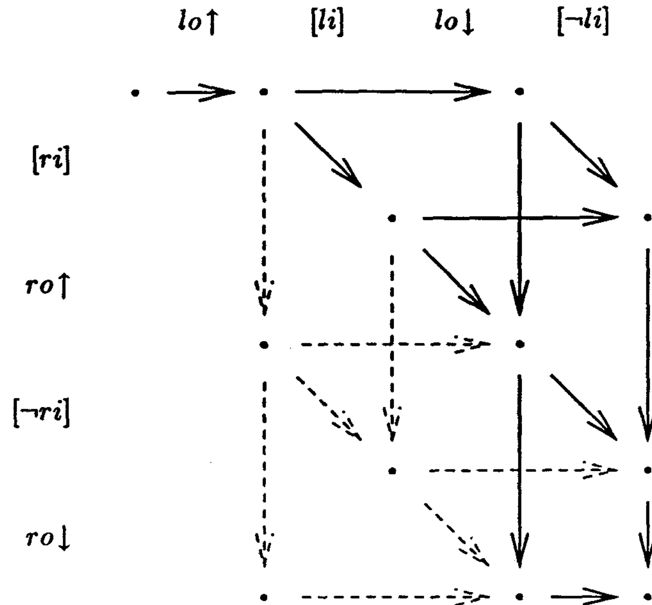


Figure 6.19: Diagram to enumerate the active/passive interleavings. Crossing a dashed arc violates either (6.1), (6.6) or (6.9).

No other interleavings (those with further unrolling of the handshaking protocol for R) meet the constraints.

The cycle period and latency of all these interleavings are derived in Figures 6.21 through 6.31 and summarized in Table 6.2.

Interleavings *d1*, *lap*, *d4* and *d5* each have at most two δ_{li} delays and two δ_{ri} delays in series. We can again split up these delays, but instead of comparing datapath implementations (there is only one type), we can compare these active/passive implementations to the passive/active in Section 6.1.3. Here,

$$\delta_{li} = \delta_o + \delta_r + \delta_c$$

$$\delta_{ri} = 0$$

$$p_{ap-iso} = p_{ap-latch} = 2\delta_o + 2\delta_r + 2\delta_c$$

The best active/passive implementations have, under this model, the same cycle

	Period					Latency				
	$\delta_{ri\Uparrow}$	$\delta_{ri\Downarrow}$	$\delta_{li\Uparrow}$	$\delta_{li\Downarrow}$	α	$\delta_{ri\Uparrow}$	$\delta_{ri\Downarrow}$	$\delta_{li\Uparrow}$	$\delta_{li\Downarrow}$	α
ap	1	2	2	1	6	0	1	1	1	3
d1	1	1	2	0	5	0	0	1	0	2
	1	1	1	1	4	0	0	0	1	1
lap	1	1	2	0	6	0	0	1	0	2
	1	1	1	1	4	0	0	1	0	2
d3	1	3	2	2	8	0	1	1	1	3
d4	1	1	2	0	4	0	0	1	0	1
	1	1	0	2	4	0	0	0	1	1
d5	1	1	2	0	5	0	0	1	0	1
	1	1	1	1	4	0	0	1	0	1
e1	2	2	3	1	8	1	1	2	1	5
e2	2	1	2	1	6	1	0	1	1	3
e4	3	1	2	2	10	1	0	1	1	4

Table 6.2: Number of each external delay in the cycle period and latency for the *CRT* interleavings that can be used as a *FIFO* stage. The interleavings *d1*, *lap*, *d4*, and *d5* each have four external delays in series.

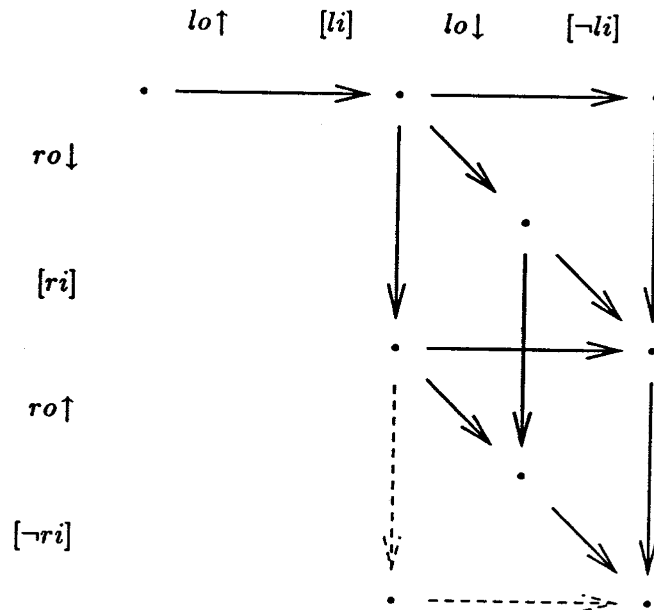


Figure 6.20: Diagram to enumerate the active/passive interleavings that contain the vacuous action $ro\downarrow$.

period as the best passive/active implementations. The active/passive scheme does have the advantage that it can be efficiently implemented without using a latched output-unit in the datapath.

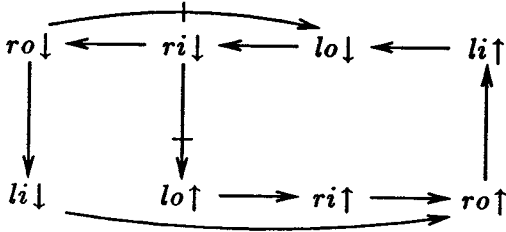


Figure 6.21: Buffer *ap*, $p = \ell + \delta_{ri\Uparrow} + \delta_{ri\Downarrow} + \delta_{li\Uparrow}$ and $\ell = \delta_{ri\Downarrow} + \delta_{li\Uparrow} + \delta_{li\Downarrow}$.

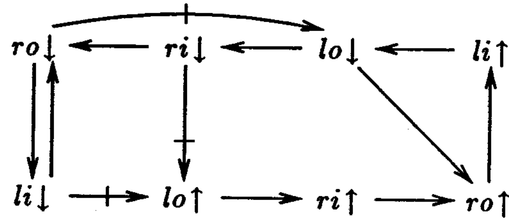


Figure 6.22: Buffer *d1*, $p = \ell + \delta_{ri\Uparrow} + \delta_{ri\Downarrow} + \delta_{li\Uparrow}$ and $\ell = \delta_{li\Uparrow}$ max $\delta_{li\Downarrow}$.

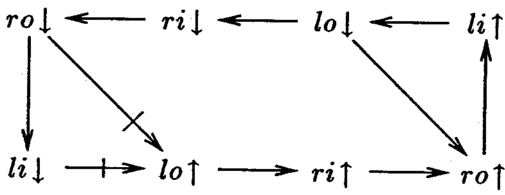


Figure 6.23: Buffer *lap*, $p = (\ell \max \delta_{li\Downarrow}) + \delta_{ri\Uparrow} + \delta_{ri\Downarrow} + \delta_{li\Uparrow}$ and $\ell = \delta_{li\Uparrow}$.

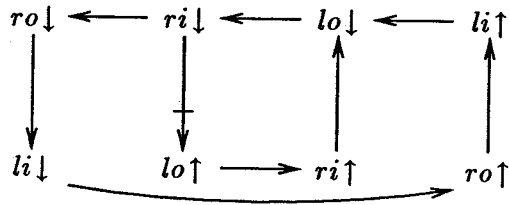


Figure 6.24: Buffer *d3*, $p = 2\ell + \delta_{ri\Uparrow} + \delta_{ri\Downarrow}$ and $\ell = \delta_{ri\Downarrow} + \delta_{li\Uparrow} + \delta_{li\Downarrow}$.

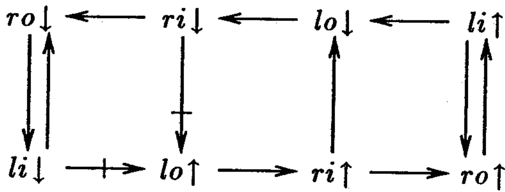


Figure 6.25: Buffer *d4*, $p = \delta_{ri\Uparrow} + \delta_{ri\Downarrow} + (2\delta_{li\Uparrow} \max 2\delta_{li\Downarrow})$ and $\ell = \delta_{li\Uparrow}$ max $\delta_{li\Downarrow}$.

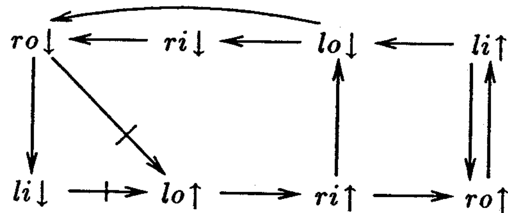


Figure 6.26: Buffer *d5*, $p = (\ell \max \delta_{li\Downarrow}) + \delta_{ri\Uparrow} + \delta_{ri\Downarrow} + \delta_{li\Uparrow}$ and $\ell = \delta_{li\Uparrow}$.

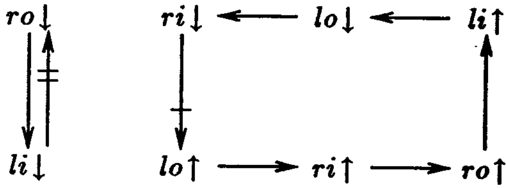


Figure 6.27: Buffer *e0*, **not** *CRT*.

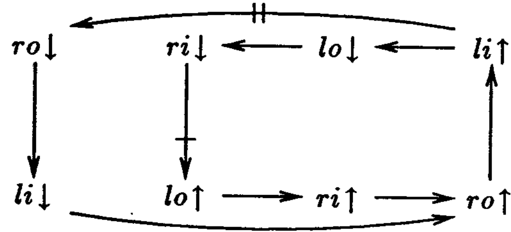


Figure 6.28: Buffer *e1*, $p = \ell + \delta_{ri\uparrow} + \delta_{ri\downarrow} + \delta_{li\uparrow}$ **and** $\ell = \delta_{ri\uparrow} + \delta_{ri\downarrow} + 2\delta_{li\uparrow} + \delta_{li\downarrow}$.

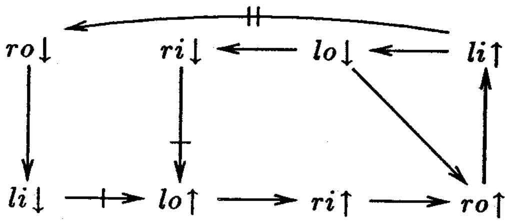


Figure 6.29: Buffer *e2*, $p = \ell + \delta_{ri\uparrow} + \delta_{ri\downarrow} + \delta_{li\uparrow}$ **and** $\ell = \delta_{ri\uparrow} + \delta_{li\uparrow} + \delta_{li\downarrow}$.

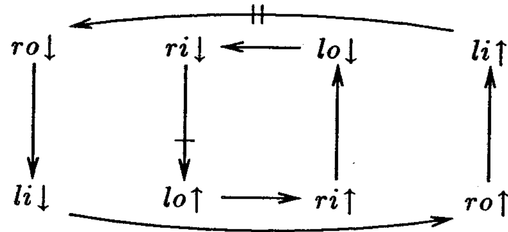


Figure 6.30: Buffer *e3*, **not** *CRT*.

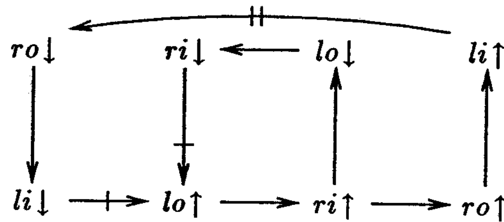


Figure 6.31: Buffer *e4*, $p = 2\ell + \delta_{ri\uparrow} + \delta_{ri\downarrow}$ **and** $\ell = \delta_{ri\uparrow} + \delta_{li\uparrow} + \delta_{li\downarrow}$.

6.2 Microprocessor

The *Asynchronous Microprocessor* described in [23] was designed using the analysis techniques of Chapter 2 and provides a good example of how to apply performance analysis to a synthesized design.

A complete CSP description of the microprocessor is given on Pages 356–7 of [23]. For the purposes of this study, we will only consider the following collection of straight-line processes that make up the control processes of the microprocessor:

$$\begin{aligned}
IMEM &= *[ID!imem[pc]] \\
FETCH &= *[PCI1; ID?i; PCI2; E1!i; E2] \\
PCADD &= *[PCI1; y := pc + 1; PCI2; pc := y] \\
EXEC &= *[E1?i; E2; Xs \bullet Ys \bullet AC!i.op \bullet ZAs] \\
ALU &= *[AC?op \bullet X?x \bullet Y?y; ZA!aluf(x, y, op)] \\
REG1 &= *[Xs \bullet X!r] \\
REG2 &= *[Ys \bullet Y!r] \\
REG3 &= *[ZAs; ZA?r]
\end{aligned}$$

The processes are straight-lined because the environment, in this case the process *IMEM*, supplies a continuous stream of *ADD R1,R2,R3* instructions. Thus, for all values of *pc*, the value of *imem*[*pc*] is this *ADD* instruction.

The performance of the above *CSP* program can be analyzed by first creating the *ER* system corresponding to this collection of straight-line handshaking expansions:

$$IMEM = *[[idi]; ido\uparrow; [\neg idi]; ido\downarrow]$$

$$\begin{aligned}
FETCH &= *[pcio\uparrow; [pcii \wedge \neg idi]; ido\uparrow; [idi]; ido\downarrow;\\
&\quad pcio\downarrow; [\neg pcii \wedge ei]; eo\uparrow; [\neg ei]; eo\downarrow]\\
PCADD &= *[[pcii]; pcio\uparrow; [\neg inci]; inco\uparrow; [inci]; inco\downarrow;\\
&\quad [\neg pcii]; pcio\downarrow; [\neg styi]; styo\uparrow; [styi]; styo\downarrow]\\
EXEC &= *[[\neg ei]; eo\uparrow; [ei]; eo\downarrow;\\
&\quad [\neg xsi \wedge \neg ysi \wedge \neg aci \wedge \neg zasi]; xso\uparrow, yso\uparrow, aco\uparrow, zaso\uparrow;\\
&\quad [xsi \wedge ysi \wedge aci \wedge zasi]; xso\downarrow, yso\downarrow, aco\downarrow, zaso\downarrow]\\
ALU &= *[[aci \wedge \neg xi \wedge \neg yi]; aco\uparrow, xo\uparrow, yo\uparrow;\\
&\quad [\neg aci \wedge xi \wedge yi]; aco\downarrow, xo\downarrow, yo\downarrow;\\
&\quad [zai]; zao\uparrow; [\neg zai]; zao\downarrow]\\
REG1 &= *[[xsi \wedge xi]; xso\uparrow, xo\uparrow; [\neg xsi \wedge \neg xi]; xso\downarrow, xo\downarrow]\\
REG2 &= *[[ysi \wedge yi]; yso\uparrow, yo\uparrow; [\neg ysi \wedge \neg yi]; yso\downarrow, yo\downarrow]\\
REG3 &= *[[zasi]; zaso\uparrow; [\neg zasi]; zaso\downarrow; [\neg zai]; zao\uparrow; [zai]; zao\downarrow]
\end{aligned}$$

The processes are connected together as shown in Figure 6.32. The cigar-shaped objects represent the datapaths used to transport values between the processes. In this analysis, we model the datapaths as delays. If the datapath intercepts the communication channel X, then we denote the delay during the rising portion of the data transfer as $\delta_{X\uparrow}$ and the delay during the falling portion as $\delta_{X\downarrow}$. All other delays in the system are assumed to be much smaller than the delays through the datapath and are modeled with the delay value zero.

Figures 6.33 and 6.34 together show the cycle period graph for this system. Some nodes are duplicated so that no cycles are required to span both figures. The cycles through Figure 6.33 result in the cycle-period constraints

$$p \geq \delta_{id\uparrow} + \delta_{id\downarrow}$$

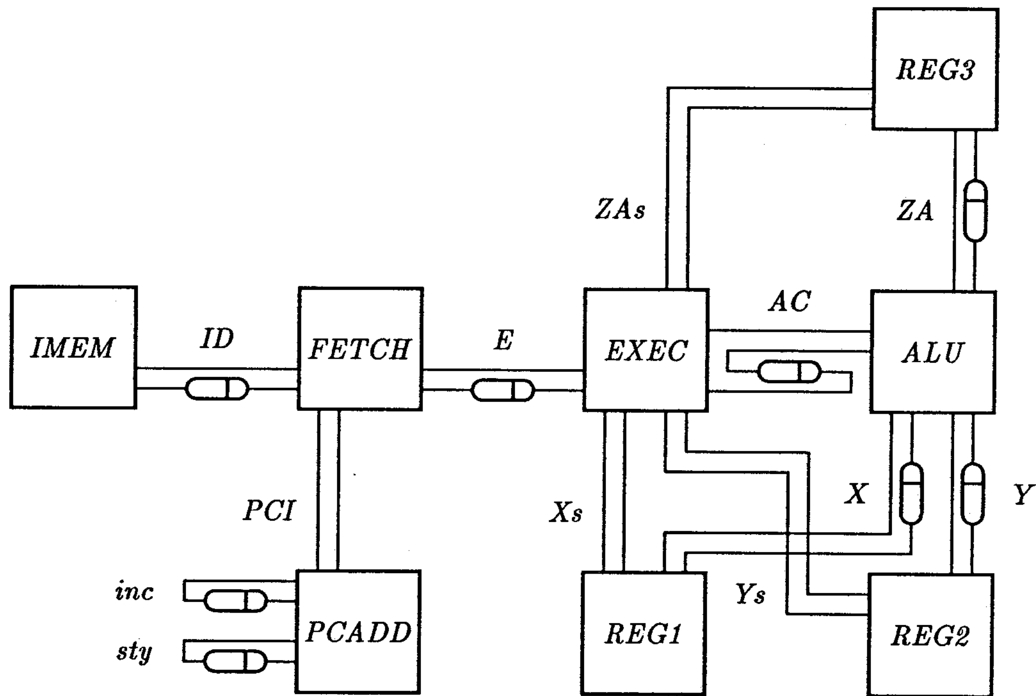


Figure 6.32: Process interconnections of the simplified asynchronous microprocessor.

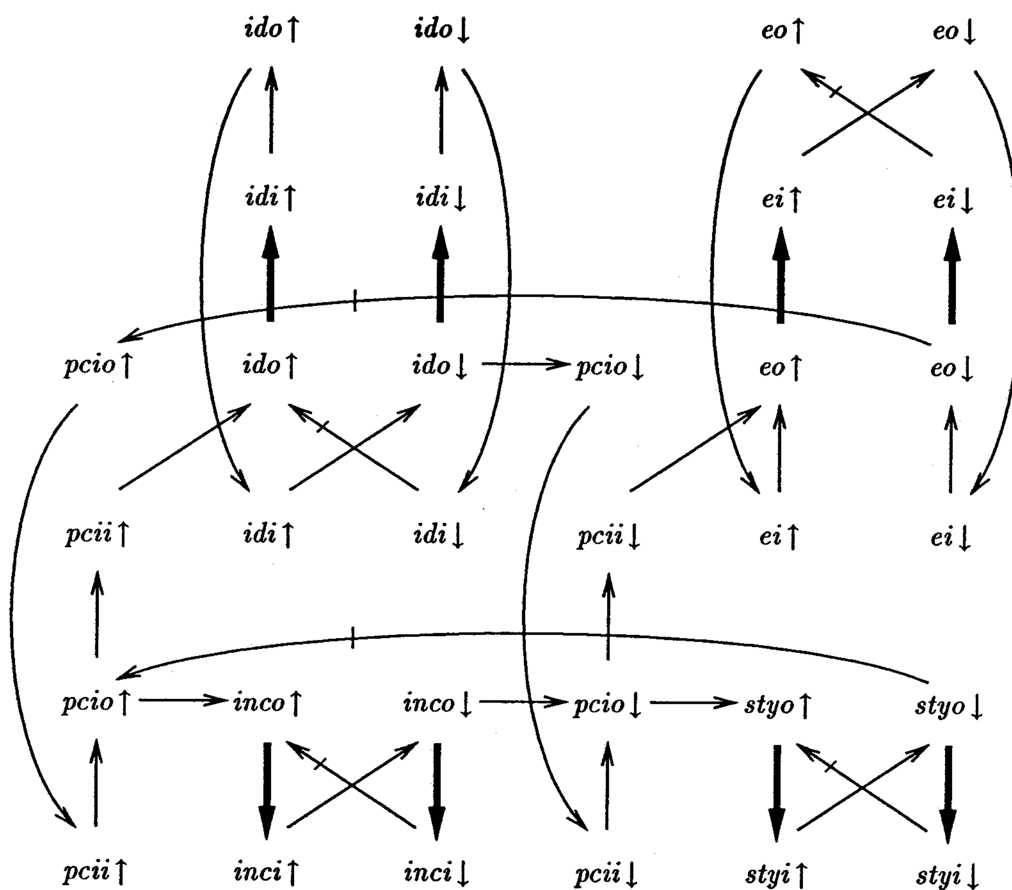


Figure 6.33: First part of the cycle period graph for the simplified asynchronous microprocessor. Bold arcs have a datapath delay associated with them. Normal arcs have a delay value of zero. A tick mark on an edges specifies that $\epsilon = 1$.

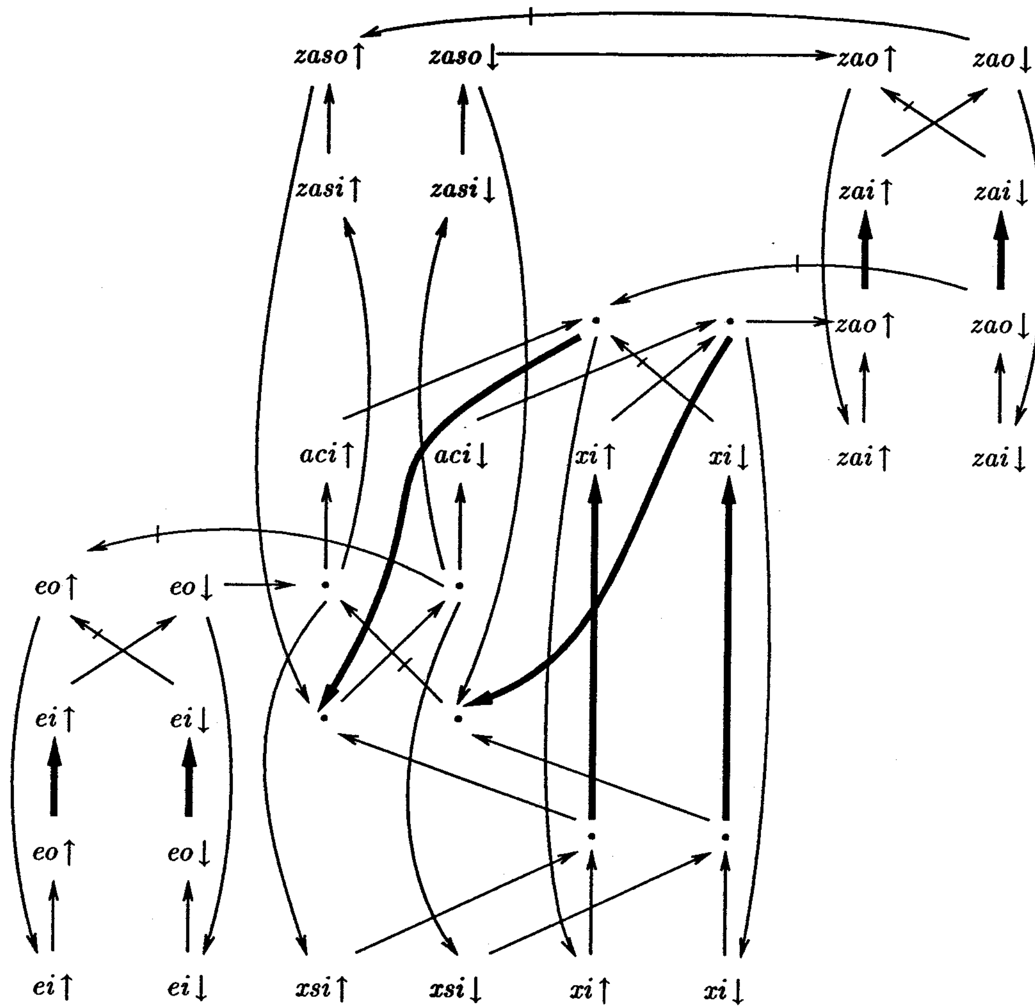


Figure 6.34: Second part of the cycle period graph for the simplified asynchronous microprocessor. For better clarity, not all nodes of the graph are shown. In particular, the process *REG2* is not included since it produces the same constraints as process *REG1*.

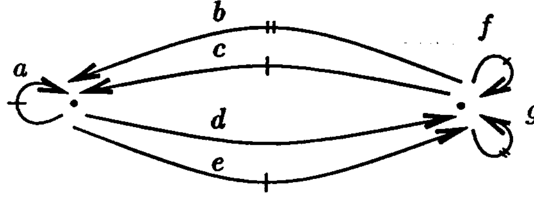


Figure 6.35: Simplified cycle-period graph corresponding to Figure 6.34. A single tick mark on the edge specifies an ε value of one. Two tick marks means $\varepsilon = 2$. Otherwise, $\varepsilon = 0$.

$$
\begin{aligned}
p &\geq \delta_{e\uparrow} + \delta_{e\downarrow} \\
p &\geq \delta_{inc\uparrow} + \delta_{inc\downarrow} \\
p &\geq \delta_{sty\uparrow} + \delta_{sty\downarrow} \\
p &\geq (\delta_{id\uparrow} \max \delta_{inc\uparrow}) + (\delta_{e\uparrow} \max \delta_{sty\uparrow})
\end{aligned}
$$

The graph in Figure 6.34 is too complex for direct cycle period analysis. By applying the transformations of Algorithm 2.1, we can simplify the cycle-period graph to that shown in Figure 6.35 with the following alpha values of the arcs:

$$
\begin{aligned}
\alpha_a &= \delta_{e\uparrow} + \delta_{ac\uparrow} \max \delta_{za\uparrow} + \delta_{za\downarrow} \\
\alpha_b &= \delta_{e\uparrow} + (\delta_{za\uparrow} \max \delta_{ac\downarrow}) \\
\alpha_c &= \delta_{ac\downarrow} \max \delta_{x\downarrow} \max \delta_{y\downarrow} \max \delta_{za\uparrow} \\
\alpha_d &= \delta_{ac\uparrow} \max \delta_{x\uparrow} \max \delta_{y\uparrow} \\
\alpha_e &= \delta_{ac\uparrow} + (\delta_{x\uparrow} \max \delta_{y\uparrow}) + \delta_{e\uparrow} \\
\alpha_f &= \delta_{ac\downarrow} + (\delta_{x\uparrow} \max \delta_{y\uparrow}) \max \delta_{za\uparrow} \\
\alpha_g &= (\delta_{za\uparrow} \max \delta_{ac\downarrow}) + (\delta_{x\uparrow} \max \delta_{y\uparrow}) + \delta_{e\uparrow}
\end{aligned}
$$

To generate this graph, first all nodes in the original graph that represent transitions on input variables are removed. Then selected output variables are removed, one by one, until this graph is produced.

We can further simplify this graph. Arcs a and e are subsumed by arcs c and d respectively, given that p is at least α_a. The self-loop created by arc g is subsumed by the two cycles formed by arcs c, d and a. Furthermore, arc f is subsumed by the cycle formed by arcs c and d. The cycles through the simplified graph impose the constraints:

$$p \geq \delta_{e\uparrow} + \delta_{ac\uparrow} \max \delta_{za\uparrow} + \delta_{za\downarrow}$$

$$p \geq (\delta_{ac\uparrow} \max \delta_{x\uparrow} \max \delta_{y\uparrow}) + (\delta_{x\downarrow} \max \delta_{y\downarrow} \max \delta_{za\uparrow} \max \delta_{ac\downarrow})$$

This analysis shows that at most two datapath delays occur in series when the microprocessor is performing a continuous stream of *ADD* instructions. The model shown here faithfully reproduces the implementation of the first fabricated $AM\,(2\mu m)$. An evaluation of the values of the datapath delays showed that the cycle period would depend on

$$\delta_{id\uparrow} + \delta_{id\downarrow}$$

or

$$\delta_{za\uparrow} + \delta_{za\downarrow}$$

depending on the propagation delay of the commercial *SRAMs* used in the instruction memory. Realizing that this was the case, and that we had fast commercial memory available, we modified the *ALU* process so that the result of the function *aluf* was stored in a local register before it was sent to process *REG3*. This modification had the effect of adding another pipeline stage to the processor and slightly complicating the mechanism needed to ensure the absence of *data hazards* [15]. The modification

split in two the data path activities modeled by the single delay named za. Only two named datapath delays occur in series in the modified version and the delay values in the datapath are smaller.

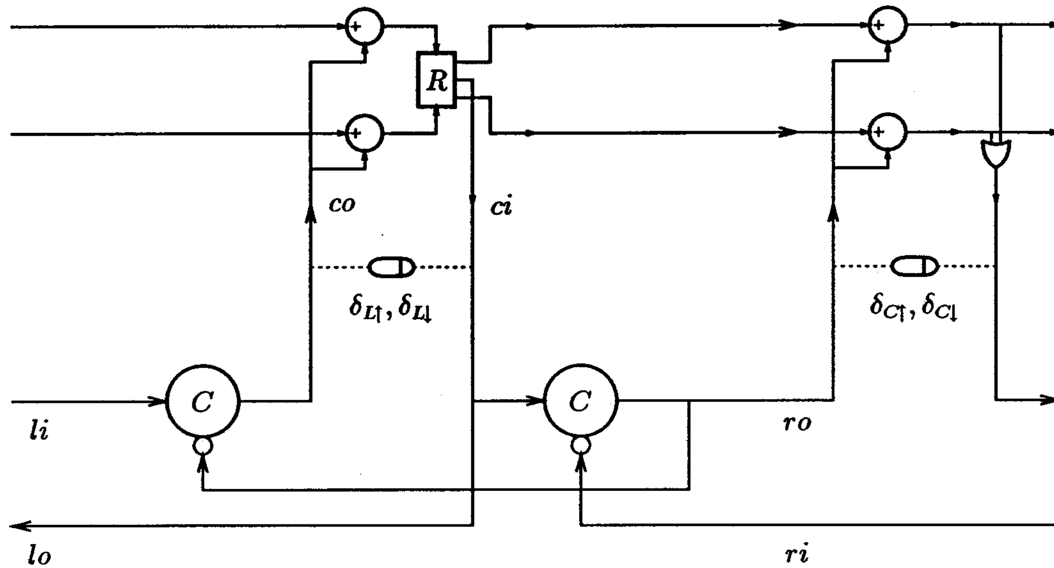


Figure 6.36: Meng's *FIFO* implementation

6.3 Other Asynchronous Pipeline Circuits

6.3.1 Meng

Teresa Meng [26] uses the circuit shown in Figure 6.36 to implement a pipelined *FIFO* buffer. The control circuit enforces more sequencing than is necessary resulting in an inefficient implementation.

To show that this is so, we will perform a cycle-period and latency analysis on Meng's implementation and on a second implementation that performs less sequencing but still enough to ensure correct operation of the datapath. Meng's pipeline control has the handshaking expansion:

$$*[[li]; co\uparrow; [ci]; lo\uparrow; [\neg ri]; ro\uparrow; [\neg li]; co\downarrow; [\neg ci]; lo\downarrow; [ri]; ro\downarrow]$$

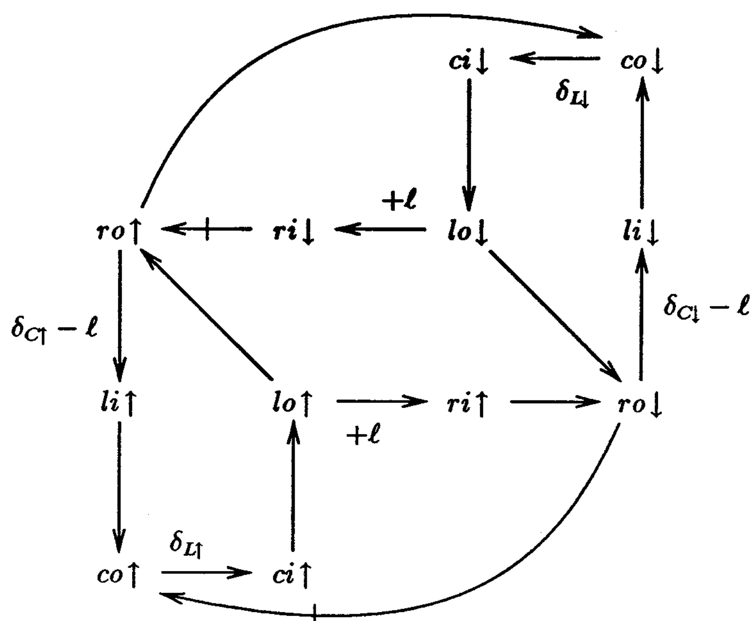


Figure 6.37: Latency/period constraint graph corresponding to Meng's *FIFO* implementation.

The constraint graph corresponding to this handshaking expansion is shown in Figure 6.37. A cycle-period analysis shows that four datapath delays occur in series. The constraints imposed by the six cycles through the graph are

$$\begin{aligned}
\ell &\geq \delta_{C\uparrow} + \delta_{L\uparrow} \max \delta_{C\downarrow} + \delta_{L\downarrow} \\
p &\geq \ell + (\delta_{L\uparrow} \max \delta_{L\downarrow}) \\
p &\geq \delta_{L\uparrow} + \delta_{L\downarrow} \\
p &\geq \delta_{C\uparrow} + \delta_{L\uparrow} + \delta_{C\downarrow} + \delta_{L\downarrow}
\end{aligned}$$

resulting in a latency and cycle period of

$$\ell = \delta_{C\uparrow} + \delta_{L\uparrow} \max \delta_{C\downarrow} + \delta_{L\downarrow}$$

$$p = \delta_{C\uparrow} + \delta_{L\uparrow} + \delta_{C\downarrow} + \delta_{L\downarrow} \max \delta_{C\uparrow} + 2\delta_{L\uparrow} \max \delta_{C\downarrow} + 2\delta_{L\downarrow}$$

Faster Pipeline Stage

An alternative handshaking expansion is

$$*[[li]; co\uparrow; [ci]; lo\uparrow; co\downarrow; [\neg ri]; ro\uparrow; [\neg li \wedge \neg ci]; lo\downarrow; [ri]; ro\downarrow]$$

In this case, $co\downarrow$ is performed earlier in the sequence. This change does not violate any requirements imposed by the datapath, since the data output of the register remains valid even after co falls. (Note that in both implementations, ro must be implemented with an isochronic fork between the control and data parts or by a latched-output stage.) The constraint graph corresponding to this handshaking expansion is shown in Figure 6.38. A cycle-period analysis shows that only three datapath delays occur in series. The constraints imposed by the four cycles through the graph are

$$\begin{aligned}
\ell &\geq \delta_{C\uparrow} + \delta_{L\uparrow} \\
p &\geq \ell + \delta_{L\uparrow} \\
p &\geq \delta_{C\uparrow} + \delta_{L\uparrow} + \delta_{L\downarrow} \\
p &\geq \delta_{C\uparrow} + \delta_{L\uparrow} + \delta_{C\downarrow}
\end{aligned}$$

resulting in a latency and cycle period of

$$\begin{aligned}
\ell &= \delta_{C\uparrow} + \delta_{L\uparrow} \\
p &= \delta_{C\uparrow} + \delta_{L\uparrow} + (\delta_{C\downarrow} \max \delta_{L\uparrow} \max \delta_{L\downarrow})
\end{aligned}$$

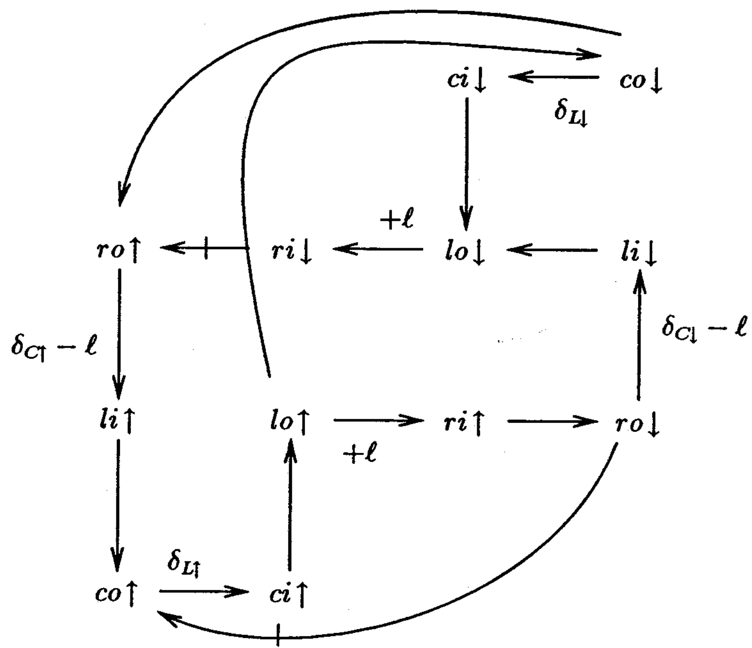


Figure 6.38: Latency/period constraint graph of a second handshaking expansion with weaker sequencing

Comparison to the Active-Passive Pipelines

We now compare these two designs to the best pipelines designed in Section 6.1.4. Because datapaths used differ slightly, we decompose them into comparable units. In Meng's pipeline, the register and computation units are separated; two completion trees are required. In the active-passive pipelines, the computation and register units are combined together into a single unit with a single completion tree. We will write the delay through the latch in Meng's pipeline as $\delta_r + \delta_c$ and the delay through the computation block as $\delta_o + \delta_c$ where the δ_r, δ_c and δ_o represent register, completion tree and output block delays, respectively. The delay through the entire datapath stage in the active-passive case will be written $\delta_o + \delta_r + \delta_c$. Up and down delays are considered equal. With these assumptions, the cycle periods of these designs are:

$$
\begin{aligned}
p_{Meng} &= 2\delta_r + 2\delta_c + 2\delta_o + 2\delta_c \\
p_{Mod} &= 2\delta_r + 2\delta_c + \delta_o + \delta_c \\
p_{lap} &= 2\delta_o + 2\delta_r + 2\delta_c
\end{aligned}
$$

The modified Meng pipeline and the lazy-active/passive pipeline both have a better cycle period than the Meng pipeline. *Mod* has a smaller cycle period if $\delta_c < \delta_o$.

6.3.2 Greenstreet, Williams, and Staunstrup

Greenstreet, Williams, and Staunstrup [14] use the implementation of a pipeline stage as shown in Figure 6.39 for their work with self-timed iterations. A number n of these pipeline stages (at least three and with at least one process initialized differently) are connected together to form a ring. If n is large, the cycle period of the ring is determined by the latency per stage of an infinite array, multiplied by n. If n is small, the cycle period of the infinite chain can contribute to the performance of the

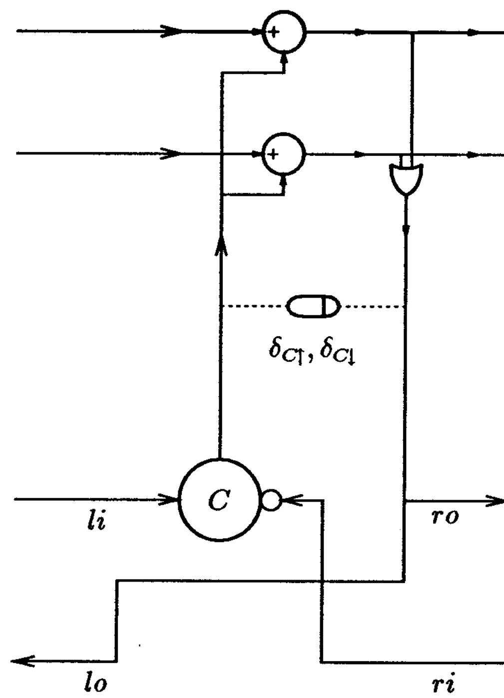


Figure 6.39: Implementation of the pipeline stage used by Greenstreet, Williams, and Staunstrup [14].

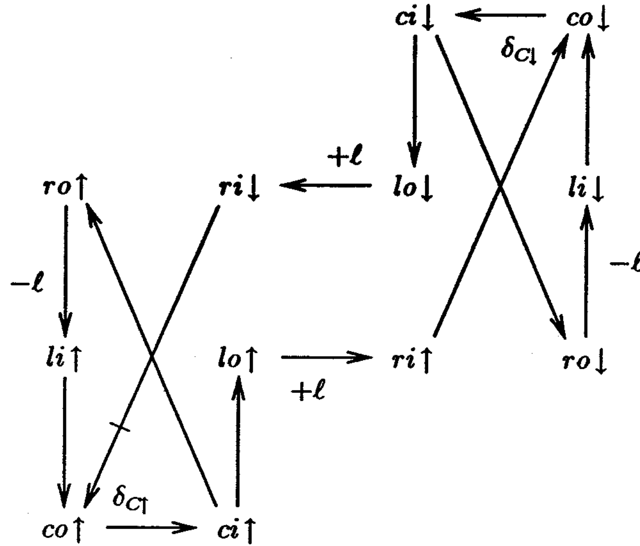


Figure 6.40: Latency/period constraint graph of the GWS pipeline stage.

system.

The handshaking for the control part of this stage is:

$$*[[li \wedge \neg ri]; co\uparrow; [ci]; lo\uparrow, ro\uparrow; [\neg li \wedge ri]; co\downarrow; [\neg ci]; lo\downarrow, ro\downarrow]$$

The corresponding constraint graph is shown in Figure 6.40. The three cycles in the graph produce the constraints

$$\begin{aligned} \ell &= \delta_{C\uparrow} \max \delta_{C\downarrow} \\ p &= 2\ell + \delta_{C\uparrow} + \delta_{C\downarrow}. \end{aligned}$$

The cycle period has four datapath delays in series and the latency has only one. While it is possible to redesign the control process in order to reduce the cycle period

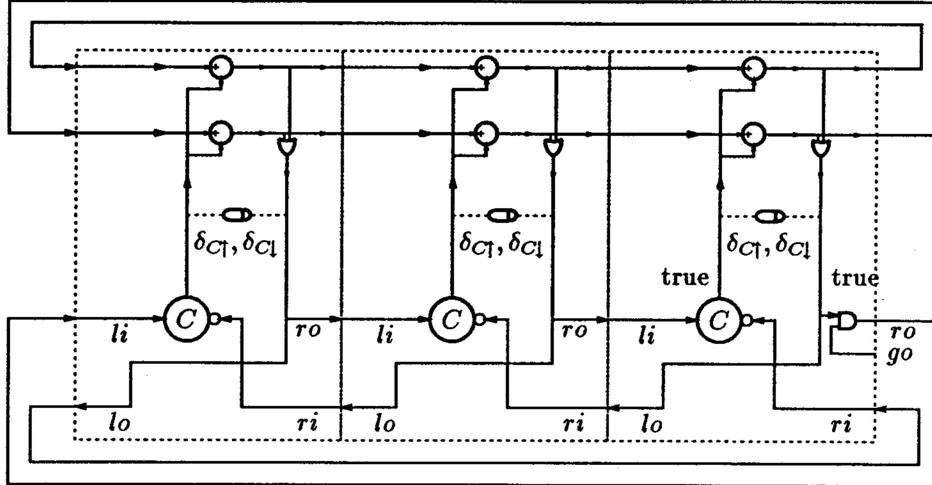


Figure 6.41: Three GWS pipeline stages connected to form a ring.

of a long chain of these stages, it is not the performance metric of interest.

What is important is the cycle period of three stages connected together to form a ring (Figure 6.41). The surprising result is that with this implementation, all *six* of the datapath delays occur in series. Even with the (non-speed-independent) optimization performed in [14, 38], all datapath delays occur in series.

The constraint graph shown in Figure 6.42 is constructed from the three pipeline stages:

$$r0 \equiv *[[li \wedge \neg ri]; co\uparrow; [ci]; lo\uparrow, ro\uparrow; [\neg li \wedge ri]; co\downarrow; [\neg ci]; lo\downarrow, ro\downarrow]$$

$$r1 \equiv *[[li \wedge \neg ri]; co\uparrow; [ci]; lo\uparrow, ro\uparrow; [\neg li \wedge ri]; co\downarrow; [\neg ci]; lo\downarrow, ro\downarrow]$$

$$r2 \equiv *[ro\uparrow; [\neg li \wedge ri]; co\downarrow; [\neg ci]; lo\downarrow, ro\downarrow; [li \wedge \neg ri]; co\uparrow; [ci]; lo\uparrow]$$

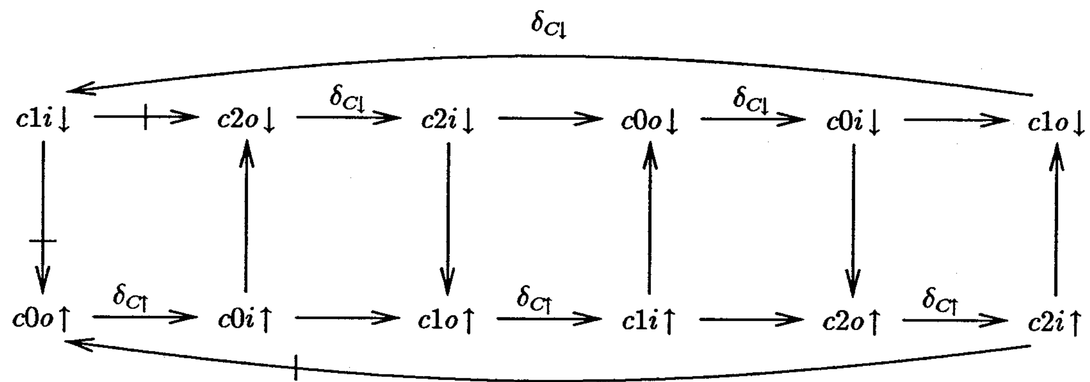


Figure 6.42: Latency/period constraint graph of three GWS pipeline stages connected to form a ring.

Process $r2$ begins with the R communication and the other two processes begin with the L communication. Note that transitions on the handshaking variables li, lo, ro and ri can be removed from the constraint graph since each transition is a copy of a transition on ci in one of the three processes. We can rewrite the handshaking expansions in this form:

$$\begin{aligned}
r0 &\equiv *[[l0i \wedge \neg c1i]; c0o\uparrow; [\neg l0i \wedge c1i]; c0o\downarrow] \\
r1 &\equiv *[[c0i \wedge \neg c2i]; c1o\uparrow; [\neg c0i \wedge c2i]; c1o\downarrow] \\
r2 &\equiv *[r2o\uparrow; [\neg c1i \wedge c0i]; c2o\downarrow; [\neg c2i]; r2o\downarrow; [c1i \wedge \neg c0i]; c2o\uparrow; [c2i]]
\end{aligned}$$

To determine the ε values on the arcs of the constraint graph, we see that the wait $[\neg ri]$ is vacuous in process $r0$ but not vacuous in process $r1$ since lo is initially true in process $r2$. The wait $[\neg li]$ is vacuous in process $r2$. There is a cycle through this constraint graph corresponding to the cycle period $3\delta_{C\uparrow} + 3\delta_{C\downarrow}$.

As the number of stages n in the ring increases, the number of datapath delays in series becomes n.

Improved Handshaking

To improve this design, we observe the requirements imposed by the datapath:

$$\begin{aligned}
\langle ci, j-1\rangle^{(-1)}\downarrow &\prec \langle co, j\rangle\uparrow \\
\langle ci, j+1\rangle\uparrow &\prec \langle co, j\rangle\downarrow \\
\langle ci, j+1\rangle^{(-1)}\downarrow &\prec \langle co, j\rangle\uparrow \\
\langle co, j+1\rangle^{(-1)}\downarrow &\prec \langle co, j\rangle\uparrow
\end{aligned}$$

The first constraint ensures that the old data in stage $j-1$ has been reset before the latch is open in stage j. The second constraint ensures that data coming into stage $j+1$ is latched before the data in stage j is lowered. The final two constraints ensure that the latch control in stage $j+1$ is lowered before new data values are sent in stage j. The third constraint is more strict and allows an implementation without an isochronic fork between the control and data parts. The fourth constraint forces an implementation that uses an isochronic fork or a latched output stage, but then the sequencing is considerably more efficient. If we enforce the fourth constraint instead of the stronger third constraint, the cycle period of an infinite chain of these stages is improved by 33 percent and the cycle period of a ring of three stages is improved by 100 percent. No improvement is observed in the cycle period of a ring with more than three stages, since the performance in constrained by the latency of each stage.

The handshaking expansion

$$\begin{aligned} lr \equiv\ & *[[\neg ri' \wedge li]; lo' \uparrow; co\uparrow; [ci]; lo'' \uparrow; ro\uparrow; \\ & [ri'' \wedge \neg li]; lo' \downarrow; co\downarrow; [\neg ci]; lo'' \downarrow; ro\downarrow] \end{aligned}$$

implements these constraints. The handshake wire from lo to ri has been replaced by two wires; one from lo' to ri', and the other from lo'' to ri''. Both lo' and lo'' are implemented with an isochronic fork since it is necessary to ensure that $lo'\uparrow$ occurs before $co\uparrow$ and that $lo''\downarrow$ occurs before $ro\downarrow$.

Figure 6.43 shows the constraint graph corresponding to an infinite array of the process lr. At most three external delays occur in series. The latency remains one external delay. Three stages connected in a ring are shown in Figure 6.44. The

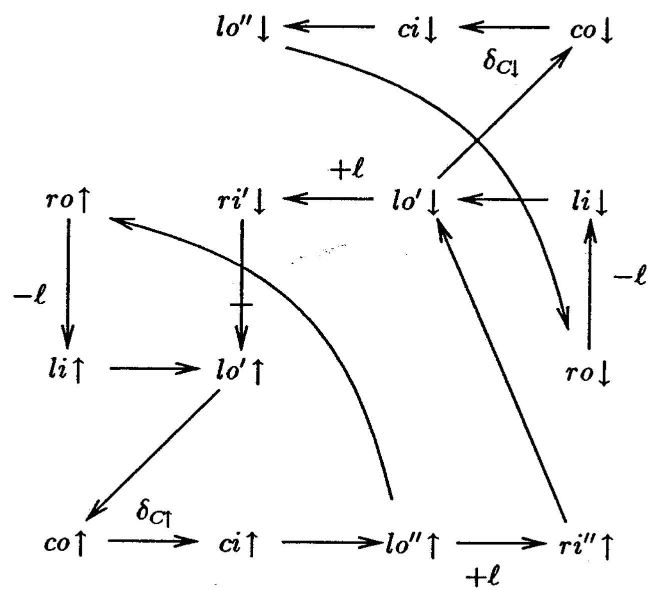


Figure 6.43: Latency/period constraint graph for an infinite array of the modified GWS pipeline stage.

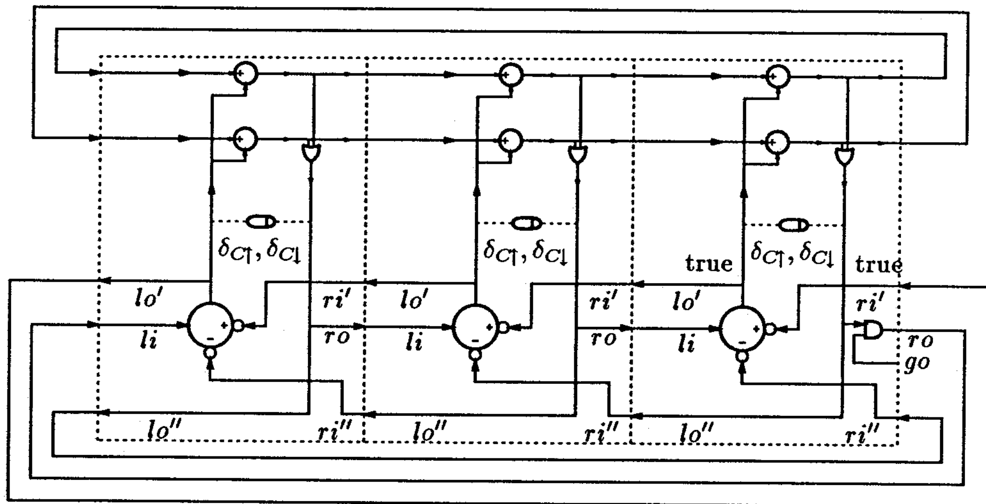


Figure 6.44: Three modified GWS pipeline stages connected in a ring.

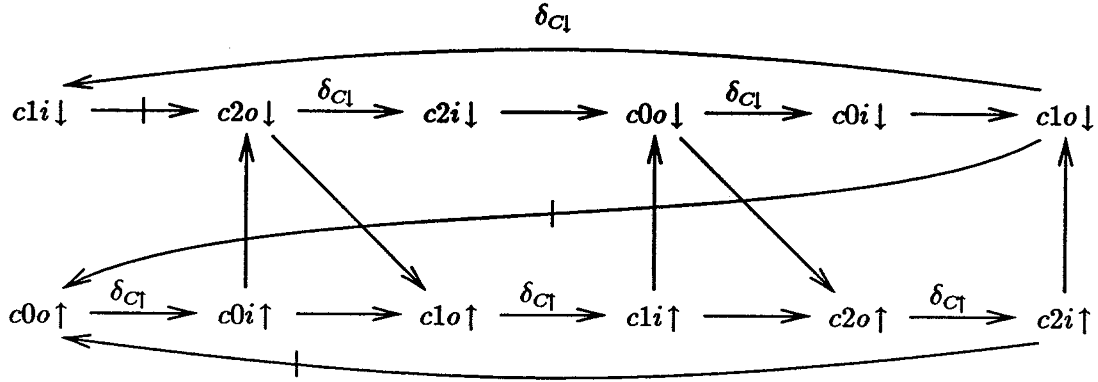


Figure 6.45: Latency/period constraint graph of three modified GWS pipeline stages connected in a ring.

rightmost process represents an implementation of

$$\begin{aligned} rl \;\equiv\; & *[ro\uparrow; [ri'' \wedge \neg li]; lo'\downarrow; co\downarrow; [\neg ci]; lo''\downarrow; ro\downarrow; \\ & [\neg ri' \wedge li]; lo'\uparrow; co\uparrow; [ci]; lo''\uparrow]\,. \end{aligned}$$

The process is needed to break deadlock in the ring. The constraint graph from this circuit (Figure 6.45) shows that at most three external delays occur in series. The constraints in the graph contributed by each lr process j are:

$$\begin{aligned} \langle co, j\rangle\uparrow &\longrightarrow \langle ci, j\rangle\uparrow \\ \langle co, j\rangle\downarrow &\longrightarrow \langle ci, j\rangle\downarrow \\ \langle ci, j-1\rangle\uparrow &\longrightarrow \langle co, j\rangle\uparrow \\ \langle ci, j+1\rangle\uparrow &\longrightarrow \langle co, j\rangle\downarrow \end{aligned}$$

$$\begin{aligned}\langle ci, j-1\rangle\downarrow &\longrightarrow \langle co, j\rangle\downarrow\\ \langle co, j+1\rangle\downarrow^{(-1)} &\longrightarrow \langle co, j\rangle\uparrow\end{aligned}$$

The last constraint has an ε value of one only when it is connected on the right to an *lr* process. The constraints in the graph contributed by each *rl* process j are:

$$\begin{aligned}\langle co, j\rangle\uparrow &\longrightarrow \langle ci, j\rangle\uparrow\\ \langle co, j\rangle\downarrow &\longrightarrow \langle ci, j\rangle\downarrow\\ \langle ci, j-1\rangle\downarrow^{(-1)} &\longrightarrow \langle co, j\rangle\downarrow\\ \langle ci, j\rangle\uparrow^{(-1)} &\longrightarrow \langle co, j+1\rangle\uparrow\\ \langle ci, j+1\rangle\uparrow &\longrightarrow \langle co, j\rangle\downarrow\\ \langle co, j+1\rangle\downarrow &\longrightarrow \langle co, j\rangle\uparrow\end{aligned}$$

Chapter 7

Performance Optimization

In this chapter, we discuss methods for optimally sizing the transistors of an asynchronous circuit with a fixed topology. Using the analysis methods from Chapter 2, a performance metric, for example, the minimum cycle period p, can be expressed in terms of the delays of an *ER* system. These delays can be estimated from the sizes of the transistors that make up the operators of the circuit, and the way these operators are interconnected, by using a simple resistance-capacitance (RC) timing model. Composing the performance metric in terms of component delays with the delay approximation of the operators in terms of transistor sizes, we get an expression for the performance of the system in terms of transistor sizes. This expression is optimized, producing optimal sizes for the transistors.

7.1 Tau Model

A simple RC switch model is used to relate each individual delay α to the widths of circuit's transistors (w's). Each transistor is modeled as a switch with a resistance inversely proportional to its width. The gate of a transistor has a capacitance to ground proportional to its width. Source and drain capacitances are also proportional

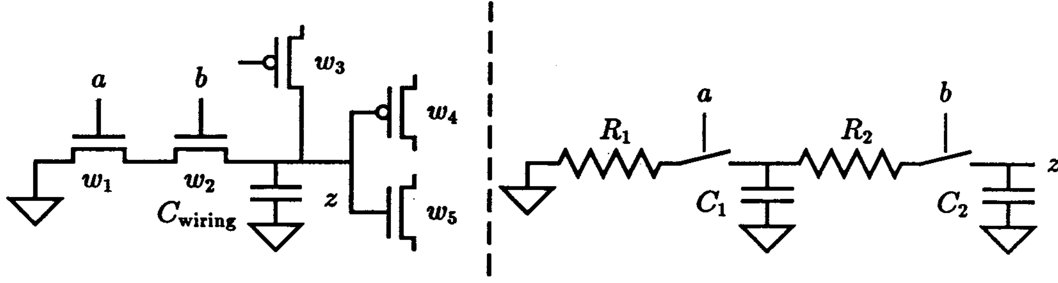


Figure 7.1: RC approximation of a CMOS pulldown.

to transistor widths. Thus, the delays between $a\uparrow$ and $z\downarrow$, and $b\uparrow$ and $z\downarrow$, of the circuit shown in Figure 7.1 are modeled as:

$$\alpha_{a\uparrow z\downarrow} = R_1C_1 + (R_1 + R_2)C_2 \tag{7.1}$$

$$\alpha_{b\uparrow z\downarrow} = (R_1 + R_2)C_2 \tag{7.2}$$

$$R_1 = \mu/w_1$$

$$R_2 = \mu/w_2$$

$$C_1 = K_{\text{int}}(w_1 + w_2)$$

$$C_2 = K_{\text{ext}}(w_2 + w_3) + C_{\text{wiring}} + K_{\text{g}}(w_4 + w_5) \quad ,$$

where μ is a constant that describes the differing per-unit-width strengths of the n- and p-channel transistors, K_{int} is the per-unit-width capacitance contributed by internal (to the series chain) drain and source terminals, K_{ext} is the per-unit-width capacitance contributed by external (the output node) drain terminals, K_{g} is the per-unit-width gate capacitance and C_{wiring} is the capacitance contributed by wiring. All capacitances are expressed in terms of transistor width, and thus $K_{\text{g}} = 1$. Each delay α is expressed in units of τ, the time needed for a unit-width n-channel transistor to switch a unit-width load. (Thus, $\mu_{\text{n}} = 1$, $\mu_{\text{p}} > 1$.) The values of K_{int}, K_{ext} and C_{wiring} are not constant, but depend on the final circuit layout that depends weakly

on the transistor widths. This dependence is normally small and is ignored in the optimization problem.

Timing models similar to the tau model are used in other transistor optimization tools (designed for synchronous circuits), such as TILOS [11], COP [21], and EPOXY [28].

Example 7.1 As an example, we will construct the optimization equations for the C-element circuit. From the cycle-period analysis,

$$\begin{aligned} p &= \alpha_{u\uparrow z\downarrow} + (\alpha_{z\downarrow x\uparrow} + \alpha_{x\uparrow u\downarrow}) \max(\alpha_{z\downarrow y\uparrow} + \alpha_{y\uparrow u\downarrow}) + \\ &\quad \alpha_{u\downarrow z\uparrow} + (\alpha_{z\uparrow x\downarrow} + \alpha_{x\downarrow u\uparrow}) \max(\alpha_{z\uparrow y\downarrow} + \alpha_{y\downarrow u\uparrow}) . \end{aligned}$$

For purposes of this example, the constants of the tau model take on these values:

$$K_{\text{ext}} = 1, \quad K_{\text{int}} = 0.5, \quad K_{\text{g}} = 1, \quad \mu_p = \mu_n = 1, \quad C_{\text{wiring}} = 0 .$$

Since the mobilities of the pull-up and pull-down devices are assumed to be identical, by symmetry, the widths of the pull-up and pull-down devices are identical. Similarly, the pull-up and pull-down delays should be identical. Thus, we can simplify the expression for the cycle period to

$$\frac{1}{2}p = \alpha_{uz} + (\alpha_{zx} + \alpha_{xu}) \max(\alpha_{zy} + \alpha_{yu}) .$$

To compute the delay between $z\downarrow$ and $x\uparrow$, we compute the external load and multiply it by the resistance of the driving transistor. So,

$$C_{\text{load}} = K_{\text{ext}}(w_{z\downarrow x\uparrow} + w_{z\uparrow x\downarrow}) + K_{\text{g}}(w_{x\downarrow u\uparrow} + w_{x\uparrow u\downarrow}) + C_{\text{wiring}}$$

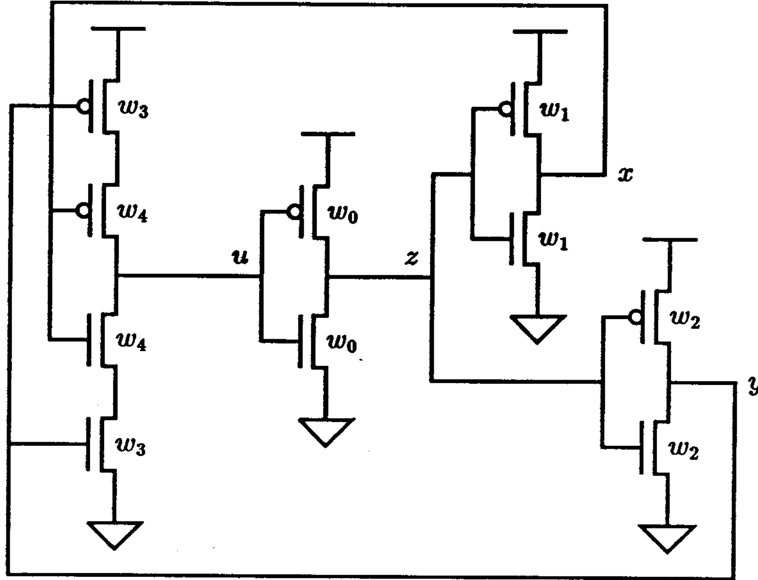


Figure 7.2: CMOS circuit for C-element and the trivial environment. Equal mobilities result in symmetric pull-up and pull-down widths as well as symmetric pull-up and pull-down delays.

$$R_{\text{drive}} = \frac{\mu_p}{w_{z\downarrow x\uparrow}}.$$

Substituting in values of the constants and applying the symmetry simplification yields:

$$\alpha_{z\downarrow x\uparrow} = \alpha_{zx} = \frac{1}{w_1}(2w_1 + 2w_4).$$

The delay between $y\uparrow$ and $u\downarrow$ is $R_1C_1 + (R_1 + R_2)C_2$, where

$$\begin{aligned}
R_1 &= \frac{\mu_n}{w_{y\uparrow u\downarrow}} \qquad R_2 = \frac{\mu_n}{w_{x\uparrow u\downarrow}} \\
C_1 &= K_{\text{int}}(w_{y\uparrow u\downarrow} + w_{x\uparrow u\downarrow}) \\
C_2 &= K_{\text{ext}}(w_{x\uparrow u\downarrow} + w_{x\downarrow u\uparrow}) + K_{\text{g}}(w_{u\uparrow z\downarrow} + w_{u\downarrow z\uparrow}) + C_{\text{wiring}}.
\end{aligned}$$

Substituting and simplifying, we get

$$\alpha_{y\uparrow u\downarrow} = \alpha_{yu} = \frac{1}{2w_3}(w_3 + w_4) + \left(\frac{1}{w_3} + \frac{1}{w_4}\right)(2w_4 + 2w_0).$$

For the other three delays, we get

$$\begin{aligned}
\alpha_{x\uparrow u\downarrow} &= \alpha_{xu} = \left(\frac{1}{w_3} + \frac{1}{w_4}\right)(2w_4 + 2w_0) \\
\alpha_{u\uparrow z\downarrow} &= \alpha_{uz} = \frac{1}{w_0}(2w_0 + 2w_1 + 2w_2) \\
\alpha_{z\uparrow y\downarrow} &= \alpha_{zy} = \frac{1}{w_2}(2w_2 + 2w_3).
\end{aligned}$$

Replacing these delay expressions in the definition of the cycle period, we get

$$\begin{aligned}
\frac{1}{4}p &= \frac{1}{w_0}(w_0 + w_1 + w_2) + \\
&\quad \left(\frac{1}{w_1}(w_1 + w_4) + \left(\frac{1}{w_3} + \frac{1}{w_4}\right)(w_4 + w_0)\right) \max \\
&\quad \left(\frac{1}{w_2}(w_2 + w_3) + \frac{1}{4w_3}(w_3 + w_4) + \left(\frac{1}{w_3} + \frac{1}{w_4}\right)(w_4 + w_0)\right).
\end{aligned}$$

Setting $w_0 = 1$, which we can do because $C_{\text{wiring}} = 0$, and simplifying, we get

$$\frac{1}{4}p = 3 + w_1 + w_2 + \frac{w_4}{w_3} + \frac{1}{w_3} + \frac{1}{w_4} + \frac{w_4}{w_1} \max \left(\frac{w_3}{w_2} + \frac{1}{4} + \frac{w_4}{4w_3}\right).$$

We can find the optimum values of the width variables by optimizing this expression with respect to the variables w_1, w_2 , w_3 and w_4. Here we use the traditional approaches from elementary calculus. An optimal solution to $f = 0.25p$ on the interior of the domain ($w_i > 0$) exists at a point where the gradient is zero or at a point of discontinuity of the gradient. However, the gradient is never zero over the interior of

the domain since:

$$\begin{aligned}\frac{\partial f}{\partial w_1} &= 1 \quad \text{if } \left(\frac{w_3}{w_2}+\frac{1}{4}+\frac{w_4}{4w_3}\right) > \frac{w_4}{w_1}\\ \frac{\partial f}{\partial w_2} &= 1 \quad \text{if } \left(\frac{w_3}{w_2}+\frac{1}{4}+\frac{w_4}{4w_3}\right) < \frac{w_4}{w_1}\end{aligned}$$

Thus, the minimum must be achieved at a point of discontinuity of ∇f, and this can only be when both expressions in the max statement have the same value. Since this is the case, we can use Lagrange Multiplier techniques to find the optimal value. Optimal values of $f(w_1, w_2, w_3, w_4)$ subject to the constraint

$$g(w_1, w_2, w_3, w_4) = \frac{w_4}{w_1} - \left(\frac{w_3}{w_2}+\frac{1}{4}+\frac{w_4}{4w_3}\right) = 0$$

are achieved when

$$\nabla f(w_1, w_2, w_3, w_4) = \lambda \nabla g(w_1, w_2, w_3, w_4). \tag{7.3}$$

The minimum value is 8.7202 and is achieved at:

$$(w_1, w_2, w_3, w_4, \lambda) = (0.4782, 1.1632, 1.8002, 0.9209, -0.7516).$$

There is another solution to (7.3) but it does not represent a minimum point. The plots in Figure 7.3 through Figure 7.6 show the cross section of f at the optimal point with respect to each of the parameters w_1, w_2, w_3 and w_4. □

7.2 Convex Objective Function

Every α derived using this simple model (and also other more accurate models) is a posynomial function (polynomial with positive coefficients and positive variables)

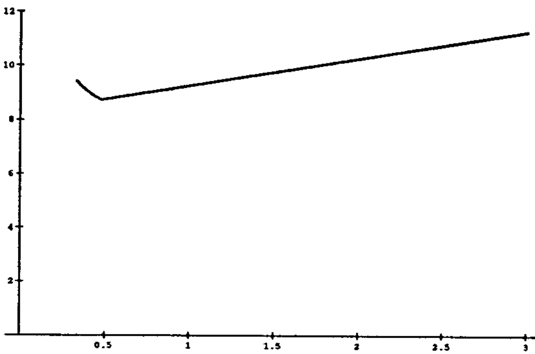


Figure 7.3: Cross section of f with respect to w_1.

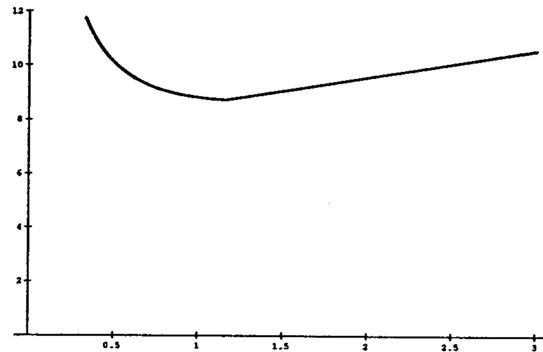


Figure 7.4: Cross section of f with respect to w_2.

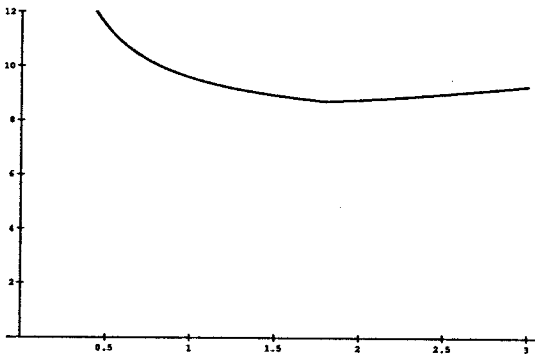


Figure 7.5: Cross section of f with respect to w_3.

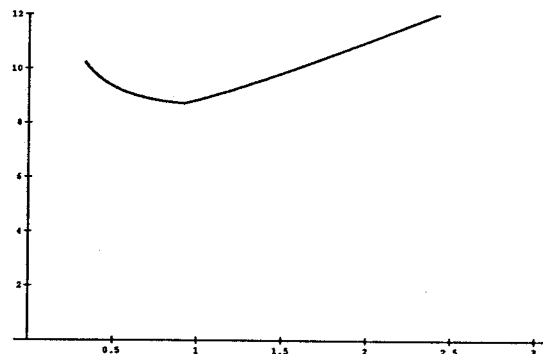


Figure 7.6: Cross section of f with respect to w_4.

of the transistor widths w's, and thus a convex function of the $\log w$'s [13]. Because both the sum and the maximum of two convex functions are convex functions, the resulting expression for p is a convex function of the $\log w$'s; and, thus, each minimum of p is global.

The addition of convex constraints, for example, to limit energy usage or to bound transistor sizes, does not alter the unique minimum property.

7.2.1 Power Constraint

There are two ways to introduce the power constraint: If the power value is some convex function of the widths, then we can add the constraint

$$power(w) \leq budget$$

and perform the minimization subject to this constraint. However, if we know that the constraint is satisfied with equality at the optimal value and the power function is linear in the widths, then we can transform the constrained optimization problem into an unconstrained problem.

Example 7.2 If $C_{\text{wiring}} = 4$ instead of 0, the performance metric equation from Example 7.1 becomes:

$$\begin{aligned} f &= \frac{1}{w_0}(w_0 + w_1 + w_2 + 2) + \left(\frac{1}{w_3} + \frac{1}{w_4}\right)(w_4 + w_0 + 2) \\ &\quad \left(\frac{1}{w_1}(w_1 + w_4 + 2)\right) \max\left(\frac{1}{w_2}(w_2 + w_3 + 2) + \frac{1}{4w_3}(w_3 + w_4)\right) \end{aligned}$$

No finite solution vector minimizes f, since larger values of the widths will always reduce the contribution of the terms with a constant numerator and a width in the denominator. In order to find a practical solution, we constrain the total transistor

width to be equal to a power budget. (In CMOS, transistor width is proportional to capacitance, and that is proportional to power.) The total capacitance that switches per cycle (all the capacitance in this circuit) is proportional to:

$$
\begin{aligned}
\textit{power} \cdot w + c_0 &= (w_0 + w_1 + w_2 + 2) + (w_4 + w_0 + 2) + \\
&\qquad (w_1 + w_4 + 2) + (w_2 + w_3 + 2) + 0.25(w_3 + w_4) \\
&= 2w_0 + 2w_1 + 2w_2 + 1.25w_3 + 2.25w_4 + 8 \\
&= \textit{budget}
\end{aligned}
$$

For example, assume a power budget of 103 units (on average $w_i = 10$). Subtracting and rewriting *power* so that $\textit{power} \cdot w = 1$, we get

$$
\begin{aligned}
\textit{power} \cdot w &= \left(\frac{2}{95}, \frac{2}{95}, \frac{2}{95}, \frac{1.25}{95}, \frac{2.25}{95}\right) \cdot (w_0, w_1, w_2, w_3, w_4) \\
&= 1
\end{aligned}
$$

As long as at the optimal solution $w_{i^*} > 0$, we can optimize the function

$$
f\left(\frac{u}{\textit{power} \cdot u}\right) \qquad \text{with } u_{i^*} = 1
$$

to obtain the constrained minimum value and the optimal solution vector. This function is convex. The performance metric f can be rewritten

$$
\begin{aligned}
f &= 1 + \frac{w_1}{w_0} + \frac{w_2}{w_0} + \frac{2}{w_0} + \frac{w_4}{w_3} + \frac{w_0}{w_3} + \frac{2}{w_3} + 1 + \frac{w_0}{w_4} + \frac{2}{w_4} + \\
&\qquad \left(1 + \frac{w_4}{w_1} + \frac{2}{w_1}\right) \max\left(1 + \frac{w_3}{w_2} + \frac{2}{w_2} + \frac{1}{4}\left(1 + \frac{w3}{w4}\right)\right)
\end{aligned}
$$

In terms of u, we get

$$\begin{aligned} f &= 1+\frac{u_1}{u_0}+\frac{u_2}{u_0}+\frac{2}{u_0}power\cdot u + \\ &\quad \frac{u_4}{u_3}+\frac{u_0}{u_3}+\frac{2}{u_3}power\cdot u + \\ &\quad 1+\frac{u_0}{u_4}+\frac{2}{u_4}power\cdot u + \\ &\quad \left(1+\frac{u_4}{u_1}+\frac{2}{u_1}power\cdot u\right)\max \\ &\quad \left(1+\frac{u_3}{u_2}+\frac{2}{u_2}power\cdot u+\frac{1}{4}\left(1+\frac{u_3}{u_4}\right)\right) \end{aligned}$$

since the $power \cdot u$ cancel in the terms that are ratios of widths. The terms remaining that contain $power \cdot u$ are posynomial since they are of the form

$$\frac{1}{u_i}power\cdot u = \sum_j power_j \frac{u_j}{u_i}$$

For this example, the function f has the optimum value 9.4895 and the optimum solution occurs at

$$\begin{aligned} (u_0,u_1,u_2,u_3,u_4) &= (1, 0.5224, 1.0948, 1.7396, 0.9258) \\ (w_0,w_1,w_2,w_3,w_4) &= (10.008, 5.2289, 10.957, 17.410, 9.2660) \end{aligned}$$

□

7.2.2 Minimum Transistor Widths

Reality places a minimum limit on the widths of individual transistors. Although this constraint is a minimum, it can be expressed in a convex form. We want $m_i \leq w_i$ for

all i. Since each $w_i > 0$, then

$$\begin{aligned} \frac{m_i}{w_i} &\le 1 \qquad \text{for all } i \\ \max\left\{\frac{m_i}{w_i} \,\middle|\, \text{for all } i\right\} &\le 1 . \end{aligned}$$

The constraint remains convex even if the power transformation is in effect,

$$power \cdot u \max\left\{\frac{m_i}{u_i} \,\middle|\, \text{for all } i\right\} \le 1 .$$

Example 7.3 Consider the minimum width of 6 for each transistor in Example 7.1. The minimum-width constraint equation becomes

$$6power \cdot u \max\left\{1, \frac{1}{u_1}, \frac{1}{u_2}, \frac{1}{u_3}, \frac{1}{u_4}\right\} \le 1 .$$

For this example, the function f has the optimum value 9.53294 and the optimum solution occurs at

$$\begin{aligned} (u_0, u_1, u_2, u_3, u_4) &= (1, 0.609272, 1.12969, 1.54954, 0.99198) \\ (w_0, w_1, w_2, w_3, w_4) &= (9.84782, 6, 11.125, 15.2596, 9.76884) . \end{aligned}$$

If the minimum width is 9.5 then three constraints are active. The function f has the optimum value 10.0372 and the optimum solution occurs at

$$\begin{aligned} (u_0, u_1, u_2, u_3, u_4) &= (1, 1, 1.01837, 1.37061, 1) \\ (w_0, w_1, w_2, w_3, w_4) &= (9.5, 9.5, 9.67448, 13.0208, 9.5) . \end{aligned}$$

□

7.3 Subgradient Algorithm

The performance metric is a continuous, nonlinear, nondifferentiable, convex function of the $\log w$'s. The nonsmooth nature of the function makes the optimization problem difficult and traditional techniques such as gradient following must be modified to provide solutions to this problem. The *subgradient* techniques described by Shor [36] provide adequate methods for finding optimal solutions.

7.3.1 The Subgradient

The subgradient of a convex function f of the vector x at the point x_0 can be defined as any vector $g_f(x_0)$ such that for all x

$$f(x) - f(x_0) \geq g_f(x_0) \cdot (x - x_0) .$$

At points where the gradient is continuous, the subgradient must equal the gradient. At points of discontinuity, any (one-sided) directional derivative can be used as the value for $g_f(x_0)$.

7.3.2 Basic Algorithm

The basic subgradient minimization algorithm is the following iteration:

$$x^{(k+1)} \quad = \quad x^{(k)} - h^{(k+1)}(x^{(k)})g_f(x^{(k)})$$

This differs from normal gradient descent methods because in order for the algorithm to converge the values of $h^{(k+1)}(x^{(k)})$ must tend to zero as k increases. Constant step-sizes are not suitable because the values of $g_f(x^{(k)})$ may not tend toward zero if, for example, the optimal value is at a point where the gradient is not continuous.

The step-sizes must be decreased with subsequent iterations; however, they must not be decreased too quickly, for in such a case all $x^{(k)}$ will be a bounded distance away from the starting point $x^{(0)}$. Theorem 2.2 in [36] (p. 25) states that given any sequence $h^{(k)}$ such that

$$h^{(k)} > 0, \qquad \lim_{k\to\infty} h^{(k)} = 0, \qquad \sum_{k=1}^{\infty} h^{(k)} = \infty$$

then the iteration

$$x^{(k+1)} = x^{(k)} - h^{(k+1)} \frac{g_f(x^{(k)})}{\|g_f(x^{(k)})\|} \tag{7.4}$$

converges to the minimum solution. However, this sequence can converge very slowly.

7.3.3 Convex Constraints

The subgradient algorithm can be used to minimize a convex function given additional convex constraints. Suppose that we minimize the convex function f subject to each convex function $f_0(x)$, $f_1(x)$, ..., $f_\ell(x)$ evaluating to at most 1. To find the minimum solution to this constrained problem, we substitute the subgradient used in (7.4) by any $g_f^*(x)$ satisfying

$$g_f^*(x) = \begin{cases} g_f(x) & \text{if for all } i,\ f_i(x) \le 1 \\ g_{f_i}(x) & \text{if } f_i(x) > 1 \end{cases}$$

If a constraint is not satisfied, then the above algorithm will decrease the value of the corresponding function until it is satisfied. If all the constraints are satisfied, the objective function f is minimized.

7.3.4 Heuristics and Space Dilation

The major reason for slow convergence of the (7.4) is that the direction specified by the subgradiant can be almost perpendicular to the direction toward the true minimum. When this occurs, a linear transformation on the domain of the objective function can help speed up the algorithm. We try to reduce the components of the gradient that are parallel to the previous gradient. We do this by shrinking the space in the direction of the previous gradient.

A vector x can be represented as:

$$x = \gamma_\xi(x)\xi + d_\xi(x),$$

where $\xi \cdot d_\xi = 0$, $\gamma_\xi(x) = x \cdot \xi$, and $d_\xi(x) = x - \gamma_\xi(x)\xi$. The operator of space dilation by the amount α in the direction ξ transforms a vector x into

$$R_\alpha(\xi)x = \alpha\gamma_\xi(x)\xi + d_\xi(x)\,.$$

This operator can be represented in matrix form as

$$R_\alpha(\xi) = I + (\alpha - 1)\xi\xi^T$$

and has the following identities:

$$\begin{aligned} R_{\alpha\beta}(\xi) &= R_\alpha(\xi)R_\beta(\xi) \\ R_1(\xi) &= I = R_\alpha(\xi)R_{1/\alpha}(\xi) \end{aligned}$$

These operators are used in Algorithm 7.1. This straight algorithm with space dilation is still too slow. We actually use Algorithm 7.2, which is described in [36]

1. Set $B_0 = I$, and $k = 0$.
2. Evaluate $g_f(x^{(k)})$.
3. Set $\tilde{g}^{(k)} = B_k^T g_f(x^{(k)})$.
4. Compute
   (a) $\xi^{(k+1)} = \tilde{g}^{(k)}/\|\tilde{g}^{(k)}\|$
   (b) $h^{(k+1)}$
   (c) $\alpha^{(k+1)}$
5. Update the solution vector $x^{(k+1)} = x^{(k)} - h^{(k+1)} B_k \xi^{(k+1)}$.
6. Update the transformation matrix $B_{k+1} = B_k R_{1/\alpha^{(k+1)}}(\xi^{(k+1)})$.
7. Increment k and go to step 2.

Algorithm 7.1: Subgradient method with space dilation along the gradient.

	n_{trans}	p_{unsized}	p_{sized}	CPU (sec)
Three stage pipeline control	59	189	143	42
Ten stage pipeline control	192	189	151	190
Ten stage pipeline control*	192	189	151	95
Simple microprocessor control*	285	646	430	369
* indicates results generated by the special-purpose algorithm				

Table 7.1: Performance of optimization tool.

(pp. 135–139). While this algorithm does not have the provable convergence properties of the previous algorithm, in practice, it converges to the optimal solution in fewer steps.

We have implemented Algorithm 7.2. In the current implementation, the user may interactively adjust the values of the parameters and possibly reset the space-dilation matrix to the identity. The algorithm terminates if the change in the value of the performance metric is less than a user-adjustable parameter (typically 0.0001) for several (typically 10) iterations. Tables B.2 and B.3 show the optimization equations for the lazy-active/passive buffer example. Tables B.4 and B.5 show the output of this tool, including the value of the performance metric at every fifth iteration.

Table 7.1 lists the results of this program when applied to a variety of circuits. The column n_{trans} denotes the number of transistors in the circuit, and thus the number of free variables in the optimization problem. The columns p_{unsized} and p_{sized} show the cycle period in units of τ of the circuit before and after optimization. In the unsized case, all transistors have equal sizes. The CPU column denotes the number of CPU seconds needed to compute the optimum value on a SUN/Sparcstation 1. The performance metric of the sized circuit is generally 20–30 percent faster than the unsized circuit. A direct implementation, using cycle enumeration to determine the cycle period, requires $O(n_{\text{trans}}^2 + n_{\text{cycles}} k_{\max})$ arithmetic operations per iteration, where

1. Set $B_0 = I$, $k = 0$, $\tilde{g}^{(0)} = 0$, and $\eta^{(0)} = 0$.
2. Evaluate $g_f(x^{(k)})$.
3. Set $\bar{g}^{(k)} = B_k^T g_f(x^{(k)})$.
4. Compute the difference of the two gradients, $r^{(k)} = \bar{g}^{(k)} - \tilde{g}^{(k)}$.
5. Calculate the ratio of the norms of $r^{(k)}$ and $\tilde{g}^{(k)}$; that is, $\beta^{(k)} = \|r^{(k)}\|/\|\tilde{g}^{(k)}\|$.
6. If $\beta^{(k)} \leq q_1$, then set
   (a) $h^{(k+1)} = h^{(k)}$
   (b) $B_{k+1} = B_k$
   (c) $\tilde{g}^{(k+1)} = \bar{g}^{(k)}$
   (d) $\eta^{(k+1)} = \eta^{(k)}$
7. If $\beta^{(k)} > q_1$, then set
   (a) $h^{(k+1)} = h^{(k)} q_2$
   (b) $B_{k+1} = B_k R_{1/\alpha}(r^{(k+1)}/\|r^{(k+1)}\|)$
   (c) $\tilde{g}^{(k+1)} = B_{k+1}^T g_f(x^{(k)})$
   (d) $\eta^{(k+1)} = h^{(k+1)} B_{k+1} \tilde{g}^{(k+1)}/\|\tilde{g}^{(k+1)}\|$
8. Update the solution vector $x^{(k+1)} = x^{(k)} - \eta^{(k+1)}$.
9. Increment k and go to step 2.

Algorithm 7.2: The r-algorithm, modified to decrease the number of function evaluations. Typically, the parameters are set as follows: $h = 1$, $q_1 = 0.9$, $q_2 = 0.95$, $\alpha = 1.5$.

n_{cycles} is the number of cycles used to form the cycle-period function, and where $k_{\max}$ is the maximum number of edges per cycle. A more sophisticated implementation uses the primal–dual algorithm (Algorithm 2.2) to solve the linear program for the cycle-period at each iteration, and requires only $O(n_{\text{trans}}^2)$ arithmetic operations per iteration.

Chapter 8

Summary and Concluding Remarks

In this thesis, we have defined an abstract representation of a computation that facilitates performance analysis. We have formulated several performance metrics that indicate the rate of operation of such an abstract system. We have formulated and proven several lemmas and theorems describing how closely we can approximate the complex, difficult-to-compute performance metrics, e.g., the timing simulation, by simple, easy-to-compute ones, e.g., the cycle-period. We have also described an efficient algorithm to compute the cycle-period. We have briefly described Martin's method for synthesizing quasi-delay-insensitive circuits from concurrent programs. We have shown how to transform a concurrent program, described at one of the various levels of the synthesis hierarchy, into the abstract representation used for performance analysis. We have also shown how to analyze effectively the performance of a system containing a linear array of identical processes. These techniques were used to compare a variety of implementations of a FIFO, and were used to improve a pair of existing designs. Finally, a tool for performance optimization was described. Within this tool, the analysis techniques of the previous chapters are used to construct a concise objective function that is then optimized.

We plan to incorporate the work described in this thesis into a computer-aided design (CAD) tool that unifies synthesis, analysis, and optimization. Such a tool would start with the description of computation as a concurrent program, and then perform a series of transformations guided by performance concerns until an asynchronous circuit that implements the computation has been constructed. Optimization methods would then be applied to compute the sizes of transistors within the circuit. We believe that such a tool should be interactive, allowing a circuit designer to decide at each level which alternatives to explore. With such a system, a good designer should quickly be able to generate efficient implementations of computations as asynchronous circuits.

8.1 Loose Ends

This work invites several questions that remain, as yet, unanswered.

A tighter bound in the termination proof of Algorithm 2.2, used to compute the cycle-period, is needed. Experimental evidence suggests, even for large systems designed using our method, that only a few iterations of the algorithm are required. The bound derived suggests that this number is $O(n^2)$ (but possibly larger if $\varepsilon > 1$), where n is the number of nodes in the collapsed-constraint graph. To get a very tight bound, we would have to consider the type of event-rule systems that are generated by the synthesis method because, for an arbitrary system, the algorithm can take $\Omega(n)$ iterations. We see this lower bound by generalizing Example 2.9.

The inability to model inherently disjunctive systems (Section 4.5) is not particularly satisfying.

Not all the possible degenerate cases of regular systems are categorized in Chapter 5. A more complete theory is required.

On the practical side, the most important follow-up would be to improve the

convergence properties of Algorithm 7.2 by customizing the step-size determination procedure and the termination test to the type of objective functions that are actually encountered during transistor sizing. The current algorithm makes almost no assumptions about the form of the objective function. Incorporating a smooth approximation into the tau model, as described in Section A.2, could also accelerate convergence.

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[2] Richard E. Bellman, Kenneth L. Cooke, and Jo Ann Lockett. *Algorithms, Graphs, and Computers.* Academic Press, New York, 1970.

[3] Steven M. Burns. Automated compilation of concurrent programs into self-timed circuits. M.S. thesis, California Institute of Technology, 1988. CS-TR-88-2.

[4] Steven M. Burns and Alain J. Martin. Syntax-directed translation of concurrent programs into self-timed circuits. In J. Allen and F. Leighton, editors, *Advanced Research in VLSI, Proceedings of the Fifth MIT Conference*, pages 35–50. MIT Press, Cambridge, MA, 1988.

[5] Steven M. Burns and Alain J. Martin. Synthesis of self-timed circuits by program transformation. In G.J. Milne, editor, *The Fusion of Hardward Design and Verification.* North-Holland, 1988.

[6] R. A. Cuningham-Green. Describing industrial processes with interference and approximating their steady-state behaviour. *Operational Research Quarterly*, 13(1):95–100, 1962.

[7] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* Ph.D. thesis, Carnegie Mellon University, 1988. CMU-CS-88-119.

[8] Jo C. Ebergen. *Translating Programs into Delay-Insensitive Circuits.* Ph.D. thesis, Technische Universiteit Eindhoven, 1987.

[9] W.C. Elmore. The transient response of damped linear networks with particular regard to wideband amplifiers. *Journal of Applied Physics*, 19(1):55–63, January 1948.

[10] Shimon Even. *Graph Algorithms.* Computer Science Press, Rockville, MD, 1979.

[11] J.P. Fishburn and A.E. Dunlop. TILOS: A posynomial programming approach to transistor sizing. In *IEEE ICCAD*, pages 326–328, November 1985.

[12] L.R. Ford, Jr. and D.R. Fulkerson. *Flows in Networks.* Princeton University Press, Princeton, NJ, 1962.

[13] Joel Franklin. *Methods of Mathematical Economics.* Springer-Verlag, Berlin, 1980.

[14] M. Greenstreet, T. Williams, and J. Staunstrup. Self-timed iteration. In *VLSI 1987*, pages 1–20, 1987. Draft.

[15] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufman Publishers, Inc., 1990.

[16] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[17] Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, New York, 1976.

[18] Tony Lee. Communication behavior of linear arrays of processes. M.S. thesis, California Institute of Technology, 1989. CS-TR-89-13.

[19] Tzu-Mu Lin. *A Hierarchical Timing Simulation Model For Digital Integrated Circuits and Systems.* Ph.D. thesis, California Institute of Technology, 1984. 5133:TR:84.

[20] Jan Magott. Performance evaluation of concurrent systems using Petri nets. *Information Processing Letters*, 18:7–13, 1984.

[21] David P. Marple and Abbas El Gamal. Optimal selection of transistor sizes in digital VLSI circuits. In Paul Losleben, editor, *Advanced Research in VLSI, Proceedings of the 1987 Stanford Conference*, pages 151–172. MIT Press, Cambridge, MA, 1987.

[22] A.J. Martin. The limitation to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Advanced Research in VLSI: Proceedings of the Sixth MIT Conference*, pages 263–278, Cambridge, MA, 1990. MIT Press.

[23] A.J. Martin, S.M. Burns, T.K. Lee, D. Borković, and P.J. Hazewindus. The design of an asynchronous microprocessor. In C.L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 351–373, Cambridge, MA, 1989. MIT Press.

[24] Alain J. Martin. An axiomatic definition of synchronization primitives. *Acta Informatica*, 16:219–235, 1981.

[25] Alain J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C.A.R. Hoare, editor, *UT Year of Programming Institute on Concurrent Programming.* Addison-Wesley, Reading, MA, 1990.

[26] Teresa H.-Y. Meng, Robert W. Brodersen, and David G. Messerschmitt. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 8(11):1185–1205, November 1989.

[27] David E. Muller and W.S. Bartky. A theory of asynchronous circuits. In *The Annals of the Computation Laboratory of Harvard University. Volume XXIX: Proceedings of an International Symposium on the Theory of Switching, Part I.*, pages 204–243, 1959.

[28] Fred W. Obermeier. *An Open Architecture for Improving VLSI Circuit Performance.* Ph.D. thesis, UC Berkeley, 1989.

[29] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.

[30] P. Penfield and J. Rubinstein. Signal delay in RC tree networks. In Charles L. Seitz, editor, *Proceedings of the Second Caltech Conference on VLSI*, pages 269–283. Caltech Computer Science Department, January 1981.

[31] C.V. Ramamoorthy and Gary S. Ho. Performance evaluation of asynchronous concurrent systems using Petri nets. *IEEE Transactions on Software Engineering*, 6(5):440–449, September 1980.

[32] Edward M. Reingold, Jurg Nievergelt, and Narsingh Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1977.

[33] Raymond Reiter. Scheduling parallel computations. *Journal of the ACM*, 15(4):590–599, October 1968.

[34] Martin Rem. Trace theory and systolic computations. Computer Science 5239:TR:87, California Institute of Technology, 1987.

[35] Charles L. Seitz. System timing. In Carver Mead and Lynn Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, Reading, MA, 1980.

[36] N.Z. Shor. *Minimization Methods for Non-Differentiable Functions.* Springer-Verlag, Berlin, 1985. Translated from Russian.

[37] Ivan E. Sutherland. A theory of logical effort (revised version). Technical Report SSA 4679, Sutherland, Sproull and Associates, Inc., September 1986.

[38] Ted Williams, Mark Horowitz, R.L. Alverson, and T.S. Yang. A self-timed chip for division. In Paul Losleben, editor, *Advanced Research in VLSI, Proceedings of the 1987 Stanford Conference*, pages 75–95. MIT Press, Cambridge, MA, 1987.

Appendix A

Accuracy of Tau Model

We provide a short analysis of the accuracy of the approximations, (7.1) and (7.2), to the actual propagation delay through the RC circuit shown in Figure 7.1. It is beyond the scope of this thesis to argue the accuracy of the tau model at describing delays of real transistors. To ease the analysis, we change the time units so that one unit corresponds to $(R_1 + R_2)C_2$, and use the dimensionless variables

$$r = \frac{R_1}{R_1 + R_2}, \qquad c = \frac{C_1}{C_2}$$

Figure A.1 shows the RC circuit with these new names for the resistances and capacitances. Typically, $r < 0.5$ and $c < 0.3$.

We now derive the time evolution of the voltages on the internal and external nodes of this circuit given various configurations of the two switches. Initially, the

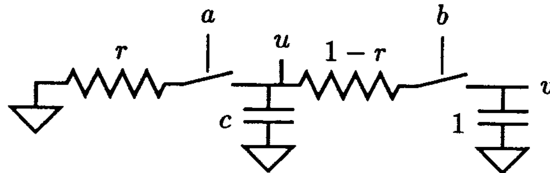


Figure A.1: RC pulldown circuit with dimensionless variables r and c.

internal node is at voltage u and the external node is at voltage v.

Switch a Tied, Switch b Cut If switch a is tied and switch b is cut, then the voltage on the internal node decays as a simple exponential.

$$u(t) = u \exp\left(-\frac{t}{rc}\right)$$

The voltage on the external node remains constant.

Switch b Tied, Switch a Cut If switch b is tied and switch a is cut, then charge sharing occurs between the internal and external nodes.

$$\begin{aligned}
s_0 &= -\frac{1+c}{c(1-r)} \\
s_1 &= 0 \\
u(t) &= \frac{u-v}{1+c}e^{s_0 t} + \frac{cu+v}{1+c} \\
v(t) &= \frac{cv-cu}{1+c}e^{s_0 t} + \frac{cu+v}{1+c}
\end{aligned}$$

Notice that if $u = v$, that is if the two nodes are at the same voltage, the voltage on both nodes remains constant. The scenario cannot reduce the voltage on the external node to half of v unless $u < 0.5$ and $c - 1/c \geq 2u$, which represents an unreasonably large (greater than unity) value for c.

Both Switches Tied If both switches are tied, the voltages on both nodes will decay toward zero.

$$s_0 = -\frac{1}{\tau_0} = \frac{2}{-1 - cr + \sqrt{(1-cr)^2 + 4cr^2}} \tag{A.1}$$

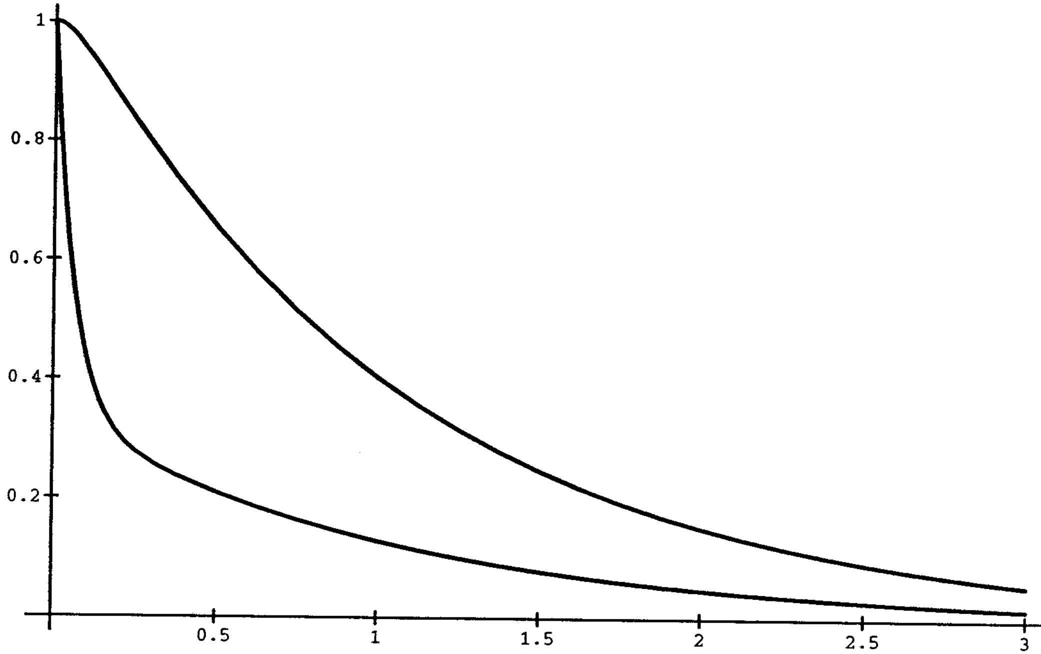


Figure A.2: Time evolution of the voltages $u(t)$ and $v(t)$ for the case of both switches tied. The initial values u and v are both set to 1. The resistance and capacitance ratios are both set to 0.3.

$$s_1 = -\frac{1}{\tau_1} = \frac{2}{-1 - cr - \sqrt{(1-cr)^2 + 4cr^2}} \tag{A.2}$$

$$u(t) = \frac{(u - rv) - u\tau_0}{\tau_1 - \tau_0} e^{s_0 t} + \frac{u\tau_1 - (u - rv)}{\tau_1 - \tau_0} e^{s_1 t} \tag{A.3}$$

$$v(t) = \frac{(v - u)cr - v\tau_0}{\tau_1 - \tau_0} e^{s_0 t} + \frac{v\tau_1 - (v - u)cr}{\tau_1 - \tau_0} e^{s_1 t} \tag{A.4}$$

The time needed to lower the voltage on the external node to one half its initial value is named the 50-percent propagation delay.

$$v(t_{50\%}) = \frac{1}{2} v \tag{A.5}$$

The α values in (7.1) and (7.2) are meant to approximate this delay. Because of the

two time constants in $(A.4)$, no closed form solution exists for $t_{50\%}$.

Actual 50% Delay The following table shows the propagation delay for various values of r and c.

$u = 1$

$t_{50\%}$	$c = 0.1$	$c = 0.3$	$c = 0.5$	$c = 0.7$	$c = 0.9$
$r = 0.1$	0.703	0.723	0.743	0.763	0.784
$r = 0.3$	0.721	0.776	0.833	0.89	0.947
$r = 0.5$	0.736	0.821	0.906	0.989	1.072
$r = 0.7$	0.748	0.857	0.964	1.070	1.174
$r = 0.9$	0.758	0.887	1.015	1.142	1.269

These values were computed by numerically solving $(A.4)$ and $(A.5)$ for $t_{50\%}$. In the limiting cases, the system degenerates to a single time constant and the propagation delay can be solved in closed form.

$$
\begin{aligned}
t_{50\%} &= \ln 2, \qquad \text{if } r = 0 \vee c = 0 \\
&= (1+c)\ln 2, \qquad \text{if } r = 1
\end{aligned}
$$

Modeled 50% Delay The tau model approximation is derived from the Elmore delay [9, 30, 19] of $v(t)$ with $v = u = 1$:

$$
\begin{aligned}
t_{\text{Elmore}} &= -\int_0^\infty t v'(t) dt \\
&= -\int_0^\infty t \left[s_0 k_0 e^{s_0 t} + s_1 k_1 e^{s_1 t} \right] dt \\
&= -\left[k_0 (t - \frac{1}{s_0}) e^{s_0 t} + k_1 (t - \frac{1}{s_1}) e^{s_1 t} \right]_0^\infty
\end{aligned}
$$

$$
\begin{aligned}
&= -\left[\frac{k_0}{s_0} + \frac{k_1}{s_1}\right] = k_0\tau_0 + k_1\tau_1 \\
&= -\frac{{\tau_0}^2}{\tau_1 - \tau_0} + \frac{{\tau_1}^2}{\tau_1 - \tau_0} \\
&= \tau_1 + \tau_0 = 1 + cr
\end{aligned}
$$

The voltage $v(t)$ is falling from 1 to 0 and thus $v'(t) \leq 0$. The Elmore delay is the centroid of $v(t)$ and represents a single time-constant approximation to the system. This approximation is exact in the limiting cases above.

$$
\begin{aligned}
t_\tau &= (1 + rc)\ln 2 \\
&= t_{50\%} \qquad \text{if } r = 0 \vee c = 0 \vee r = 1
\end{aligned}
$$

Percent Relative Error We define the percent relative error of the approximation by

$$
\%err_{\text{rel}} = 100\frac{t_\tau - t_{50\%}}{t_{50\%}}
$$

The following table shows the errors for various values of r and c.

$u = 1$

$\%err_{\text{rel}}$	$c = 0.1$	$c = 0.3$	$c = 0.5$	$c = 0.7$	$c = 0.9$
$r = 0.1$	−0.4	−1.2	−2.0	−2.8	−3.8
$r = 0.3$	−0.9	−2.7	−4.3	−5.8	−7.0
$r = 0.5$	−1.1	−2.9	−4.3	−5.4	−6.2
$r = 0.7$	−0.8	−2.1	−2.9	−3.4	−3.8
$r = 0.9$	−0.3	−0.8	−1.0	−1.1	−1.2

The approximation is quite good, with no more than 7% error over a wide range of parameter values. Over the range of typical values, $r \leq 0.5$ and $c \leq 0.3$, the errors are all less than 3%.

Internal Node Not Fully Charged The voltage u is state information and cannot, in general, be determined statically from the circuit description. The tau model assumes the worst-case initial voltage for the internal node. Large overestimations of delay occur if the initial voltage on the internal node is zero. The inability to model the effect is one of the key contributors to the error of the tau model.

$u = 0$

$t_{50\%}$	$c = 0.1$	$c = 0.3$	$c = 0.5$	$c = 0.7$	$c = 0.9$
$r = 0.1$	0.693	0.692	0.691	0.690	0.689
$r = 0.3$	0.690	0.682	0.671	0.658	0.644
$r = 0.5$	0.684	0.659	0.625	0.585	0.547
$r = 0.7$	0.676	0.625	0.555	0.474	0.403
$r = 0.9$	0.664	0.583	0.474	0.341	0.204

From this table, we see an error of 25% if $r = 0.5$ and $c = 0.3$ and an error of 11% if $r = 0.3$ and $c = 0.3$. The error is less pronounced if $u = 0.5$, being less than 7% over

the range of typical values.

$u = 0.5$

$t_{50\%}$	$c = 0.1$	$c = 0.3$	$c = 0.5$	$c = 0.7$	$c = 0.9$
$r = 0.1$	**0.698**	**0.707**	**0.717**	0.727	0.737
$r = 0.3$	**0.705**	**0.730**	**0.755**	0.780	0.803
$r = 0.5$	**0.710**	**0.743**	**0.773**	0.801	0.826
$r = 0.7$	**0.712**	**0.747**	**0.776**	0.800	0.821
$r = 0.9$	0.712	0.744	0.770	0.791	0.809

To compensate for this error, we can compute the Elmore delay of $v(t)$, with u set to an intermediate value.

$$\begin{aligned} t_{\text{Elmore}} &= k_0\tau_0 + k_1\tau_1 \\ &= \frac{(1-u)cr - \tau_0}{\tau_1 - \tau_0}\tau_0 + \frac{\tau_1 - (1-u)cr}{\tau_1 - \tau_0}\tau_1 \\ &= \tau_0 + \tau_1 - (1-u)cr \;=\; 1 + cru \end{aligned}$$

Using this estimate, we see an error of less than 5% over the range of typical values. The error, however, can be quite large for large r and c. However, this technique is rarely practical, since it is difficult to determine without simulation what initial value u to choose.

Figures B.13 through B.16 show the SPICE-generated waveforms of selected nodes of a circuit implementing the lazy-active/passive buffer. All but the first and the fifth waveforms correspond to internal nodes. Notice that the voltage on some internal nodes is constant while on others it varies almost from rail to rail. We do not consider it practical to use this information during optimization and choose a worst-case initial voltage for the internal nodes. However, this example does show a major source of

error.

A.1 Tied Transistors in the Pull-up Chain

One source of capacitance is ignored by the delay model as it stands. The capacitance of internal nodes in the pull-up chain of an element can influence (make longer) the delay of the down-going transitions of the element. (The same argument holds for pull-down chains and up-going transitions.) This is because there can be a conducting chain of transistors between the output node to the internal node. The capacitance on the nodes of this conducting chain must be charged as well.

If a circuit variable x is connected to a gate in the pull-up chain and to a gate in the pull-down chain, then the capacitance of the internal nodes beyond x (toward the power rail) can never contribute to the delay of the down-going transition. This is because in order for the down-going transition to occur, all the n-channel transistors in the pull-down chain must be tied and thus the p-channel transistor in the pull-up chain with gate x must be cut (by non-interference). While this fact does not say for sure whether the capacitance of an internal node in a pull-up chain influences a down-going delay, it does say when such is possible, and thus provides an upper bound on the delay. A more accurate determination requires simulation.

Compare the α values determined by SPICE simulation as shown in Figures B.17 and B.18 with those shown in Figures B.19 and B.20. The results of the simulations described by the second set of figures show a better fit between simulated and modeled delay values.

A.2 Maximum Approximation

Another inaccuracy occurs when the inputs switch almost simultaneously. The time between the inputs $t(a\uparrow)$ and $t(b\uparrow)$ and the output $t(z\downarrow)$ is characterized using the tau-model as

$$t(z\downarrow) \;=\; \max(t(a\uparrow) + \alpha_{a\uparrow z\downarrow}, t(b\uparrow) + \alpha_{b\uparrow z\downarrow})\,.$$

Assuming that $t(a\uparrow) + \delta = t(b\uparrow)$ and substituting for each α in the dimensionless model, we get

$$\begin{aligned}
t(z\downarrow) &= t(b\uparrow) + \max(\alpha_{a\uparrow z\downarrow} - \delta, \alpha_{b\uparrow z\downarrow}) \\
t(z\downarrow) - t(b\uparrow) &= \max((R_1C_1 + (R_1 + R_2)C_2)\ln 2 - \delta, (R_1 + R_2)C_2 \ln 2) \\
&= \max((rc + 1)\ln 2 - \delta, \ln 2) \\
&= \ln 2 + \max(rc \ln 2 - \delta, 0)\,.
\end{aligned}$$

For $\delta \geq 0$, the value of $t_{50\%}$ is determined from $(A.4)$ after first setting the initial condition of the internal variable to

$$u \;=\; \exp\left(-\frac{\delta}{rc}\right) \tag{A.6}$$

to account for the charge pulled off this node while switch a is tied and switch b is cut. Figure A.3 compares the actual $t_{50\%}$ values—generated by numerically solving $(A.4)$—with the tau-model estimates.

A more accurate (and also smooth) approximation can be constructed from the

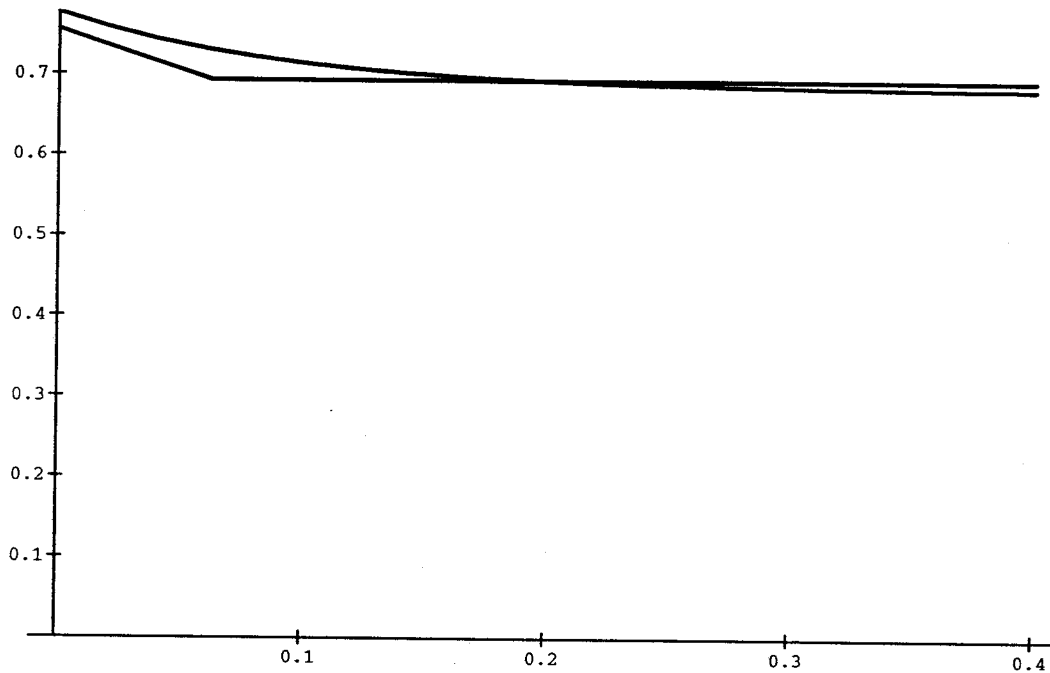


Figure A.3: Plot comparing $t_{50\%}$ and $t(z\downarrow) - t(b\uparrow)$ derived using the tau-model. Units on both axes are multiples of the fundamental time $(R_1+R_2)C_2$. The functions are plotted versus δ, the time after $a\uparrow$ fires that $b\uparrow$ fires. That is $\delta = t(b\uparrow) - t(a\uparrow)$. The parameters r and c are 0.3 and 0.3, respectively. An underestimation of the delay by 7% is observed at $t = rc$.

Elmore delay of $v(t)$ with u set at some intermediate value. Using $(A.6)$ for u, we get

$$t(z\downarrow) - t(b\uparrow) = \begin{cases} (1+rc)\ln 2 - \delta & \text{if } \delta \leq 0 \\ \left(1 + rc\exp\left(-\frac{\delta}{rc}\right)\right)\ln 2 & \text{if } \delta \geq 0\,. \end{cases}$$

The derivatives of both pieces match at $\delta = 0$.

Example A.1 Using this smooth approximation, the optimization equation from Example 7.1 becomes

$$\begin{aligned} \delta &= \frac{w_4}{w_1} - \frac{w_3}{w_2} \\ \tau &= \frac{1}{4}\left(1 + \frac{w_4}{w_3}\right) \\ f &= 3 + w_1 + w_2 + \frac{w_4}{w_3} + \frac{1}{w_3} + \frac{1}{w_4} + \frac{w_4}{w_1} + \\ &\quad \begin{cases} \tau\exp(-\delta/\tau) & \text{if } \delta \geq 0 \\ \tau - \delta & \text{if } \delta \leq 0 \end{cases}. \end{aligned}$$

This expression can be minimized by traditional gradient-following techniques. The minimum value is 8.806 and is achieved at:

$$(w_1, w_2, w_3, w_4) = (0.5136, 1.1422, 1.8478, 0.8972)$$

Figures A.4 through A.7 show the respective cross sections at the optimal point. □

This approximation is quite accurate although it is significantly more complicated than the maximum approximation. However, it is potentially simpler to optimize such a smooth function because there are no discontinuities in the gradient.

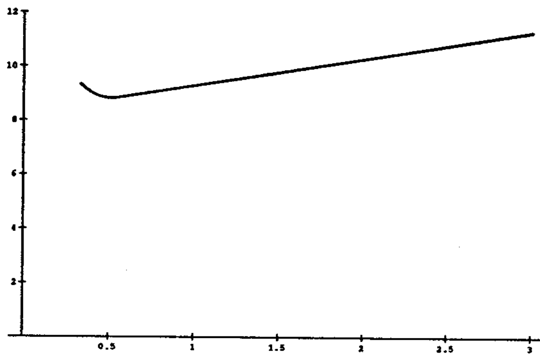


Figure A.4: Cross section of smooth f with respect to w_1.

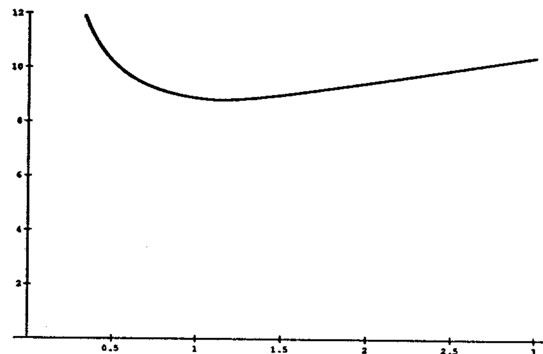


Figure A.5: Cross section of smooth f with respect to w_2.

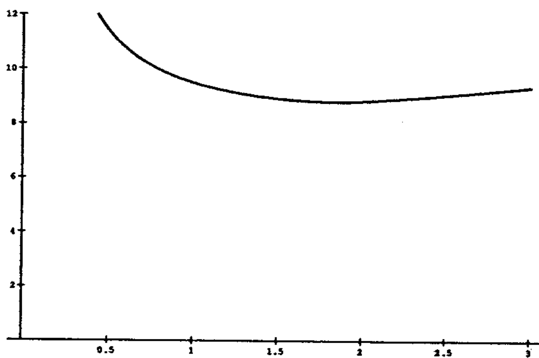


Figure A.6: Cross section of smooth f with respect to w_3.

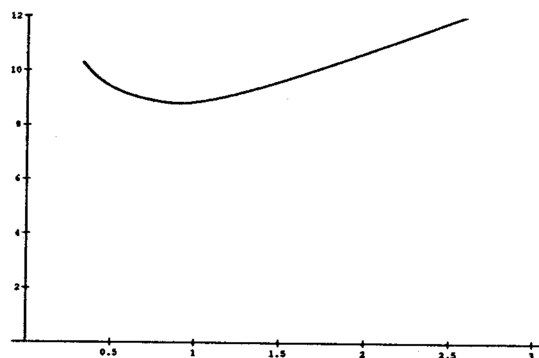


Figure A.7: Cross section of smooth f with respect to w_4.

A.3 Transistor Models

As stated earlier, it is beyond the scope of this thesis to argue the accuracy of the tau model at describing delays of real transistors. Instead, we will refer only to some examples in the following appendix. Figures B.9 and B.10 show the waveforms provided by a SPICE simulation comparing an implementation of the lazy-active/passive buffer with optimized and non-optimized transistor sizes. Figures B.13 and B.14 show the waveforms of the internal nodes in these cases. Table B.1 and Figure B.17 compare the predicted delay with the SPICE simulations. In Table B.3 and Figure B.19, the comparison is redone with the tied model. In Figures B.21 and B.22, the waveforms of a larger circuit are compared.

Appendix B

Detailed Example

In the appendix, we perform a complete performance analysis and transistor optimization of a circuit implementing the lazy-active/passive buffer.

```
process active {

        x.i -> x_.o -
        ~ x_.o -> x.o +
        ~ x.i -> x_.o +
        x_.o -> x.o -

}

process passive {

        ~ x.i [-1] -> x.o +
        x.i -> x.o -

}

# top level production rules

        ~ l.i [-1] & ~ x [-1] & ~ r.o [-1] -> _l.o +
        g & _l.o -> l_.o -
        ~ l_.o -> x +
        ~ l_.o -> l.o +

        x & l.i -> _l.o -
        ~ _l.o -> l_.o +
        l_.o -> l.o -

        r.i & x & l_.o -> r_.o -
        ~ r_.o -> r.o +
        r.o -> x -
        ~ x & ~ r.i -> r_.o +
        r_.o -> r.o -

instance passive p

# instance active a
# connect l.o a/x.i
# connect l.i a/x.o

# instead of making an instance of active,
#       connect the wires of the left channel

connect l.o l.i

connect r.o p/x.i
connect r.i p/x.o
```

Figure B.1: Input representation of a single lazy-active/passive buffer.

```
"$period"
with
        "$power" max "$widths" <= 1
where
        "$l(l.o)" = "$w(l.o+ _l.o-)" + "$w(l.o- _l.o+)" + ( "$w(l_.o+ l.o-)" + "$w(l_.o- l.o+)" ) * 1 + 4
        "$r(l.o-)" = "$l(l.o)" * "$w(l_.o+ l.o-)" ^ -1
        "$a(l_.o+ l.o-)" = "$r(l.o-)"
        "$l(_l.o)" = "$w(_l.o+ l_.o-)" + "$w(_l.o- l_.o+)" + ( "$w(r.o- _l.o+)" + "$w(l.o+ _l.o-)" ) * 1 + 4
        "$r(_l.o+)" = "$l(_l.o)" * ( "$w(l.o- _l.o+)" ^ -1 + "$w(x- _l.o+)" ^ -1 + "$w(r.o- _l.o+)" ^ -1 ) * 2.5
        "$a(l.o- _l.o+)" = "$a(r.o- _l.o+)" + ( ( "$w(l.o- _l.o+)" ^ -1 + "$w(x- _l.o+)" ^ -1 ) * ( "$w(x- _l.o+)" + "$w(r.o-
_l.o+)" ) + "$w(l.o- _l.o+)" ^ -1 * ( "$w(l.o- _l.o+)" + "$w(x- _l.o+)" ) ) * 1.25
        "$a(x- _l.o+)" = "$a(r.o- _l.o+)" + "$w(x- _l.o+)" ^ -1 * ( "$w(x- _l.o+)" + "$w(r.o- _l.o+)" ) * 1.25
        "$a(r.o- _l.o+)" = "$r(_l.o+)"
        "$l(x)" = "$w(x+ _l.o-)" + "$w(x+ r_.o-)" + "$w(x- _l.o+)" + "$w(x- r_.o+)" + ( "$w(r.o+ x-)" + "$w(l_.o- x+)" ) * 1 + 4
        "$r(x-)" = "$l(x)" * "$w(r.o+ x-)" ^ -1
        "$a(r.o+ x-)" = "$r(x-)"
        "$l(r.o)" = "$w(r.o+ x-)" + "$w(r.o+ r.i-)" + "$w(r.o- _l.o+)" + "$w(r.o- r.i+)" + ( "$w(r_.o+ r.o-)" + "$w(r_.o-
r.o+)" ) * 1 + 4
        "$r(r.o-)" = "$l(r.o)" * "$w(r_.o+ r.o-)" ^ -1
        "$a(r_.o+ r.o-)" = "$r(r.o-)"
        "$l(l_.o)" = "$w(l_.o+ l.o-)" + "$w(l_.o+ r_.o-)" + "$w(l_.o- x+)" + "$w(l_.o- l.o+)" + ( "$w(_l.o+ l_.o-)" + "$w(_l.o-
l_.o+)" ) * 1 + 4
        "$r(l_.o-)" = "$l(l_.o)" * ( "$w(g+ l_.o-)" ^ -1 + "$w(_l.o+ l_.o-)" ^ -1 )
        "$a(_l.o+ l_.o-)" = "$r(l_.o-)"
        "$r(x+)" = "$l(x)" * "$w(l_.o- x+)" ^ -1 * 2.5
        "$a(l_.o- x+)" = "$r(x+)"
        "$r(l.o+)" = "$l(l.o)" * "$w(l_.o- l.o+)" ^ -1 * 2.5
        "$a(l_.o- l.o+)" = "$r(l.o+)"
        "$r(_l.o-)" = "$l(_l.o)" * ( "$w(x+ _l.o-)" ^ -1 + "$w(l.o+ _l.o-)" ^ -1 )
        "$a(x+ _l.o-)" = "$a(l.o+ _l.o-)" + "$w(x+ _l.o-)" ^ -1 * ( "$w(x+ _l.o-)" + "$w(l.o+ _l.o-)" ) * 0.5
        "$a(l.o+ _l.o-)" = "$r(_l.o-)"
        "$r(l_.o+)" = "$l(l_.o)" * "$w(_l.o- l_.o+)" ^ -1 * 2.5
        "$a(_l.o- l_.o+)" = "$r(l_.o+)"
        "$l(r.i)" = "$w(r.i+ r_.o-)" + "$w(r.i- r_.o+)" + ( "$w(r.o- r.i+)" + "$w(r.o+ r.i-)" ) * 1 + 4
        "$r(r.i+)" = "$l(r.i)" * "$w(r.o- r.i+)" ^ -1 * 2.5
        "$a(r.o- r.i+)" = "$r(r.i+)"
        "$l(r_.o)" = "$w(r_.o+ r.o-)" + "$w(r_.o- r.o+)" + ( "$w(l_.o+ r_.o-)" + "$w(r.i- r_.o+)" ) * 1 + 4
        "$r(r_.o-)" = "$l(r_.o)" * ( "$w(r.i+ r_.o-)" ^ -1 + "$w(x+ r_.o-)" ^ -1 + "$w(l_.o+ r_.o-)" ^ -1 )
        "$a(r.i+ r_.o-)" = "$a(l_.o+ r_.o-)" + ( ( "$w(r.i+ r_.o-)" ^ -1 + "$w(x+ r_.o-)" ^ -1 ) * ( "$w(x+ r_.o-)" + "$w(l_.o+ r_.o-
)" ) + "$w(r.i+ r_.o-)" ^ -1 * ( "$w(r.i+ r_.o-)" + "$w(x+ r_.o-)" ) ) * 0.5
        "$a(x+ r_.o-)" = "$a(l_.o+ r_.o-)" + "$w(x+ r_.o-)" ^ -1 * ( "$w(x+ r_.o-)" + "$w(l_.o+ r_.o-)" ) * 0.5
        "$a(l_.o+ r_.o-)" = "$r(r_.o-)"
        "$r(r.o+)" = "$l(r.o)" * "$w(r_.o- r.o+)" ^ -1 * 2.5
        "$a(r_.o- r.o+)" = "$r(r.o+)"
        "$r(r_.o+)" = "$l(r_.o)" * ( "$w(x- r_.o+)" ^ -1 + "$w(r.i- r_.o+)" ^ -1 ) * 2.5
        "$a(x- r_.o+)" = "$a(r.i- r_.o+)" + "$w(x- r_.o+)" ^ -1 * ( "$w(x- r_.o+)" + "$w(r.i- r_.o+)" ) * 1.25
        "$a(r.i- r_.o+)" = "$r(r_.o+)"
        "$r(r.i-)" = "$l(r.i)" * "$w(r.o+ r.i-)" ^ -1
        "$a(r.o+ r.i-)" = "$r(r.i-)"
```

Figure B.2: Optimization equations for the lazy-active/passive buffer.

```
        "$power" = ( "$w(1.o- _1.o+)" + "$w(x- _1.o+)" + "$w(r.o- _1.o+)" + "$w(g+ 1_.o-)" + "$w(_1.o+ 1_.o-)" + "$w(1_.o-
x+)" + "$w(1_.o- 1.o+)" + "$w(x+ _1.o-)" + "$w(1.o+ _1.o-)" + "$w(_1.o- 1_.o+)" + "$w(1_.o+ 1.o-)" + "$w(r.i+ r_.o-)" + "$w(x+ r_.o-
)" + "$w(1_.o+ r_.o-)" + "$w(r_.o- r.o+)" + "$w(r.o+ x-)" + "$w(x- r_.o+)" + "$w(r.i- r_.o+)" + "$w(r_.o+ r.o-)" + "$w(r.o-
r.i+)" + "$w(r.o+ r.i-)" ) * 0.00238095
        "$widths" = ( "$w(1_.o+ r_.o-)" ^ -1 max "$w(x+ _1.o-)" ^ -1 max "$w(1_.o- x+)" ^ -1 max "$w(r.i- r_.o+)" ^ -1 max "$w(1_.o-
1.o+)" ^ -1 max "$w(r_.o+ r.o-)" ^ -1 max "$w(1.o- _1.o+)" ^ -1 max "$w(r.o- _1.o+)" ^ -1 max "$w(r_.o- r.o+)" ^ -1 max "$w(r.i+ r_.o-
)" ^ -1 max "$w(g+ 1_.o-)" ^ -1 max "$w(r.o+ x-)" ^ -1 max "$w(_1.o+ 1_.o-)" ^ -1 max "$w(x- r_.o+)" ^ -1 max "$w(x- _1.o+)" ^ -
1 max "$w(_1.o- 1_.o+)" ^ -1 max "$w(x+ r_.o-)" ^ -1 max "$w(r.o+ r.i-)" ^ -1 max "$w(r.o- r.i+)" ^ -1 max "$w(1.o+ _1.o-)" ^ -
1 max "$w(1_.o+ 1.o-)" ^ -1 ) * 4
        "$c0" = ( ( "$a(r_.o- r.o+)" + "$a(r.o+ r.i-)" + "$a(r.i- r_.o+)" max "$a(r_.o- r.o+)" + "$a(r.o+ x-)" + "$a(x-
r_.o+)" ) + "$a(r_.o+ r.o-)" + "$a(r.o- _1.o+)" max "$a(r_.o- r.o+)" + "$a(r.o+ x-)" + "$a(x- _1.o+)" ) + ( ( "$a(_1.o+ 1_.o-
)" + "$a(1_.o- 1.o+)" + "$a(1.o+ _1.o-)" max "$a(_1.o+ 1_.o-)" + "$a(1_.o- x+)" + "$a(x+ _1.o-)" ) + "$a(_1.o- 1_.o+)" + "$a(1_.o+ r_.o-
)" max "$a(_1.o+ 1_.o-)" + "$a(1_.o- x+)" + "$a(x+ r_.o-)" )
        "$c1" = ( "$a(_1.o+ 1_.o-)" + "$a(1_.o- 1.o+)" + "$a(1.o+ _1.o-)" max "$a(_1.o+ 1_.o-)" + "$a(1_.o- x+)" + "$a(x+ _1.o-
)" ) + ( "$a(_1.o- 1_.o+)" + "$a(1_.o+ 1.o-)" ) + "$a(1.o- _1.o+)"
        "$c2" = ( "$a(r_.o- r.o+)" + "$a(r.o+ r.i-)" + "$a(r.i- r_.o+)" max "$a(r_.o- r.o+)" + "$a(r.o+ x-)" + "$a(x-
r_.o+)" ) + ( "$a(r_.o+ r.o-)" + "$a(r.o- r.i+)" ) + "$a(r.i+ r_.o-)"
        "$period" = "$c0" max "$c1" max "$c2"
initially
        "$w(1_.o+ r_.o-)" = 15.665
        "$w(x+ _1.o-)" = 15.665
        "$w(1_.o- x+)" = 24.7685
        "$w(r.i- r_.o+)" = 24.7685
        "$w(1_.o- 1.o+)" = 24.7685
        "$w(r_.o+ r.o-)" = 15.665
        "$w(1.o- _1.o+)" = 24.7685
        "$w(r.o- _1.o+)" = 24.7685
        "$w(r_.o- r.o+)" = 24.7685
        "$w(r.i+ r_.o-)" = 15.665
        "$w(g+ 1_.o-)" = 15.665
        "$w(r.o+ x-)" = 15.665
        "$w(_1.o+ 1_.o-)" = 15.665
        "$w(x- r_.o+)" = 24.7685
        "$w(x- _1.o+)" = 24.7685
        "$w(_1.o- 1_.o+)" = 24.7685
        "$w(x+ r_.o-)" = 15.665
        "$w(r.o+ r.i-)" = 15.665
        "$w(r.o- r.i+)" = 24.7685
        "$w(1.o+ _1.o-)" = 15.665
        "$w(1_.o+ 1.o-)" = 15.665
```

Figure B.3: Optimization equations for the lazy-active/passive buffer (cont.).

```
iter: 0 143.355 h:1 alpha:1.5 zero:0/21
iter: 5 125.288 h:.903688 alpha:1.5 zero:0/21
iter: 10 120.874 h:.796236 alpha:1.5 zero:0/21
iter: 15 116.99 h:.70156 alpha:1.5 zero:0/21
iter: 20 116.119 h:.618141 alpha:1.5 zero:0/21
iter: 25 116.042 h:.544642 alpha:1.5 zero:20/21
iter: 30 115.244 h:.479881 alpha:1.5 zero:0/21
iter: 35 115.415 h:.422821 alpha:1.5 zero:0/21
iter: 40 115.168 h:.372546 alpha:1.5 zero:0/21
iter: 45 114.81 h:.328249 alpha:1.5 zero:0/21
iter: 50 114.726 h:.289219 alpha:1.5 zero:0/21
iter: 55 114.631 h:.254829 alpha:1.5 zero:0/21
iter: 60 114.588 h:.224529 alpha:1.5 zero:0/21
iter: 65 114.524 h:.197831 alpha:1.5 zero:0/21
iter: 70 114.518 h:.174308 alpha:1.5 zero:0/21
iter: 75 114.514 h:.153582 alpha:1.5 zero:0/21
iter: 80 114.498 h:.135321 alpha:1.5 zero:20/21
iter: 85 114.508 h:.119231 alpha:1.5 zero:0/21
iter: 90 114.495 h:.105054 alpha:1.5 zero:0/21
iter: 95 114.492 h:0.0925622 alpha:1.5 zero:0/21
iter: 100 114.49 h:0.0815562 alpha:1.5 zero:0/21
iter: 105 114.488 h:0.0718588 alpha:1.5 zero:0/21
iter: 110 114.488 h:0.0633145 alpha:1.5 zero:0/21
iter: 115 114.487 h:0.0557861 alpha:1.5 zero:0/21
iter: 120 114.486 h:0.0491529 alpha:1.5 zero:0/21
iter: 125 114.485 h:0.0433084 alpha:1.5 zero:0/21
iter: 130 114.485 h:0.0381589 alpha:1.5 zero:20/21
iter: 135 114.485 h:0.0344837 alpha:1.5 zero:20/21
iter: 140 114.485 h:0.0303834 alpha:1.5 zero:0/21
iter: 145 114.485 h:0.0267707 alpha:1.5 zero:0/21
iter: 150 114.485 h:0.0235876 alpha:1.5 zero:20/21
iter: 155 114.485 h:0.0207829 alpha:1.5 zero:0/21
iter: 160 114.484 h:0.0183117 alpha:1.5 zero:0/21
iter: 165 114.484 h:0.0161344 alpha:1.5 zero:20/21
iter: 170 114.484 h:0.0142159 alpha:1.5 zero:20/21
iter: 175 114.484 h:0.0125256 alpha:1.5 zero:0/21
iter: 180 114.484 h:0.0110363 alpha:1.5 zero:0/21
iter: 185 114.484 h:0.00972399 alpha:1.5 zero:0/21
iter: 190 114.484 h:0.00856777 alpha:1.5 zero:20/21
iter: 195 114.484 h:0.00754902 alpha:1.5 zero:0/21
iter: 200 114.484 h:0.00665141 alpha:1.5 zero:0/21
iter: 205 114.484 h:0.00586053 alpha:1.5 zero:20/21
iter: 210 114.484 h:0.00516369 alpha:1.5 zero:0/21
iter: 215 114.484 h:0.0045497 alpha:1.5 zero:0/21
iter: 220 114.484 h:0.00400872 alpha:1.5 zero:0/21
iter: 225 114.484 h:0.00353207 alpha:1.5 zero:0/21
iter: 230 114.484 h:0.00311209 alpha:1.5 zero:20/21
iter: 235 114.484 h:0.00281236 alpha:1.5 zero:0/21
iter: 240 114.484 h:0.00247796 alpha:1.5 zero:0/21
iter: 245 114.484 h:0.00218332 alpha:1.5 zero:0/21
iter: 250 114.484 h:0.00192371 alpha:1.5 zero:0/21
iter: 255 114.484 h:0.00169497 alpha:1.5 zero:20/21
iter: 260 114.484 h:0.00149343 alpha:1.5 zero:0/21
iter: 264 114.484

minimum 114.484 constraint 1
variables:
        "$w(l.o- _l.o+)" = 41.9465
        "$w(r.o- _l.o+)" = 13.4876
        "$w(x- r_.o+)" = 34.4755
        "$w(x- _l.o+)" = 31.7336
        "$w(x+ r_.o-)" = 20.5728
        "$w(r.o- r.i+)" = 4
```

Figure B.4: Output of optimization tool for the lazy-active/passive buffer.

```
        "$w(l_.o+ r_.o-)" = 9.0725
        "$w(l_.o- x+)" = 26.9904
        "$w(l_.o- l.o+)" = 11.6961
        "$w(r.i+ r_.o-)" = 33.7778
        "$w(_l.o+ l_.o-)" = 10.4381
        "$w(x+ _l.o-)" = 20.727
        "$w(r_.o+ r.o-)" = 10.3686
        "$w(g+ l_.o-)" = 62.3914
        "$w(_l.o- l_.o+)" = 16.5043
        "$w(r.i- r_.o+)" = 17.0989
        "$w(r_.o- r.o+)" = 16.3941
        "$w(r.o+ x-)" = 18.3244
        "$w(r.o+ r.i-)" = 6.24478
        "$w(l.o+ _l.o-)" = 9.75601
        "$w(l_.o+ l.o-)" = 4
definitions:
        "$l(r.o)" = 72.8194
        "$r(r.o+)" = 11.1045
        "$a(r_.o- r.o+)" = 11.1045
        "$l(r.i)" = 65.1215
        "$r(r.i-)" = 10.4282
        "$a(r.o+ r.i-)" = 10.4282
        "$l(r_.o)" = 56.9341
        "$r(r_.o+)" = 12.4528
        "$a(r.i- r_.o+)" = 12.4528
        "$l(x)" = 156.824
        "$r(x-)" = 8.55819
        "$a(r.o+ x-)" = 8.55819
        "$a(x- r_.o+)" = 14.3228
        "$r(r.o-)" = 7.0231
        "$a(r_.o+ r.o-)" = 7.0231
        "$l(_l.o)" = 54.186
        "$r(_l.o+)" = 17.5419
        "$a(r.o- _l.o+)" = 17.5419
        "$a(x- _l.o+)" = 19.3232
        "$l(l_.o)" = 82.7014
        "$r(l_.o-)" = 9.24855
        "$a(_l.o+ l_.o-)" = 9.24855
        "$l(l.o)" = 71.3986
        "$r(l.o+)" = 15.2612
        "$a(l_.o- l.o+)" = 15.2612
        "$r(_l.o-)" = 8.16839
        "$a(l.o+ _l.o-)" = 8.16839
        "$r(x+)" = 14.5259
        "$a(l_.o- x+)" = 14.5259
        "$a(x+ _l.o-)" = 8.90373
        "$r(l_.o+)" = 12.5273
        "$a(_l.o- l_.o+)" = 12.5273
        "$r(r_.o-)" = 10.7284
        "$a(l_.o+ r_.o-)" = 10.7284
        "$a(x+ r_.o-)" = 11.4489
        "$c0" = 114.484
        "$r(l.o-)" = 17.8497
        "$a(l_.o+ l.o-)" = 17.8497
        "$a(l.o- _l.o+)" = 22.8665
        "$c1" = 85.9216
        "$r(r.i+)" = 40.701
        "$a(r.o- r.i+)" = 40.701
        "$a(r.i+ r_.o-)" = 12.6923
        "$c2" = 94.4019
        "$period" = 114.484
        "$power" = 1
        "$widths" = 1
```

Figure B.5: Output of optimization tool for the lazy-active/passive buffer (cont.).

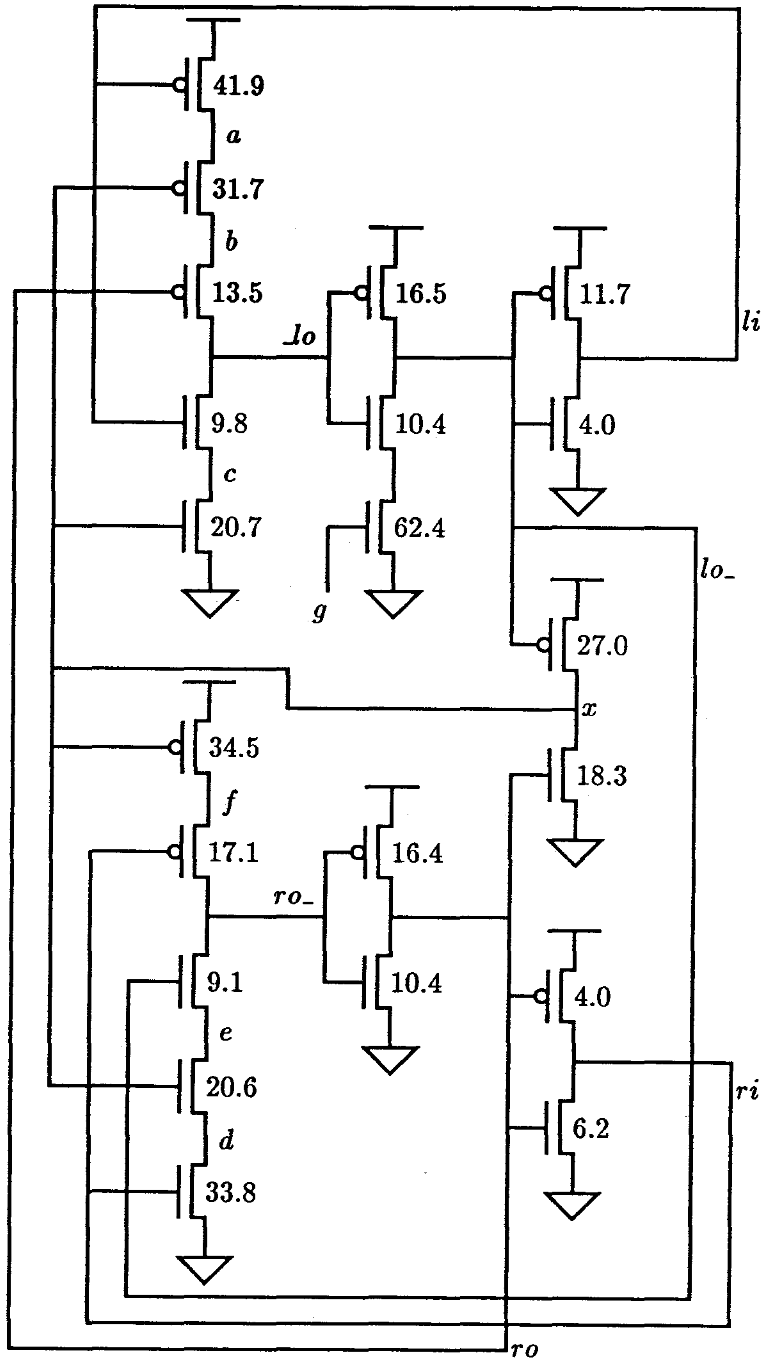


Figure B.6: Circuit for lazy-active/passive buffer.

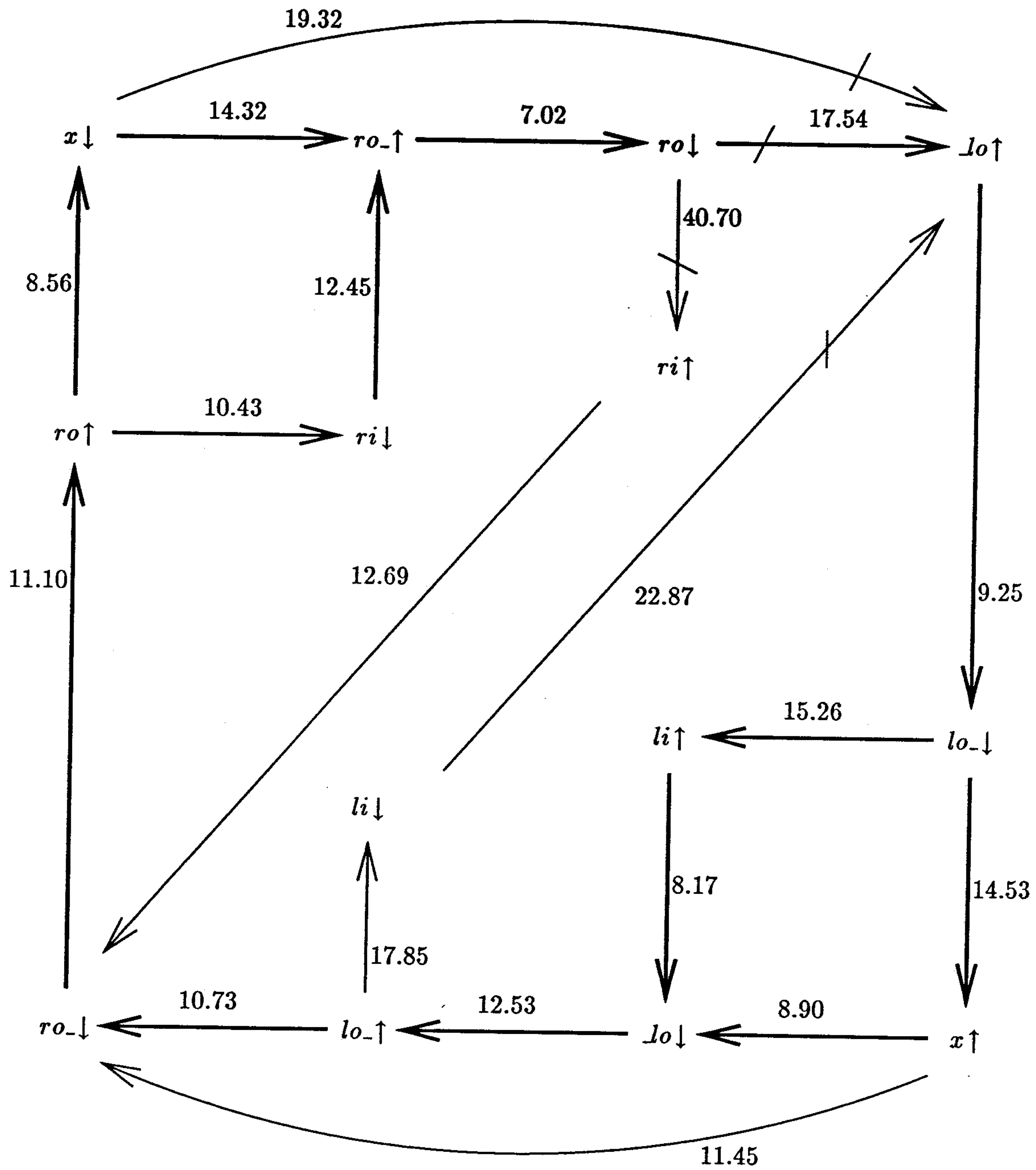


Figure B.7: Period constraint graph with optimized transistors.

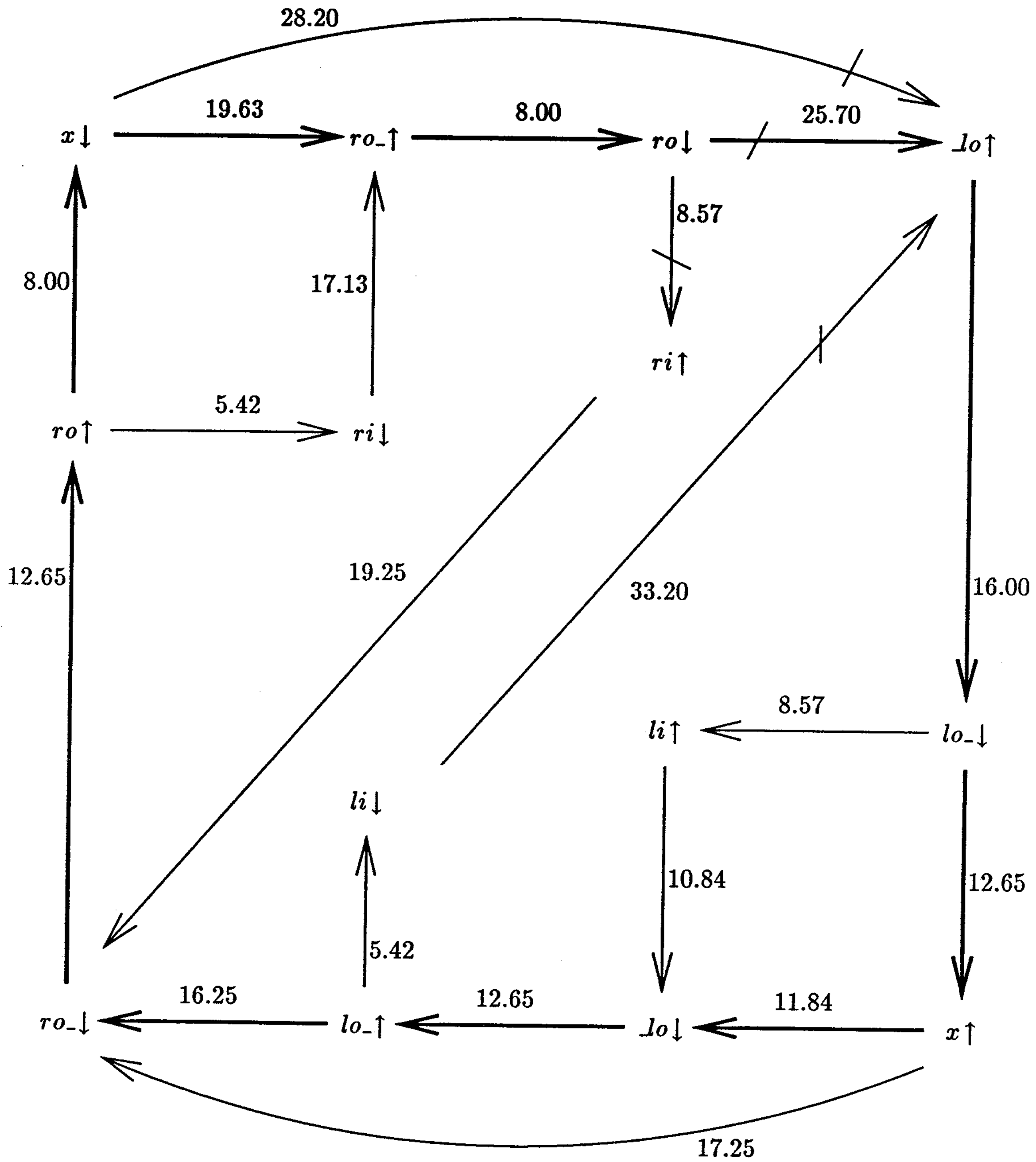


Figure B.8: Period constraint graph with nonoptimized transistors.

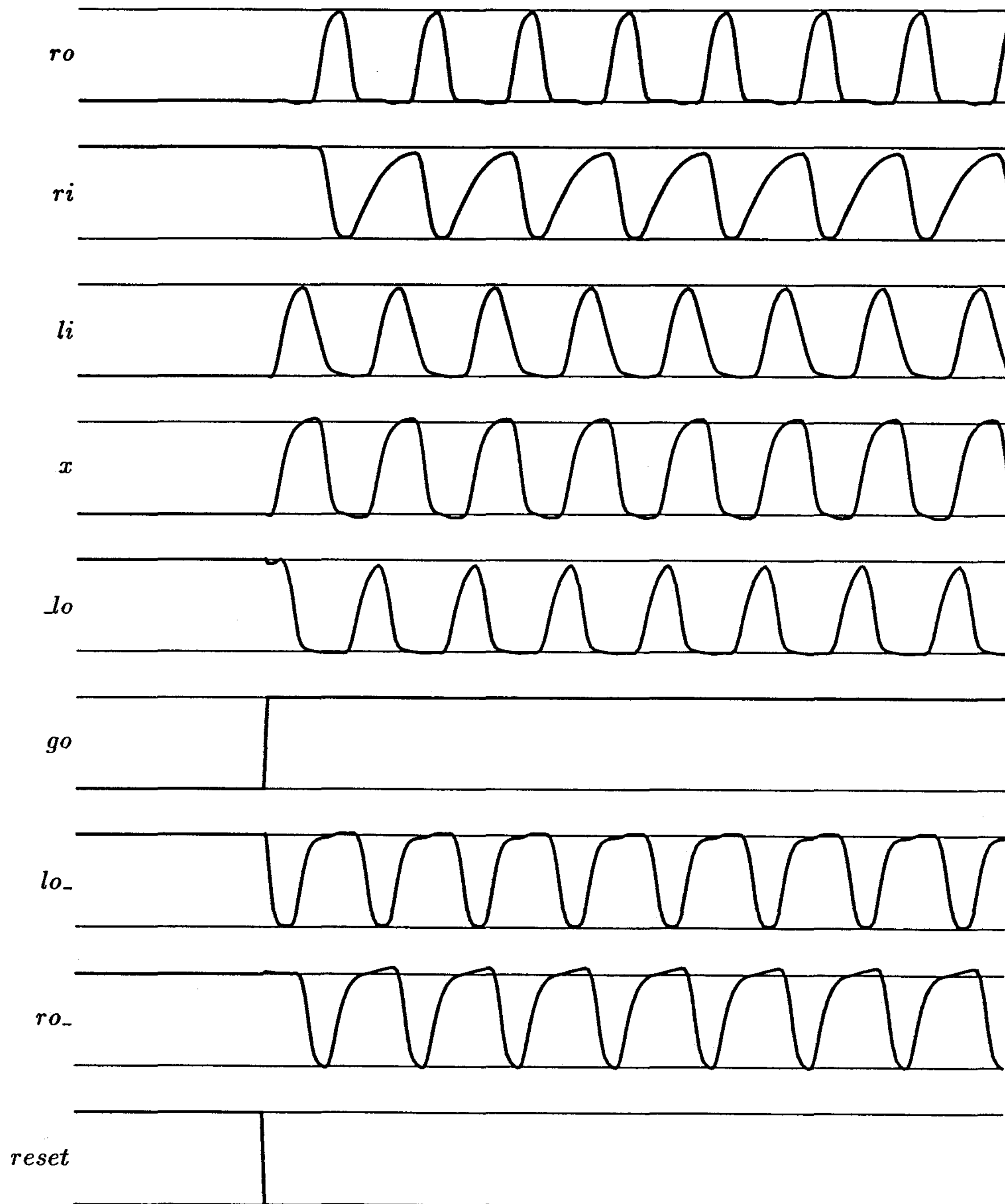


Figure B.9: SPICE output of optimized lazy-active/passive buffer. $p = 5.2ns$ Improvement: 52%

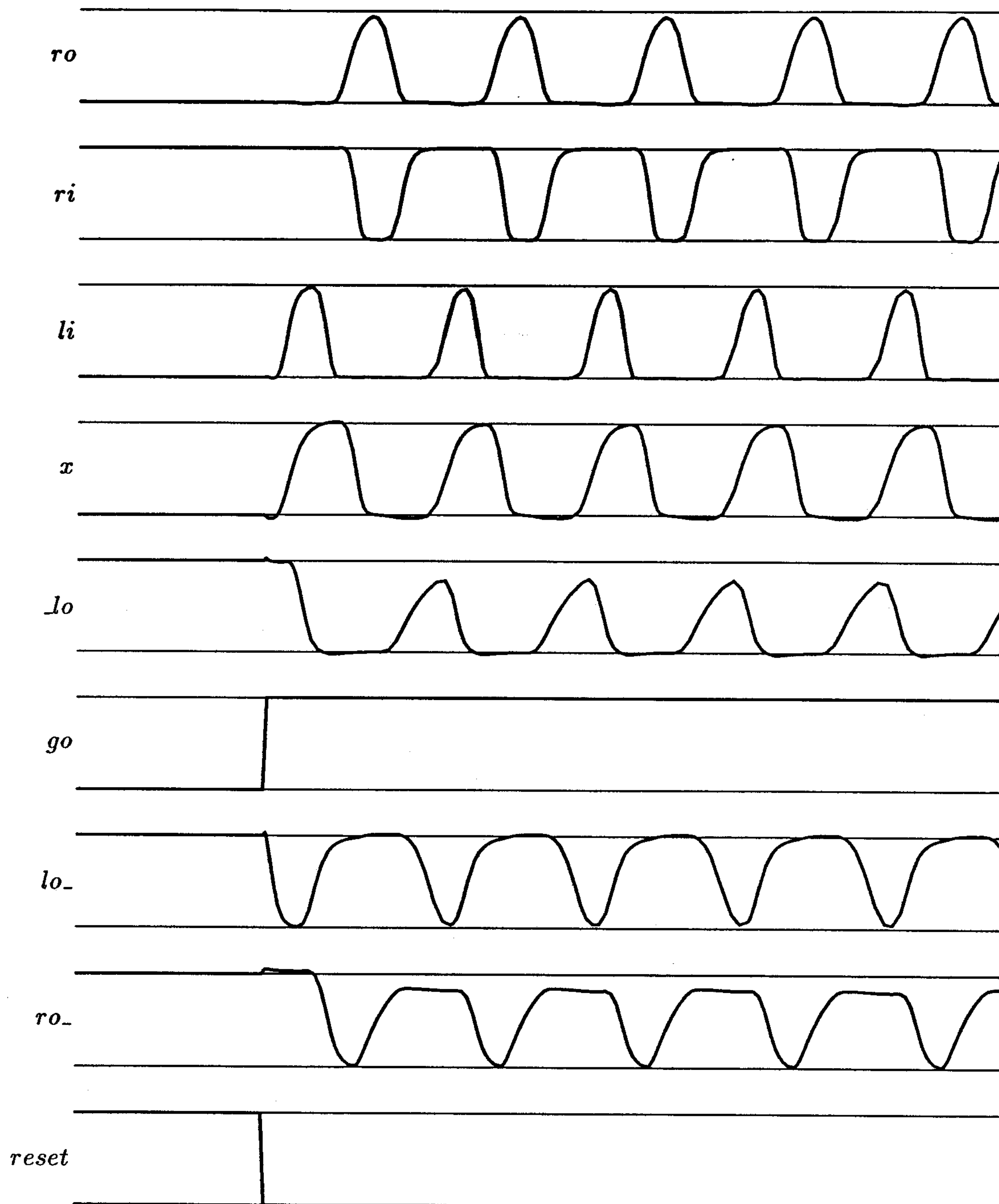


Figure B.10: SPICE output of nonoptimized lazy-active/passive buffer. $p = 8.0ns$

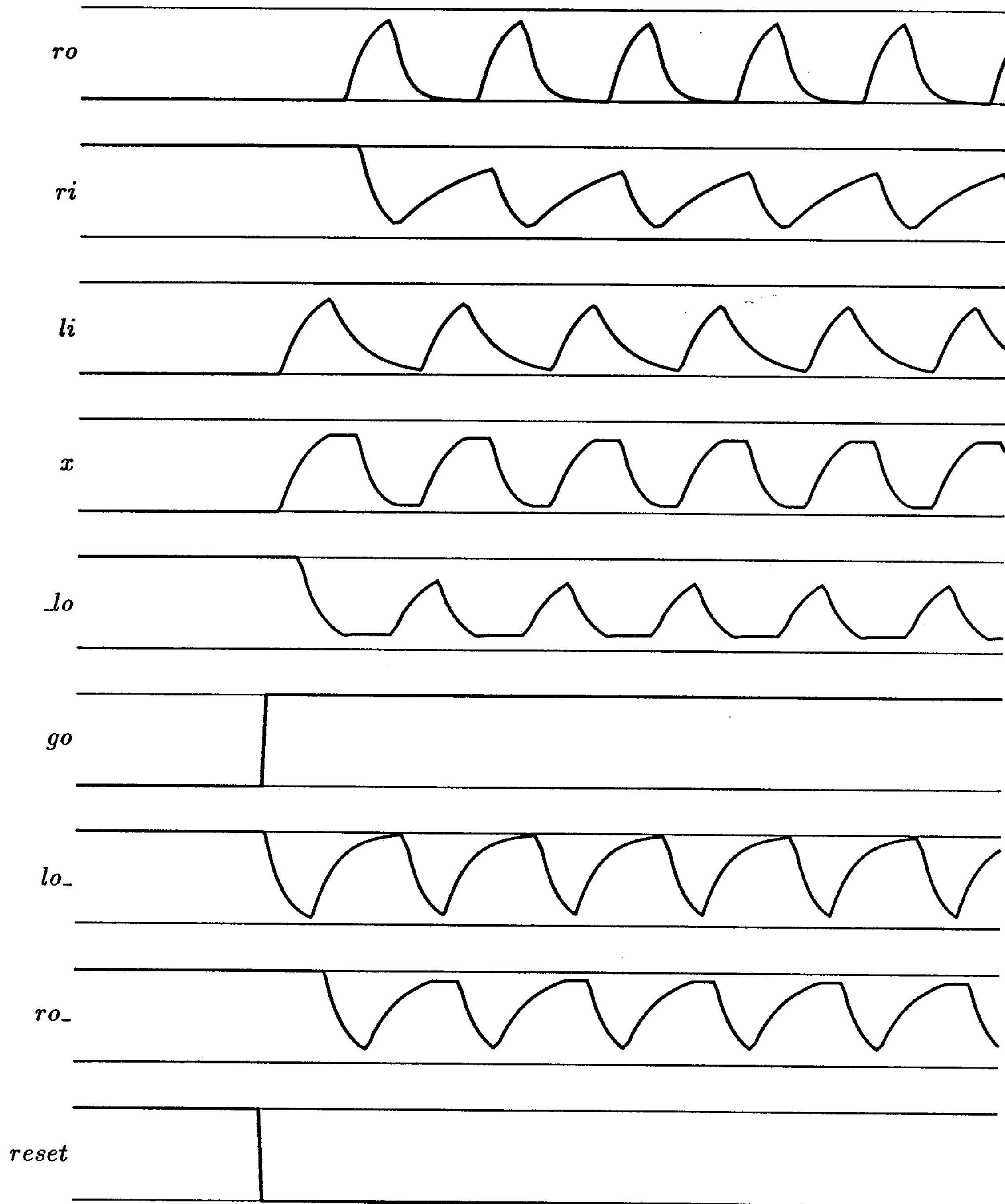


Figure B.11: SPICE output of optimized lazy-active/passive buffer using the switch model. $p = 6.9ns$ Improvement: 38% Predicted improvement: 25%

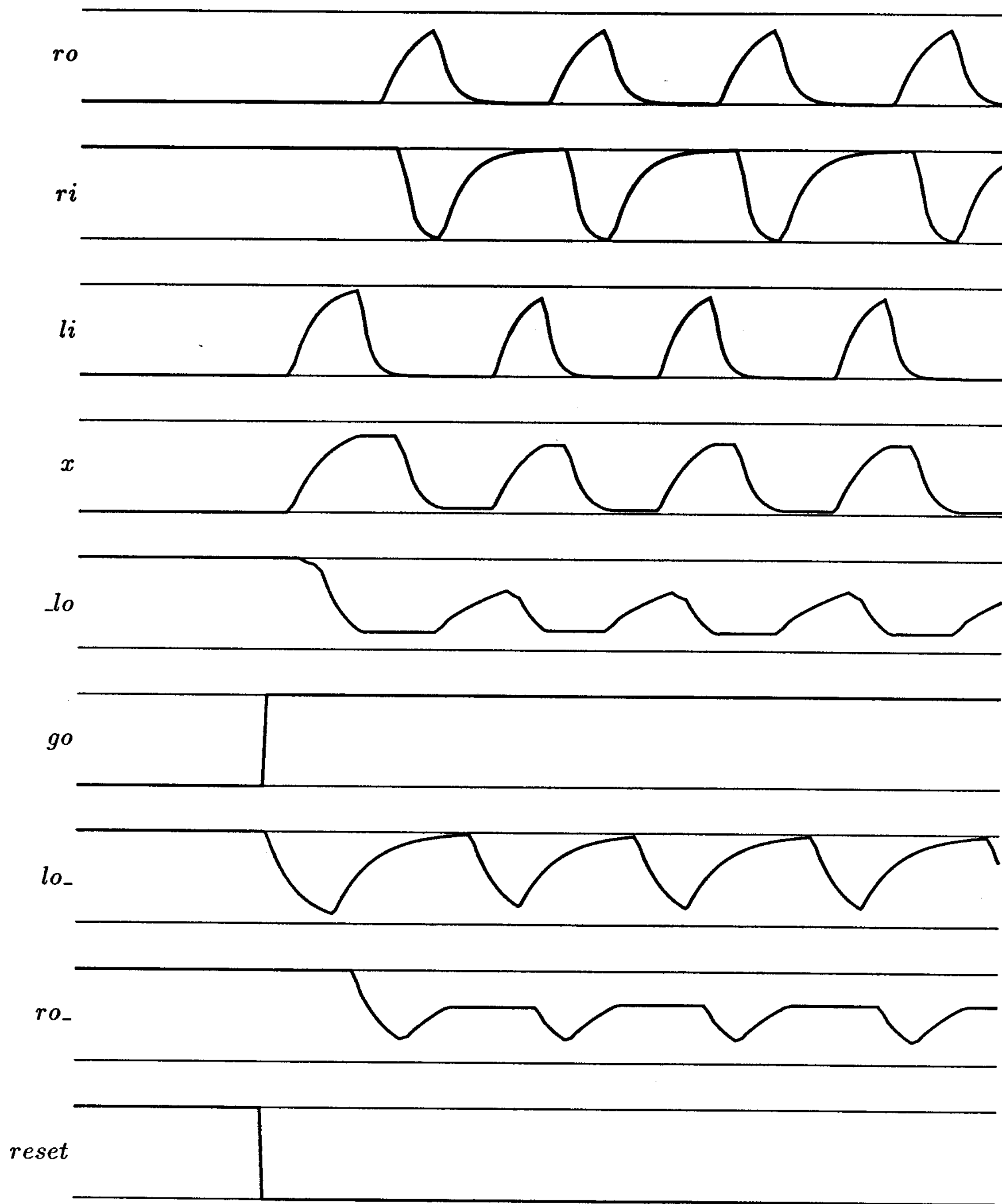


Figure B.12: SPICE output of nonoptimized lazy-active/passive buffer using the switch model. $p = 9.5ns$

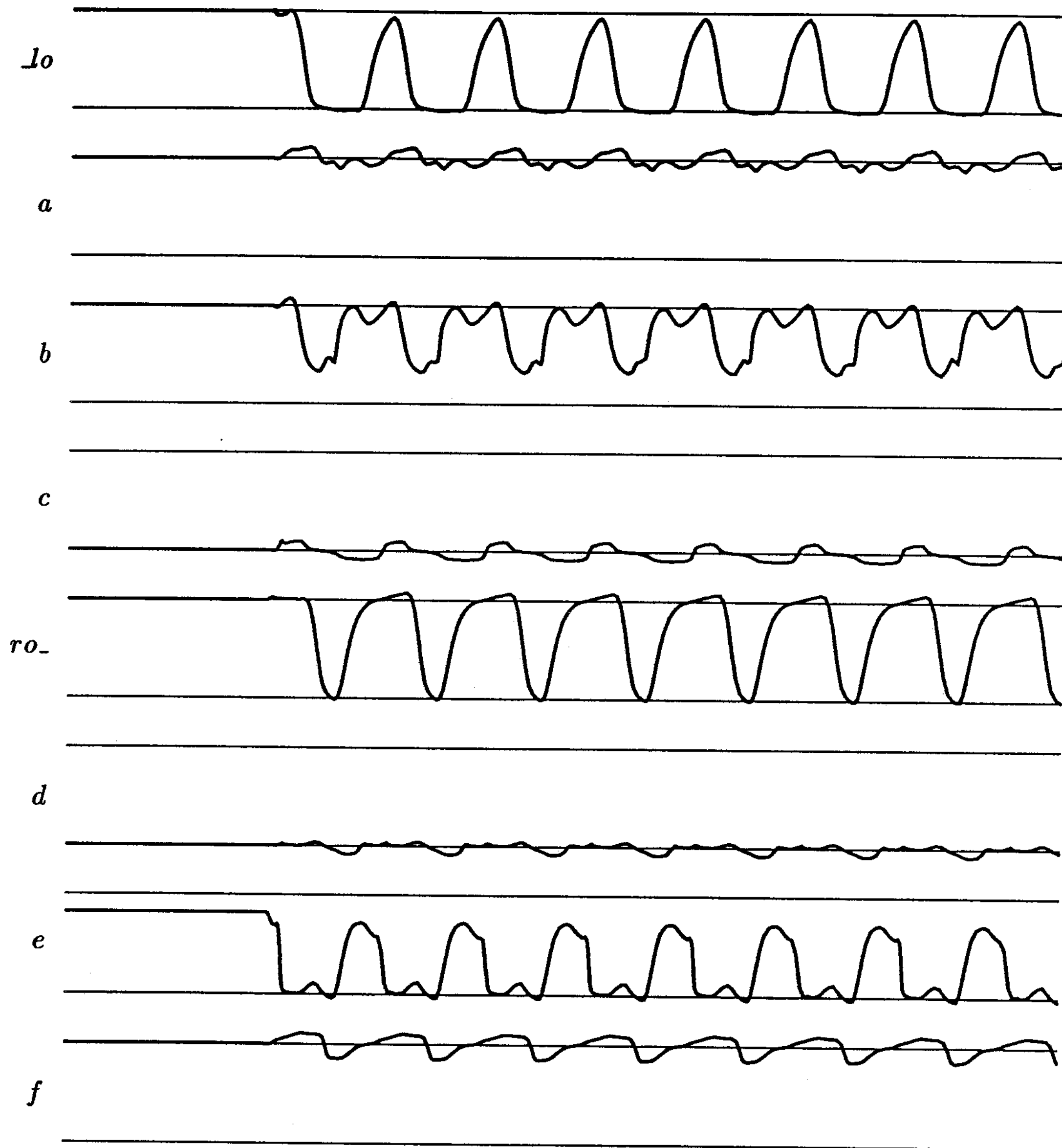


Figure B.13: SPICE output of optimized lazy-active/passive buffer.

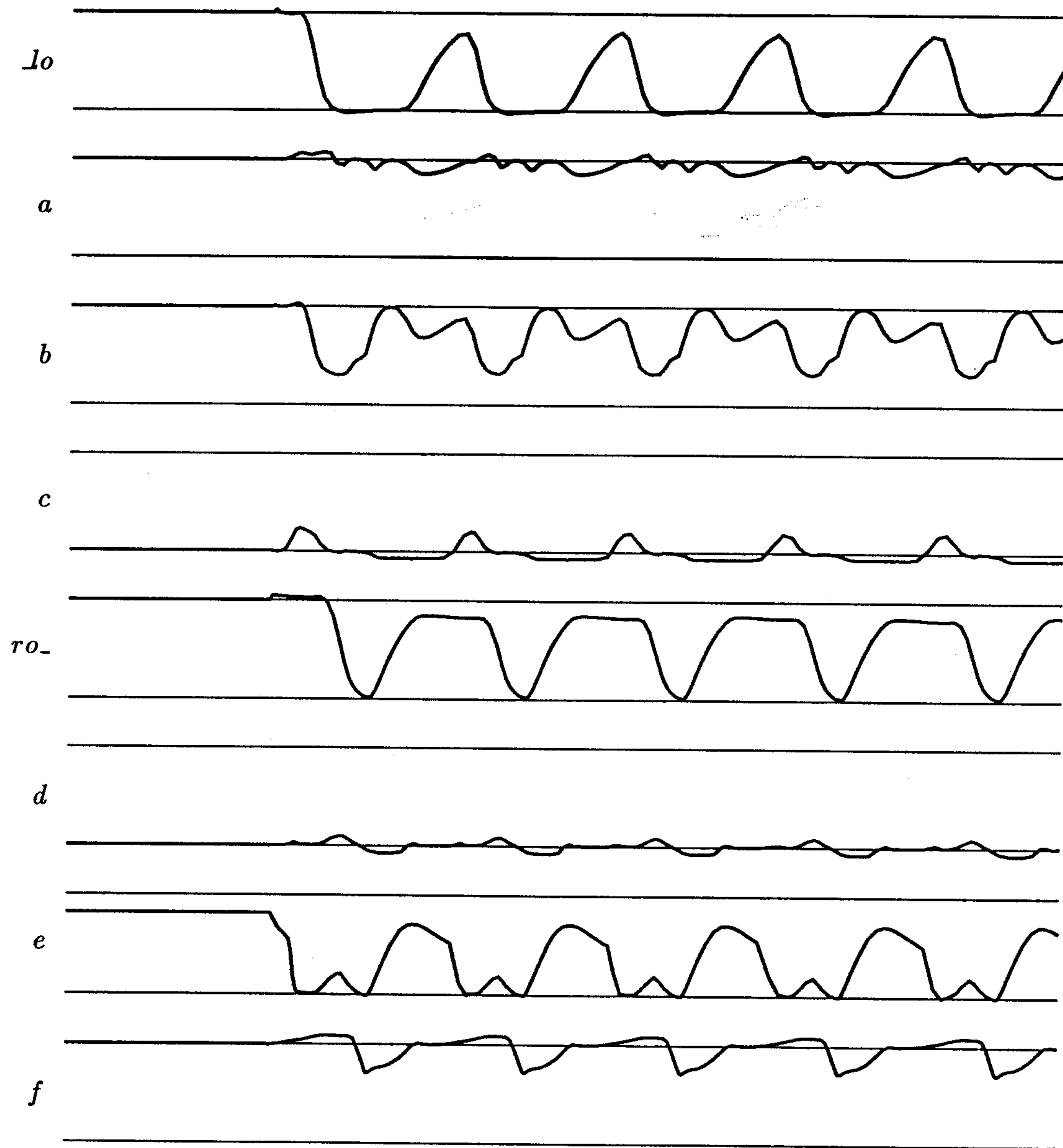


Figure B.14: SPICE output of nonoptimized lazy-active/passive buffer.

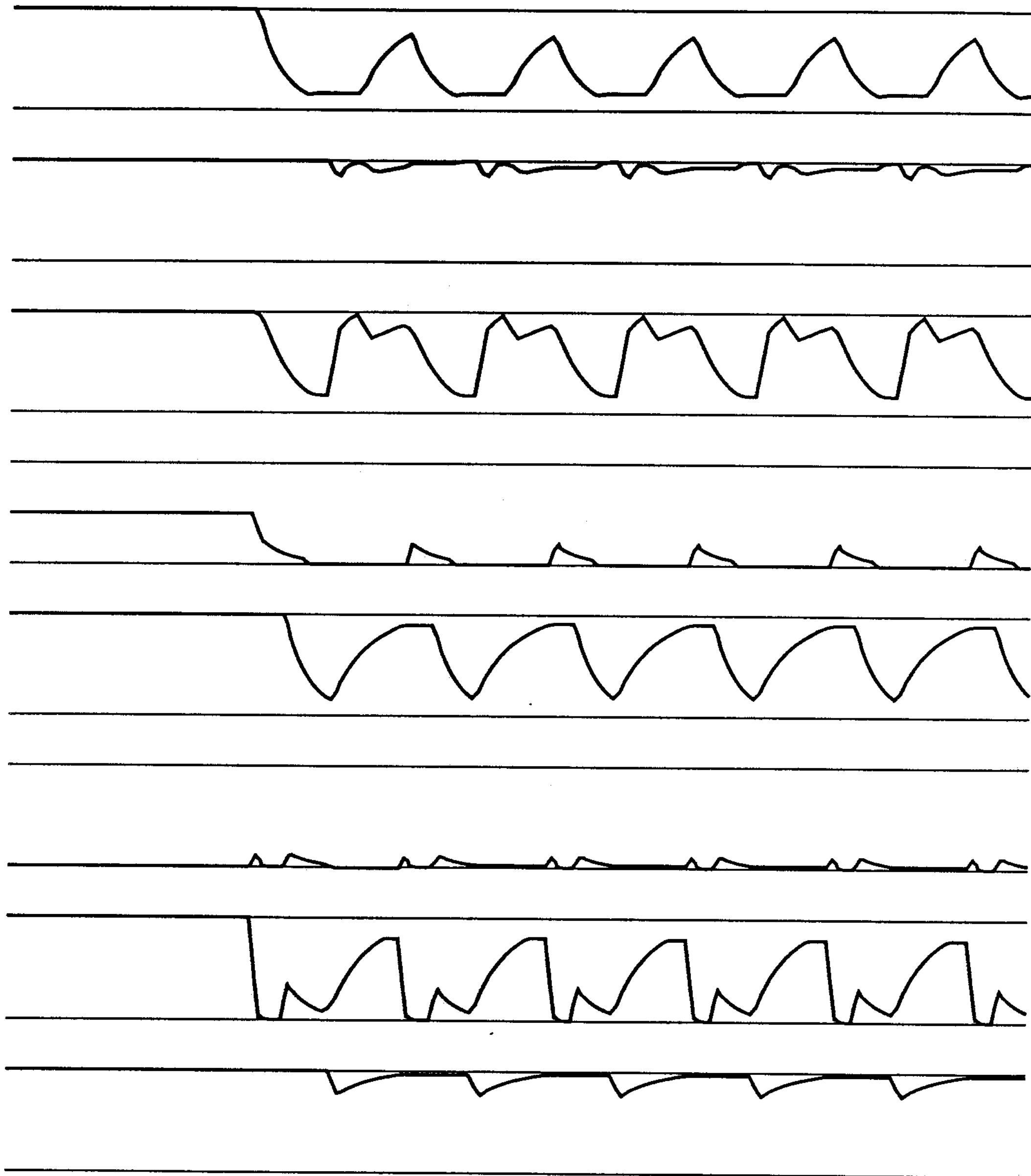


Figure B.15: SPICE output of optimized lazy-active/passive buffer using the switch model.

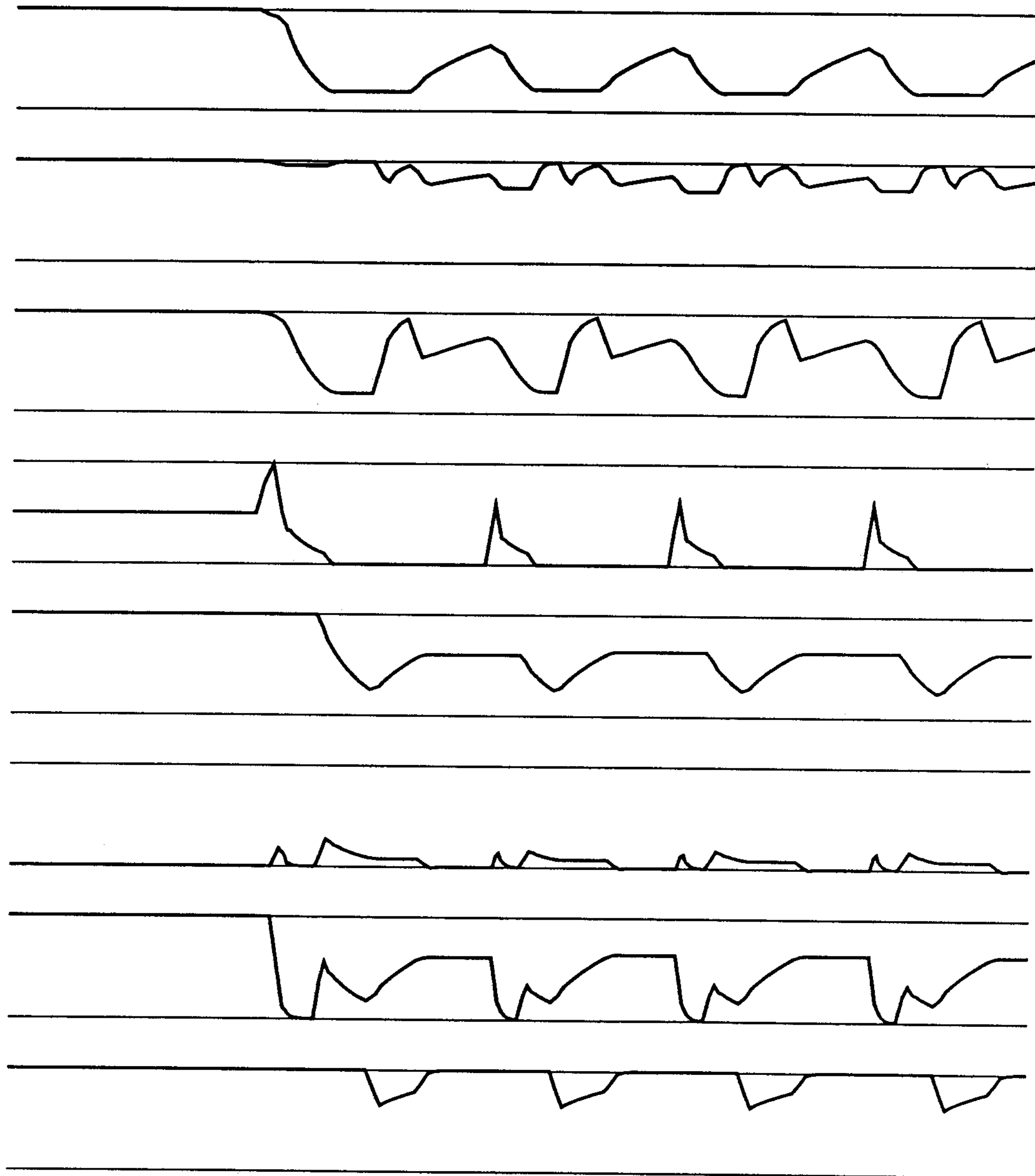


Figure B.16: SPICE output of nonoptimized lazy-active/passive buffer using the switch model.

Graph Edge	Predicted (τ)	Simulated (ns)
$ri\downarrow\ ro_\uparrow$	12.4528	0.716107
$lo_\downarrow\ x\uparrow$	14.5259	0.557763
$x\uparrow\ ro_\downarrow$	11.4489	1.6227
$_lo\uparrow\ lo_\downarrow$	9.24855	0.487441
$lo_\uparrow\ ro_\downarrow$	10.7284	0.495459
$ro\downarrow\ _lo\uparrow$	17.5419	0.616859
$x\downarrow\ ro_\uparrow$	14.3228	0.736862
$ro_\downarrow\ ro\uparrow$	11.1045	0.4743
$ro\uparrow\ ri\downarrow$	10.4282	0.391314
$ro_\uparrow\ ro\downarrow$	7.0231	0.383195
$lo_\uparrow\ li\downarrow$	17.8497	0.716005
$x\downarrow\ _lo\uparrow$	19.3232	1.73692
$lo_\downarrow\ li\uparrow$	15.2612	0.569415
$ro\downarrow\ ri\uparrow$	40.701	1.17362
$li\uparrow\ _lo\downarrow$	8.16839	0.589842
$ri\uparrow\ ro_\downarrow$	12.6923	2.11115
$_lo\downarrow\ lo_\uparrow$	12.5273	0.525745
$li\downarrow\ _lo\uparrow$	22.8665	2.37577
$ro\uparrow\ x\downarrow$	8.55819	0.375331
$x\uparrow\ _lo\downarrow$	8.90373	0.601494

Table B.1: Transistor simulation, simple model: predicted α's (τ) vs. simulated α's (ns).

Graph Edge	Predicted (τ)	Simulated (ns)
$ri\downarrow\ ro_\uparrow$	12.4528	1.0093
$lo_\downarrow\ x\uparrow$	14.5259	0.794033
$x\uparrow\ ro_\downarrow$	11.4489	1.97755
$_lo\uparrow\ lo_\downarrow$	9.24855	0.518611
$lo_\uparrow\ ro_\downarrow$	10.7284	0.664136
$ro\downarrow\ _lo\uparrow$	17.5419	0.794643
$x\downarrow\ ro_\uparrow$	14.3228	1.00696
$ro_\downarrow\ ro\uparrow$	11.1045	0.797499
$ro\uparrow\ ri\downarrow$	10.4282	0.418945
$ro_\uparrow\ ro\downarrow$	7.0231	0.562676
$lo_\uparrow\ li\downarrow$	17.8497	0.80195
$x\downarrow\ _lo\uparrow$	19.3232	2.36428
$lo_\downarrow\ li\uparrow$	15.2612	0.863115
$ro\downarrow\ ri\uparrow$	40.701	2.28707
$li\uparrow\ _lo\downarrow$	8.16839	0.488083
$ri\uparrow\ ro_\downarrow$	12.6923	1.79777
$_lo\downarrow\ lo_\uparrow$	12.5273	0.756253
$li\downarrow\ _lo\uparrow$	22.8665	3.44526
$ro\uparrow\ x\downarrow$	8.55819	0.421288
$x\uparrow\ _lo\downarrow$	8.90373	0.557165

Table B.2: Switch-level simulation, simple model: predicted α's (τ) vs. simulated α's (ns).

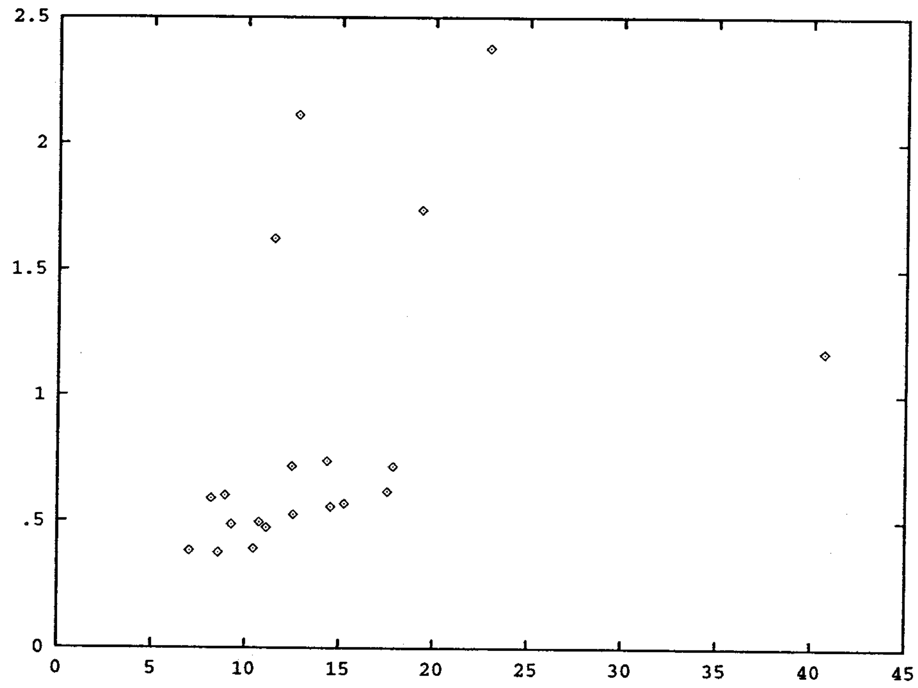


Figure B.17: Transistor simulation, simple model: simulated α's (ns) vs. predicted α's (τ).

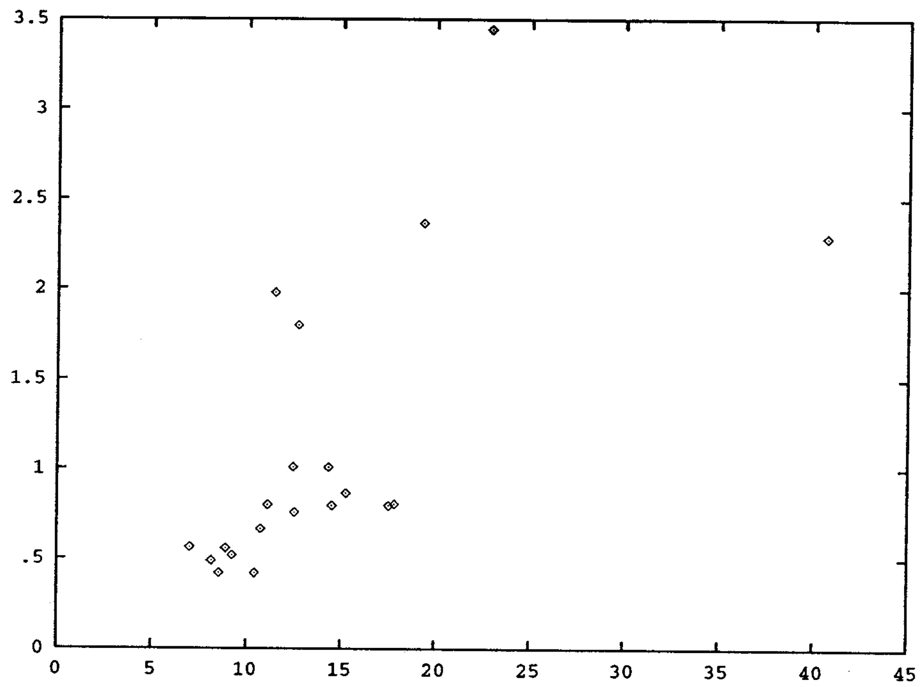


Figure B.18: Switch-level simulation, simple model: simulated α's (ns) vs. predicted α's (τ).

Graph Edge	Predicted (τ)	Simulated (ns)
$x\uparrow\ ro_\downarrow$	12.9682	1.54882
$lo_\downarrow\ x\uparrow$	14.4692	0.540642
$ri\downarrow\ ro_\uparrow$	13.8993	0.653855
$_lo\uparrow\ lo_\downarrow$	9.21432	0.49932
$lo_\uparrow\ ro_\downarrow$	12.1956	0.545544
$ro\downarrow\ _lo\uparrow$	19.1171	0.642331
$x\downarrow\ ro_\uparrow$	15.7815	0.66685
$ro_\downarrow\ ro\uparrow$	11.0317	0.478837
$lo_\uparrow\ li\downarrow$	18.5313	0.713662
$ro\uparrow\ ri\downarrow$	10.3968	0.387837
$ro_\uparrow\ ro\downarrow$	6.97786	0.366419
$x\downarrow\ _lo\uparrow$	20.9894	1.67563
$lo_\downarrow\ li\uparrow$	15.2094	0.551436
$li\uparrow\ _lo\downarrow$	9.63698	0.491809
$ro\downarrow\ ri\uparrow$	42.4117	1.1761
$_lo\downarrow\ lo_\uparrow$	12.4524	0.500672
$ri\uparrow\ ro_\downarrow$	14.0595	2.05501
$li\downarrow\ _lo\uparrow$	24.1886	2.37168
$ro\uparrow\ x\downarrow$	8.51459	0.374842
$x\uparrow\ _lo\downarrow$	10.3771	0.502603

Table B.3: Transistor simulation, tied model: predicted α's (τ) vs. simulated α's (ns).

Graph Edge	Predicted (τ)	Simulated (ns)
$x\uparrow$ $ro_\downarrow$	12.9682	2.1009
$lo_\downarrow$ $x\uparrow$	14.4692	0.728569
$ri\downarrow$ $ro_\uparrow$	13.8993	0.54395
$_lo\uparrow$ $lo_\downarrow$	9.21432	0.763381
$lo_\uparrow$ $ro_\downarrow$	12.1956	0.806885
$ro\downarrow$ $_lo\uparrow$	19.1171	0.957902
$x\downarrow$ $ro_\uparrow$	15.7815	0.589523
$ro_\downarrow$ $ro\uparrow$	11.0317	0.711888
$lo_\uparrow$ $li\downarrow$	18.5313	0.875744
$ro\uparrow$ $ri\downarrow$	10.3968	0.506599
$ro_\uparrow$ $ro\downarrow$	6.97786	0.40268
$x\downarrow$ $_lo\uparrow$	20.9894	1.95011
$lo_\downarrow$ $li\uparrow$	15.2094	0.899785
$li\uparrow$ $_lo\downarrow$	9.63698	0.339696
$ro\downarrow$ $ri\uparrow$	42.4117	2.03018
$_lo\downarrow$ $lo_\uparrow$	12.4524	0.783103
$ri\uparrow$ $ro_\downarrow$	14.0595	2.52057
$li\downarrow$ $_lo\uparrow$	24.1886	3.1538
$ro\uparrow$ $x\downarrow$	8.51459	0.470423
$x\uparrow$ $_lo\downarrow$	10.3771	0.510912

Table B.4: Switch-level simulation, tied model: predicted α's (τ) vs. simulated α's (ns).

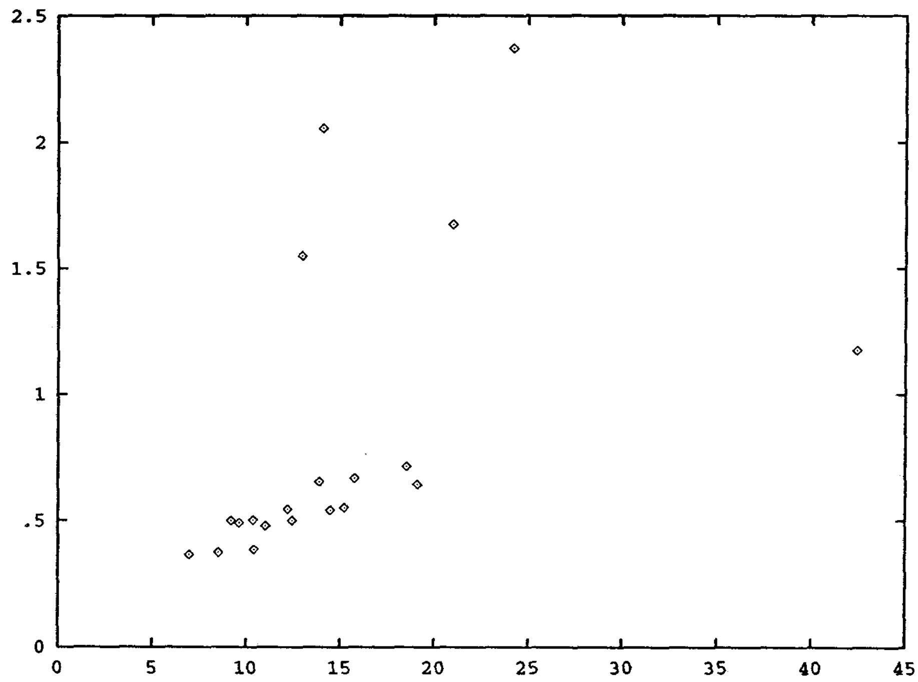


Figure B.19: Transistor simulation, tied model: simulated α's (ns) vs. predicted α's (τ).

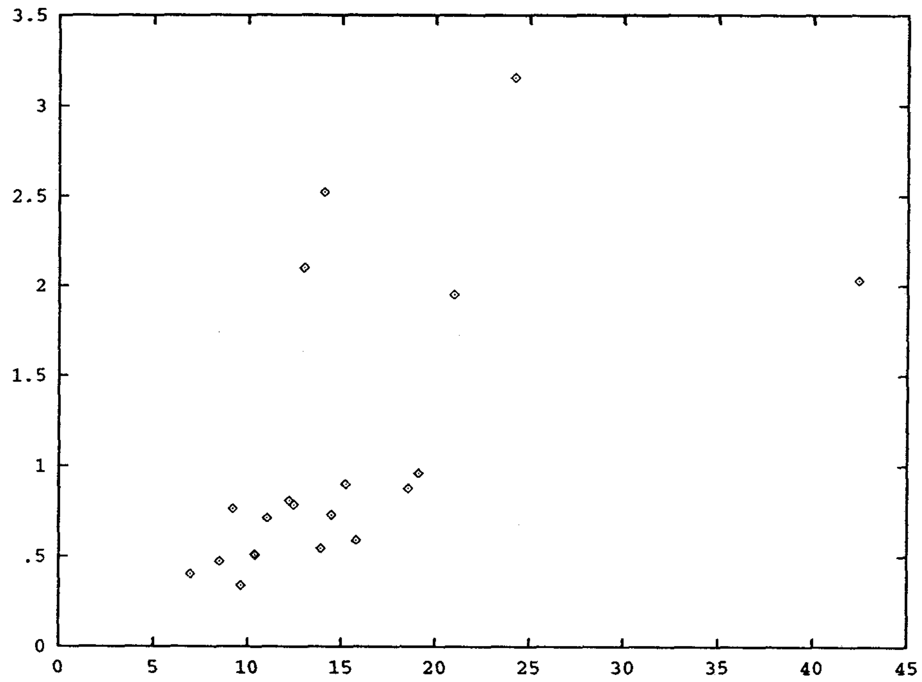


Figure B.20: Switch-level simulation, tied model: simulated α's (ns) vs. predicted α's (τ).

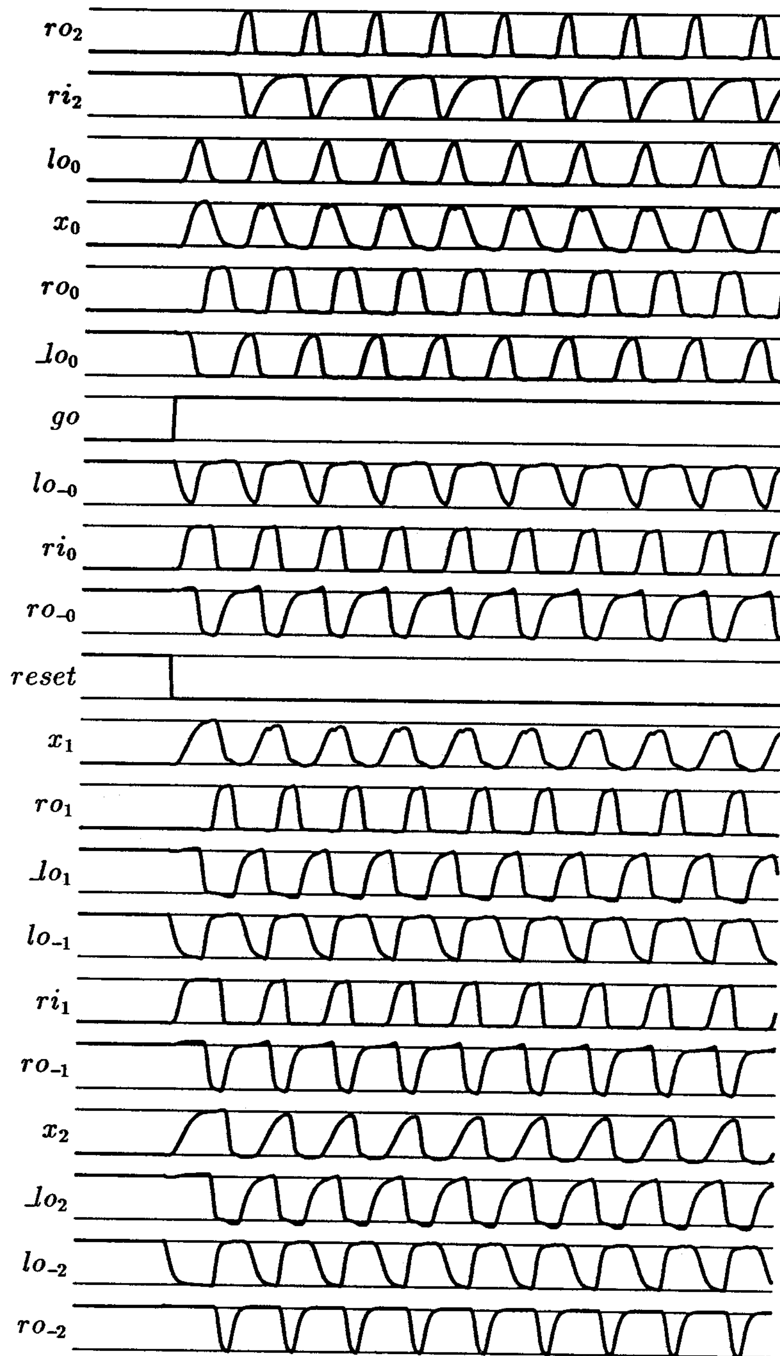


Figure B.21: SPICE output for three-stage optimized lazy-active/passive pipeline. $p = 7.3ns$ Improvement: 15% Predicted Improvement: 34%

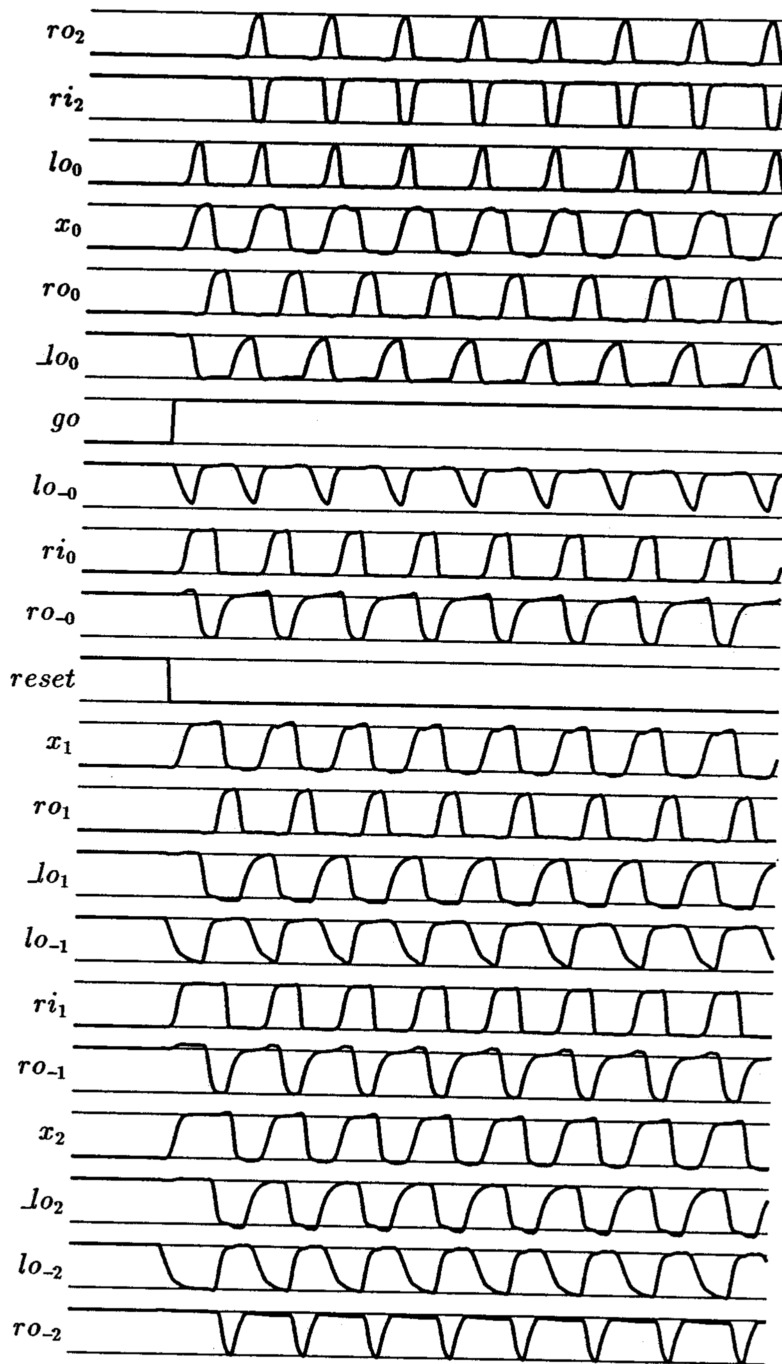


Figure B.22: SPICE output for three-stage nonoptimized lazy-active/passive pipeline. $p = 8.4ns$

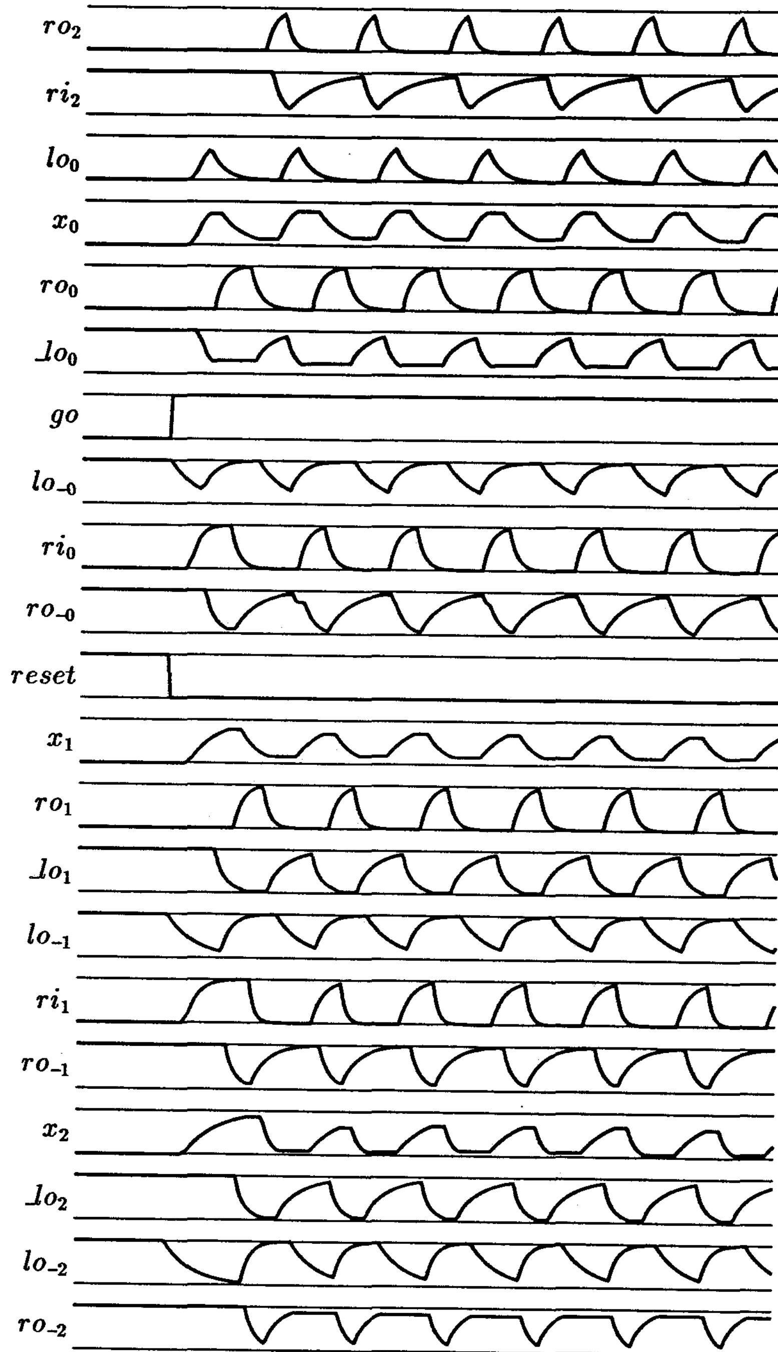


Figure B.23: Switch-level SPICE output for three-stage optimized lazy-active/passive pipeline. $p = 10.4ns$ Improvement: 30% Predicted Improvement: 34%

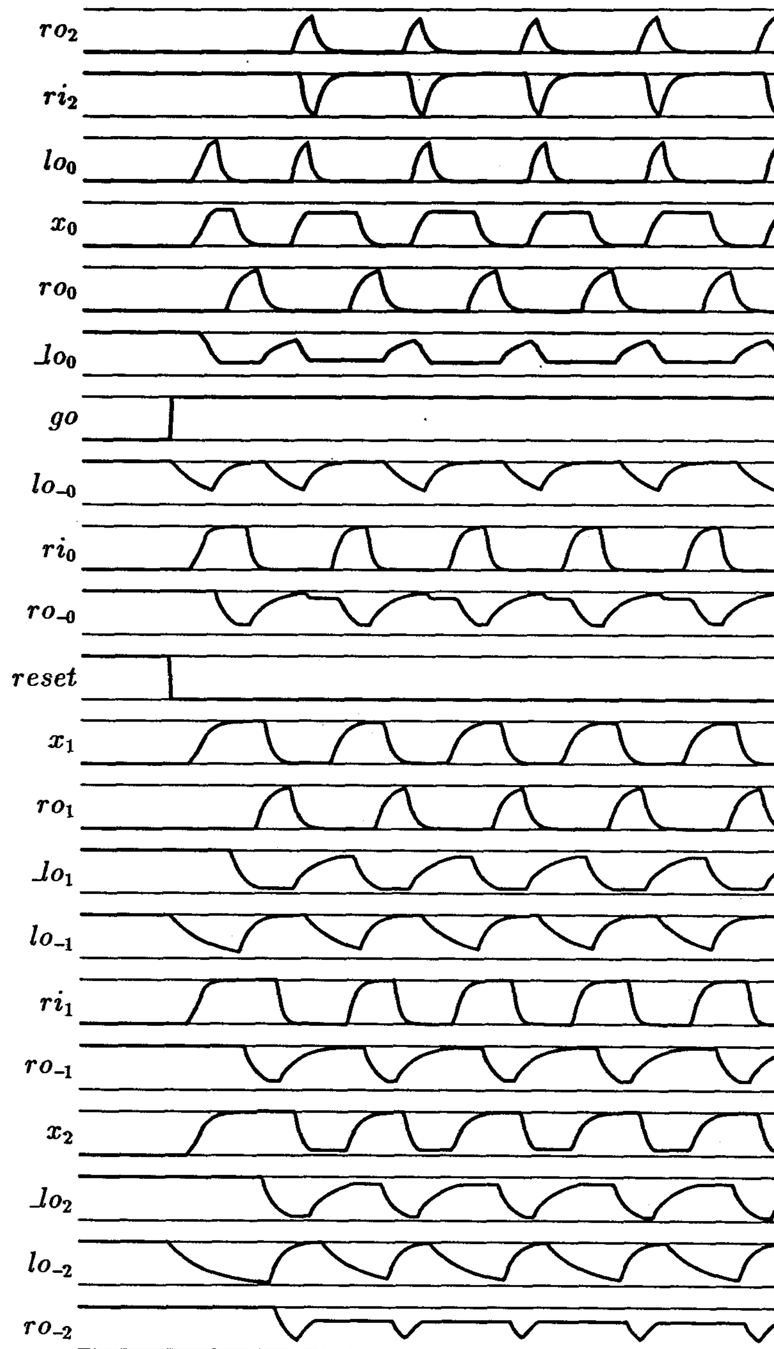


Figure B.24: Switch-level SPICE output for three-stage nonoptimized lazy-active/passive pipeline. $p = 13.5ns$