STARI: A TECHNIQUE FOR HIGH–BANDWIDTH COMMUNICATION

Mark Russell Greenstreet

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

January 1993

Abstract

This dissertation presents a novel technique for high–bandwidth communication called STARI, *Self–Timed At Receiver's Input*. STARI is a hybrid of synchronous and self–timed design techniques. STARI is provably correct, well suited for CMOS implementation, and offers a practical, cost effective approach to interprocessor communication in a multiprocessor. To demonstrate STARI, a CMOS chip has been implemented and is described here.

STARI is motivated by limitations of purely synchronous or purely self–timed hardware. This dissertation begins with an asymptotic analysis of the performance of self–timed pipelines. It is shown that self–timed pipelines can achieve linear speedup under an assumption of uniform interconnect delays. However, practical designs have non–uniform interconnect because of the hierarchy of packaging. In these designs, the performance is limited by the latency of the slowest interconnect.

To exploit the high–bandwidth offered by STARI in a multiprocessor, each processor node must have a memory system that offers high–bandwidth data transfers to the STARI interface and low latency access to the local CPU. Dual–port memories allow both of these goals to be met. Consistency problems that arise when using dual–port memories are identified and two software solutions are presented and analyzed.

Acknowledgements

I am grateful to the many people who have made this work possible. My readers, Andrea LaPaugh, Kai Li, and Ken Steiglitz have not only provided helpful comments about this manuscript, but they have provided support and advice throughout my time at Princeton. It is a great pleasure to offer a special thanks to Ken whose kindness, wit, and scholarship I can only hope to emulate.

Wonderful friends have made my time at Princeton enjoyable and memorable. They are too numerous to enumerate, but I must mention Matt Blaze, Pei Cao, Michael Cox, Kirk and Mira Kanzelberger, Margaret Nelson, and Norman Ramsey.

My appreciation to ATT Bell Laboratories for inviting me to use their facilities to test my chip. A special thanks to Bryan Ackland for making his lab available and to Alex Dickinson for cheerfully answering my many questions.

I am also appreciate assistance from the people at Intel Scientific Computers. In particular, I would like to thank Paul Close, Steve Nugent, Justin Rattner, and Anthony Zilka.

Associated with five years of graduate school is some money. Much of this work was made possible by an ATT Bell Laboratories Fellowship. I spent a productive summer at DEC Systems Research Center and another at Matsushita Information Technology Laboratory. The fabrication of my chip was funded by DARPA/ONR contract N00014-88-K-0459. Other support for this work came from the Defense Advanced Research Projects Agency and the Office of Naval Research under contract N00014-91-J-4039, the National Science Foundation under grants CCR-9-20893, MIP-8912100 and MIP-9201484, Intel Supercomputer Systems Division, and Digital Equipment Systems Research Center and External Research.

I owe a special debt of gratitude to Jørgen Staunstrup. He convinced a stubborn skeptic of the beauty and utility of formal methods when we were both at Aarhus University. Jørgen has been an extraordinary teacher, colloborator, mentor, and friend.

Colleagues and friends from Caltech, ESL, and DEC Systems Research Center deserve special mention. My thanks to Ted Williams for introducing me to the fascinating world of self–timed circuits and to John Hershberger for music, chemistry, good food, and some algorithms. Thanks to Joe Gray and Lars Jorgensen for our time at ESL as we all learned to be silicon hackers. Thanks to John Dillon, Leslie Lamport, Jim Saxe, and Chuck Thacker for a fun summer at DEC SRC where may of the ideas in this dissertation took form.

The most important lessons of life are learned at home. From my parents, Vernon and Shirley, I have learned that I am always loved, to celebrate the life God has given us, and to face challenges with confidence, diligence, and integrity. Thanks mom and dad.

I cannot possibly express my thanks to my wife, Susan. Her companionship, love, joy, and support have made this work possible and kept it in perspective.

To my parents and my wife

Contents

Abstract ii

Acknowledgements iii

1 Introduction 1

2 Performance of Self–Timed Pipelines 8

2.1 Self–timed Pipelines 10

2.2 Basic Properties of the Self–Timed Pipeline 12

2.3 Performance of Self–Timed Pipelines 13

2.4 Fixed Processing and Storage Times 15

2.5 Bounded Processing and Storage Times 16

2.6 Exponential Processing and Storage Times 17

2.7 Exponentially Distributed Processing Times with Fixed Offset 19

2.8 Conclusions 20

3 STARI 23

3.1 The schedule 25

3.1.1 Fixed delays 26

3.1.2 Bounded C-element delays 30

3.1.3 Bounded skew 31

3.2 Engineering considerations 33

3.3 Where did the synchronization errors go? 35

3.4 Conclusions 38

4 Implementing STARI **40**
4.1 Designing a Self–Timed FIFO 43
4.1.1 Self–timed codes 43
4.1.2 A simple FIFO 44
4.1.3 The Muller–C element 46
4.2 The Receiver 48
4.2.1 The input amplifier 50
4.2.2 The clock buffer and shift register 54
4.2.3 The demultiplexer 56
4.2.4 The FIFO 59
4.3 The Transmitter 61
4.4 Test Results 64

5 Cache Consistency with Dual–Port Memory **68**
5.1 Dual–Port Memories 69
5.2 Consistency Problems 72
5.3 Virtual–Memory Based Consistency Schemes 74
5.4 Performance 79
5.5 Virtually Addressed Caches 91
5.6 Related Work 92
5.7 Conclusions 94

6 Conclusions **95**
6.1 Contributions 95
6.2 Future Research 96

A Properties of Self–Timed Pipelines **116**

B Timing Verification of STARI **123**
B.1 `Synchronized Transitions` 124
B.1.1 Notation 124
B.1.2 Proof rules 125
B.1.3 Miscellany 126

B.2 Verification of STARI 127
B.2.1 The program 127
B.2.2 The invariant 130
B.2.3 The proof 132
B.2.4 The correctness of STARI 142
B.3 Remarks on the Verification 143
B.4 Related Work 146

C Analysis of Cache Consistency Schemes **148**
C.1 Write–Through Overhead 148
C.1.1 The single stage write buffer 149
C.1.2 Unbounded buffer without read bypassing 149
C.1.3 Unbounded buffer with read bypassing 150
C.1.4 How big is infinity? 150
C.2 Workload Statistics 152

List of Figures

1 A self–timed pipeline. Processors are labeled f and C-elements C. . . 10
2 Tokens in a self–timed pipeline 11
3 Definition of loop_i 12
4 Processor utilization for a self–timed pipeline 21
5 STARI communication 24
6 Stage–to–stage transfer times 27
7 Waiting for bubbles 28
8 Minimum cycle time vs. FIFO occupancy 35
9 STARI chip block diagram 42
10 Data tokens and bubbles in a self–timed FIFO 45
11 Three stages of a self–timed FIFO 45
12 A CMOS implementation of a Muller C–element 46
13 Block diagram of the receiver 48
14 The input amplifier 51
15 Dependence of delay on pulse width (data from simulation) 52
16 The shift register and clock buffer 54
17 A latched with tightly matched set–up and hold times 55
18 The demultiplexer 56
19 Simplified version of the ring counter 57
20 State transitions in the counter of figure 19 58
21 A ring counter with no long signal wires 58
22 Two FIFO stages as implemented on the prototype chip 60
23 Block diagram of the transmitter 61

24 Timing race in the FIFO 65
25 Block diagram of a Video RAM 69
26 A processor node using dual–port memory 70
27 Consistency problems with dual–port memory 73
28 Page sets in the two–list scheme 75
29 Physical addresses with aliased dual–port memory 78
30 Page sets with physical address aliasing 79
31 Write–Through overhead vs. write–buffer capacity 87
32 Write–Through overhead vs. Memory cycle time 88
33 Write–Through overhead vs. Frequency of shared writes 89
34 `Synchronized Transitions` program for STARI 128
35 An invariant for the program of figure 34 131
36 Work–load statistics 155

Chapter 1

Introduction

Communication is an essential aspect of all computation. As CPU performance continues to increase, communications demands grow proportionately [Amd67]. Accordingly, I/O bandwidths that were required by supercomputers a few years ago are now required for workstations. The demand for high bandwidth communication is even greater in multiprocessors where the communications network must not only support I/O activity but must provide for communication between processors as well. To be cost effective, multiprocessors cannot rely on exotic technology [Dal90]. Multiprocessors require high bandwidth communication, and the circuits that implement this communication should be integrated in the same VLSI process (i.e. CMOS in 1992) that is used for the processor, and the interconnection should use the same printed circuit technology that is used for the rest of the system.

This dissertation presents a novel technique for high performance communication called STARI, *Self–Timed At Receiver's Input*. STARI offers high bandwidth communication; it is well suited for CMOS implementation; and it can be used to transfer data over controlled impedance transmission lines such as the printed traces of a suitable printed circuit board or coaxial cables. These properties make STARI an attractive technique for implementing interprocessor communication in a multiprocessor.

Communication is the act of conveying information from one agent, **the transmitter**, to another, **the receiver**. Each communication action requires two agreements: an agreement on energy levels, the encoding of the information; and an agreement on time, when the transmitter may change these energy levels and when the receiver should measure them. Both of these agreements must be carefully considered when designing a multiprocessor. If a communication technique is wasteful of energy, then the power consumption of the multiprocessor will be high, and it may be uneconomical or infeasible to supply the power or remove the heat. If a communication technique is wasteful of time, then the performance of the multiprocessor will be limited by the bandwidth of interprocessor communication.

Three of the most common other approaches to communication are:

Synchronous: The transmitter and receiver operate at the same frequency as given by a common frequency reference, the "clock." The delays between the source of the clock, the transmitter and receiver must be tightly controlled (typically the tolerances must be much smaller than one period of the clock). No synchronization [CM73] is required.

Phase locked: The transmitter and receiver are controlled by separate clocks that are closely matched in frequency. Data is transmitted with a redundant coding that allows the receiver to extract a signal that corresponds to the transmitter's clock. Delays from the transmitter to the receiver do not need to be tightly controlled. Synchronization is required.

Self–Timed: Neither the transmitter nor receiver are controlled by clocks. Instead, data is transmitted with a redundant encoding that includes control information to regulate the operations of the receiver, and an acknowledgement signal from the receiver regulates the operations of the transmitter. No synchronization is required. Self–timed designs are discussed in greater detail in chapters 2 through 4.

It might seem that synchronous communication is the most efficient as it does not require a redundant encoding of data. However, the requirement for tight control of

delays throughout the system makes synchronous systems scale poorly. For example, as more processors are added to a synchronous multiprocessor, the clock period must be increased to tolerate larger uncertainities in the delays associated with distributing the clock to each processor [FK85, KS90, DS91]. At the other extreme, self–timed designs [Sei79, Dal87] are very tolerant of uncertainities of delays. However, in a purely self–timed design, the transmitter must wait for an acknowledgement from the receiver for the current value before sending the next one. This precludes having multiple bits simultaneously in transit along the same transmission line, and the bandwidth of such a design is limited by the inverse of the connection's round–trip latency.

Phase locked designs do not suffer from the scalability problems of synchronous designs [Now89], nor are they limited by round–trip latencies like self–timed circuits. However, phase-locked designs have circuitry requirements that are poorly suited for high levels of VLSI integration. First, a phase-locked loop must include analog circuitry to implement the voltage controlled oscillator and low pass filter. On the other hand, microprocessors, network controllers, and related components are purely digital designs. Fabrication processes that are optimized for high density digital designs often have poor characteristics for analog designs. Second, the phase comparator in the phase-locked loop must include circuits with gain–bandwidth products much higher than the bit rate. Accordingly, the phase detector is the critical bottleneck in a phase-locked design, and the other circuitry is underutilized. Because of these considerations, most phase-locked designs implement the phase-locked loop in a fast, low density, bipolar process, and other functions are implemented is a higher density CMOS process. Accordingly, both the transmitter and receiver in these designs typically require several chips (e.g., see [AMD89]). A further disadvantage of phase-locked designs is the need for synchronization. Because the time required for reliable synchronization can be two orders of magnitude greater than the bit–transfer time, blocks of data must be double buffered if high bandwidths are to be realized. This double buffering can be the dominant source of latency in phase-locked communication.

STARI provides a novel mechanism for the agreement on time required for communication. STARI combines synchronous and self–timed design methods to produce

a communications technique that suffers neither from the clock skew problems of synchronous designs nor from the round–trip latency problems of self–timed interfaces. As in a synchronous design, a global clock ensures that the transmitter and receiver operate at exactly the same frequency. Unlike a synchronous system, the uncertainity in the delays between the clock generator, the transmitter, and the receiver may be much greater than the clock period. Data is sent from the transmitter using a self–timed encoding, and a self–timed FIFO at the input of the receiver compensates for skew. The tolerance of skew is proportional to the length of the FIFO. With a sufficiently long FIFO, the skew tolerance can be arbitrarily large. No synchronization is required with a STARI interface.

STARI was developed using formal methods for hardware verification. I discovered STARI while trying to derive sufficient conditions to allow a self–timed transmitter to safely send data items before receiving all outstanding acknowledgements and thereby circumvent the bottleneck of round–trip delays. STARI is a very simple solution to this problem: the transmitter operates without acknowledgements from the receiver, and by design, the FIFO is guaranteed to operate at the rate of the global clock. This is proven using methods for timing verification for self–timed circuits based on formal methods for concurrent programming. In particular, the self–timed circuit is described as a concurrent program with real–time assertions. The correctness of STARI is formulated as an invariant of this program.

Dissertation overview

The structure of this dissertation reflects the chronology of the research that it describes. Although there have been many studies of the impact of clock–skew on the performance of synchronous processor arrays [FK85, KS90, DS91], and although self–timed designs have been proposed as a solution to the clock skew problem [Sei79, FK85, MBM88], there has be a paucity of analysis to support claims that self–timed processors scale well. Chapter 2 analyzes the performance of self–timed pipelines. Unlike other studies [Bur91, Wil91, Thi91a] that consider specific, small pipelines, and fixed delays, the analysis in chapter 2 addresses issues of scalability by considering asymptotic performance and issues of uncertainity by employing

probabilistic delay models. We show that self–timed processors can achieve linear asymptotic speedups under a wide variety of processing time distributions.

The asymptotic analysis of chapter 2 assumes that the multiprocessor is implemented on a uniform fabric, i.e. that all connections between circuit elements are roughly equivalent. This is a simplistic assumption that does not correspond to the computing technology of today or the foreseeable future. In practice, there is a packaging hierarchy [Bak90] of, for example, transistors on a semiconductor chip, chips on a printed circuit board, and boards in a rack. At each level of the hierarchy, connections are longer than those of the previous level. Accordingly, wires on a die can be driven much faster than those on the circuit board that connect separate chips, and the wires on the board can be driven faster than those of the backplane that connect boards in a rack. The self–timed communication protocols described in chapter 2 are limited by the latency of the interconnections; therefore, the performance of a self–timed multiprocessor is determined by the longest latency interconnect. Even though the circuits of a single chip are capable of operating at very high rates, these circuits are idle most of the time, waiting for off–chip communication.

STARI is presented in chapter 3. STARI combines synchronous and self–timed design techniques to produce a communication protocol that can operate at higher bandwidths than either purely synchronous or purely self–timed designs. Conditions that ensure correct operation are presented, and the consequences of these conditions for designing a real implementation are described. A formal statement and proof of the timing correctness of STARI is given in appendix B.

STARI is a general communications technique that could be implemented in a wide variety of technologies, for example CMOS integrated circuits connected by transmission lines, high–speed optical fibers [Sau89], or Boy Scouts with semaphore flags. In chapter 4, a CMOS implementation of STARI is described. The choice of CMOS is motivated by the desire to use STARI in multiprocessor applications. Here, high levels of integration are desirable to minimize the size and cost of each individual processor. As CMOS is the current technology of choice for implementing microprocessors, network controllers, memory, etc., it is the natural choice for STARI.

The choice of CMOS as the circuit technology leads directly to the choice to use

electrical (as opposed to optical) interconnection. Due to the low dispersion of optical fiber, optical interconnect can support higher bandwidths than electrical interconnect of similar cost. However, silicon is poorly suited for electo–optical applications [Sze81] because transitions between the conduction and valence bands are indirect[1]. Thus, optical interconnection requires multiple fabrication technologies (typically CMOS for controllers and buffers, and gallium arsenide for electro–optic devices). In many multiprocessor applications, the higher degree of integration that is offered by CMOS outweighs the higher bandwidths that can be achieved with optical interconnect. Accordingly, the implementation of STARI presented in this dissertation uses CMOS integrated circuits with electrical interconnect between the transmitter chip and the receiver chip.

The design presented in chapter 4 is based on the MOSIS 2 micron CMOS process (VTI n–well). In simulation, the design operates at data rates of 700 Mbits/sec. over a two–wire connection. Chips have been fabricated and tested at speeds up to 100 Mbits/sec. Due to a design error, it was not possible to test the chips at higher data rates. A second fabrication with corrected chips is planned.

STARI provides high bandwidth between processors. Chapter 5 examines how this impacts multiprocessor design. In particular, we consider processor nodes that employ dual–port memories so that the local CPU and the STARI interface can access data independently. For cache–based designs, dual–port memories introduce two issues of consistency. First, a consistency scheme must ensure that each active cache line was loaded *after* the corresponding locations of the dual–port memory were last written by the STARI interface. Second, when the STARI interface reads data from the dual–port memory, this data must correspond to the latest values written by the processor. Chapter 5 presents a software consistency scheme and analyzes its efficiency.

Chapter 2, the analysis of self-timed pipelines, is adapted from a prior journal paper by the author of this dissertation; see [GS90]. An earlier publication of this work is [GS88b]. STARI was described in [Gre92] which gives a proof that is similar

[1] A transition is indirect if the electron must loose momentum to a phonon when making the transition. In such transitions, the efficiency of photon production is usually very low.

to the proof presented in appendix B.

The main contributions of this dissertation are:

- Asymptotic analysis of the performance of self–timed pipelines with probabilistic delay models. In chapter 2, it is shown that self–timed pipelines can achieve linear speedup.

- STARI. Chapter 3 describes STARI, a novel approach to high bandwidth communication. STARI is a hybrid of synchronous and self–timed design techniques, and overcomes limitations that either has by itself. STARI is provably correct, simple, and well suited for VLSI (e.g. CMOS) implementation.

- An implementation of a STARI interface. Chapter 4 presents a CMOS implementation of STARI and examines many issues that arise in the design of CMOS circuits for high performance communication.

- A consistency scheme for processors with cache and dual–port memory. Chapter 5 identifies consistency problems that arise when using dual–port memories in processor architectures that include caches, and two software solutions are presented and analyzed.

Chapter 2

Performance of Self–Timed Pipelines

Many algorithms have implementations on a one–dimensional pipeline such that, in principle, the pipeline can be made arbitrarily long, with a proportionate increase in throughput on problems of the same size, and with no increase in either the complexity of the processor or the communication bandwidth. Examples of such designs include systolic arrays for signal processing applications [KRY81], chips for matching subsequences in DNA strings [LL85] and a machine [KS89] for the lattice–gas model of Frisch, Hasslacher, and Pomeau [FHP86] Each of these designs is a simple concatenation of identical stages, and the system has **linear speedup**, n stages have n times the throughput of one.

Pipelines can be implemented using either synchronous or self–timed circuitry. In a synchronous design, interprocessor communication is regulated by a clock, one or more signals that are distributed globally. The values of these signals in this ensemble are periodic in time; this period of the ensemble value is called the **clock period**. Each clock period is divided into **epochs** (also called phases) that are defined according to the values of the clock signal. The epochs in which a transmitter may change the values of its output must be disjoint from the epochs in which the corresponding receiver samples these values. Otherwise, the sampled value (e.g. a voltage level) may not correspond to a legal discrete value (e.g. "true," or "false" by

some level map). A failure of the receiver to obtain a well–defined value is called a synchronization failure [CM73]. Such failures can lead to the computation of incorrect results or other malfunctions of the computing system.

Due to, among other causes, variations in operating conditions, limited manufacturing tolerances, and imprecise circuit models it is not possible to control the exact time at which the clock events arrive at the various subcircuits. The uncertainity in the relative timing of clock events is called **clock skew**. The separation between the epochs in which a transmitter changes the values of its output and the epochs in which corresponding receiver samples these values must be greater than the clock skew. Otherwise, the two epochs may not appear disjoint to the receiver, and a synchronization failure may result. In general, clock skew grows with the total complexity of a synchronous design. There are many theoretical arguments [FK85, KS90, GS90, DS91] that synchronous processor arrays cannot achieve linear speed–up due to clock skew. Furthermore, a strong empirical argument against the practicality of large, synchronous systems is their scarcity. Most buses (e.g. VME and FutureBus [HP90a]) are asynchronous, and many architectures that began with synchronous timing (e.g. the original proposal for the Connection Machine [Hil85] and its implementation, the CM-2 [Thi87]) have evolved to asynchronous implementations (e.g. the CM-5 [Thi91b]).

To avoid the problems of clock skew, some designers [MB59, Sei79, Sin81, Udd84, MFR85, MBM88, Mar90] have promoted purely self–timed designs. Many analyses have appeared recently that address the performance of self–timed processors assuming deterministic delays of logic elements and wiring [Bur91, Wil91, Thi91a]. Each of these shows that an analysis of parallel program performance originally by Reiter [Rei68] can be applied to self–timed circuits. Each of these studies assumes fixed times for each circuit element to perform each operation, variations in these delays are neglected. As it is the *uncertainity* of delays that gives rise to clock skew problems in synchronous designs, the delays in a self–timed design should be modeled with random variables to provide a suitable basis for comparison. In the remainder of this chapter, a generic self–timed pipeline is described, and its performance is analyzed assuming several probabilistic delay models.

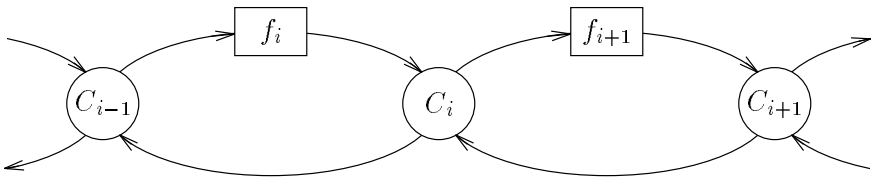


Figure 1: A self–timed pipeline. Processors are labeled f and C-elements C.

2.1 Self–timed Pipelines

In self–timed pipelines, the flow of computation is controlled by *completion* and *acknowledge* signals. As we will show, this form of control does not suffer from the asymptotic throughput limitations of synchronous designs mentioned in the introduction of this chapter. We consider self-timed rings (circular pipelines). When stage i completes a computation, it sends a completion signal to stage $i+1$ (modulo n, the number of stages in the ring). When stage i receives new input data, it sends an acknowledge signal to stage $i-1$. Figure 1 shows the self–timed pipeline analyzed in this chapter. The boxes labeled f perform the computation; the circles labeled C store the results and control the flow of data. In this figure, data flows to the right on the upper branches, and acknowledge signals flow to the left on the lower branches.

Register C_i is the output register for f_i and simultaneously the input register for f_{i+1}. The contents of C_i cannot change until two things happen: (1) f_i completes its computation of a new result, and (2) C_{i+1} acknowledges receipt of a result (which means that the old result from C_i has been used and can be overwritten).

We can visualize this sequence of events by placing tokens on the signal lines. Figure 2(a) shows a situation at time t_1: The token at the upper left input of C_i is the completion signal from f_i; the token at the lower right input is the acknowledge signal from C_{i+1}. At time t_2 (figure 2(b)), C_i stores the new result, transferring the completion token to its upper right output and the acknowledge token to its lower left output, the lower right input of C_{i-1}. At time t_3 (figure 2(c)), f_{i+1} computes a new result, transferring the completion token to the upper left input of C_{i+1}.

What we have described in this section summarizes well known ideas from the

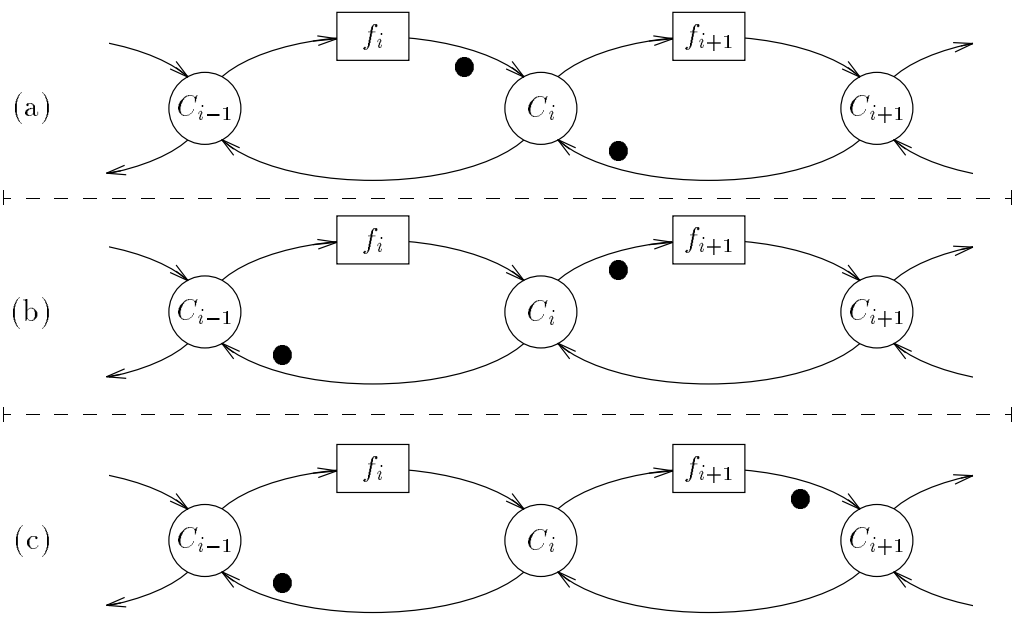


Figure 2: Tokens in a self–timed pipeline. Part (a) shows the state when C-element C_i is enabled to fire. (b) shows the state after firing; and (c) show the state after processor f_{i+1} has completed its computation.

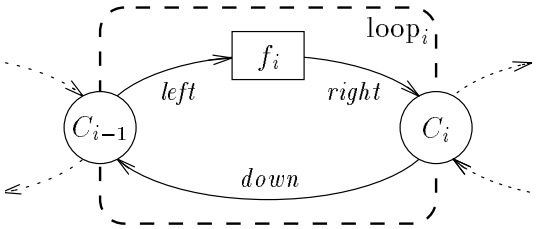


Figure 3: Definition of loop$_i$

literature of self–timed circuits and sets the stage for the analysis of such pipelines. The token–passing view of the pipeline is a simple instance of a Petri Net [DNS89]. As will be shown in chapter 4, the operations of the storage element C_i are analogous to those of a Muller C-element [Sei79]. In [Mil65] it is shown that three–stage pipelines of this form function correctly regardless of the delays of the f and C elements. In [GSW87], pipelines of any number (at least three) of stages are shown to function correctly.

2.2 Basic Properties of the Self–Timed Pipeline

From the rules governing the pipeline, it is easy to see that the number of tokens is invariant and equal to the number of stages. Let loop$_i$ be the cycle (input of f_i) $\rightarrow$ (upper left input of C_i) $\rightarrow$ (lower right input of C_{i-1}) as indicated in figure 3. Each loop contains exactly one token, and it is either a completion or an acknowledge token. We say that each such loop is in one of the three states *left*, *right*, or *down*, according to the location of the token. By the rules governing the flow of data and signals, the number of loops in the *down* state is invariant.

We call the processor f_i **active** if loop$_i$ is in state *left*. This corresponds to the intuitive notion of active: f_i begins a new computation when the token (i.e., new data) arrives at its input, and transfers the token from *left* to *right* when the computation is completed. In the same spirit, we say that C_i is **active** when loop$_i$ is in the *right* state and *loop*$_{i+1}$ is in the *down* state. This means that C_i is allowed to fire, sending an acknowledge back to C_{i-1} and data forward to f_{i+1}.

If all loops were in the *left* state, all processors would be active. However, once

these computations were completed, all loops would be in the *right* state, and the pipeline would be deadlocked. For progress to be made, C-elements must be enabled to fire; this requires that there be loops in the *down* state.

We refer to a loop in the *down* state as a **bubble**. Let n be the number of loops in the pipeline, and b be the number of bubbles. The number of bubbles, b, in a self–timed ring is invariant; this number is determined by the initialization of the ring. Theorem 1 in appendix A shows that $1 \leq b < n$ is a sufficient condition to ensure that a ring of n stages does not deadlock. The number of active processors is bounded from above by $n - b$. The number of active C-elements is bounded from above by $\min(b, n - b)$ because every active C-element must have a loop in the *down* state to its right (and there are b such loops) and a loop in the *right* state to its left (and there are at most $n - b$ such loops).

2.3 Performance of Self–Timed Pipelines

We describe the performance of a pipeline with two related quantities: throughput and utilization. The *throughput* is the total rate of computations made by all processors. If the throughput is proportional to the number of processors in the limit as the number of processors goes to infinity, we say that the pipeline has the **linear speedup** property. The *utilization* U of a processor is the fraction of the time that the processor spends computing. In the cases considered here, all processors have the same utilization. A utilization of 100% indicates that every processor is computing all of the time. The pipeline has linear speedup if the utilization is bounded from below by a positive constant as the number of processors goes to infinity. In the analysis of the following sections, we will derive exact values and bounds for utilization under various conditions.

In general, the times for computation and storage operations are random variables. To analyze the properties of self–timed rings, we consider infinite executions. In probabilistic terms, an "event" assigns specific values to the random variables corresponding to each operation. Because the word "event" also has connotations of any incident associated with an execution, we will use the term **realization** to refer

to an initial state and an assignment of values to each random variable of an infinite execution. Corollary 2.1 in appendix A shows that each execution is completely determined by its realization. For the sake of simplicity, we assume that the random variables for computation times are independent and identically distributed; t_f denotes a random variable with this distribution. Likewise, we assume that storage times are independent and identically distributed and write t_c to denote one such variable.

Because we consider infinite executions, any specific realization, in general, has probability zero. Therefore, we consider aggregate properties of pipelines, such as the average time between operations by the same processor. The properties that we develop here will hold for all realizations except for some (possibly non–empty) set with probability zero. Using standard terminology from probability, we will say that a property holds for *almost every* realization if it holds for a set of probability one. Likewise, we will say that a random variable has a particular value *almost surely* if it has the value with probability one.

To compute the utilization of a pipeline, we must determine the fraction of the time each processor is active. By the strong law of large numbers, the average time per computation for any processor in a given realization is $E[t_f]$ almost surely [Fel50a]. The **average waiting time** for a processor is the average time between starting successive computations. For each realization, the average waiting time is the same for every processor. For the pipelines considered in this dissertation, the average waiting time is the same for almost every realization (see appendix A, theorem 3.1), and we denote it by T. In this case, we define

$$U = \frac{E[t_f]}{T}.$$

Like all other benchmarks, the utilization of a pipeline must be interpreted with care. A high utilization could mean that the pipeline is very efficient. On the other hand, it could mean that the processors are very slow, and that greater throughput could be obtained by adding more storage elements (to increase the amount of pipelining) and thereby lowering the utilization.

2.4 Fixed Processing and Storage Times

We first consider pipelines with fixed t_f and t_c and a single bubble ($b = 1$). Each time f_i completes a computation, loop_i must cycle through states *left*, *right*, and *down*. The average waiting time T is the sum of the average times spent in each of the three states *left*, *right*, and *down*. The time spent in the *left* state is always t_f, and the time spent in the *right* state is at least t_c. Let T_{down} be the average time spent in the *down* state. We have $T \geq t_f + t_c + T_{\text{down}}$. By a simple balancing argument, $nT_{\text{down}} \geq T$. Combining these two relationships yields $T \geq (n/(n-1))(t_f + t_c)$. To obtain a second bound on T, note that the bubble must travel around the ring each cycle; so $T \geq nt_c$. We claim that for any pipeline the greater of these two bounds gives the exact value for T; therefore, the utilization U of the pipeline is given by

$$U = \begin{cases} \dfrac{n-1}{n}\dfrac{t_f}{t_f + t_c} & , n \leq \bar{s} \\ \dfrac{t_f}{nt_c} & , n \geq \bar{s} \end{cases}$$

where $\bar{s} = (t_f/t_c) + 2$.

To see this, consider first the case when $n \geq \bar{s}$. After the bubble leaves loop_i (to loop_{i-1}), at least $(n-2)t_c$ time units elapse before the bubble arrives at loop_{i+1}. Therefore, processor f_i has completed its computation by the time the bubble appears at the lower–right input of C_i. The rate of progress is completely determined by the progress of the bubble, and the bubble spends exactly t_c time units in each loop. Thus, $T = nt_c$ justifying the throughput in this case. The other case, when $n \leq \bar{s}$, follows by a similar argument. As $n \to \infty$, the utilization of the pipeline with a single bubble is $\Omega(1/n)$.

To improve utilization, more bubbles must be added to the pipeline. Let s be the average spacing of bubbles, $s = n/b$. First consider a pipeline where all the bubbles are initially equally spaced. Because all processing and storage times are identical and the initial pipeline configuration is symmetric, each bubble will propagate identically (see Appendix 2, Corollary 2.1). Since the bubbles are indistinguishable, the analysis of the single bubble case applies with n replaced by s. Appendix 2, Theorem 3 shows that for a large class of pipelines, including this case, the initial configuration of

bubbles does not affect the utilization of a pipeline. We conclude

$$U = \begin{cases} \frac{s-1}{s}\frac{t_f}{t_f+t_c} & , s \leq \bar{s} \\ \frac{t_f}{st_c} & , s \geq \bar{s}\,. \end{cases} \tag{1}$$

In either case, the throughput depends only on the average spacing of bubbles. In particular, it is independent of the length of the pipeline. Thus, for any fixed spacing, linear speed–up is achieved. Optimal performance is achieved by choosing $s = \bar{s}$, which yields

$$U_{\text{opt}} = \frac{t_f}{t_f+2t_c} \quad , s = \bar{s}\,. \tag{2}$$

Thus, linear speed–up is achieved by pipelines with fixed processing and storage times. Furthermore, utilizations arbitrarily close to 100% can be achieved for $t_f \gg t_c$.

2.5 Bounded Processing and Storage Times

We now consider pipelines where t_f and t_c are specified by their means and maxima. We derive lower bounds for throughput by deriving an upper bound for T. To do this, consider a pipeline where each computation requires time exactly $\max(t_f)$ and each storage requires time exactly $\max(t_c)$. We will call this the **slow pipeline**. Let U', and T' refer to the slow pipeline. By definition, $T' = \max(t_f)/U'$, and from appendix A, corollary 3.2, we have $T \leq T'$ which yields $U \geq (E[t_f]/\max(t_f))U'$. Since these $\max(t_f)$ and $\max(t_c)$ are fixed, the analysis of the previous section applies, and U' can be calculated. This yields the bounds

$$U \geq \begin{cases} \frac{s-1}{s}\frac{E[t_f]}{\max(t_f+t_c)} & , s \leq \frac{\max(t_f)}{\max(t_c)} + 2 \\ \frac{E[t_f]}{s\max(t_c)} & , s \geq \frac{\max(t_f)}{\max(t_c)} + 2\,. \end{cases} \tag{3}$$

Again, linear speedup is achieved for any fixed s. If t_c is sufficiently small and $\max(t_f)$ is sufficiently close to $E[t_f]$, then utilizations arbitrarily close to 100% can be achieved for an appropriate choice of s. Upper bounds for U can be obtained in a similar manner using $\min(t_f)$ and $\min(t_c)$.

It may seem pessimistic to estimate the utilization based on the worst–case delays. One would hope for a pipeline whose performance is determined primarily by the average processing and storage times. However we can propose pipelines where the bounds given above are tight in the limit as the number of processors goes to infinity. For example, consider a pipeline where t_f is $1/p$ with probability p and zero with probability $1-p$, and $t_c = 0$. This approximates the behavior of a pipeline where most operations are fast, but infrequently computations take a long time. If the pipeline is operated with a single bubble, we can show that $U = p + 1/n$. In contrast, for a pipeline with processing times fixed at $E[t_f] = 1$ and one bubble the analysis of Section 5 applies, and $U_{\text{fixed}} = (n-1)/n$. In the limit as the number of processors goes to infinity,

$$\frac{U}{U_{\text{fixed}}} = \frac{E[t_f]}{\max(t_f)}.$$

Thus, in the single–bubble case, linear speedup is achieved, but utilization is limited by the worst–case processing time. Better utilization can be achieved by adding more bubbles; however, this example shows that the performance of self–timed systems is not necessarily determined by average case–timings (as has often been claimed).

2.6 Exponential Processing and Storage Times

We now consider a case where the worst–case processing and storage times are not bounded. We show that linear speed–up (relative to expected processing time) is still achieved. To exploit techniques from queuing theory, we make two simplifying assumptions:

1. The operations of the processor and the C-element are combined into a single operation. This could be achieved by taking the acknowledge signal to C_{i-1} from the output of f_{i+1} instead of the output of C_i.

2. The time for the combined operation of storage and computation, $t_c + t_f$, is exponentially distributed with mean τ.

With this change, the *left* state is never visible. When loop_{i-1} is in state *right* and loop_i is in state *down*, a transition is enabled that brings loop_{i-1} into state *down* and

loop_i into state *right*. Because this could be effected by delaying the acknowledge inputs of C-elements, the throughput of this modified pipeline is less than or equal to the throughput of the original pipeline (appendix A, theorem 2).

A **segment** refers to a sequence of adjacent loops between bubbles plus the bubble on the right. Each segment can be viewed as a queue. A transition which lengthens a segment corresponds to an arrival to the corresponding queue; a transition which shortens a segment corresponds to a departure. A segment of length one corresponds to an empty queue (no departure possible since bubbles cannot be destroyed). Let λ be the rate of arrivals to a queue; the average waiting time T is s/λ. A segment can grow if the successor segment has a length greater than one, and the time for the transition which lengthens the segment is exponentially distributed with mean τ. Thus, $\lambda = (1 - p_1)/\tau$, where p_1 is the probability that the length of a segment is one. Accordingly, the utilization of processors can be determined once the distribution of segment lengths is known.

By the symmetry of the ring, the segment lengths are identically distributed. Furthermore [Kao90], these lengths are independent random variables as $b \to \infty$. This is an M/M/1 queuing system [All78]. The departure rate is $1/\tau$: whenever a segment includes more than two loops, the bubble can move. The expected length of an M/M/1 queue is $(1-p_1)/p_1$; accordingly, the expected length of a segment is $1/p_1$. Since the average segment length must be $n/b = s$ by the definition of segments, we have $p_1 = 1/s$. From this, we have $T = \tau s^2/(s-1)$ and

$$\lim_{n\to\infty} U \quad = \quad \frac{s-1}{s^2}\frac{E[t_f]}{\tau}. \tag{4}$$

From this result it follows that choosing $s = 2$ maximizes the asymptotic value of U, and that this maximum value is $U = E[t_f]/4\tau$. As mentioned above, this is a lower bound for utilization, because the simplifications made for the analysis were conservative. For any fixed s, linear speedup is realized.

We have shown that asymptotically good utilizations can be realized even when the worst–case processing time is unbounded. In contrast, the clock period of a synchronous system must be longer than the worst–case delay. The above result demonstrates that a self–timed pipeline can operate with performance determined by

the average processing time instead of the worst–case.

2.7 Exponentially Distributed Processing Times with Fixed Offset

We now show that a self–timed pipeline can achieve high utilization even when processing times are unbounded. We consider processing times that are exponentially distributed with an offset:

$$\mu(t) = \begin{cases} 0 & , t < t_0 \\ \lambda e^{-\lambda(t-t_0)} & , t \geq t_0 \,. \end{cases}$$

To simplify the analysis, we assume $t_c = 0$. (The analysis of section 2.6 corresponds to $t_f = 0$ and t_c exponential with parameter τ.)

We cannot give exact analytic results for this case; instead, we derive approximate results for three cases depending on the spacing of bubbles, and we present the results of Monte–Carlo simulations that confirm these approximations.

1. Close spacing: $s \ll \lambda t_0$. Because there are many bubbles, a loop will most often proceed directly from the *right* to the *down* state without waiting. In the limiting case as s goes to zero, the average time spent in the *right* state is zero. By arguments similar to those in section 2.4, $(s-1)T_{\text{down}} = E[t_f]$. Occasional waiting in the *right* state can only slow down the pipeline; thus,

$$U \leq \frac{s-1}{s} \quad , s \ll t_0 \,. \tag{5}$$

   Monte–Carlo simulation shows that this bound is close to the actual utilization.

2. Intermediate spacing: $s \approx \lambda t_0$. Both waiting for new data and waiting for bubbles occur frequently. Based on Monte–Carlo simulation, we observed that

$$U \leq \frac{\lambda t_0}{\lambda t_0 + 1} \quad , s \approx t_0 \,. \tag{6}$$

   is an upper bound and reasonable approximation. This is the region of maximum utilization.

3. Large spacing: $s \gg \lambda t_0$. Waiting for bubbles to arrive is the primary cause of lost utilization. We present a more detailed analysis of this case below.

We compute an estimate based on the rate of flow of bubbles around the pipeline for the case that $s \gg \lambda t_0$. Since t_c is zero, a loop is a bubble for a positive amount of time only if it becomes a bubble while its predecessor is still computing a new result. Estimating the actual waiting time between computations with the average, T, this happens with probability approximately $e^{-\lambda(T-t_0)}$. When a bubble stops at a stage, the average length of the stay is $1/\lambda$. This yields:

$$T \approx \frac{s}{\lambda} e^{-\lambda(T-t_0)} .$$

It is easy to verify that in the limit as s goes to infinity

$$T \approx t_0 + \frac{1}{\lambda} \log \left(\frac{s}{\log s + \lambda t_0} \right) .$$

Since this assumes the bubble always arrives after t_0, this analysis provides a lower bound on average waiting time, and thus an upper bound on utilization. In the limit as $s \to \infty$, $T \to \infty$, and the bubble arrives after t_0 with probability approaching one. This yields

$$U \approx \frac{\lambda t_0 + 1}{\lambda t_0 + \log \left(\frac{s}{\log s + \lambda t_0} \right)} \quad , s \gg \lambda t_0 . \tag{7}$$

These three upper bounds provide a good approximation to the utilization of the self–timed pipeline. Figure 4 shows data obtained from Monte–Carlo simulations along with the three upper bounds for a pipeline with $t_0 = 0.9$, $\lambda = 10$, (i.e. mean 1.0, variance 0.1, processing times) and $t_c = 0$. For distributions with λt_0 sufficiently large, utilizations arbitrarily close to 100% can be realized, even though the worst–case processing time is unbounded.

2.8 Conclusions

We have shown that self–timed pipelines can achieve linear speedup with utilizations close to 100%, under a wide variety of processing time distributions, including

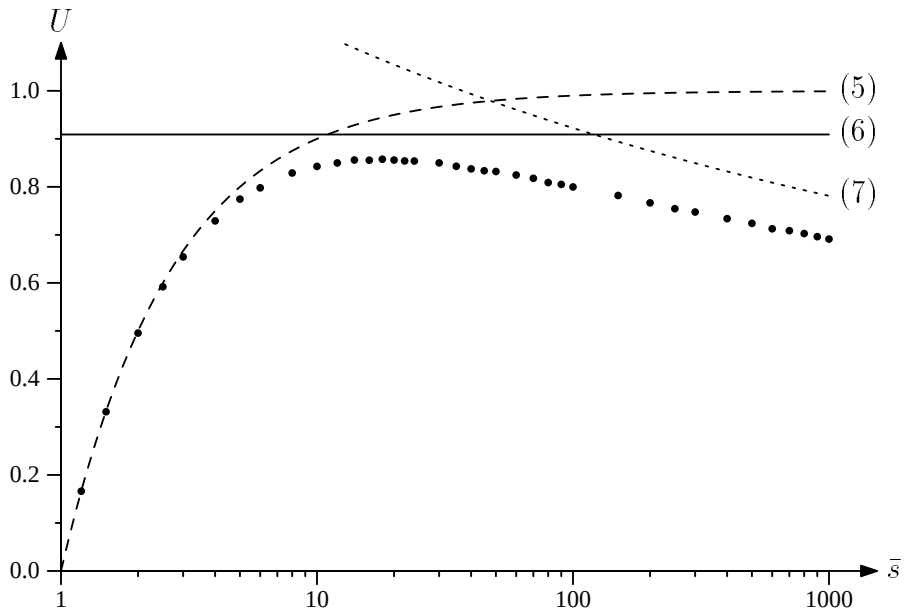


Figure 4: Utilization, U, vs. bubble spacing, $\bar{s}$, with $\lambda t_0 = 10$. Curves for upper bounds are labeled by their equation numbers. The dots are the results of Monte Carlo simulation.

unbounded distributions. The variation in processing time is absorbed by bubbles, processors which are temporarily idle. By introducing bubbles, the pipeline can operate at a rate which is closer to the average processing time than the worst–case.

The theorems presented in appendix A provide a general framework for analyzing the throughput of self–timed pipelines. In particular, using corollary 3.2, performance bounds (both upper and lower) for many other processing time distributions can be derived from the results presented in sections 2.4 through 2.7. The C-element protocol has a natural extension to higher dimensions. The bounds from section 2.5 (bounded distributions) also apply to this case. In particular, assuming bounded processing and communication time, two–dimensional processor networks can also realize linear speedup. For higher dimensional networks embedded in a two–dimensional implementation (e.g. a chip or printed circuit board), the delays of non–nearest neighbor communication may limit performance to less than this.

The techniques presented here can also be applied to synchronous designs. The C-element ring can be used instead of a buffer chain to distribute clock pulses in a pipeline with locally synchronous stages. Feedback between stages in the C-element

protocol guarantees correct functioning independent of all delays; thus, this design does not suffer from the limitations of buffer chains described in [GS90, DS90]. This approach was proposed in [Su89] without performance analysis. It combines the simplicity of synchronous design (for the individual stages) and the robustness of self–timed designs (for interstage timing). The analysis presented in this chapter shows that such a pipeline can achieve linear speed–up.

Like the other studies mentioned in the introduction to this chapter [Bur91, Wil91, Thi91a], the analysis presented here assumes that all connections between circuit elements are roughly equivalent. In practice, this is not the case [Bak90]. Wires on a single chip can be driven much faster than the wires on a circuit board that connect separate chips. Likewise, circuit board wires can be driven much faster than the wires of the backplanes that connect boards in a rack, and similar observations apply to the cables that connect rack, etc. The performance of a self–timed pipeline based on the protocols described in this chapter is limited by the longest latency interconnect. Thus, even though the circuits on a single chip are capable of operating at very high rates, these circuits are idle most of the time, waiting for off–chip communication. Typically, self–timed designs are most successful in applications where the ratio of on–chip computation to off–chip communication is large (e.g. [WH+87]).

In a self–timed circuit, each data transfer requires two communications to be performed sequentially: first, the receiver makes a request; then, the transmitter sends the data. It is this "round–trip" for each transfer that limits the performance of pure self–timed designs with significant off–chip communication. The observation that leads to STARI is that the transmitter does not necessarily have to wait for a request before sending new data. Any condition that guarantees that new data will only arrive at the receiver after the receiver makes its request is sufficient.

Chapter 3

STARI

This chapter presents a novel approach to hardware communication called STARI: *Self–Timed At Receiver's Input*. As illustrated in figure 5, STARI combines synchronous and self–timed design techniques to produce an interface that can operate at higher bandwidths than either purely synchronous or purely self–timed designs. The key component of STARI is a self–timed FIFO that receives data from a synchronous transmitter and delivers data to a synchronous receiver. The transmitter and the receiver operate at the same clock rate; however, STARI can tolerate large skews. During each cycle (period of the global frequency reference), the transmitter sends one completion encoded datum [Sei79] that is inserted into the FIFO upon arrival. Likewise, the receiver removes one item from the FIFO each cycle. Once properly initialized, the FIFO never overflows or underflows.

Correct operation of the STARI circuit requires that the FIFO can complete each insert and remove operation within one cycle. When this requirement is met, the FIFO appears as a synchronous component to both the transmitter and the receiver, and both the transmitter and the receiver appear to the FIFO as well–behaved self–timed systems. The transmitter produces a new data token each clock cycle, just as if the FIFO were another component synchronous to its own clock. Because the FIFO performs each insert operation within one cycle, the FIFO acknowledges (see chapter 2.1) the previous data token before the next token arrives. Thus, the synchronous transmitter satisfies the self–timed signaling conventions of the self–timed

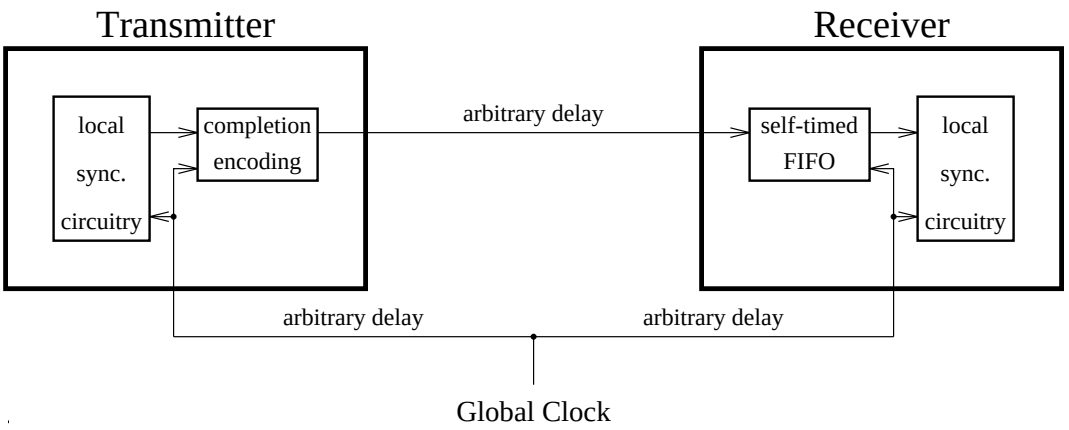


Figure 5: STARI communication

FIFO. Likewise, the prompt response of the FIFO to acknowledge tokens from the receiver guarantees that the receiver does not issue an acknowledge token until the corresponding data token is present. Therefore, the interface between the FIFO and the receiver is correctly timed.

The self–timed FIFO in a STARI interface compensates for clock skew between the transmitter and receiver. Thus, the performance of a STARI interface is not constrained by the skew problems that limit the speed of synchronous designs. Furthermore, the transmitter in a STARI interface does not depend on acknowledge tokens from the receiver. As shown in figure 5, the acknowledge signal from the self–timed FIFO is ignored. Accordingly, the performance of a STARI interface is not constrained by the round–trip delays that limit the speed of self–timed designs. As will be described in chapter 4, a STARI interface can operate a speeds of only a few gate delays per data transfer.

FIFO's are often used to interface circuits operating with independent clocks. Typically, the FIFO generates flow control signals for the transmitter and receiver. These signals direct the transmitter to suspend sending if the FIFO is nearly full, and the receiver to stop requesting data if the FIFO is nearly empty. These signals must be synchronized to the clocks of the transmitter and receiver respectively, and this introduces the possibility of synchronization failure [CM73]. Instead of using

flow control signals, static timing analysis guarantees that the FIFO in STARI can neither overflow nor underflow. This allows STARI to operate without the possibility of synchronization failures and at higher rates than traditional FIFO based designs.

Other communication schemes that depend on close or exact frequency matching at the transmitter and receiver but are tolerant of phase variations have been implemented [RG85, HN86, JH88, AMD89]; these schemes use clock forwarding or employ a phase–locked loop to regenerate the transmitter's clock at the receiver. The STARI technique offers performance comparable to these approaches; because of its simplicity, STARI can be used in applications where phase–locked methods are less practical. For example, STARI circuitry can be incorporated into the pad drivers and receivers of a VLSI chip, offering much higher chip–to–chip communication bandwidths than other signaling techniques. Another consequence of STARI's simplicity is that it can be formally proven to be correct. Appendix B presents such a proof. The author of this dissertation is not aware of any formal correctness proofs for other approaches.[1]

3.1 The schedule

The heart of a STARI interface is a self–timed FIFO. A FIFO is a very simple pipeline where the function computed at each stage is the identity function (i.e. the data input of each stage's C-element is connected directly to the input of the stage). Many of the observations and results of the previous chapter are useful here. The correct operation of a STARI interface depends on the timely operation of the FIFO: each data value from the transmitter must be inserted into the FIFO before the next one arrives, and the FIFO must output a value before the receiver applies the corresponding acknowledge. Consider what happens when a value arrives at the input of the FIFO. The time at which this value is inserted into the FIFO depends on three things:

1. the time for an enabled C-element to perform its operation;

2. the time at which the data value arrives;

[1] Gordon has published a correctness proof for the network interface chip used in the Cambridge Fast Ring [GH85]. Gordon's proof assumes that bits from the ring behave like other synchronous signals and does not consider the off–chip phase–locked loop used for clock extraction(see [HN86]).

3. and the time at which the input stage receives an acknowledge signal from the second stage of the FIFO for the previously input value.

Of course, the time at which the second stage acknowledges the previous value, depends on when the third stage acknowledges the value before that, and so on. Accordingly, the time at which a value is inserted into the FIFO depends upon the times of operations for all of the FIFO stages and the times of the acknowledge signals from the receiver. To verify the timing of a STARI interface, a global schedule for operations is required. In this section, three such schedules are presented. The first schedule assumes fixed delays for each C-element operation and exact periodicity of the transmitter and receiver (no variations of skew). Although this model is unrealistic, it presents the essential features of the more sophisticated models in a simple, intuitively appealing way. Next, a schedule with bounded delays for C-elements and fixed skew is presented. Finally, a schedule where both C-element delays and the skew may vary is described. Appendix B presents a formal proof for the case with bounded C-element delays and fixed skew. This proof applies to the fixed delay model as a special case. A proof for the bounded delay case is similar but requires much more formal apparatus.

3.1.1 Fixed delays

Let the C-elements of the FIFO be numbered from 1 to n. The input of the FIFO is called $\mathtt{y}(0)$, the output of the i^{th} FIFO stage is called $\mathtt{y}(i)$, and the receiver generates an acknowledge by copying the value of $\mathtt{y}(n)$, the output of a FIFO, to $\mathtt{y}(n+1)$, the output of a latch controlled by the receiver's local clock. The FIFO has n stages, and $n+1$ transfers occur between the output of a value by the transmitter and the arrival of that value at $\mathtt{y}(n+1)$, the receiver's latch.

Consider the case when period of every cycle of the transmitter and receiver is both exactly equal to π (some constant, not necessarily 3.14159265...lustra). Then, the time from when the transmitter applies a data token to C-element 1 to when the receiver generates an acknowledge token for the same data token (not necessarily the next acknowledge if there is more than one datum in the FIFO) is fixed. This

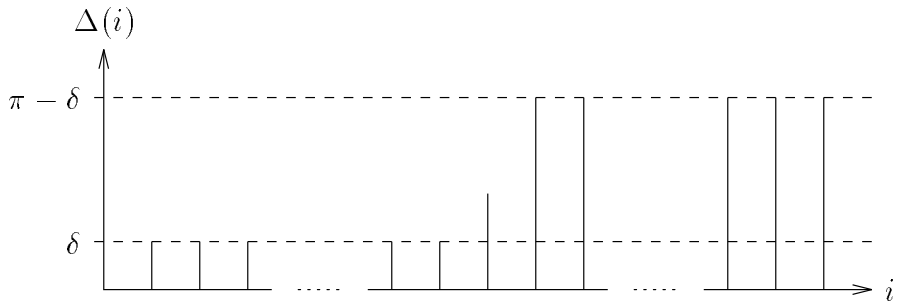


Figure 6: Stage–to–stage transfer times

time is called the *latency* of the FIFO and is denoted by λ. When in addition each FIFO operation has a fixed delay, δ, then each internal state transition of this FIFO occurs exactly once every π time units. Thus, the delay between a data token arriving at C-element $i-1$ and its transfer to C-element i is fixed. This delay is called the **transfer delay** and is denoted by $\Delta(i)$.

Figure 6 shows $\Delta(i)$ for a typical STARI interface with fixed delays. This data could be obtained, for example, by simulation of the circuit. The key observation from this figure is that the FIFO can be divided into two regions: in the left region (the region that includes the first stage of the FIFO), each stage has a transfer delay of δ; in the right region, each stage has a transfer delay of $\pi - \delta$. There is one stage between these two regions with a transfer delay that is between these two values. This behavior has a simple explanation. C-elements in the left region receive new data tokens after they receive the acknowledge tokens; we say that they "wait for data." Accordingly, the data token is transferred δ time units after its arrival.

C-elements in the right region receive data tokens before they receive the acknowledge token for the previous data token. These stages wait for bubbles from the receiver. Let i be in this region. Consider what happens when the C-element i receives an acknowledge from C-element $i+1$ at time ν (see figure 7. τ denotes the current time).

$\tau = \nu$: C-element i receives an acknowledge token from C-element $i+1$.

$\tau = \nu + \delta$: C-element i outputs the data token currently output by C-element $i-1$.

Assume that the stage that updates `x(i-1)` is also waiting for an acknowledge

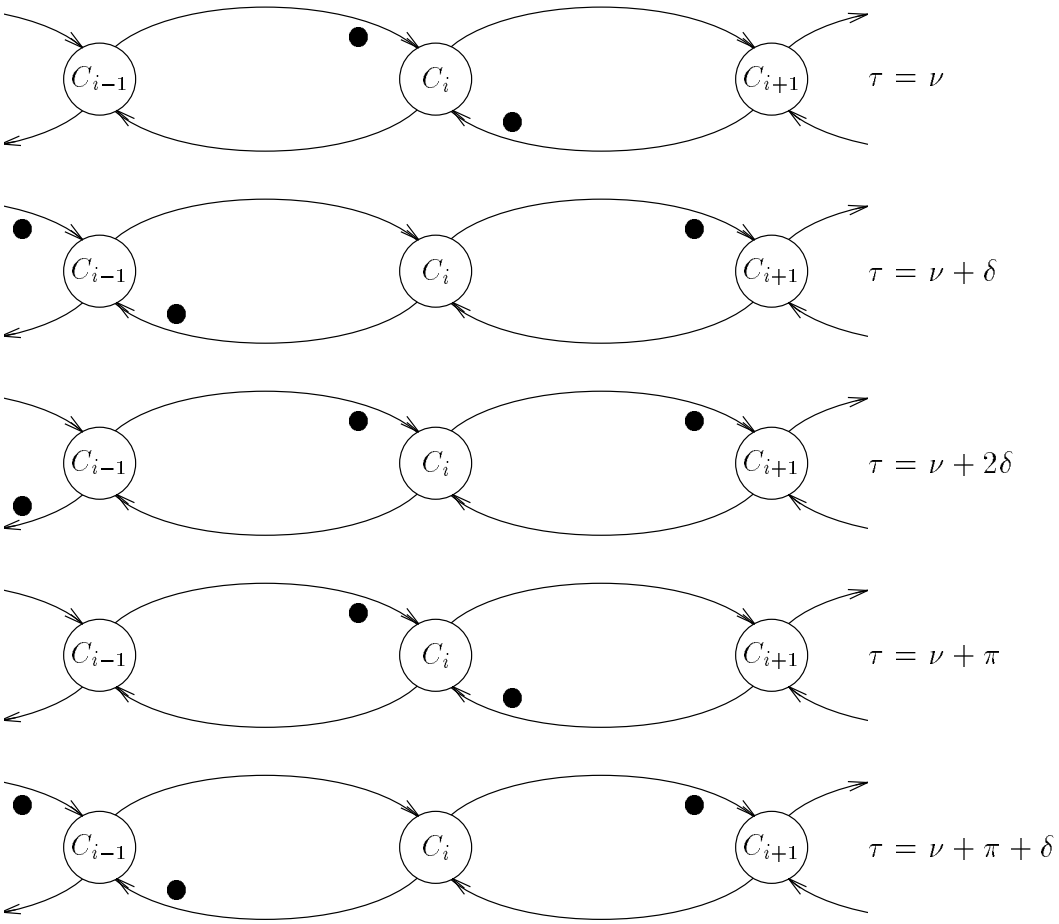


Figure 7: Waiting for bubbles

token...

$\tau = \nu + 2\delta$: C-element $i-1$ outputs a new data token. By the assumption that internal transitions are periodic with period π...

$\tau = \nu + \pi + \delta$: C-element i outputs the new data token output by C-element $i-1$.

Thus, the transfer delay from `x(i-1)` to `x(i)` is $\pi - \delta$ as observed in figure 6.

The final case is the stage between the two regions. This stage also waits for bubbles, but its predecessor waits for data. Thus, C-element $i-1$ outputs a new data token more than δ time units after the previous data token was output by C-element i. Since data tokens arrive later (relative to the time of the previous transfer), they don't wait as long as in the case described in the preceding paragraph.

These observations make it possible to determine the transfer delay of each stage given the clock period π, the FIFO latency λ, and the C-element delay δ. In particular, there are $n+1$ data transfers between applying a data token to `y(`$n+1$`)` and the arrival of that token at `y(`$n+1$`)` Let k be the number of these transfers with delay δ. Then, $n-k$ transfers have delay $\pi - \delta$ and one transfer has delays between these two values. A simple pigeon–hole argument yields

$$k \;=\; \mathtt{n} - \left\lfloor \frac{\lambda - (n+1)\delta}{\pi - 2\delta} \right\rfloor \tag{8}$$

and

$$\Delta(i) \;=\; \begin{cases} \delta & \text{, if } i < k \\ \lambda - k\delta - (n-k)(\pi - \delta) & \text{, if } i = k \\ \pi - \delta & \text{, if } i > k. \end{cases} \tag{9}$$

In particular, a valid schedule is possible if

$$\begin{aligned} &\pi > 2\delta \\ \text{and}\quad &(n+1)\delta < \lambda < (n+1)(\pi - \delta)\,. \end{aligned} \tag{10}$$

The implications of these observations on the implementation of a STARI interface are discussed in section 3.2.

3.1.2 Bounded C-element delays

The model of fixed delays for C-element operations and exact periodicity of the transmitter and receiver used in the previous section is unrealistic. However, its simplicity revealed key properties of STARI and provides the basis for verifying STARI with more realistic assumptions.

Consider a FIFO with bounded delays; a C-element stage takes time *at most* δ to fire once it has received a new data token and a new acknowledge token. Any delay from zero to δ is allowed. Furthermore, assume that the transmitter and receiver have exact periodicity as in the previous section. This means that the latency of the FIFO is fixed. This model closely resembles many real systems where variations in the clock period from one cycle to the next are very small. It is shown that the same bounds apply as for the fixed delay case.

When the C-element delays vary, the exact schedule of the previous section no longer applies. Obviously, the delay to transfer a value from one stage to the next may be lower than in the fixed delay case. Furthermore, as the total latency of the FIFO remains fixed, transfer delays can also be greater than in the fixed delay case. However, the *total* time for a data token to propagate from the input to any given stage is no greater than in the fixed delay case. This can be seen by an argument analogous to the one given for the proof of theorem 2 in appendix A. The only situation in which a transfer delay is greater in the bounded delay model than in the fixed delay model is when a data token arrives early and must wait longer for a bubble.

The schedule for the FIFO with bounded delays is a schedule for the total delay from the time that a data token arrives at the input of the FIFO until it is output by C-element i as given by $\Psi(i)$ defined below:

$$\Psi(i) \;=\; \sum_{j=1}^{i} \Delta(i)\,. \tag{11}$$

Because Ψ is derived from Δ, this schedule identifies a "waiting for data" region ($i < k$ with k defined by equation 8) and a "waiting for bubbles" region ($i \geq k$). In a realization, a stage in the "waiting for data" region may end up waiting for a bubble

because of a data token arriving early; however, it will not wait longer than the time allowed by the "waiting for data" schedule from equation 11. Likewise, a stage in the "waiting for bubbles" region may wait for a data token, but not so long as to violate the schedule.

The arguments given here and in the previous section are informal, operational, and should evoke the reader's skepticism. In appendix B these arguments are made rigorous. The STARI interface is described as a concurrent program. The property that the FIFO maintains a schedule is a real–time property that is formulated as an invariant. This invariant is verified using simple assertional arguments. Since the fixed delay model of the FIFO is a special case of the bounded delay model, verifying the bounded delay formulation also establishes correct operation under the fixed delay model.

3.1.3 Bounded skew

In the previous two section, the transmitter was assumed to output exactly one data token every π time units. Likewise, the receiver was assumed to be exactly periodic. Although it is possible to build extremely stable clock generators, no clock generator can be absolutely precise[2]. Furthermore, the path from the global clock through the transmitter to the data input of the FIFO includes circuits whose delays may vary with temperature, power supply voltage or data values. Likewise the total delay from the global clock to the acknowledge input of the FIFO may vary as the FIFO operates. In this section, bounded variations of the relative skew between the transmitter and receiver are considered. Three approaches are considered: an "engineer's" steady–state argument, an argument by construction and analogy, and a formal proof.

The steady–state argument notes that the STARI interface operates correctly for any fixed skew in the interval described by equation 10. In practice, operating conditions such as the ambient temperature and the power supply voltage change very slowly in comparison with the time to transmit a single data token. Thus, one could postulate that the interface is never any significant distance from the fixed skew

[2] A simple argument from quantum mechanics applies, as the product of the uncertainity in the time of an event and the energy of the event must be greater than Plank's constant [FLS65].

scenarios described in the previous sections, and that these analyses apply to slowly changing skew as well. This approach admits to large variations in the skew, but it does not take into account dependencies of delays upon data values.

In the second approach, one adds a virtual first stage in front of the FIFO. Some of the delay of the path from the global clock through the transmitter to FIFO input is attributed to this stage. This virtual stage is guaranteed to always wait for data tokens (so it's operation is independent of the acknowledge signal from the real FIFO) The arguments of the previous section apply, and the bounded delay model allows this virtual stage to delay successive tokens by different amounts (from 0 to δ). This corresponds to variations of the arrival time of a data token without any assumption that the variations occur slowly. A symmetric arguments adds a virtual last stage behind the FIFO to account for variations in the arrival time of acknowledge tokens.

Both of the preceding approaches lend intuitive credibility to the claim that a STARI interface can operate correctly in the presence of varying skew. Neither constitutes a proof. Using proof techniques akin to those presented in appendix B, it is possible to verify rigorously that STARI is tolerant to dynamic variations in skew as well as variations of C-element delay. As in the previous two cases, a schedule is constructed that has "waiting for data" and "waiting for bubbles" parts. Unlike the previous two cases, the border between these two regions can change as the FIFO operates; this corresponds to variations of the clock skew. When a C-element fires, either its data input is from a "waiting–for–data" stage in which case its output will satisfy the "waiting–for–data" schedule, or its acknowledge input is from a "waiting–for–bubbles" stage, in which case its output will satisfy the "waiting–for–bubbles" schedule. The formal proof is similar to the proof presented in appendix B. Sufficient conditions to guarantee correct timing are summarized below:

1. The time between the arrival of any two successive data tokens at the input of the FIFO is at least 2δ.

2. The time between the arrival of a data token from the transmitter at the input of the FIFO and removing it from the output must be at least $(n+1)\delta$.

3. The time between the arrival of any two successive acknowledge tokens at the

output of the FIFO is at least 2δ.

4. The time between the arrival of an acknowledge token at the output and its removal at the input of the FIFO must be at least $(n+1)\delta$.

Conditions 1 and 3 correspond to the minimum cycle time of the FIFO; conditions 2 and 4 correspond to the minimum propagation delay (ripple through) of the FIFO. Conditions 2 and 4 may appear to require some kind of handshaking between the transmitter, receiver and FIFO. This is not the case; it is sufficient to show that the distribution network for the global clock guarantees these properties.

3.2 Engineering considerations

The arguments of the previous section that show that the FIFO satisfies the required schedule also provide guidelines that are useful when implementing such an interface. As shown in equation 10, π, the period of the clock for the transmitter and receiver, must be greater than 2δ; this is the minimum cycle time of the FIFO.

The latency of the FIFO is λ, and equation 10 gives bounds for λ:

$$(n+1)\delta \;<\; \lambda \;<\; (n+1)(\pi-\delta)\,.$$

The lower bound corresponds to the minimum time to propagate a value from the stage 1 of the FIFO to stage n. and the upper bound corresponds to the minimum time to propagate a bubble from the stage n to stage 1. The range between these two bounds gives Λ, the tolerance of the interface to variations in skew

$$\Lambda \;=\; (n+1)(\pi-2\delta)\,. \tag{12}$$

This equation suggests three ways to increase skew tolerance: increase π (slow down the clock), decrease δ (build faster circuits), or increase n (build a bigger FIFO). Each of these observations is intuitively plausible; equation 12 quantifies this intuition.

The occupancy of the FIFO, N, is the number of data values in the FIFO. If the skew is fixed, N alternates between two values increasing by one when a value arrives from the transmitter, and decreasing by one when a value is acknowledged by the

receiver; if the skew varies, N can take on a wider range of values. The time–average of N is denoted by $\bar{N}$, and in either case $\bar{N} = \lambda/\pi$ (this is a special case of Little's formula for queues [Kle75a]). Because the time allowed for a data value to propagate from the data input of stage 1 to the data output of stage n is at least $(\mathtt{n}+1)\delta$; the clock period π must be greater than $\Pi_d(\bar{N})$ as defined below

$$\Pi_d(\bar{N}) \;=\; \frac{(n+1)\delta}{\bar{N}}\,.$$

Conversely, the time allowed for a bubble to propagate from the acknowledge from $\mathtt{y}(n+1)$ to $\mathtt{y(0)}$ must be at least $(n+1)\delta$, therefore, the clock period π must be greater than $\Pi_b(\bar{N})$ as defined below

$$\Pi_b(\bar{N}) \;=\; \left(1 + \frac{\bar{N}}{n-\bar{N}}\right)\delta\,.$$

These two bounds are equivalent to the bounds for λ from equation 10. Let

$$N_0 \;=\; \frac{\mathtt{n}+1}{2}\,.$$

If $\bar{N} \le N_0$, then the Π_d bound is tight, and the FIFO is said to be *data–limited*. Conversely, if $\bar{N} \ge N_0$, then the Π_b bound is tight, and the FIFO is said to be *bubble–limited*.

Figure 8 illustrates these bounds. For a given clock period, p, the intersections of the $\pi = p$ line with the curves for Π_d and Π_b are the endpoints of the interval of feasible operation. As p is increased, this interval becomes wider, and as p is decreased, this interval becomes narrower, vanishing at $p = 2\delta$, the minimum clock period.

For many applications, the exact latency of the FIFO is not critical. For a fixed offset between the transmitter and receive clock phases, the FIFO latency must be congruent to this offset modulo the clock period. When the interval of feasible operation is wide enough, there may be several latencies that satisfy this requirement. In this situation, it is desirable to initialize the FIFO to an occupancy near the center of the interval of feasible operation. This maximizes the STARI interface's robustness to variations in skew. The center of this interval corresponds to a FIFO occupancy of

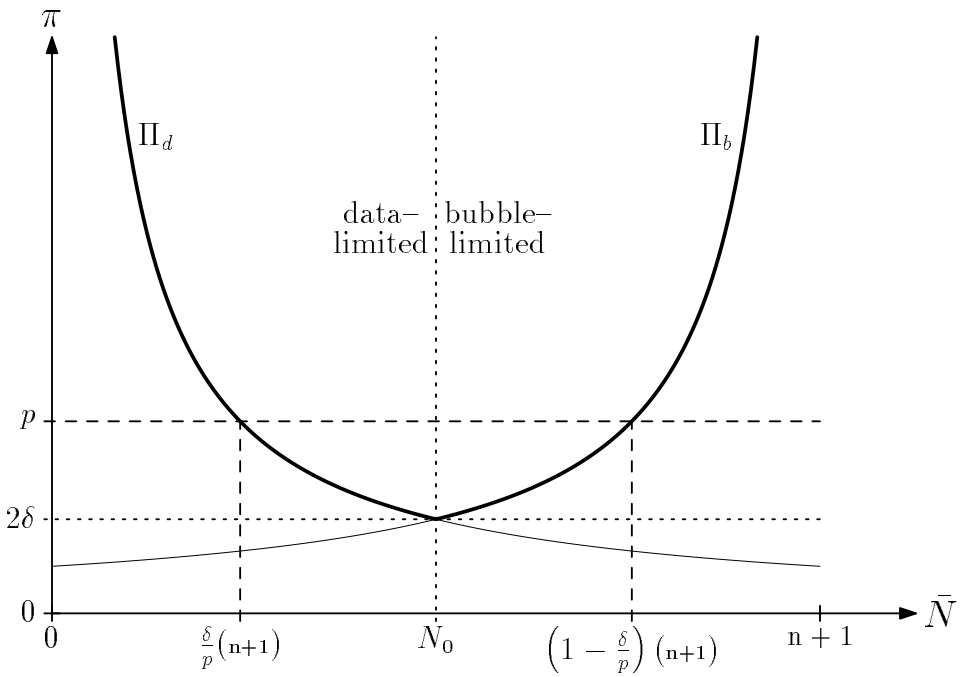


Figure 8: Minimum cycle time vs. FIFO occupancy

$n/2$. This suggests that the FIFO should be initialized to a state where it is roughly half–full. This observation is used in the design of the initialization circuitry of the FIFO chip as described in chapter 4.2.

3.3 Where did the synchronization errors go?

A STARI interface provides reliable communication between two processors operating at the same clock frequency in spite of large and even varying skew. In fact the skew can vary by many clock cycles (if Λ as defined by equation 12 is sufficiently large) *while the interface is operating.* Large timing uncertainities such as these are traditionally associated with synchronization failures [CM73]. This section examines the absence of such synchronization failures in STARI.

Most important, STARI is known to be free of synchronization failures because it has been formally proven (see appendix B) that the timing requirements of the transmitter, receiver and FIFO are all satisfied. One cannot prove a communication

technique to be free of synchronization failures by implementing an instance of the design and making laboratory measurements. Synchronization failure is a probabilistic phenomenon. Accordingly, one could build an interface where the probability of failure is extremely small (e.g. one failure per 10^{1000} bits transferred) and never observe a failure. However, another implementation of the same technique could have a much higher probability of failure (perhaps one failure per 1000 bits transferred) and be plagued errors. The correctness proof for STARI is the only compelling argument that it is free from synchronization failures.

Synchronization failures arise in circuits that must "make a decision." Typically, this decision involves some sampling event. If a signal changes sufficiently before the sampling event, then the circuit has one behavior, if the signal changes sufficiently after the sampling event, then the circuit has another behavior. Even though these signals have discrete interpretations, the representations of these discrete values (e.g. voltages) are continuous quantities. By the mean–value theorem [Apo67], there must be some intermediate time at which the signal can change that produces an intermediate behavior that doesn't have a valid digital interpretation. See [Cha84] for a thorough exposition of this argument.

The self–timed FIFO of a STARI interface does not perform any sampling events in the sense described above. A FIFO stage waits until it has *both* a data token and an acknowledge token before it performs any action, and it performs the same action regardless of the order in which the tokens arrive. If the delays through the transmitter increase (or the delays through the receiver decrease), more stages will wait for data, but they will process exactly the same sequence of values, only later. Likewise, if the delays through the transmitter decrease, more stages will wait for bubbles, but the sequence of values is unchanged. As long as the skew remains in the bounds specified by equation 10, the sequence of values processed by each stage and then input by the receiver is independent of the skew.

The formal proof in appendix B verifies an invariant of the STARI interface. It does not consider initialization, and this offers an opportunity for synchronization failures. Two initialization schemes are described below: the first is a simple, practical scheme that requires a single synchronization event when the interface is initialized.

After initialization, no further synchronization is required. The second scheme is less practical, but it demonstrates that even initialization can be performed without synchronization.

As mentioned in section 3.2, the goal of initialization is to reach a state where the FIFO is half–full and the transmitter and receiver are each performing one operation per period of the global clock. The most direct way to implement this is to include a reset signal that forces the FIFO to a half full state. To start communication, the transmitter sends the first datum, waits m clock cycles (for some suitably large m) and then sends one value each period of the clock. The receiver detects the first datum and synchronizes this detection to its version of the global clock. After $m-1$ cycles, the receiver starts removing one datum from the FIFO each clock period. This produces the desired state of a half–full FIFO with the transmitter and receiver each performing one operation every period of the global clock.

In the scheme described above, it is the estimate of the skew modulo the clock period that introduced the need for a synchronization. To see this, observe that the time of arrival of the first datum from the transmitter can vary continuously, but receiver's synchronizer must choose a discrete cycle on which to start removing data from the FIFO. The mean–value argument says that there must be a time at which the first datum can arrive such that the receiver fails to exhibit valid digital behavior (e.g. applies a "half" acknowledge to the FIFO).

For many synchronizers, the probability of synchronization failure drops exponentially with the time allowed for synchronization [Sei79]; by chosing m, the number of cycles between the first and second datum sent by the transmitter, sufficiently large, the probability of synchronization failure can be made arbitrarily small. Observe that the pause of m cycles only occurs during initialization. Once the STARI interface is initialized it can operate at the full clock rate and with a latency that is independent of m.

To eliminate the synchronization during initialization, the cycle on which the receiver starts removing data from the FIFO must not depend on the skew. This can be done by starting the system with a slow clock. In particular, the clock should be slow enough that the transmitter and receiver can operate as a single synchronous system.

Initialization is performed synchronously at this low clock rate. Once initialization is completed, the clock rate is increased to the high–speed rate. This can be done maintaining the restrictions presented in section 3.1.3; therefore, it is provably free of timing hazards.

Although the second scheme has the aesthetic advantage of requiring no synchronization, it is less practical than the first approach. First of all, it requires a clock generator that can run at two different speeds. Second, it requires the FIFO to be large enough to absorb the absolute skew between the transmitter and receiver. In the first scheme, the FIFO only had to compensate for variations in the skew as the circuits operate plus one period of the global clock. In general, the second scheme will require a larger FIFO and have higher latency than the first approach.

3.4 Conclusions

This chapter presented STARI, a new approach to hardware communication. Like many local area networks (e.g. Ethernet [MB76], the Cambridge Fast Ring [HN86], Autonet [SB+90]), STARI allows multiple bits to be in flight between the transmitter and receiver at the same time, and the exact delay of the interconnection between the transmitter and the receiver is not critical. All of these must align the incoming data to the receiver's clock. Most existing networks do this by one of two methods:

Oversampling the input to identify the transitions between successive bits. This approach is taken by Ethernet, RS-232, the BBN Butterfly [RG85], among other schemes.

Phase–locking a local oscillator to the incoming bit stream. The regenerated clock is used to store the data. Once an entire block is received, control of the buffer is transferred to the local processor which can then read the received data. This approach is taken by the Cambridge Fast Ring, and the Autonet, among other schemes.

STARI is novel in that it uses a self–timed FIFO to synchronize incoming data. Unlike oversampling or phase–locking techniques, STARI does not require local circuitry that

can operate at switching rates much greater than the incoming bit rate.

STARI is provably correct. It demonstrates the application of methods from concurrent programming (e.g. invariants [LS84] that express a schedule for operations) can be applied to a timing verification problem that is not amenable to standard techniques. Furthermore, STARI dispels engineering folklore that large uncertainities in skew inevitably lead to the need for synchronization and therefore to an inevitable risk of synchronization failures. With a sufficiently large and fast FIFO, a STARI interface can tolerate variations of skew much greater than the clock period without any possibility of synchronization failure.

STARI is simple. The heart of STARI is a self–timed FIFO which is a concatenation of simple stages. An implementation of a self–timed FIFO is presented in the next chapter. STARI does not require a special fabrication process (i.e. one with carefully controlled analog characteristics). Accordingly, STARI interfaces are suitable for use on a wide variety of chip such as network interfaces, processors, or memories.

The principle disadvantage of STARI is that the data which is sent by the transmitter must be completion encoded so that the self–timed FIFO can distinguish between successive bits. Such a code must have more than two symbols (if it is to carry any information in addition to the completion status). Therefore, the connection between the transmitter and receiver must use more than one wire, or multi–level signaling [Smi88] (e.g. ternary) must be used. If more than one wire is used (the approach presented in chapter 4), then the delays of these wires must be closely matched. This is practical for printed traces for point–to–point interconnection on a circuit board or over a backplane. On the other hand, it is not suitable for a local area network. Completion encoded data can be conveyed over a single wire using multi–level signaling. However, multi–level signaling requires more complicated, analog circuitry than two–level signaling and would make STARI less suitable for implementation using generic, digital fabrication technologies. These considerations suggest that STARI is well suited for high speed communication between chips on a board, or between processors communicating over a backplane. Existing methods (such a fiber optic technology [AMD89]) are probably better suited for communication over longer distances.

Chapter 4

Implementing STARI

This chapter describes the design of a prototype chip to demonstrate the STARI technique. Limited neither by clock skew nor round–trip delays, STARI interfaces can support bandwidths that are limited only by the switching speeds of the circuits used to implement the transmitter and receiver and by the bandwidth of the interconnect. To obtain a very high bandwidth design, many circuit level optimizations are employed

The STARI chip is designed to provide high–bandwidth communications between two conventional (e.g. TTL) systems. The transmitter half of the chip inputs parallel data at clock rates that are suitable for microprocessor based board level design, performs a ten–to–one multiplexing operation, and outputs serial data for transmission to a STARI receiver. Conversely, the receiver inputs high bit rate, serialized data, demultiplexes it, and outputs parallel data. In the following, the global clock that controls serial data transfer is called the **fast clock**. At either end of the STARI interface, the derived clock that controls parallel data transfers is called the **slow clock**.

In the course of designing the STARI chip, various performance bottlenecks were identified, and architectural and circuit level solutions were devised to overcome these bottlenecks and maximize the bandwidth realized by the design. The first bottleneck was the stage–to–stage transfer time of the FIFO. The FIFO is in the receiver half

of the chip, and, as described above, the receiver functions as a ten–to–one demultiplexer. By demultiplexing the serial data *before* applying it to the FIFO, the FIFO can be implemented with ten parallel channels, each of which operates at one tenth the incoming bit rate. The FIFO design is described in sections 4.1 and 4.2.4.

The demultiplxer is a synchronous design (see section 4.2.3). The clock for the demultiplexer is sent from the transmitter along with the data. Since the clock and data signals traverse equivalent paths, the delays of the two signals can be closely matched; there is much less problem with skew than in a purely synchronous design where the clock must be distributed to *many* different destinations along different paths than the data signals. As described in section 4.3, the transmitter consists of two multiplexers, one that outputs the forwarded clock, and the other that outputs data. This is a simple design that exports issues of initialization and facilitates experiments and testing. In a production design, initialization logic would be included on chip, and only one of the multiplexers would be externally visible.

Once the FIFO bottleneck has been eliminated, the interconnect between the transmitter and receiver must be considered. To achieve high bandwidths, it must be possible to have multiple bits in flight between the transmitter and the receiver. This means that the interconnection must be a controlled impedance transmission lines [Bak90], and that the transmitter and receiver must be controlled impedance sources and loads. Furthermore, to avoid excessive power consumption, low voltage swing (one volt) signaling is used between the transmitter and receiver. Circuit simulation using CAzM [EK+89c] (a Spice–like [Nag75] simulator) revealed that the input amplifier that converts one volt swing signals off–chip to five volt swing signals on chip is the bottleneck in this design. A sense–amp style design (see section 4.2.1) is used to improve the performance of the input amplifier.

Figure 9 is a simplified block diagram of the chip that reflects the optimizations described in the preceding paragraphs.

In simulation, the STARI chip sustains transfer rates exceeding 700 Mbits/sec. This corresponds to a fast clock rate of 350 MHz., with separate data transfers during the high and low portions of each clock cycle. The multiplexing factor is ten;

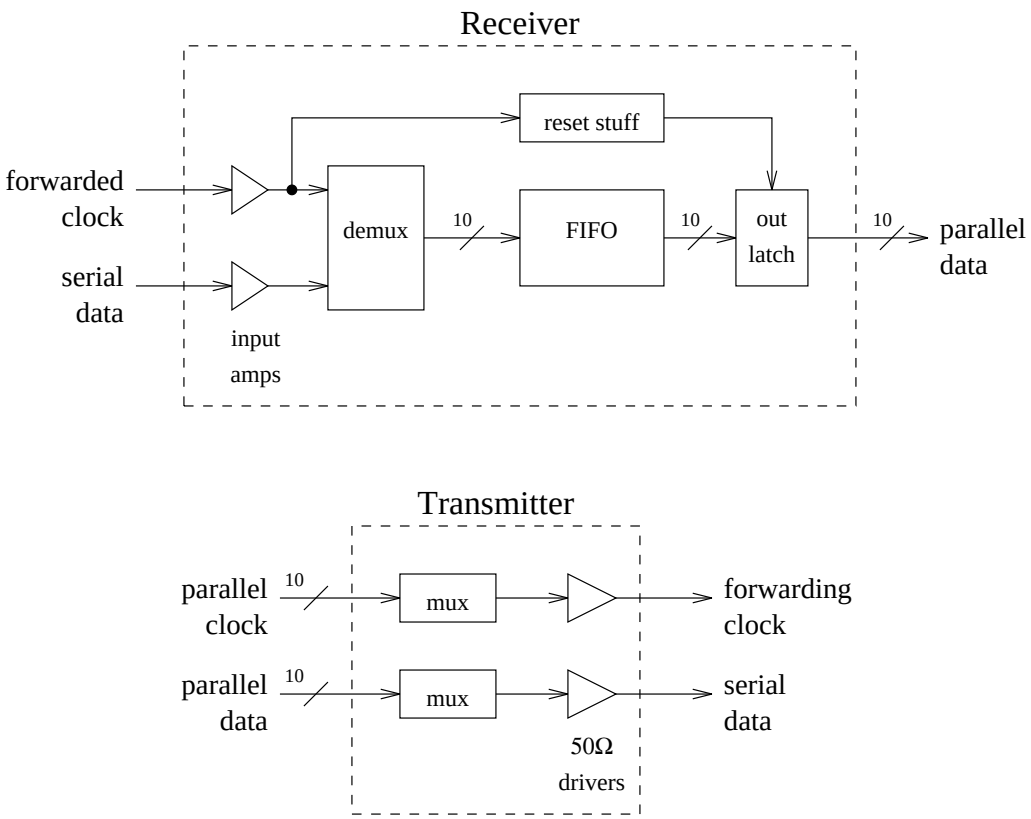


Figure 9: STARI chip block diagram

accordingly, the slow clock rate is 70 MHz. To obtain this performance, each subcircuit of the STARI chip was designed to operate at fast clock rates up to 500 MHz. Chips have been fabricated and tested. Three design errors were revealed: one is a wiring error that was overlooked due to an unfortunate partitioning of the design for simulation; the other two are due to subtle analog effects that were not revealed by the circuit simulator. Using test vectors that compensated for the design errors, correct operation was observed at rates of 100 Mbits/sec., the maximum rate supported by the test equipment.

4.1 Designing a Self–Timed FIFO

The heart of STARI is a self–timed FIFO. As self–timed design are not included in the core of most computer science programs, this section presents some basic techniques for designing a self–timed FIFO. Section 4.2.4 describes the refinements to this design that were included in the design of the STARI chip.

4.1.1 Self–timed codes

A self–timed circuit requires a self–timed encoding [Ver88, BB91] for data values. This code must represent data values (e.g. "true" and "false") and allow successive data values to be distinguished. The prototype chip uses dual–rail coding [Sei79]. Each Boolean signal, `y`, is encoded with two signals, `y.t` and `y.f` as shown below:

`y.t`	`y.f`	`y`
`low`	`low`	`E` (empty)
`low`	`high`	`F` (false)
`high`	`low`	`T` (true)
`high`	`high`	illegal

As a shorthand, we write

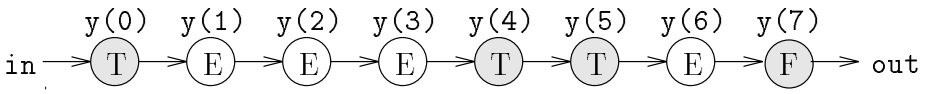
$$\text{valid}(\texttt{q}) \equiv ((\texttt{q} = \texttt{FALSE}) \vee (\texttt{q} = \texttt{TRUE}))$$

"Empty" values function as spacers between "valid" values allowing a circuit, for example, to distinguish between two consecutive true values and one value that is

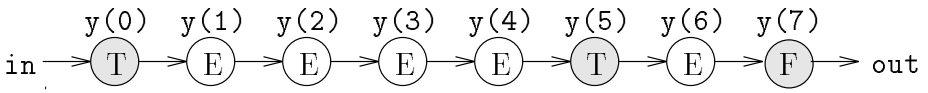
asserted for a long time. In operation, a dual–rail code signal alternates between empty and valid values.

4.1.2 A simple FIFO

When a new value is applied to the input of the FIFO, the value is copied from one stage to the next until a stage is encountered that has not yet acknowledged the previous input. A stage continues to hold a value until the next value of the input sequence arrives and it has an acknowledge from its successor. Accordingly, the same value of the input sequence may be held by many adjacent stages simultaneously. This is what permits the FIFO to hold fewer values than its maximum capacity. For example, a FIFO with `nstages` = 8 holds the sequence `F`, `E`, `T`, `E`, `T` when it is in the state depicted below:



From this state, the fourth stage of the FIFO can fire to produce the state



This state also corresponds to the sequence `F`, `E`, `T`, `E`, `T`, which makes sense, since internal FIFO operations should not alter the sequence of values that are held in the FIFO.

Blocks of adjacent stages that have the same value correspond to a single value in the input sequence. Bubbles are represented by a pair of adjacent stages that hold the same value; data tokens are represented by pairs that hold different values. As illustrated in figure 10, data tokens propagate from left to right (i.e. in the direction of increasing stage indices) and bubbles propagate from right to left in the FIFO.

Figure 11 shows a segment of a self–timed FIFO. Each stage consists of two Muller C–elements that transfer the `.t` and `.f` components of a data token, and a `NOR` gate that computes the acknowledge signal from the stage's output. As illustrated below, when both inputs of a Muller C–element are the same, the output takes on this value, when the inputs are different, the output retains its value.

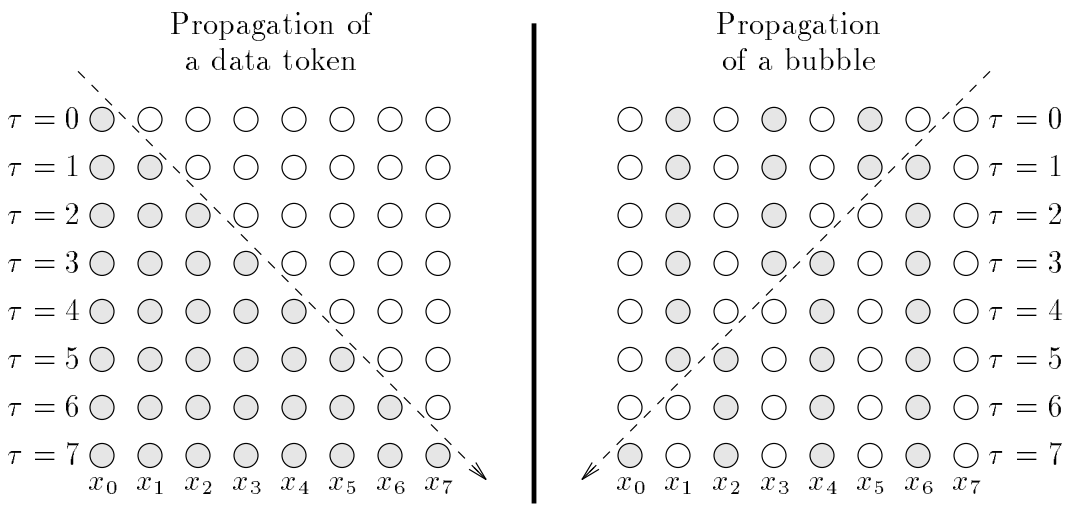


Figure 10: Data tokens and bubbles in a self–timed FIFO

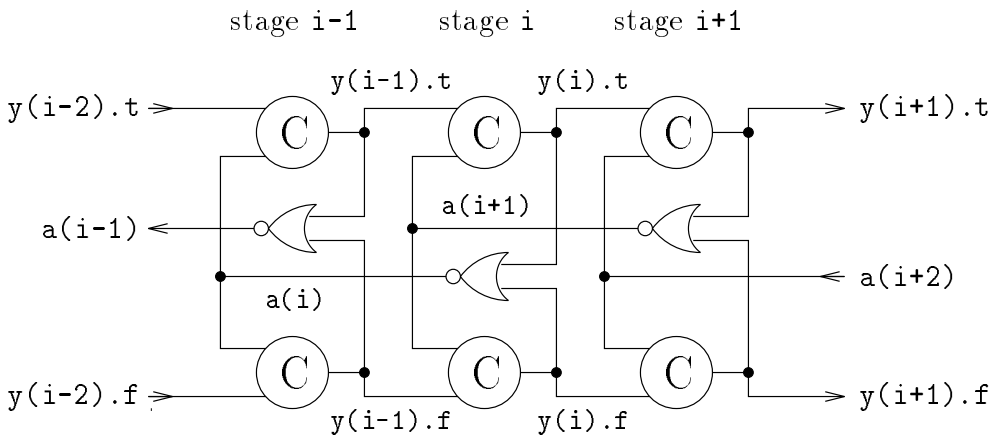


Figure 11: Three stages of a self–timed FIFO

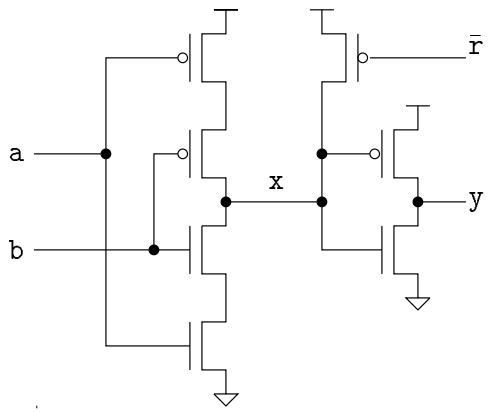


Figure 12: A CMOS implementation of a Muller C–element

Muller C–element		
inputs		output
0	0	0
0	1	unchanged
1	0	unchanged
1	1	1

To understand the operation of the FIFO, consider a state where the value of `y(i-1).t` is `T`, and the values of `y(i)` and `y(i+1)` are `E`. This means that new data is available for stage `i`, and its successor has acquired its current output. Because `y(i+1)` has an empty value, `a(i+1)` will eventually be true. This enables the upper C–element of stage `i`. When this C–element fires, `y(i).t` goes high, and therefore the value of `y(i)` becomes `T`. Thus, the data token that was at `(y(i-1), y(i))` has been transferred to `(y(i), y(i+1))`; likewise the bubble that was at `(y(i), y(i+1))` has been shifted one stage to the left. The transition from a valid value to empty is analogous.

4.1.3 The Muller–C element

Figure 12 shows a common CMOS implementation of a Muller C–element. The signal $\bar{r}$ is the reset input for initialization. Initialization is discussed later; the

current discussion of the operation of the C–element assumes that $\bar{r}$ is high. When both inputs, a and b, are high, the internal node x is pulled low, and the inverter drives the output y high. Conversely, when both inputs are low, x is driven high, and the output goes low. When a and b have different values, the internal node x floats, retaining its value due to the capacitance of the node, and y retains its value. Thus, the circuit exhibits the behavior of a C–element.

The use of dynamic storage on the node x leads to a simple, compact, and fast realization of a C–element. It also requires that issues of refreshing and charge sharing [GD85] be addressed. The internal node x is driven high each time an empty value passes through the stage. A slow clock rate greater than a few kilohertz guarantees adequate refreshing. As the chip is intended for high speed operation, refreshing these nodes is not a problem. Charge sharing occurs if the b input changes before the a input. In this case, the voltage on node x may shift slightly. With standard layout techniques, the capacitance of the node between two series n–channel (or series p–channel) transistors is much less than the drain and gate capacitances of node x, and the small change of voltage on node x will still leave it far from the inverter's threshold. Thus, neither refreshing nor charge sharing present serious problems in this design.

As described in chapter 3.2, the initialization procedure for the STARI interface should produce a state in which the FIFO is approximately half full. This is achieved by first bringing the FIFO into the state where all stages hold the empty value (i.e. a completely empty FIFO). When the transmitter starts sending, the receiver waits long enough for the FIFO to fill half way, and then it starts removing data from the FIFO.

To empty the FIFO, an empty value is applied to the input of the FIFO, and the $\bar{r}$ signal of each stage is driven low. A simple induction argument shows that this brings the FIFO to a state where all stages hold the empty value. The base case is y(0) which is the input of the FIFO and has the empty value by the definition of the initialization procedure. Now, assume that y(i) outputs the empty value. Then, the n–channel transistors of the C–elements whose gates are connected to y(i).t and y(i).f are in the cut–off region (no current flows through them). Therefore, the

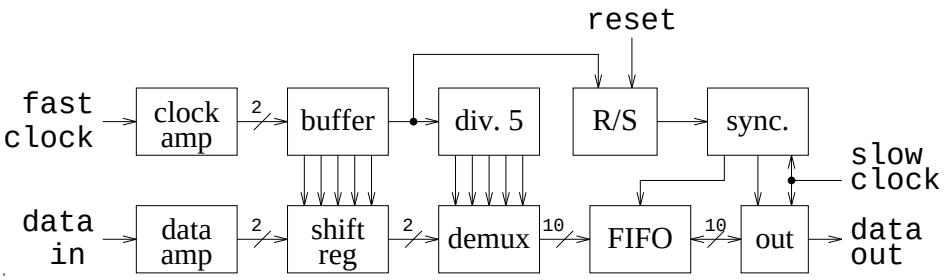


Figure 13: Block diagram of the receiver

p–channel transistors whose gates are connected to $\bar{\mathtt{r}}$ will eventually pull the internal nodes `x.t` and `x.f` high, which produces an empty value on `y(i+1)`. Thus, the FIFO eventually reaches a state in which each stage outputs the empty value.

4.2 The Receiver

Figure 13 shows a block diagram of the receiver portion of the STARI chip. A distinguishing feature of the design is the choice to demultiplex the incoming data *before* it is input to the FIFO. Circuit simulation indicated that the 5 ns cycle time of the FIFO was the critical bottleneck on the chip. As described below, the input amplifier and demultiplexer can operate at rates greater than 1.5 ns per bit. Thus, this organization leads to a three–fold improvement in performance over a design where the data is input directly to the FIFO.

The demultiplexer (which is described in greater detail in section 4.2.3) is based on the high–speed synchronous design techniques of Yaun and Svensson [YS89]. Accordingly, the transmitter forwards the fast clock along with the data it is sending. Since both the clock and data signals traverse the same path, these delays can be very closely matched[1]. Clock forwarding does not address the problems of clock skew between the transmitter and receiver. Although the forwarded clock allows the demultiplexer to be based on synchronous design techniques, the phase relationship (skew) between the forwarded fast clock of the transmitter and the local slow clock

[1]Note that even in a STARI interface that uses dual–rail coded data, the delays of the `.t` and `.f` signals must be closely matched if the design is to operate with multiple bits simultaneously in flight.

of the receiver is unspecified by the design and may vary during operation. Thus, the characteristic self–timed FIFO of a STARI interface appears between the output of the demultiplexer and the synchronous circuitry (e.g. the box labeled "out" in figure 13) of the receiver. As discussed in the previous chapter, the latency of the connection between the transmitter and the receiver is not critical in a STARI interface. Including a multiplexer at the transmitter and a corresponding demultiplexer at the input of the receiver implements the identity function and therefore only adds latency to the connection.

The receiver operates as follows. The signals from the transmitter employ low voltage swing signaling on low impedance transmission lines. To achieve bandwidths that are not limited by the latency of chip–to–chip interconnection, it must be possible to have multiple bits simultaneously in flight. This requires impedance matching of the transmitter and receiver to the transmission line used for chip–to–chip interconnection. To maintain acceptable power dissipation while driving low impedance lines, low voltage signaling must be employed. The prototype chip uses one–volt signaling on 50–ohm transmission lines. Separate amplifiers convert the clock and data signals from the one–volt swing off–chip levels to the five volt swing on chip levels. To maximize the feasible data transfer rate, the delays of these amplifiers must be closely matched. Since both amplifiers are fabricated on the same die, their electrical properties are very closely matched. However, due to the non–linearities inherit in digital circuits, the amplifier delays also depend on the data values (intersymbol interference). The design of the amplifier to minimize these effects is discussed in section 4.2.1.

The amplifiers output both true and complemented versions of their inputs. To maximize throughput, separate data values are sent during the high and low portions of the clock signal. These two streams are separated by the latches immediately following the input amplifier. Then, the clock is buffered further to provide sufficient drive for the demultiplexer. Associated with each buffer stage for the clock, there is a shift register stage for the data. This arrangement provides a data signal at the output of the shift register whose timing corresponds to the clock signal output by the buffer. Section 4.2.2 describes the design of this shift register.

The demultiplexer consists of a five–stage ring counter and ten latches. The outputs of the ring counter are the clocks to the latches. Thus, each of the two data channels from the shift register is demultiplexed by a factor of five to provide ten data channels for the FIFO. The demultiplexer outputs dual–rail coded signals for the FIFO. Section 4.2.3 describes the demultiplexer in greater detail.

As described in section 4.2.4, The FIFO is ten bits wide and 36 stages deep (i.e. 18 instances of the cell of figure 22) and each channel can operate at up to 200 Mbits/sec. At the maximum input rate of 700 Mbits/sec, each FIFO channel must operate at 70 Mbits/sec. This means that the window of feasible skew for this chip is greater than 300 ns. It is nearly inconceivable that this much variation in delay would occur in any practical system. The FIFO size was chosen to ensure a robust design. Since the prototype chip is pad bound, there is no area penalty in building a large FIFO.

The initialization circuit for the receiver consists of an R/S flip–flop and a static shift register for synchronization. When the external `reset` signal is asserted, the FIFO is initialized to an empty state as described in section 4.1.3. After the `reset` signal is no longer asserted, the transmitter may start sending. The protocol is that the transmitter sends the first clock edge, waits eighty cycles of the fast clock (i.e. eight cycles of the slow clock), and then sends data at the full rate. The first edge of the clock signal from the transmitter clears the R/S latch. After four cycles of the slow clock for synchronization, the reset is lifted from the FIFO, allowing it to accept data from the transmitter. After thirteen more cycles of the slow clock, the reset is removed from the "out" block, which then starts generating acknowledgements and removing data from the FIFO. The lag from when the transmitter starts sending to when the receiver starts removing data from the FIFO allows the FIFO to fill to a half–full state.

4.2.1 The input amplifier

To achieve high performance, the input amplifier must have high–bandwidth and the delays of the clock and data amplifiers must be closely matched. The second requirement means that the delay of the amplifier must be independent of data patterns. The input amplifiers used on the STARI chip have two distinguishing features:

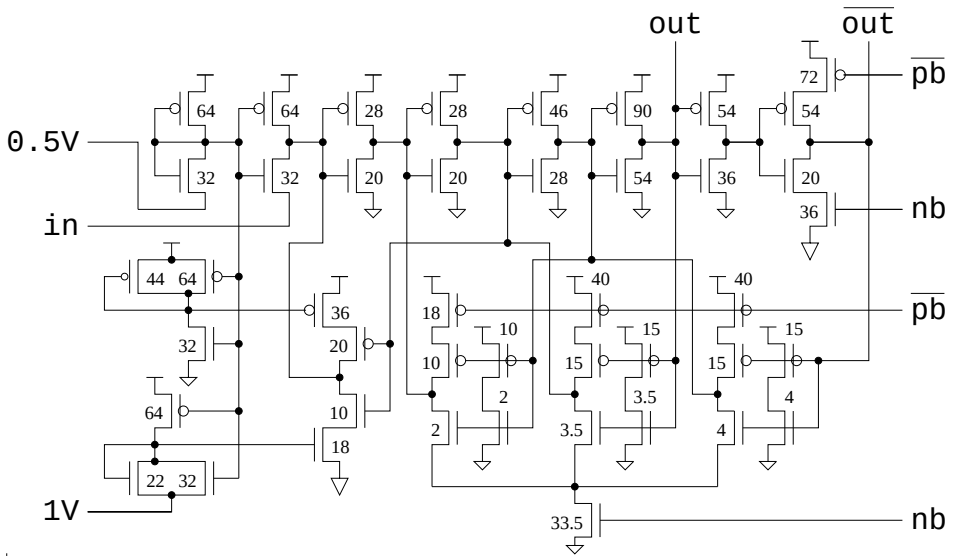


Figure 14: The input amplifier

1. The interstage fan–outs are small (< 2).

2. Negative feedback is used to improve speed and reduce variations in delay.

Figure 14 shows the input amplifier used for clock and data signals. The input stage is adapted from Knight's design presented in [KK88]: a biased inverter is used to convert the off–chip low voltage swing signals to five volt swing signals on chip. Also adapted from Knight's design is the use of a buffer with much smaller devices following the biased inverter stage. As the biased inverter has a high impedance output (since both transistors are in or close to saturation at all times), the small load is essential to obtain reasonable switching times. The input amplifier on the STARI chip includes four additional buffer stages that are not present in the design of [KK88].

The buffer chain has small interstage fan–outs. It is well known [MC79] that the minimum propagation delay through a buffer chain is achieved by an interstage fan–out of e (2.718...). There are two reasons to use smaller fan–outs in the input amplifier. First, the total delay of the amplifier is not critical; instead, the design

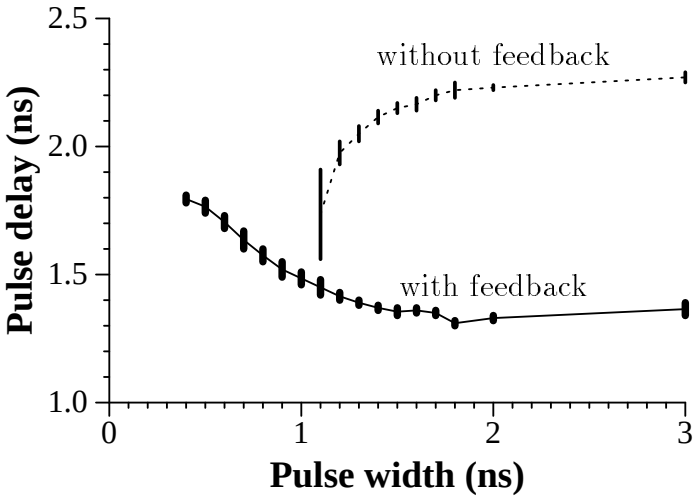


Figure 15: Dependence of delay on pulse width (data from simulation)

objective is to maximize throughput. As multiple bits may be simultaneously propagating though different stages of the amplifier, throughput can be greater than inverse propagation delay. Smaller fan–outs allow stages to switch faster, thus increasing throughput. Second, the analysis in [MC79] assumes that the drive capability of a stage depends only on the shape factors of its transistors. However, the drive capability also depends on the slope of the input signal [Hor83, LM84]. Due to the slowly changing output of the biased inverter, smaller fan–outs are employed for the first stages of the buffer chain than for later ones.

Given that small fan–out buffer chains have higher throughput than large fan–out stages, one might consider a very long buffer chain with fan–outs of $1+\epsilon$ for some very small value value of ϵ. However, another consideration excludes this approach. The delays through the clock and data amplifiers must be well matched. Any mismatch in the delay must be added to the minimum time per bit. If a buffer chain is very long, then the uncertainity in its total delay will also be large. Circuit simulation indicated that the design shown in figure 14 represents a good trade–off with four stages after the biased inverter to the non–inverted output, and five stages to the inverted output.

The second distinguishing feature of the design is the use of negative feedback. This is similar to a static sense amplifier [Bak90] with many, cascaded stages. Viewed

as a linear circuit (an extremely coarse approximation), negative feedback [HW89] provides an increase in amplifier bandwidth and reduction in output impedance for a reduction in voltage gain. Since very little gain is needed in the buffer chain, this is a favorable trade–off. Furthermore, by shifting the dominant pole to a higher frequency, negative feedback reduces the dependency of delay on pulse width. As shown in figure 15, without negative feedback, the delay of the amplifier depends strongly on the pulse width (a simple model of varying data patterns). Negative feedback significantly reduces this dependence thereby better matching the delays of the clock and data channels. The "error–bars" in figure 15 show the variation in delays of rising and falling edges according to circuit simulation. "Slow–slow" transistor parameters were used in the simulations that generated this figure.

In simulation, the best performance was observed when the feedback at each stage was sufficient to pull the output of the corresponding inverter about one volt above ground or below the power supply. As the relative strength of the feedback depends on the relative strength of n–channel and p–channel transistors (which are not tightly matched by the fabrication process), the design includes two provisions to compensate for extremely mismatched devices. The signals `nb` and $\overline{\texttt{pb}}$ ("n–boost" and "p–boost") can be used to selectively enable or disable n– and p–channel transistors in the amplifier. Although it would be straightforward to build a circuit that automatically set these signals, in the current design they are connected to input pads of the chip to increase controllability of the chip during test. This level of control is adequate for most of the buffer, but it is too coarse for use with the feedback stage connected to the output of the biased inverter.

The high output impedance of the biased inverter makes it benefit significantly from the application of feedback. It also makes this stage particularly sensitive to variations in the transistor parameters. The on–chip biasing stages shown in figure 14 provide close tracking of the drive of the feedback stages to the drive capability of the biased inverter.

Given the basic design, circuit simulation was used to determine transistor sizing. The principle objective was to minimize variations in the delay of the clock and data channels over a wide variety of data patterns. The choices of transistor widths was

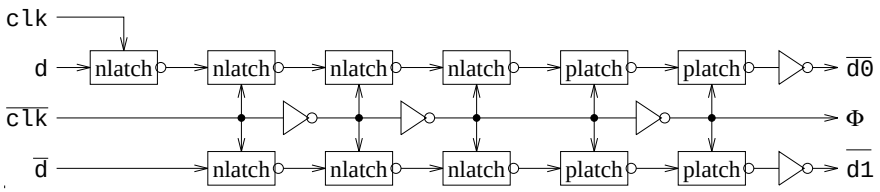


Figure 16: The shift register and clock buffer

restricted to widths that are available by geometries that minimize the drain diffusion capacitance to gate width ratio. In this low fan–out design, the capacitances of the drain regions are critical. Figure 14 shows the final choice of transistor sizing (as shape factors). This amplifier supports data rates exceeding 500 Mbits/sec. with a maximum variation of delay (rising and falling edge, and over variations of pulse width) of less than 0.25 ns.

4.2.2 The clock buffer and shift register

Once the data and the forwarded clock have been amplified and buffered, the data must be latched to the forwarded clock. To maximize the robustness of the design, the set–up/hold window of the first latch must be as small as possible. This section describes the design of a latch that is optimized for this application. The shift register that keeps the data aligned with the clock through the buffer chain is briefly described.

Figure 16 shows the clock buffer chain and data shift register of the receiver. Inputs `clk` and $\overline{\texttt{clk}}$ are the true and inverted versions of the clock signal from the input amplifier for the clock channel. The `d` and $\overline{\texttt{d}}$ inputs are the outputs of the input amplifier for the data channel. The latches are transparent: latches labeled "nlatch" pass data when the clock input is high and hold their value when the clock input is low; latches labeled "platch" pass data on a low clock and hold it while the clock is high. The upper shift register of figure 16 acquires the data values that are sent with the high portion of the clock, and the lower shift register acquires the values that are sent with a low clock. Thus, the input data stream is demultiplexed by a factor of two when entering the shift register.

As described above the interval between the maximum set–up and maximum hold

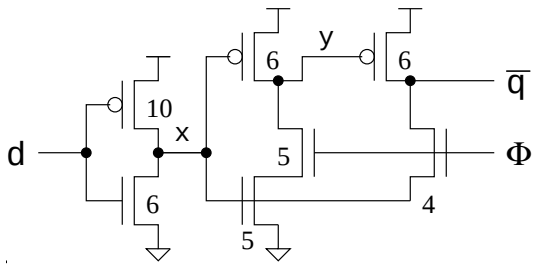


Figure 17: A latched with tightly matched set–up and hold times

times of the first latch in each shift register should be minimized. To reliably latch the data from the transmitter, the data signal must not change in this interval. By minimizing the width of this interval, the receiver is made more robust to variations of delay in the two channels. The transparent latch used in this design is shown in figure 17. This latch is an adaptation of the latch designs presented in [YS89]. The inverter at the input of the latch provides a delay necessary for the negative hold time. The use of a pass transistor controlled by the clock for the pull–down path of `q` helps achieve set–up and hold times that are nearly independent of the data value. If the `d` input is high, the critical path to store this value consists of `x` going low followed by `y` going high and $\bar{\texttt{q}}$ going low concurrently. A low value of `d` is stored once `x` goes high and `y` goes low; $\bar{\texttt{q}}$ can go high *after* the clock goes low. In either case, the critical path has two steps, and by careful sizing of the transistors, the dependence of set–up and hold times on data value is very small (less than 0.1 ns). The latch shown in figure 4.9 is an n-latch: data is acquired when the clock input is high and held when the clock is low. A p–latch is the dual circuit obtained by exchanging n–channel and p–channel devices.

The second critical timing issue in the design of the shift register is the race between the clock and data paths. Correct operation of the shift register requires that the clock path through the inverter be faster than the data path through the latch. This is a three–against–one race (three stages in the latch against one in the clock buffer chain), and simulation showed that the margin is always greater than 1 ns, which is generous for this design with a 3 ns clock period.

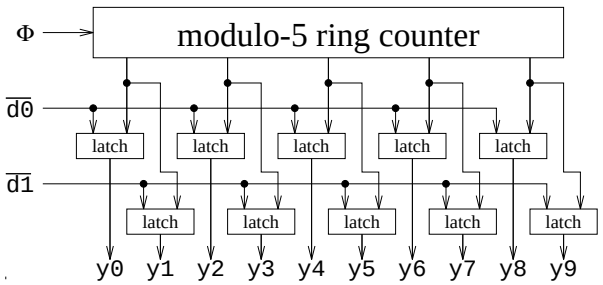


Figure 18: The demultiplexer

To provide sufficient drive to the demultiplexer, each stage in the clock buffer chain has greater drive capacity than its predecessor; the final stage has shape factors of 110 and 220 for the n– and p–channel transistors respectively. Likewise, each stage in the shift register has greater drive capacity than its predecessor as well. At the input of the shift register "n–latches" are used. In the middle, a change is made to "p–latches" that acquire data while the clock is low. This complements the counter in the demultiplexer which changes when the clock is high and guarantees stable data for the demultiplexer's latches.

4.2.3 The demultiplexer

The critical circuit of the demultiplexer is the counter that controls the demultiplexer's operation. This section describes the design of the counter used in the STARI chip. This counter has two distinguishing features:

1. The critical timing path for each half–cycle (high or low portion) of the clock is a single gate. This is the minimum possible.

2. All communication is nearest neighbor. The counter is a systolic design with no long signal wires.

Figure 18 shows a block diagram of demultiplexer. The design is simple, a modulo–5 ring counter strobes the latches, five latches for each data channel. The latches are the same as those described in the previous section. A simplified version of the counter

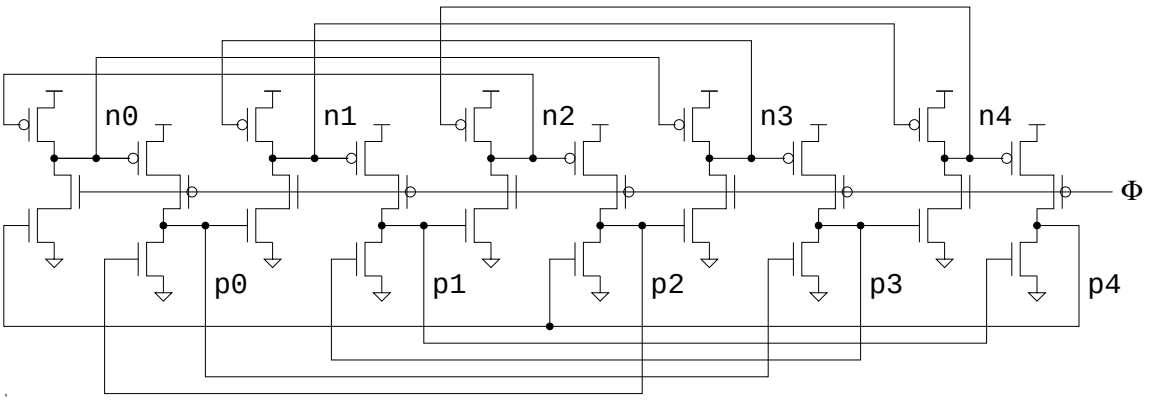


Figure 19: Simplified version of the ring counter

is shown in figure 19. For the sake of simplicity, no initialization circuitry is shown. To understand the operation of this counter, assume that it is initially in the state shown on the first line of figure 20. When Φ goes high, `n1` is pulled low (the "set" action) which in turn pulls `n4` high (the "reset" action). This brings the counter to the state shown on the second line of figure 20. Then when Φ goes low, a set action brings `p1` high and the subsequent reset action brings `p4` low. Figure 20 shows five complete clock cycles, returning the counter to its original state. Only the set action depends on the value of the clock, accordingly, the critical delay for this counter is switching a node through two series transistors. However, this design requires long wires, and the capacitance of these wires limits the performance of the design.

Figure 21 shows the ring counter that is used on the STARI chip. It is based on the design of figure 19. Retiming [LS86] has been employed to replace, the long–wires of the original with half–registers (implemented with inverting C–elements). This allows the `e` signal (the signal that enables the reset action) to be derived from a closer stage and provides buffering of the signal. The behavior of the new counter is slightly different than that of the counter shown in figure 19. The output of each `N0` or `N1` stage is low only for three half–cycles of Φ instead of four as shown for the simple counter in figure 20. This change is insignificant in the context of the demultiplexer. The outputs of this counter (signals `n0`, `n1`, `n2`, `n3`, and `n4`) are buffered through

Φ	n0	n1	n2	n3	n4	p0	p1	p2	p3	p4
0	0	1	1	1	0	1	0	0	0	1
1	0	0	1	1	1	1	0	0	0	1
0	0	0	1	1	1	1	1	0	0	0
1	1	0	0	1	1	1	1	0	0	0
0	1	0	0	1	1	0	1	1	0	0
1	1	1	0	0	1	0	0	1	1	0
0	1	1	1	0	0	0	0	0	1	1
1	0	1	1	1	0	0	0	0	1	1
0	0	1	1	1	0	1	0	0	0	1

Figure 20: State transitions in the counter of figure 19

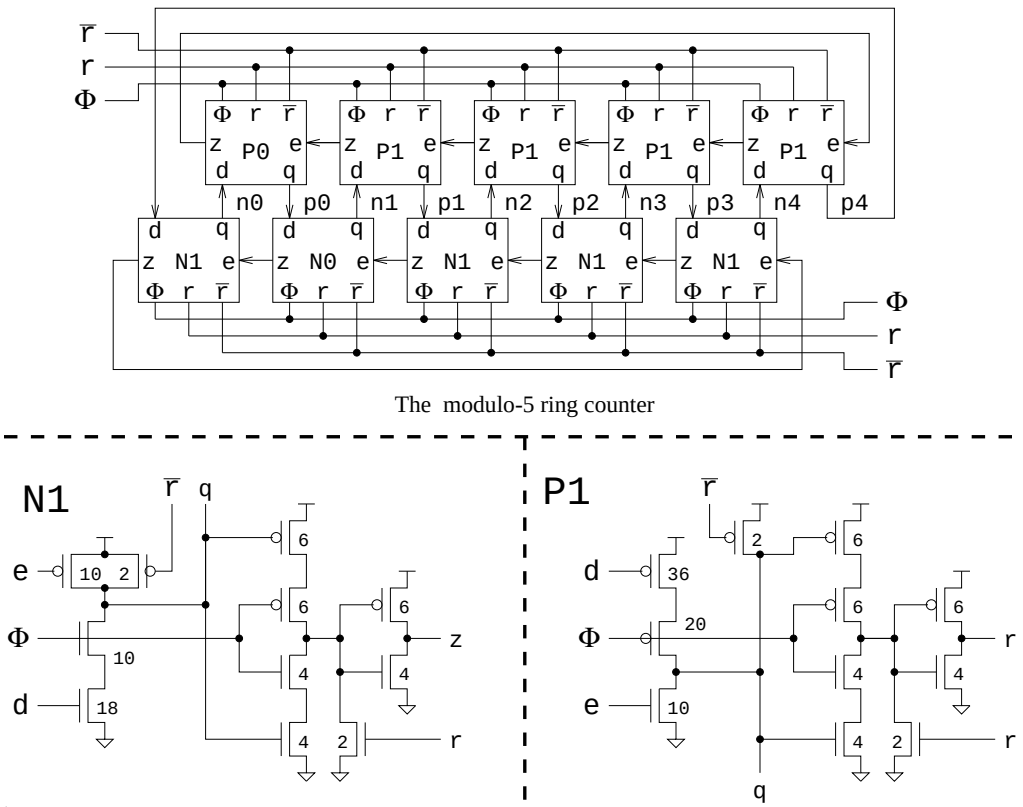


Figure 21: A ring counter with no long signal wires

a pair of inverters and the buffered signals strobe the latches of the demultiplexer. In simulation, the new counter can operate at clock frequencies exceeding 650 MHz (worst–case parameters), nearly double the speed required by the receiver.

4.2.4 The FIFO

Two modifications were made to the FIFO design presented in section 4.1 to improve performance. One of these, a logic optimization to remove inverters, resulted in a malfunction of the prototype. The other, transistor sizing to minimize the time per operation, appears to have been successful. Both modifications are described here.

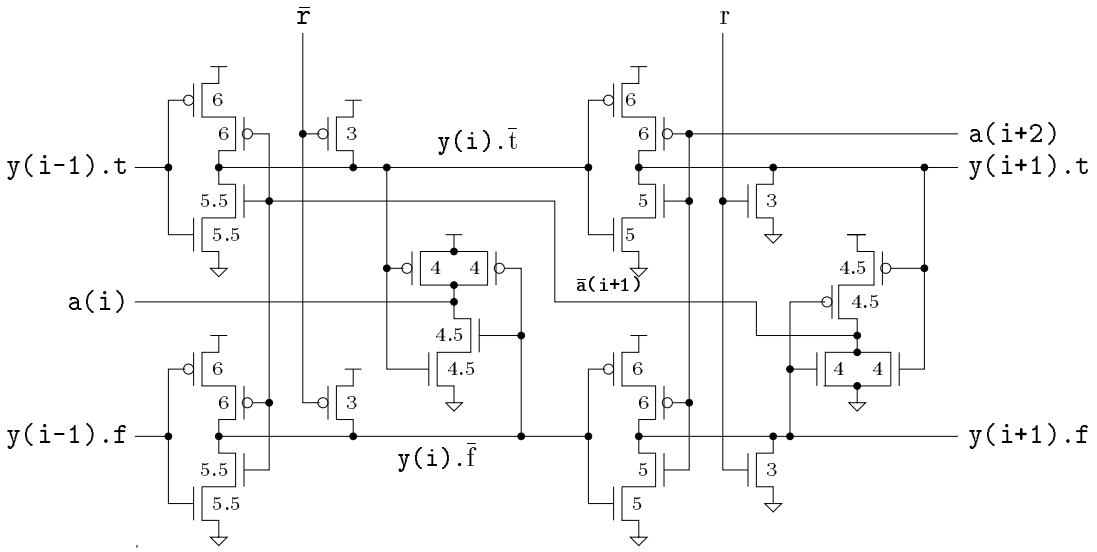
The design choice that lead to a malfunction is based on the observation that the dual–rail code described in section 4.1.1 can be inverted to produce the code:

$y.\bar{t}$	$y.\bar{f}$	y
low	low	illegal
low	high	T (true)
high	low	F (false)
high	high	E (empty)

This suggests a design where even–numbered stages output values in the original encoding and odd–numbered stages output values in the inverted dual–rail code. To implement this, the outputs of the C–elements are inverted. This is accomplished by *deleting* the inverter at the output of each C–element. Furthermore, the odd indexed acknowledge signals are generated by NAND gates.

These changes produce a faster FIFO by reducing the delay through each stage (δ in chapter 3.1.1). However, it also compromises the noise margins [GD85] of the design. Although the modified FIFO is quite robust in circuit simulation, tests of the chip (see section 4.4) suggest that this modification causes the FIFO to malfunction at low data rates. This is discussed further in section 4.4.

Once this structural change was made, transistor sizes were adjusted to optimize performance. To estimate the speed of the FIFO, an eight stage FIFO was simulated with `y(0).t` connected to `a(1)` through an inverter and `a9` connected to `y(8).t` through a buffer. The input `y(0).f` was grounded. In this configuration, the FIFO



Shape factors are indicated for each transistor. All transistors have 2 micron gate lengths. `i` is an odd integer.
Warning: this design malfunctions. See section 4.4.

Figure 22: Two FIFO stages as implemented on the prototype chip

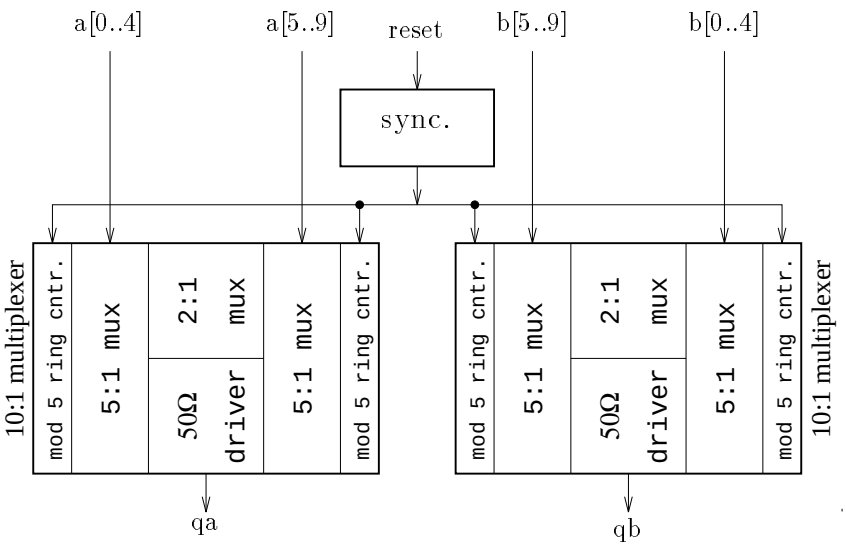


Figure 23: Block diagram of the transmitter

operates as a chain of coupled ring oscillators. Maximizing the rate of oscillation minimizes the time per data transfer of the FIFO. A design was obtained which can sustain one data transfer (i.e. transferring both a valid and an empty value) every five nanoseconds. The design uses the MOSIS 2 micron n–well process, and simulation used worst–case SPICE parameters. The final design is shown in figure 22. As will be described in the next section, this performance would be a bottleneck for a direct implementation of the STARI interface, but it is more than adequate for the implementation of STARI used on the prototype chip.

4.3 The Transmitter

As shown in figure 23 the transmitter consists of two ten–to–one multiplexers. One multiplexer generates the high–speed data output; the other generates the forwarded clock. This design exports the implementation of the initialization sequence described in section 4.2 to off–chip logic, simplifying the design of the chip and facilitating experiments.

The transmitter circuitry is purely synchronous. The modulo–5 counters that

generate control signals are similar to the counter used in the receiver's demultiplexer. In addition to controlling the demultiplexer, the outputs of the counters are used to generate two–phase non–overlapping clock signals at the slow clock rate. These two–phase clock signals control the input registers of the transmitter and are output for use off–chip. The five–to–one and three–to–one multiplexers are implemented using the single–phase clocking techniques described in [YS89], and the 50Ω output drivers are based on the design of [KK88]. The output impedance of these pads is controlled by selecting which of four parallel output transistors are enabled to drive the pad [Kni89]. To achieve 700 Mbits/sec. performance (operation with a 350 MHz fast clock), three aspects of the design were identified as critical:

1. initializing the two multiplexers to the same control state;

2. distributing control signals within each multiplexer; and

3. clock distribution.

These correspond to the signals that must be transmitted long distances on the chip.

The uncertainities in the delays from the circuit board, through the pin, lead–frame, and bonding wires to the die preclude a design where the external reset signal is synchronized to the high–speed clock. A synchronizer must be used. To ensure that both multiplexers are reset to the same state, one synchronizer is used for both multiplexers. As pad spacing restrictions require that the two multiplexers be geographically separated, the global synchronization signal must be distributed to widely separated circuits. It is not practical to implement drivers that can convey the synchronization signal across the diameter of the chip in a single clock cycle. Instead, a shift–register tree is used. The root of this tree is the output of the synchronizer, and the leaves are the circuits that require the reset signal. The length (measured in half–registers) from the root to each leaf is the same. This transforms the problem of reset signal distribution to a problem of clock–signal distribution.

The multiplexers are controlled by signals derived from a modulo–5 ring counter. It is not practical to distribute these signals across an entire multiplexer. Accordingly, each five–to–one multiplexer has its own modulo–5 counter and the outputs of each

counter are distributed with shift–registers. As in the design of the systolic counter, this is an example of removing long wires by duplicating function, a common design technique that is especially important in high–performance circuits. Of course, the reliable operation of these counters and shift registers requires reliable distribution of the high–speed clock.

To solve the problems of clock distribution, the circuits are designed to operate with a sinusoidal clock. Because the dimensions of the chip are much smaller than the wavelength of the clock, the clock can be distributed as a standing wave with very little phase variation on the chip [Chi90]. This requires that the clock distribution wires have resistances that are much lower than the absolute values of the impedances of the capacitive loads presented by MOS circuits. This requirement is readily satisfied by implementing the primary clock distribution network on a single level of metal and only using vias to other layers near the network's leaves.

Although the clock signal will be uniform on the chip, this signal may have a much different amplitude and phase than the clock signal applied at the board to the chip. Although phase shifts are inconsequential in a STARI interface, a clock signal of inadequate or excessive amplitude can damage the chip or cause it to malfunction. The path from the pin through the lead frame to the die has significant inductance, and the chip appears as a capacitive load to the clock signal. This is a series resonant circuit. MOSIS, the silicon broker used for fabricating the STARI chip, does not publish lead inductance data for their packages. Bakoglu [Bak90] suggests a typical lead inductance of 7 nH for a 68 pin grid array package. This estimate is used in the following calculations. The on–chip clock capacitance is about 15 pF with gate capacitance accounting for over 90% of this figure and interconnect capacitance for the rest.

The resonant frequency of the clock circuit based on the estimates of the preceding paragraph is 485 MHz. If the chip is operated with a 350 MHz sinusoidal clock, the amplitude of the on–chip clock signal will be more than three times that of the off–chip clock. This has significant consequences for the chip design. First, it emphasizes the importance of circuits that can operate with a sinusoidal clock: a high frequency square wave clock would be distorted by severe ringing. Second, the amplitude of the

off–chip clock must be carefully adjusted to provide the desired on–chip signal. In the current design, my intention was to characterize the clock circuit of a fabricated chip by time domain reflectometry (TDR) and set the amplitude of the clock generator accordingly. This is acceptable for a prototype chip; circuits to provide for automatic adjustment of the clock level would be necessary for a chip that would be used in a real product.

4.4 Test Results

On October 31, 1991, three variations of the STARI chip were submitted to MOSIS for fabrication on run N1AZ (VTI 2 micron, n–well CMOS). The three variations differed in the implementation of the receiver's input amplifier as described below:

1. The design described in section 4.2.1.

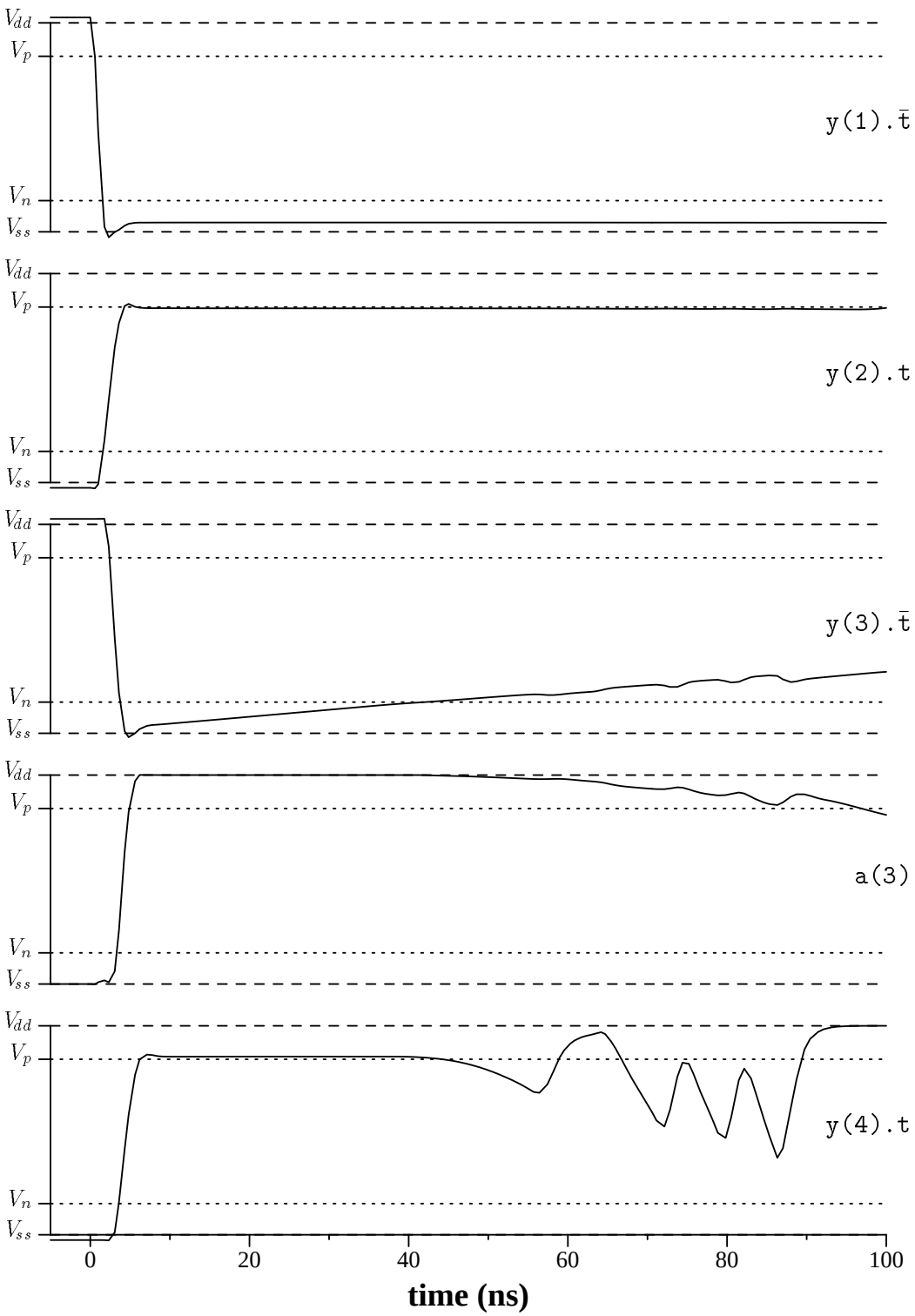
2. An amplifier without negative feedback but otherwise like the design described in section 4.2.1.

3. An amplifier without negative feedback and without the the biased inverter input stage (input signals must be five–volt swing). In addition, twenty–five probe points were included in this version.

Fabricated chips were received from MOSIS in January 1992. A design error in the transmitter prevented testing of the chips with the equipment available at Princeton University. In July 1992, I tested the chips at ATT Bell Laboratories, Holmdel.

Three errors were discovered in the design. First, a wiring error in the shift register that distributes the reset signal in the transmitter was discovered while preparing test vectors for the chip. It was possible to avoid this error by modifying the test vectors. However, it caused dynamically stored values not to be restored. Accordingly, the chip only operates for a few milliseconds after initialization. This is adequate for further testing, but prevents the chip from being used in a complete system.

The second error is in the five–to–one multiplexer of the transmitter. The error is manifested as a latch malfunction. The malfunction is not revealed by simulation but appears to be caused by either a timing race or cross–talk between signals.



$V_{ss} = 0\text{V}, \quad V_{dd} = 5\text{V}, \quad V_n = 0.746\text{V}, \quad V_p = 4.2\text{V}$

Figure 24: Timing race in the FIFO

The third error is an analog timing race in the FIFO. Figure 24 shows a simulation of this race using Spice parameters from MOSIS for the chips that were tested. As in section 4.1, `y(0)` is the input of the FIFO, `y(n)` is the data output of the n^{th} FIFO stage, and `a(n)` is the acknowledge output of the n^{th} FIFO stage. V_n is the threshold voltage of an n–channel transistor with both source and drain at V_{ss}, and V_p is the threshold voltage of a p–channel transistor with both source and drain at V_{dd}. In this simulation, the FIFO is initially reset (all stages hold the "empty" value). When the simulation time, τ, is zero, the data input `y(0).t` goes high causing the following events: `y(1).`$\bar{\texttt{t}}$ makes a high to low transition; `y(2).t` makes a low to high transition; `y(3).`$\bar{\texttt{t}}$ makes a high to low transition; and `a(3)` makes a low to high transition. When τ = 4.9ns, `a(3)` crosses V_p turning off the pull–up for `y(2).t`. At this time, `y(2).t` is at 4.27 volts, then charge sharing causes `y(2).t` to drop to 4.17 volts. As this is slightly[2] below V_p, `y(3).`$\bar{\texttt{t}}$ is slowly pulled towards V_{dd}. When τ = 42ns, `y(3).`$\bar{\texttt{t}}$ crosses V_n, and shortly after that, `y(4).t` exhibits oscillatory behavior.

The critical timing is that `a(3)` crosses V_p before `y(2).t` is sufficiently above this threshold. As a consequence, the circuit fails to operate as a FIFO.

Although this problem is revealed by simulation, it was not observed while designing the chip. This is because the timing depends critically upon wiring capacitances. When designing the chip, most capacitances were manually estimated due to limitation of the CAD tools[3]. As figure 24 show, the race is extremely close; inaccuracies in the manual estimates hid this error during design. After testing revealed the problem, more accurate counting of the wiring capacitances showed the race in simulation as well.

In spite of these three errors, it was possible to devise test sequences that exercised most of the circuitry of the chip. The available test equipment could generate a fast clock at rates up to 50 MHz (i.e. operating the interface at 100 Mbits/sec.). At this

[2]Due to drain induced barrier lowering [GD85], `y2.t` is well below the threshold of the p–transistor that pulls up `y3.t`

[3]The circuit extractor in Magic models diffusion regions as fixed capacitors. However, the incremental capacitance of the diffusion/substrate junction depends strongly on the reverse bias of the junction [GD85]. As a consequence, diffusion capacitances extracted by Magic are much greater than the actual capacitance.

rate all circuits in the receiver functioned correctly. All receiver circuitry except for the FIFO functioned correctly with a 10 MHz fast clock as well.

In the transmitter, the ripple counter of the synchronizer was observed to operate. Limitations of the test set–up prevented testing the operation of the transmitter's synchronizer. As noted above, there was a wiring error in the distribution tree for the synchronized reset signal in the transmitter. Using suitable test vectors, the rest of the transmitter could be tested in spite of this wiring error. The ring counters that generate control signals for the multiplexers function correctly, and the two–phase slow and all control signals within the multiplexer (the signals derived from the outputs of the ring counters) were observed to be correct. Data was observed to be correctly multiplexed by the first stage of the multiplexer. As noted above, data was lost in a latch between the first two stages of the multiplexer. This error prevented fully exercising the multiplexers and the 50Ω output pads.

Each of the errors can be corrected by simple modifications to the design. A correction of the chip and refabrication is planned.

Chapter 5

Cache Consistency with Dual–Port Memory

STARI offers high bandwidth communication between processors of a multiprocessor or multicomputer. To exploit the bandwidth offered by STARI, each processor's memory must support high bandwidth transfers to and from the STARI interface. Furthermore, to achieve acceptable performance, these transfers should have minimal impact on local memory accesses. These goals can be achieved using dual–port memories that allow the network and the processor to access the memory independently. When used in conjunction with caches, dual–port memories introduce consistency problems. In particular, when the processor performs a write to the cache, data in the dual–port memory may become stale. Conversely, when the network writes to the dual–port memory, data in the cache may become stale. Since network reads and writes of the dual–port memory are not visible on the processor's memory bus, these problems cannot be solved by traditional, snooping techniques. As a consequence, existing designs do not use caching in conjunction with dual–port memory.

In this chapter, we examine dual–port memories and the consistency problems associated with their use. We propose two software solutions to these problems. The first scheme is suitable for machines with relatively small caches. The second refines the first by using alias bits provided by most cache components and allows machines with large caches to achieve consistency efficiently. Both schemes can be

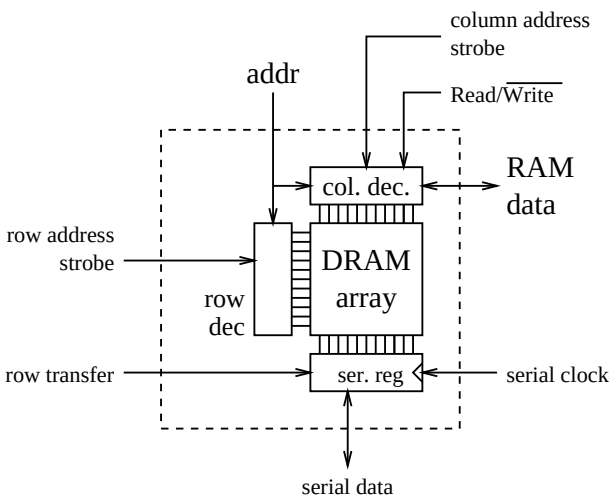


Figure 25: Block diagram of a Video RAM

easily implemented in virtual memory management systems, and neither requires special hardware support. Our performance analysis shows that these schemes can be used with very low overhead.

5.1 Dual–Port Memories

Dual–port memories are attractive components for interprocessor communication because they allow simultaneous memory operations through separate ports. We are particularly interested in using Video RAMs[1] (VRAMs, see figure 25) because they provide the high density of dynamic memories, and are suitable for applications with large data sets. Originally intended for video display applications [PR+84], VRAMs have been used for multiprocessor applications [Nic88] and for high–speed networks [SB+90].

Figure 26 depicts a node of a multiprocessor that uses dual–port memories for communication. This figure reflects several architectural assumptions:

[1]Video RAMs are dynamic memories that implement a second port using a large, fast shift register.

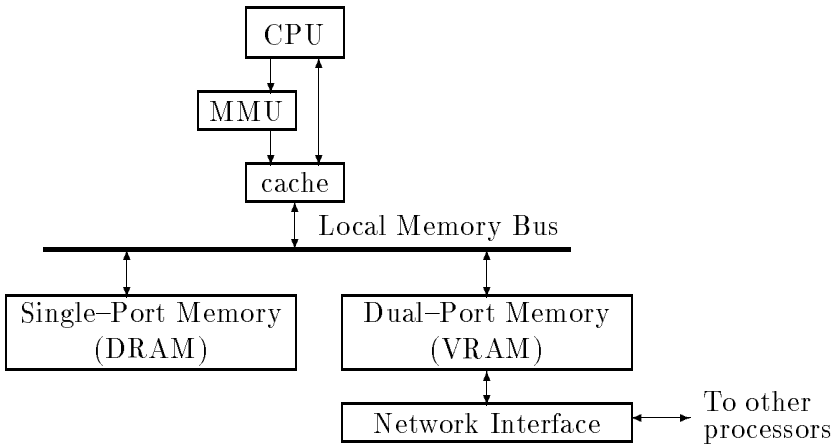


Figure 26: A processor node using dual–port memory

- The processor includes a cache. Caches are an essential part of almost every high performance processor design, matching the short cycle time of the CPU to the longer access time of memories. Previous work using dual–port memories has been based on cacheless designs [Nic88, JKP89, Ser90], or caching has been employed only for the single–port memory, and the dual–port memory has been uncached [SB+90]. In this paper, we present an efficient scheme that allows caching of dual–port memory.

- The processor supports virtual memory, and the cache is physically addressed. Address translation is performed by the MMU (memory management unit), which resides between the processor and the cache. With a physically addressed cache, our schemes can be realized without any modifications to the cache hardware. This allows our schemes to be employed with stock microprocessors and their on–chip caches. We consider virtually addressed caches in section 5.5. We also assume that write–through or copy–back policies can be specified on a per–page basis as is the case in many off–the–shelf processors (e.g. the Motorola MC88200 [Mot88] and the Intel i860 [Int89]).

- The physical memory space is implemented using a combination of dual–port

memories (VRAM) and conventional, single–port memories (DRAM). This reflects the economic reality that VRAM storage currently costs about twice as much per bit as DRAM. It is the responsibility of the application program or the operating system to identify shared data structures and store them in the dual–port memory.

- To maintain a clear focus on the issues raised by using dual–port memory, we assume that each node in the multiprocessor has only one CPU.

Using dual–port memories, the CPU and the network interface can access data independently. This independence offers numerous architectural advantages over traditional, DMA based designs. In general, good CPU performance requires low latency memory accesses, and network performance requires high memory bandwidth. Architectural opportunities offered by dual–port memories include:

No arbitration

Because the CPU and the network access the memory through separate ports, no arbitration or buffering of access is needed. This simplifies the node design, and can improve the performance of both ports. For example, we have explored a design that exploits the guarantee that the full memory bandwidth is always available to eliminate the need for flow control hardware. In some designs, the dual–port approach could completely eliminate the need for arbitration of the local memory bus, allowing the bus to be implemented with a simple, high–performance, synchronous design.

Lighter memory bus load

Because transfers between the memory and the network do not use the local memory bus, the bus load is lighter than for a DMA based design. For a uniprocessor node, this does not appear to significantly improve performance (assuming that CPU memory accesses can preempt network transfers). Given a tolerable cache hit ratio, the CPU will only need the bus occasionally. However, the advantages of removing network traffic from the memory bus could be more significant for designs where the bus load is higher, for example, if each node were a shared–bus multiprocessor.

Independent interleaving

By separately interleaving the CPU and network ports, each port can be optimized for its application. For example, in the common case where blocks transferred between processors are larger than a cache line (e.g. a page), a higher degree of interleaving can be exploited on the network interface than on that of the CPU.

5.2 Consistency Problems

As described in the previous section, dual–port memories are attractive components for interprocessor communication; however, they also introduce consistency problems if the processor includes a cache. Generally, programmers assume that memory is consistent; when the memory is read, the value of the read should be the value of the immediately preceding write. Inconsistencies can arise when there are multiple physical copies of the same logical datum. In the case of the processor node of figure 26, the same datum can be simultaneously stored in both the cache and the dual–port memory. If no precautions are taken, this can result in inconsistent memory states.

Figure 27 illustrates these problems. Initially, the cache is empty and the value `xxx` is stored at location `p` in the dual–port memory. Then, the CPU reads location `p`, loading the cache with the tuple `(p, xxx)`, and values for location `p` exist in both the dual–port memory and in the cache. From this state, there are two ways to produce an inconsistent state. The first is depicted by the upper path in figure 27. In this case, the CPU writes to location `p` and the corresponding cache entry is updated, but the dual–port memory continues to hold the old value. A subsequent read of location `p` by the network will receive the old value, `xxx`, whereas the value of the most recent write is `yyy`. We refer to this as **stale dual–port memory data**. The second form of inconsistency is shown in the lower path of figure 27. Here a write from the network port of the dual–port memory modifies location `p` in the memory but not in the cache. A subsequent read of location `p` by the CPU will receive the old value, `xxx`, whereas the value of the most recent write is `zzz`. We refer to this as

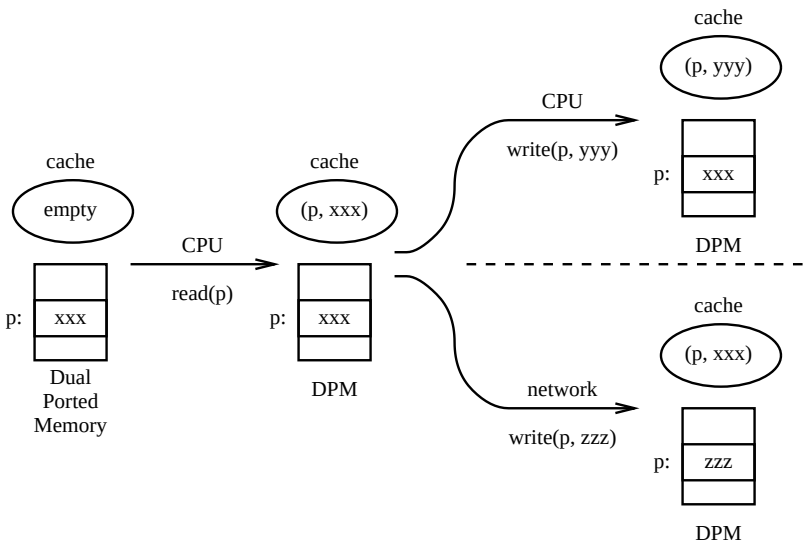


Figure 27: Consistency problems with dual–port memory

stale cache data.

The consistency problems just described must be considered even for designs based on traditional DMA with single–port memories (see [HP90b]), and numerous solutions based on "snooping" and "directories" have been proposed. In a snooping scheme, all memory transactions are performed on a common bus, and the cache monitors all memory transactions on the local bus taking appropriate actions to ensure consistency [McC84, KE+85, AB86, TS87, EK89a]. In directory–based schemes, copies of each node's cache directory are maintained by the memory system and used to identify accesses to shared data [Tan76, Tuc86, AS+88, CG+88]. When a processor accesses an item held in another cache, appropriate actions are taken to maintain consistency. Both snooping and directory–based schemes require close interaction between clients of the memory while memory accesses are taking place. They introduce arbitration for buses, cache directories, and cached data. Because of these interactions, neither snooping nor directory schemes can fully exploit the architectural advantages of dual–port memories.

5.3 Virtual–Memory Based Consistency Schemes

To ensure memory consistency while retaining the independence of the CPU and network hardware, we propose a software-based consistency scheme. Our scheme can be integrated into the processor's virtual memory system. Because virtual memory systems provide software management of memory accesses, it is natural to use these facilities to ensure that accesses to shared data are performed consistently. We note that many existing parallel programming paradigms use virtual memory pages for interprocessor communication [Smi85b, LH86, RT+88, CF89, LH89, LE90]. In this section, we first show that selective write–through is an effective approach to the stale dual–port memory data problem. Then, we present two solutions to the stale cache data problem.

Selective Write–Through

Write–through caches have often been used to ensure that the data in main memory remains consistent with cache entries. There are two requirements for efficient use of a write–through strategy. First, the average number of processor cycles between writes must be greater than the memory cycle time; otherwise, the processor must stall frequently. Second, the write pipeline must include enough stages of buffering to avoid stalling on bursts of writes. Current processor cycle times are sufficiently shorter than memory access times so that serious stalling will occur if write–through is used for all writes. However, for typical applications, only a small fraction of all writes are writes to shared variables. For example, in all of the programs considered in [EK88], less than one sixth of all writes were to shared data. This suggests that write–through for shared variables is practical.

Many caches support selection of write–through or copy–back policies on a per page basis [Mot88, Int89]; a bit in each page table entry is used to specify which policy to use. When the virtual memory system allocates page frames for an application, it can set this bit according to the intended use of the page. Pages for shared variables are allocated with a write–through policy, and pages for private data are allocated for copy–back. This solves the problem of stale memory data and, as will be shown

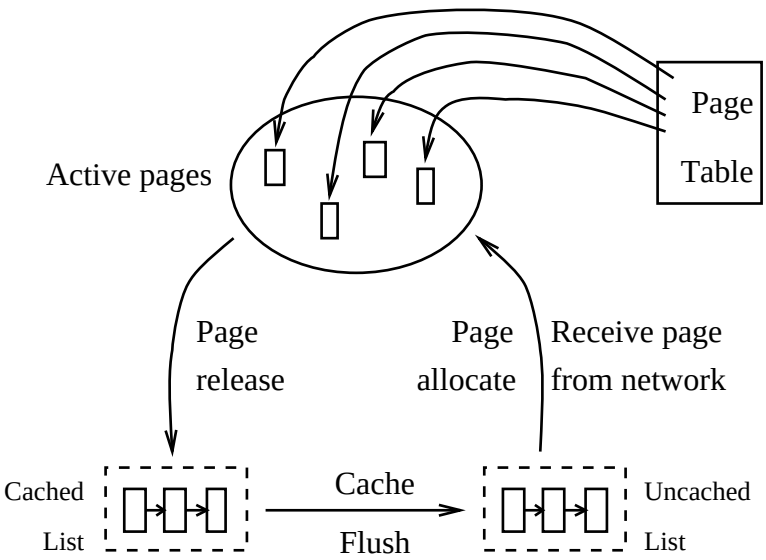


Figure 28: Page sets in the two–list scheme

in section 5.4, incurs little overhead.

Two–List Scheme

Our first solution to the stale cache data problem is called the two–list scheme. This scheme is motivated by the following, simple observation: if the cache is flushed each time the network writes to the dual–port memory, there will never be stale data in the cache. However, flushing on every network write is expensive and impractical. Through careful management of the page table, the two–list scheme can reduce the frequency of cache flushing by several orders of magnitude.

In a typical virtual memory system, the pages of physical memory are partitioned into two sets: **active** and **free**. Virtual to physical translations map virtual page to pages in the active set. The page frames in the free set are not the target of any translation. When the virtual memory system needs a new physical page frame, it transfers the page from the free set to the active set. A physical page is returned to the free set when the corresponding virtual page is paged out or released.

When the cache is flushed, all physical pages become uncached[2], not just the page currently being allocated. Furthermore, the pages in the free set remain uncached because they are inaccessible to user processes. All physical page frames that were on the free list before a flush may be allocated after the flush without further flushing to maintain consistency. The two–list scheme maintains two lists for free page frames: a list of **uncached** pages, and a list of (possibly) **cached** pages (see Figure 28).

- Initially, the uncached set contains all dual–port memory pages, and the cached set is empty.

- When a page is allocated, it is removed from the uncached list.

- When a page is released, application programs cannot reference it; however, the cache may continue to hold lines corresponding to the page. Therefore, pages are added to the cached list when they are released.

- When the uncached list becomes empty, the cache is flushed (making all pages uncached), the pages of the cached list are transferred to the uncached list, and the cached list becomes empty.

The two–list scheme maintains two essential invariants. First, the active pages, the pages in the cached list, and the pages of the uncached list form three disjoint sets. Second, every page in the uncached list is, in fact, uncached. The correctness of the scheme follows from these invariants.

By maintaining a sufficient number of page frames in the uncached set, the virtual memory system can ensure that cache flushes are infrequent and that the overhead is low. Our analysis will show that the two–list approach works well for machines with relatively small caches. However, for designs with large caches, flushing is expensive, and the overhead of the two–list scheme can be unacceptable. Our next consistency scheme overcomes this problem.

[2] We say that a page p is **uncached** if there are no blocks in the cache corresponding to a location in p. Conversely, if there is some block in the cache corresponding to p, we say that p is **cached**.

Physical Address Aliasing

To further reduce the cache flushing overhead requires a scheme that allows the same dual–port memory page to be released and reallocated without an intervening cache flush. This can be accomplished with cache tags that distinguish between different allocations of the same page. Because only cache lines corresponding to the most recent allocation of the page have the most recent tag, these tags solve the stale cache data problem. Explicitly adding an extra tag to each cache line requires designing a new cache. This section describes how the same effect can be accomplished using off–the–shelf cache components.

The key observation behind our scheme is that off–the–shelf processors and caches support a much larger physical address space than is typically implemented on a single node of a multiprocessor. The node can be designed so that either most physical addresses do not map to any memory location or so that many physical addresses map to the same memory location. Our scheme exploits the second approach. In particular, we use a node design where each page of the dual–port memory corresponds to many physical address page frames. We call these replicas **aliases**. Since aliases have different physical addresses, they are distinguished by the physical address cache. If the virtual memory system assigns each page of the dual–port memory a different alias each time the page is allocated, stale data in the cache will never be accessed. We call this technique **physical address aliasing**.

As a concrete example of a plausible implementation, consider a node with 32-bit physical addresses and 2 Mbytes of dual–port memory. Let the address decoder select the dual–port memory when the most significant bit of the physical address, p_{31}, is low. This design is depicted in figure 29. In this design, physical address bits p_0 through p_{20} select locations within the dual–port memory, and bits p_{21} through p_{30} are ignored, providing 1024 replicas of each location in the dual–port memory (reducing cache flushing overhead by three orders of magnitude).

By combining the physical address aliasing and two–list techniques, the cache flushing overhead can be greatly reduced. The virtual memory system maintains a counter k which holds the value of the current alias ($0 \leq k < A$, where A is the number of aliases for each page). The consistency actions taken by the virtual memory system

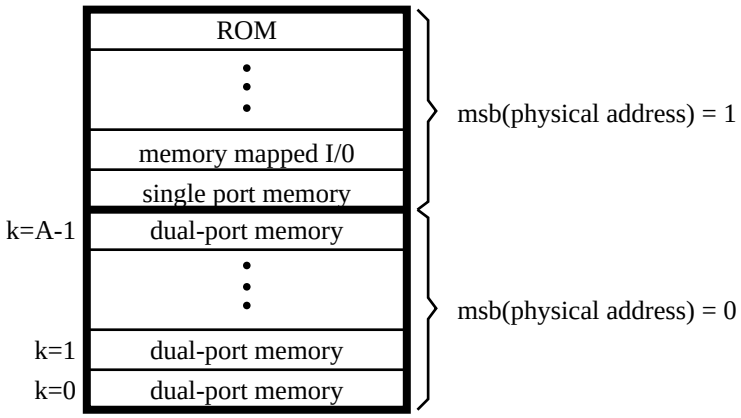


Figure 29: Physical addresses with aliased dual–port memory

are (see figure 30):

- Initially, the uncached set contains all dual–port memory pages, the cached set is empty, and k is set to 0.
- When a page is released, it is added to the cached list.
- When a page is allocated, it is taken from the uncached list, and its alias is set to k.
- When the uncached list becomes empty, the value of k is checked:
  (a) If k is less than its maximum value, k is incremented, the pages of the cached set are transferred to the uncached set, and the cached set becomes empty.
  (b) If k is equal to its maximum value, $A - 1$, k is set to 0, the alias fields of all page table entries associated with the dual–port memory are set to 0, the cache is flushed, and the pages of the cached set are transferred to the uncached.

The physical address aliasing scheme maintains two invariants in addition to those of the two–list scheme: no cache block has an alias greater than k, and if the alias of

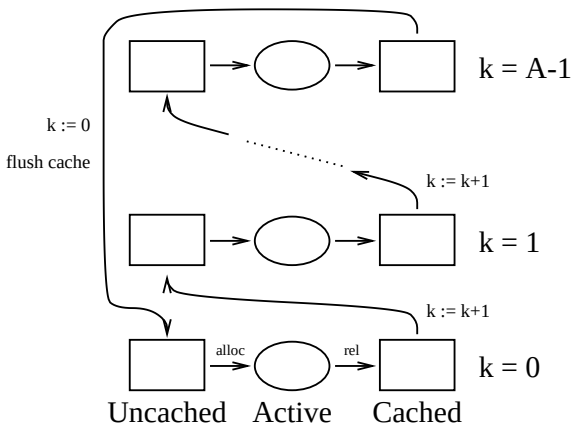


Figure 30: Page sets with physical address aliasing

a cache block is equal to k, its page does not correspond to any page on the uncached set. The correctness of the scheme follows from these invariants.

Physical address aliasing improves the original two–list scheme greatly because cache flushing is performed much less often. The flush step (b) in the physical address aliasing scheme requires changing the alias field of all page table entries associated with the dual–port memory. The additional overhead is only a few instructions per page. The cost of the flush step is slightly more than that of the original two–list scheme, but it is performed much less often. In practice, the overhead due to the flush step is negligible.

5.4 Performance

Our consistency schemes incur overhead due to additional cache misses because of cache flushes and due to additional memory bus traffic because of write–throughs. In this section, we analyze these overheads. First we consider the cost of cache–flushing, and show that it is proportional to the cache size and the rate of receiving pages from the network and inversely proportional to the number of pages held in

the free lists and the number of aliases per page. We show that the simple two–list scheme is adequate for designs with small caches, and that cache flushing overhead is insignificant for nearly any practical design when the scheme with physical address aliasing is used. Next, we consider the overhead due to the write–through caching policy for shared data. This overhead depends on the design of the caches write buffer. We consider a design with a single–stage write buffer and two designs with large buffers. The efficiency of our schemes depend on workload parameters. We show that excellent performance is achieved for parameters from published workloads and that this performance is robust; that is, the schemes tolerate large variations from the published values. Finally, we examine limitations of the analytical model used to derive these results.

We estimate overhead in cycles per instruction (CPI). Since most architectures have CPI figures slightly greater than one, this is a conservative measure. For example, an overhead of 0.01 CPI indicates that the consistency scheme accounts for *at most* 1% of the total execution time.

Modeling cache–flushing overhead

The cache flushing operations of the two–list and aliasing schemes incur overheads for two reasons. First, a cache flush itself requires time. A cache flush in our schemes requires scanning the cache and invalidating lines that correspond to the dual–port memory. However, most caches do not provide for selective invalidation. In practice, the flushing step will invalidate all cache lines, and this requires writing back any dirty cache lines. In either case, the flush time is roughly proportional to the cache size. Second, the cache miss ratio will be higher after a flush. To reload the entire contents of the cache (the worst case), requires time proportional to the size of the cache. Thus, the total overhead for a cache flush is proportional to the cache size.

To keep the amortized flushing overhead small, cache flushes must be infrequent. The frequency of cache flushes (and therefore the cache flushing overhead) is proportional to the rate of receiving pages from the network, and inversely proportional to the number of pages that can be received between flushes. The rate of incoming pages is a workload parameter. The number of pages that can be received between

flushes depends upon the choice of the consistency schemes and the policies of the virtual memory system.

Combining the above observations, we obtain,

$$C_f \leq \frac{N_{cache}(T_l + T_a)}{I_p F} \tag{13}$$

where,

C_f = Cache flushing overhead (in CPI).

N_{cache} = Size of the cache (in cache lines).

T_l = Worst–case time to write an entire cache line (in cycles).

T_a = Worst–case time to access the first word in a cache line (in cycles).

I_p = Average number of instructions executed between receiving pages from the network.

F = Number of pages that can be received between cache flushes.

The formula above reflects the worst case where all cache lines are dirty when the cache is flushed, and where all cache lines are live (i.e., would have been referenced again before being evicted if the cache had not been flushed).

When the two–list scheme is employed without aliases, $F = \alpha P_D$, where α is the fraction of pages that the virtual memory system keeps free, and P_D is the number of physical page frames in the dual–port memory. When aliases are employed, $F = \alpha P_A$, where P_A is the size of the physical address space mapped to the dual–port memory (in pages). Typically, $P_A \gg P_D$, which allows the alias scheme to have very low overheads.

We illustrate these estimates of C_f with an example. Consider a processor that has an eight–byte wide data bus and an 8K byte cache with 32 byte cache lines. Assume that wrap around cache line replacment is employed, the first access to a portion of a cache line requires six processor cycles, and the remaining three transfers take two cycles each. For this processor, $N_{cache} = 512$, $T_l = 18$ cycles, and $T_a = 6$

cycles. The processor has 2 Mbytes of dual–port memory implemented with VRAM and pages are 4 Kbytes; thus $P_D = 512$. Physical addresses are 32 bits; By using the top address bit to select the dual–port memory, $P_A = 524,288$ (i.e. 2 Gbytes). I_p is a workload parameter. For shared virtual memory multiprocessing, read faults occur at intervals for which we use a worst–case value of 15,000 (see table 36, page 155). The value of α depends on the policies of the memory management software. We use $\alpha = 0.05$ in the following examples (again, see table 36, page 155).

With these parameters, $C_f < 0.016$ CPI with the two–list scheme, and it vanishes for all practical purposes when aliases are used. However, if a larger cache is used, the overhead of the two–list scheme without aliases can become unacceptable. For example, with a 1 Mbyte cache, C_f increases to 0.6 CPI. With aliases, C_f drops to less than 0.002 CPI. We conclude that the two–list scheme without aliases is practical for designs with small caches. When aliases are used, the cache flushing overhead is negligible for any practical design.

Modeling write–through overhead

Caching shared data with a write–through policy causes overhead for two reasons. First, a write–through may be blocked by a previous memory reference. Second, a write–through may cause a subsequent reference to block. The scenarios under which blocking occurs depend on the size and configuration of write buffers in the design. To simplify the analysis, we assume the following configuration:

Assumption W1: The write buffer for the dual–port memory is separate from the write buffer for the local memory.

Within this framework, we consider three variations:

1. A single–stage write buffer.

2. An unbounded write buffer. Reads may not bypass pending writes.

3. An unbounded write buffer. Reads may bypass pending writes.

A real buffer will have some fixed capacity; however, the analysis is much simpler in the unbounded case than for arbitrary buffer capacity, and the results are more perspicuous. We also show that the behavior of an unbounded buffer can be closely approximated by a small buffer. To derive estimates of the overheads, we use the following workload model:

Assumption W2: The number of instructions executed per write to the dual–port memory is geometrically distributed (i.e., the discrete version of an exponential distribution) with mean $1/p_{sw}$. Stated otherwise, p_{sw} is the probability that an instruction is a write to shared data, (i.e. the dual–port memory). We are not aware of any characterization of the distribution of shared writes for real applications.

Assumption W3: The number of instructions executed per read of the dual–port memory is geometrically distributed with mean $1/p_{ms}$. As reads only occur due to cache misses, p_{ms} is the probability that an instruction causes a cache miss for shared data.

Assumption W4: The time to perform a write is T_c, a constant (worst–case memory cycle time).

We first consider the single stage write buffer. Blocking occurs when: (1) the processor issues a write to the dual–port memory and the write buffer is already full, or (2) the processor issues a read to the dual–port memory and a write is in progress. In appendix C.1.1 we show (equation 16, page 149)

$$C_{w,1} \leq \frac{T_c(T_c-1)}{2} p_{sw}(p_{sw}+p_{ms})$$

$C_{w,1}$ denotes the overhead due to the write–through policy for a single stage write buffer.

Intuitively, $T_c p_{sw}$ is an upper bound (ignoring stalling) of the utilization of the write buffer, i.e. the probability that the write buffer is occupied when the processor executes a cycle. $(T_c-1)/2$, corresponds to the expected time remaining to complete the current write, if such a write is in progress. The product of these two terms

$$\frac{T_c(T_c-1)}{2} p_{sw}$$

describes the expected overhead per read or write to the dual–port memory. The probability that an instruction accesses the dual–port memory is at most $(p_{sw} + p_{ms})$. The bound for $C_{w,1}$ given above follows from these observations.

Typically, $p_{sw} \gg p_{ms}$, and blocking writes are the dominant source of overhead. This can be alleviated by increasing the size of the write buffer.

If an unbounded write buffer is assumed, overhead only arises due to blocking reads. In a simple design where reads cannot bypass pending writes

$$C_{w,\infty} \quad \leq \quad \frac{T_c(T_c - 1)}{2(1 - T_c p_{sw})} p_{sw} p_{ms}, \quad T_c p_{sw} < 1$$

$C_{w,\infty}$ denotes the overhead due to the write–through policy for an unbounded write buffer without bypassing (see equation 17, page 150).

Again, $T_c p_{sw}$ corresponds to the probability that the write–buffer is occupied. Because multiple writes may be pending in the buffer, the expected time remaining to complete all outstanding writes, given that there is at one, is greater than for the single stage buffer. In appendix C.1.2 a queueing analysis shows that this increase is by at most a factor of $1/(1 - T_c p_{sw})$ and yields a bound of

$$\frac{(T_c - 1)}{2(1 - T_c p_{sw})}$$

for the expected overhead per read to the shared memory (there is no overhead for writes, because writes never block with an unbounded buffer). The probability that an instruction is a read (and a cache miss) to the dual–port memory is p_{ms}. The bound for $C_{w,\infty}$ given above follows from these observations.

Although the expected overhead per stalling operation is greater, far fewer operations stall when the unbounded write buffer is used. The net effect is a significant reduction in total overhead.

The write buffer can be further refined by allowing reads to bypass pending writes. This can be implemented, for example, by including hardware to scan the addresses in the write buffer when a read is performed. If a match is found, data is supplied by the buffer instead of by the dual–port memory. In appendix C.1.3 we show (equation 18, page 150)

$$C_{wb,\infty} \quad = \quad \frac{{T_c}^2}{2} p_{sw} p_{ms}, \quad T_c p_{sw} < 1\,.$$

$C_{wb,\infty}$ denotes the overhead due to the write–through policy for an unbounded write buffer with bypassing.

$T_c p_{sw}$ is again the probability that a write is in progress. $T_c/2$ is the expected time for for an in progress write to complete. Assuming that an in progress write is allowed to complete and that all other writes are bypassed when a read to the dual–port memory is issued

$$\frac{(T_c - 1)}{2(1 - T_c p_{sw})}$$

gives the expected overhead per read to the shared memory. The bound for $C_{wb,\infty}$ given above follows from these observations.

These estimates for C_w depend on four parameters: memory access time T_c; write buffer capacity; the probability that an instruction is a write to shared data, p_{sw}; and the probability that an instruction is a cache miss to shared data, p_{ms}. For example, the processor we considered had $T_c = 9$. Using the worst case values from table 36 for estimates of p_{sw} (0.02), and p_{ms} (0.004), we obtain $C_{w,1} \leq 0.018$ CPI. If an unbounded write buffer is used without bypassing of reads, $C_{w,\infty}$ is less than 0.0036 CPI. When bypassing is allowed, overhead is even lower with $C_{wb,\infty} \leq 0.0033$ CPI. In the next section, we consider the sensitivity of these estimates to variations in the parameters of the model.

Robustness of the schemes

The overhead incurred by using our consistency schemes depends on both hardware parameters (e.g. cache size and memory access time) and workload statistics (e.g. rate of receiving pages from the network, cache miss rate, rate of writes to the dual–port memory). We considered examples using architectural parameters from a real design, and worst–case workload statistics from three studies. Under these assumptions, the overhead due to flushing is less than 0.02 CPI for the two–list scheme without aliases when used with a small cache, and negligible when aliases were used for any practical cache size. The overhead due to the write–through policy for shared data is less than 0.02 CPI when a single stage write buffer is employed, and less than 0.004 CPI with a large buffer. We now examine the robustness of our schemes to variations in the

architectural and workload parameters.

We first consider cache flushing overhead, C_f, for the scheme with aliases. In the example above, the overhead was less than 0.002 for a processor with a 1 Mbyte cache. We note that C_f is proportional to $1/I_p$, the rate of receiving pages from the network, and pages are received in response to page faults for shared data. $C_f I_p = N_{cache}(T_l + T_a)/F$ is the amortized flushing cost per page fault. For the design considered, this cost is 30 cycles per page fault. Clearly, the overhead of the virtual memory management software will be much greater than this. Therefore, the cache flushing cost will be much less than the overhead of the virtual memory management software for any value of I_p. Furthermore, this relationship will continue to apply even if the cache is substantially larger or the memory access time is substantially longer than the values considered in this paper. When physical address aliasing is employed, the overhead due to cache flushing is insignificant for any practical design.

We now consider C_w. Our estimates depended on four parameters, the write–buffer capacity, T_c, p_{sw}, and p_{ms}. Both C_w and the overhead due to cache miss processing for shared data are proportional to p_{ms}. If the other three parameters have reasonable values, then C_w will be less than the cache miss overhead. Therefore, we focus this discussion on T_c, p_{sw}, and the write–buffer capacity.

We first consider the write–buffer capacity. We derived bounds for C_w for unbounded buffers with and without bypassing of writes by reads. The unbounded buffers are an idealization; real designs will have bounded buffers. The bounded buffer will have more overhead than the unbounded one due to blocking writes. In appendix C.1.4 we show (equation 19)

$$C_{ww} \quad < \quad p_{sw} T_c \zeta^{(T_c - \frac{k}{p_{sw}})}$$

where C_{ww} is the overhead due to blocking writes, k is the capacity of the write buffer, and ζ is the largest real root of $S(z)$ (see appendix C, equation 20),

$$S(z) \quad = \quad p_{sw} z^{T_c} - z + 1 - p_{sw} \, .$$

In appendix C.1.4, it is also shown (equation 21) that $\zeta > (p_{sw} T_c)^{-1/(T_c - 1)}$.

The intuition for these bounds comes from comparing the operation of a bounded buffer with that of an unbounded one. It can be shown that at the point of executing

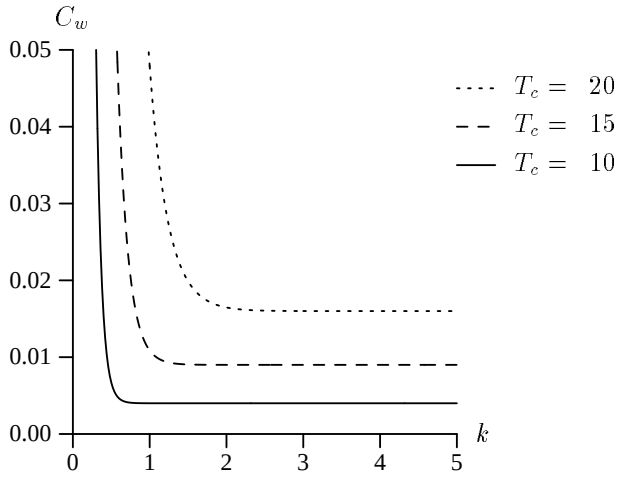


Figure 31: Write–Through overhead vs. write–buffer capacity

any particular instruction, the bounded buffer will hold at most as many pending writes as the unbounded one. Therefore, a write will stall with a k-bounded buffer only if an unbounded buffer processing the same workload holds at least k items. The probability that an unbounded buffer holds more than k items drops exponentially with k for a large variety of arrival and service time distributions. This corresponds to the exponential form for bound for C_{ww} given above. The base of the exponential ζ is based on an approximation due to Kingman (see [Kle76a]). Overhead decreases for larger values of ζ. The bound of $\zeta > (p_{sw}T_c)^{-1/(T_c-1)}$ corresponds to a buffer with an exponential distribution for the memory cycle time.

Figure 31 shows C_{ww} as a function of k for various values of T_c when the other parameters are held at the values used in the above example (i.e. $p_{sw} = 0.02$, and $p_{ms} = 0.004$). We see that C_w quickly approaches its asymptotic value, and the rate depends on T_c. For example, a write buffer of length three is adequate for $T_c \leq 20$. For designs with long memory access times, larger write buffers are needed.

Figure 32 shows C_w as a function of T_c when p_{sw} and p_{ms} are held at their nominal values. Three write–buffer configurations are considered, a single stage buffer ($k = 1$), and unbounded buffers ($k = \infty$) with and without bypassing of reads. For all three configurations and small values of T_c, C_w grows roughly quadratically with T_c. This

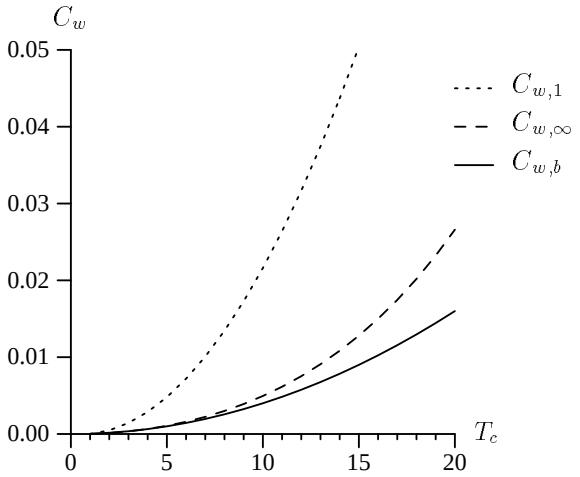


Figure 32: Write–Through overhead vs. Memory cycle time

is because both the occupancy of the buffer and the waiting time for blocked accesses grow with T_c. The two unbounded buffers become qualitatively different for large values of T_c. When $T_c p_{sw}$ approaches one, the expected write buffer occupancy grows rapidly with increasing T_c. If reads cannot bypass writes, each read must wait for the buffer to empty before it can be performed. With bypassing, only the current write must complete before the read can be performed. Thus, C_w is less sensitive to large values of T_c when the write buffer supports bypassing of reads.

The dependence of C_w on p_{sw} is shown in figure 33. Qualitatively different behavior is observed for each of the three write buffer configurations. We first consider the single stage write buffer. $C_{w,1}$ is dominated by C_{ww}. Increasing p_{sw} increases both the rate of arrivals to the write buffer and the probability that the write buffer is already occupied. Thus, both C_{ww} and $C_{w,1}$ grow quadratically with increasing p_{sw}.

We consider next the case when $k = \infty$ without bypassing. $C_{w,\infty} = C_{wr}$ which is proportional to L, the average number of items in the write buffer. For small values of p_{sw}, L grows linearly with p_{sw}. This linear approximation describes $C_{w,\infty}$ well for $T_c p_{sw} < 0.5$. After this point, the $(1 - T_c p_{sw})$ term in the denominator of the expression for $C_{w,\infty}$ becomes significant, corresponding to a rapid growth in L. $T_c p_{sw} = 0.5$ when $p_{sw} = 0.055$ (for $T_c = 9$ in the above example). This is nearly three

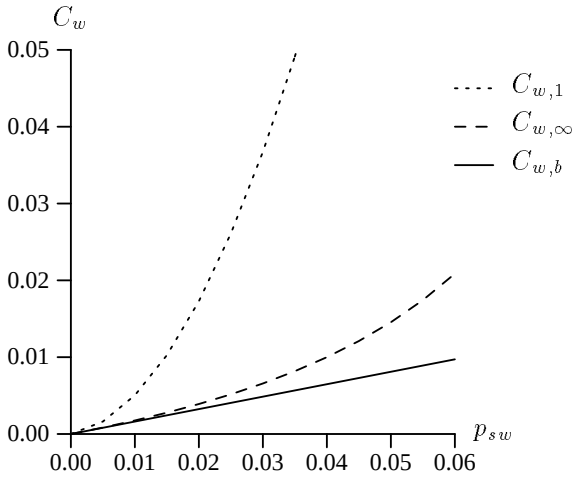


Figure 33: Write–Through overhead vs. Frequency of shared writes

times the worst–case for p_{sw} from table 36 (page 155) and should not be a problem in practice.

When bypassing is included, increasing p_{sw} increases the probability that a write is in progress when a cache miss occurs. This probability grows linearly for small values of p_{sw} and is bounded by 1.0. This accounts for the low values of $C_{wb,\infty}$ for all values of p_{sw} in figure 33. However, if $p_{sw}T_c$ is greater than one, the write buffer occupancy will grow without bound. In this case, approximating the behavior of an infinite write buffer with a finite one is no longer possible. In the example above, this occurs for $P_{sw} > 0.11$, as this is more than five times the worst–case value for p_{sw} from table 36 (page 155), this should not be a problem in practice.

Our consistency schemes incur overhead due to occasional cache flushes and due to the additional memory traffic arising from the write–through caching policy for shared data. When physical address aliasing is employed, the overhead due to cache flushing is much less than the overhead due to other virtual memory management tasks for any conceivable workload. Thus, it is the overhead of write–throughs that must be carefully considered. The most important architectural provision to reduce this overhead is sufficient write buffering. A two– or three–stage write buffer is adequate for the workloads we considered. Allowing reads to bypass writes offers a small

performance improvement, but it is probably not justified unless the cycle time of the dual–port memory is much longer than is typical for current memories and processors.

Limitations of the Model

The analysis in this section was based upon a model that was chosen in part for its mathematical tractability. Real designs will not conform exactly to our model. We now address some of the issues.

In the analysis, we assumed that writes to the shared memory are separated by geometrically distributed intervals. This assumption allowed us to employ embedded Markov chain techniques to analyze the performance of the write buffer. We are aware of no studies of the time distribution of memory references of shared programs. Typical trace generation schemes [EK88, SA88], record the sequence of memory references, but not the exact timings. If an application has a distribution of shared writes that is "burstier" than the geometric distribution, the observed write–through overheads will be higher than predicted by our analysis.

Our analysis assumed constant memory cycle times. There are many reasons this is not the case for real systems. For example, most dynamic memories support fast page mode accesses. If a reference has the same row address as its predecessor, a memory cycle can be performed much faster than the worst–case T_c assumed in our analysis. This increases the throughput of writes and decreases C_w. Another significant departure from the model occurs if the dual–port memory is partitioned into independently addressable banks. Partitioning the dual–port memory also lowers C_w. In particular, such partitioning can be used to avoid the singularity at $T_c p_{sw} = 1$.

We assumed that separate write buffers are used for the dual–port memory and the private, single–port memory. Separating the buffers allowed us to ignore writes to the private memory (due to eviction of dirty cache lines) in our analysis. To consider designs with combined write buffers would require more workload statistics and examining more architectural alternatives. Since the rate of dirty evictions is typically much less than the rate of writes to shared data, the results would not differ significantly from those presented here.

The derivation of workload parameters in appendix C.1 is pessimistic. In particular, we obtained worst–case values for I_p, p_{ms}, and p_{sw} by multiplying the extreme values of parameters from published studies [Li86, EK88, OA89]. We believe that applications that are correspond to the worst–case values of several of these parameters are pathological and extremely rare.

The analysis presented in this section is conservative. All approximations are upper bounds, and the workload statistics are based on published worst–case values. We believe that this supports the viability of our schemes: they perform well, even with unrealistically hostile assumptions. If typical values of workload statistics were used, the predicted overheads would be much lower.

5.5 Virtually Addressed Caches

Virtually addressed caches [HE+86, CG+88, HH+90], are becoming increasingly popular, primarily because large, direct–mapped, virtually addressed caches can be easily built using commercially available static RAMs. Cache hits can be obtained quickly because no address translation is necessary to select a cache line. However, the two–list and alias schemes exploit address translation to distinguish different instantiations of the same virtual page. Although we are not aware of any way to adapt the two–list scheme for use with virtually addressed caches, the alias technique can still be applied.

When large virtually addressed caches are used, some provision must be made to distinguish between address spaces. Otherwise, a cache flush is required at each change of address space (e.g., for each context switch). A common solution is to add an address space identifier to the tag field of each cache line. On a cache hit, the address space identifier of the cache line is compared with that of the current process. If they match, the hit is accepted; otherwise, a miss occurs.

Address space identifiers can be used to implement the alias consistency scheme. Instead of using one identifier for each address space, each page is allocated with an identifier that is stored in the per process page table. Each time a page frame is allocated for the same virtual page, the new allocation receives a new identifier. As in

the alias scheme for physically addressed caches, a cache flush is required only when the set of possible identifiers for a page is exhausted. On a cache hit, the identifier of the cache line is compared with that in the current per process page table. If they match, the hit is accepted; otherwise, a miss occurs. A similar scheme has been proposed by Smith [Smi85a] (see section 5.6). To perform the comparison quickly, a TLB should be used to hold the identifiers for active pages; the TLB must be flushed on changes of address space. As discussed in [Jou89], comparisons to validate a cache hit can be performed without increasing cache latency by overlapping the comparisons with other operations of the execution pipeline.

5.6 Related Work

There is a large body of literature on cache consistency. Most of the related work focuses on the cache consistency problems in DMA-based memory systems, and on the cache coherency problems for shared–memory multiprocessors. Although several multiprocessors have been implemented using dual–port memories for low–latency communication [Par87, Nic88, JKP89, Ser90, SB$^+$90], to the best of our knowledge, all of these designs make the dual–port memory uncachable.

Many processors make the DMA memory buffers uncachable or perform DMA operations through the cache [Smi82]. Our analysis shows that making shared data uncachable incurs an overhead of 0.3 to 1.5 CPI for typical multiprocessor applications. Performing DMA operations through the cache effectively flushes the cache for large data transfers. Thus, either of these approaches incur large performance penalties. Our schemes enjoy the simplicity of these schemes while providing a clear performance advantage.

Our physical address aliases resemble the "one–time–identifier" (OTI) method proposed by Smith [Smi85a]. The OTI method adds a register to the TLB to generate a unique tag for each TLB entry, and a cache flush is performed when this register overflows. The OTI scheme is implemented by special purpose hardware, whereas our schemes are implemented in the virtual memory software. This makes our approach more flexible than the OTI approach. Furthermore, the OTI scheme requires a new

identifier each time a new TLB entry is created, while our alias scheme only increments the alias counter each time the uncached set of pages is exhausted. Therefore, the alias scheme can be implemented with fewer identifier bits or less cache flushing overhead than the OTI method. Smith gives several variations and optimizations such as support for virtual address caches, process migration, etc. These techniques can be applied directly to our schemes.

Snooping cache schemes [AB86, EK89b] are used for solving both the cache consistency problem with DMA operations and the cache coherency problem of shared–memory multiprocessors. In such schemes, all caches "snoop" all memory bus transactions and take appropriate actions to preserve consistency. When a DMA operation writes to a location currently held in the cache, the cache either invalidates [KE+85] or updates [McC84, TS87] its copy, solving the stale cache data problem. To solve the stale memory data problem, either a write–through policy is used (e.g. the VAX 8800), or the cache provides the data on DMA reads to locations that it holds [McC84, KE+85, TS87]. Being a hardware scheme, snooping does not force the programmer to adopt a particular (i.e., page–oriented) communication paradigm. However, the snooping cache method does not solve the cache consistency problem with dual–port memories because data transfers via the second port do not use the processor's memory bus.

Directory–based schemes [Tan76, CG+88, AS+88] use a copy of the cache directory to determine when consistency actions are needed. This is the approach used in the IBM 3090 [Tuc86]. In this machine, data transfers between caches, main memory, and I/O devices are managed by "system control elements" (SCEs). Each SCE maintains a copy of the directory for each cache in the system. When a DMA operation reads a cached location, the SCE detects the reads and requests data from the appropriate cache. This requires arbitration for the cache, and nullifies the independence of the network and cache designs that dual–port memories offer. To solve the stale cache data problem, the 3090 provides a special instruction that allows the CPU to access each set of the cache, purging entries on a particular page. Such a sophisticated cache design could be used to solve the cache consistency problem with dual–port memories; however, our schemes are simpler and appear better suited for large scale

multiprocessors.

5.7 Conclusions

Using dual–port memories such as VRAMs to implement part of the main physical memory space is an attractive approach to exploit high bandwidth interprocessor communication. However, they introduce consistency problems that cannot be solved by traditional snooping techniques. We have presented software–based solutions for these problems. Our solutions employ a combination of cache flushing and selective write–through of shared data. Our solutions are simple: they do not require special hardware, and they can be easily incorporated into virtual memory management software.

We have shown that our schemes are practical and efficient. Although cache flushes are required, our schemes ensure that these flushes are infrequent, and the overhead due to cache flushing is insignificant. The write–through policy for shared data is the dominant source of overhead for our schemes. This overhead is reduced by write buffering. With a write buffer of two or three stages, the overhead due to the write–through policy can be kept to less than one percent of the total execution time.

Chapter 6

Conclusions

This dissertation has addressed the design of hardware for high bandwidth communication, focusing on issues of timing and synchronization. A new approach for addressing these issues called STARI, "Self-Timed At Receiver's Input," was presented. It is motivated by the observation that it is straightforward to distribute a frequency reference but much harder to control the relative phase of a signal at each point in the system. As in a synchronous system, STARI requires an exact match in frequency (i.e. rate of operation) of the transmitter and the receiver; however, the tolerance of phase variation (i.e. skew) is large, but bounded. STARI is a hybrid of synchronous and self–timed designs: the synchronous part of the design assures the transmitter and receiver operate at the same frequency; the self–timed part of the design compensates for phase variations. STARI is simple to implement and particularly well suited for communication in tightly coupled designs, such as multiprocessors and multicomputers.

6.1 Contributions

A novel approach to communication, STARI, was presented. STARI combines synchronous and self–timed circuits. It is simple and provably correct.

A chip that implements the STARI technique was designed, fabricated and tested. Although STARI is provably correct, the chip failed to operate due to an analog race

that was neither represented in the circuit model used for the chip nor manifested in simulation before the chip was sent to fabrication. Once the chip was tested and the error identified, careful circuit simulation (aimed at recreating the problem) showed the underlying phenomenon. This experience revealed limitations of discrete models (either as employed in formal proofs or less formally by typical simulators).

In simulation, the chip operated at a rate of 700 Mbits/sec over the range of Spice parameters provided by the vendor. In test, the chip operated at 100 Mbits/sec, the maximum speed supported by available test equipment that could support test sequences that avoided the problems of the chip.

The performance of self–timed circuits with probabilistic delays was asymptotically analyzed. It was shown that self–timed pipelines can achieve linear speed–ups. For various delay distributions, exact results, upper and lower bounds, and approximations of the pipeline's processor utilization were derived.

A software coherency scheme was proposed for dual–port memory. This consistency scheme can be implemented as a part of each processor's virtual memory management software. An analysis of the scheme based on published workload statistics shows that the overheads are negligible.

6.2 Future Research

For each question answered by the research described in this dissertation, many more were raised. The performance analysis of self–timed circuits considered one–dimensional pipelines. Extending this work to two–dimensional (e.g. meshes) and higher dimensional networks is a natural extension of this work. The simple chain–of–queues analogy that applied to the one–dimensional case does not apply to networks of higher connectivity. Although higher dimensional network can be modeled as Markov chains, the size of the state space grows exponentially with the size of the processor array, and it is not clear how to obtain asymptotic results from such analysis.

The exponential processing–time assumption used to obtain exact results for self–timed pipelines is probably not realistic. Developing models for the processing time

distributions of self–timed circuits would further clarify the relative strengths an weaknesses of these designs. To obtain asymptotic performance estimates for processing time distributions that are non-exponential requires finding the fixed–point distribution for a chain of G/G/1 queues. Given the sparsity of analytical results in this area [Kle76b], this appears to be a very difficult problem.

The most serious omission of the asymptotic analysis is the absence of a suitable model for the impact of the packaging hierarchy. Such a model would necessarily be *ad–hoc*; however a model that is both realistic and mathematically tractable would help bridge the areas of theory and practice in hardware performance analysis.

An obvious topic for further work with STARI is to implement a working chip. Such a design is in progress. Two of the three bugs show pitfalls of designing high performance digital circuits and suggest areas for further research in design verification. In particular, better formalisms linking the continuous, analog behavior of a circuit to discrete, digital abstractions are needed. This topic has been addressed to a limited extent by work in qualitative reasoning (e.g. [dK85, Wil85]). However the circuit models considered in these analyses are too simplistic to capture the behavior of high performance circuits; they would conclude that the circuit can exhibit one of many behaviors, including the desired one, but not excluding many undesired, unrealizable ones. The more quantitative approaches described in [AE+89] shows promise for these applications. New techniques are necessary to apply these methods to circuits where neither the device parameters nor inputs to the circuit are completely determined.

STARI presents a new solution to the timing issues of a communications interface, one where an exact match of frequency is assumed but delays within the system are not specified. The formal methods described in this dissertation provide a framework for evaluating untraditional designs, such as STARI. Further development of these methods is likely to lead to more novel and useful designs.

The correctness proof for STARI was manually generated. Standard timing verification methods are not applicable to STARI, and most theorem–proving programs [Gor85, BM88, GG89, BDV90] do not support theories of real numbers and inequalities. Manual proofs are too tedious and error–prone to be practical. Jim

Saxe [Sax92] has suggested that the following mechanical procedure could be applied to STARI (and to similar timing verification problems for self–timed circuits, see [HC90]) by augmenting a traditional theorem prover with a decision procedure for linear programs [PS82]:

1. For each transition in the program, let s_1 and s_2 denote states before and after executing the transition.

2. Compute critical pairs [KB69] of the transition's precondition in state s_1, the invariant in state s_1, the action of the transition, and the *negation* of the invariant in state s_2. If this yields a contradiction, then the transition has been shown to maintain the invariant.

3. Otherwise, consider the set of linear inequalities derived in step 2. If this is an infeasible linear program, then the transition has been shown to maintain the invariant.

Although this approach is not complete for all possible programs and invariants, it appears to be sufficient for STARI and similar timing verification problems. One area for further research is to implement linear programming in a theorem prover and study proof techniques for programs with real–time properties (e.g. STARI).

The timing requirements of STARI are simple: the FIFO must insert each datum before the next arrives, and output each datum before the corresponding acknowledge arrives. On the other hand, devising an invariant that guarantees such conditions is often the hardest part of the verification. There has been much work on systematically deriving invariants (e.g. [Lam87, CM88]); applying these methods to hardware verification is a promising area for further research.

The performance analysis of the memory consistency protocols for multiprocessors based on dual–port memories showed that the efficiency of these protocols depended primarily on two workload ratios: (1) the rate of writes to shared data relative to the rate at which they can be committed to main memory, and (2) the rate of remote page faults (in a shared virtual memory environment) relative to the cost of a cache flush. It is reasonable to expect each of these ratios to increase (become less favorable) with time.

Processor cycle times are decreasing faster than memory cycle times [HP90a]. Accordingly, write traffic will eventually become a bottleneck in the consistency schemes presented in this dissertation. The following observations suggest that this should not be a critical problem. Memory can be interleaved, and memory latency is not crucial for the write traffic. A high–bandwidth serial interconnect (e.g. using STARI) separate from the main memory bus could transmit writes from the processor to interleaved memory modules. This provides a scalable solution for the write traffic. It also introduces two more consistency problems. First, before transmitting a page, all pending writes (to that page) must complete. The context switch time for a local processor to change from running an application process to servicing a remote page request should be sufficient for this. Second, this allows reads to bypass pending writes. Existing schemes for weak consistency [AH90, GL+90] address this problem.

In chapter 5, it was argued that cache flush overheads are small compared to the cost of handling remote page faults. This reveals the most speculative aspect of these consistency schemes. Page–based consistency has been shown to work well with simple applications such as matrix multiplication [Li86], but much less data is available for large applications. Recent studies [PL92] suggest that page–based coherency works well for many applications on small–scale multiprocessors. More study is needed to confirm these claims for larger multiprocessors and more applications.

Bibliography

[AB86] James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[AE+89] Harold Abelson, Michael Eisenberg, et al. Intelligence in scientific computing. *Communications of the ACM*, 32(5):546–562, May 1989.

[AH90] Sarita V. Adve and Mark D. Hill. Weak ordering - a new definition. In *Seventeenth Annual International Symposium on Computer Architecture, Seattle.*, pages 2–14. IEEE, May 1990.

[AL89] Martín Abadi and Leslie Lamport. Composing specifications. In J.W. de Bakker et al., editors, *Proceedings of the REX Workshop, "Stepwise Refinement of Distributed Systems"*. Springer-Verlag, 1989. LNCS 430.

[AL92] Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. In J.W. de Bakker et al., editors, *Proceedings of the REX Workshop, "Real-Time: Theory in Practice"*. Springer-Verlag, 1992. LNCS 600.

[All78] Arnold O. Allen. *Probability, Statistics, and Queuing Theory, with Computer Science Applications*. Academic Press, New York, 1978.

[Amd67] G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceeding AFIPS 1967 Spring Joint Computer Conference*, pages 483–485, Atlantic City, NJ, 1967.

[AMD89] AMD Inc. *The SUPERNET Family for FDDI*, 1989.

[Apo67] Thomas M. Apostle. *Calculus*, volume 1. John Wiley and Sons, Inc., New York, second edition, 1967.

[AS+88] Anant Agarwal, Richard Simoni, et al. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 280–289, May 1988.

[Bak90] H.B. Bakoglu. *Circuits, Interconnections, and Packaging for VLSI.* Addison-Wesley, 1990.

[Bar84] H. Barrow. Proving the correctness of digital hardware designs. *VLSI Design*, 5(7), July 1984.

[BB91] Mario Blaum and Jehoshua Bruck. Coding for skew-tolerant parallel asynchronous communications. Technical Report RJ 8268, IBM Almaden Research Center, July 1991. submitted for publication.

[BDV90] David A. Basin and Peter Del Vecchio. Verification of combinational logic in Nuprl. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, pages 333–357. Springer-Verlag, 1990. Lecture Notes in Computer Science, 408.

[BM88] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook.* Academic Press, Boston, 1988.

[Bur91] Steven M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits.* PhD thesis, Computer Science Department, California Institute of Technology, Pasadena, CA, January 1991. Tech. Report Caltech-CS-TR-91-01.

[CF89] Alan L. Cox and Robert J. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with PLATINUM. Technical Report 263, Department of Computer Science, University of Rochester, March 1989.

[CG+88] David R. Cheriton, Anoop Gupta, et al. The VMP multiprocessor: Initial experience, refinements and performance evaluation. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 410–421, May 1988.

[Cha84] Daniel M. Chapiro. *Globally-Asynchronous, Locally-Synchronous Systems.* PhD thesis, Department of Computer Science, Stanford University, October 1984. Tech. Report STAN-CS-84-1026.

[Che86] Marina C. Chen. Very-high-level parallel programming in Crystal. Technical Report YALEU/DCS/RR-506, Department of Computer Science, Yale University, December 1986.

[Chi90] Vernon L. Chi. Salphasic distribution of clock signals. Technical Report 90-026, Department of Computer Science, University of North Carolina, June 1990.

[Chr90] Jacob Michael Christensen. Efficient synthesis of hierarchical circuit descriptions. Master's thesis, Department of Computer Science, Technical University of Denmark, August 1990.

[CM73] T.J. Chaney and C.E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, C-22(4):421–422, April 1973.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[Coh88] A.J. Cohn. A proof of correctness of the VIPER microprocessors: The first level. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 27–71. Kluwer, Norwell, MA, 1988.

[Coh89] A.J. Cohn. A proof of correctness of the VIPER microprocessors: The second level. In G. Birtwistle and P.A. Subrahmanyam, editors, *Trends in Hardware Verification and Automated Theorem Proving*, pages 1–91. Springer Verlag, New York, 1989.

[Dal87] William J. Dally. *A VLSI Architecture for Concurrent Data Structures.* Kluwer Academic, Boston, 1987.

[Dal90] William J. Dally. Network and processor architecture of message–driven computers. In Robert Suaya and Graham Birtwistle, editors, *VLSI and Parallel Computation*, pages 140–222. Morgan Kaufmann, San Mateo, CA, 1990.

[Dil88] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1988. Published in book form as part of the ACM Doctoral Dissertation Award Series by the MIT Press, Cambridge, MA, 1989.

[Dil89] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, June 1989.

[dK85] Johan de Kleer. How circuits work. In Daniel G. Bobrow, editor, *Qualitative Reasoning About Physical Systems*, pages 205–280. MIT Press, Cambridge, MA, 1985.

[DNS89] David L. Dill, Steven M. Nowick, and Robert F. Sproull. Specification and automatic verification of self-timed queues. Technical Report CSL-TR-89-387, Computer Systems Laboratory, Stanford University, August 1989.

[DS90] Marios D. Dikaiakos and Kenneth Steiglitz. Comparison of tree and straight-line clocking for long systolic arrays. Technical Report CS-TR-271-90, Department of Computer Science, Princeton University, June 1990.

[DS91] Marios D. Dikaiakos and Kenneth Steiglitz. Comparison of tree and straight-line clocking for long systolic arrays. *Journal of VLSI Signal Processing*, 2(4), 1991.

[EK88] Susan J. Eggers and Randy H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 373–382, May 1988.

[EK89a] Susan J. Eggers and Randy H. Katz. The effect of sharing on the cache and bus performance of parallel programs. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–270, April 1989.

[EK89b] Susan J. Eggers and Randy H. Katz. Evaluating the performance of four snooping cache coherency protocols. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 2–15, June 1989.

[EK+89c] D.J. Erdman, S.W. Kenkel, et al. CAzM: A numerically robust, table-based circuit simulator. Technical Report TR89-23, Microelectronics Center of North Carolina, Research Triangle Park, North Carolina, July 1989.

[Eur] European Silicon Structures, Bracknell, Berkshire RG12 3DY, United Kingdom. *Designing in MODEL*, doc. no. ES2-014-0055 edition. SOLO Reference Manual.

[Fel50a] William Feller. *An Introduction to Probability Theory and Its Applications*, volume 2. John-Wiley and Sons, 1950.

[Fel50b] William Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. John-Wiley and Sons, 1950.

[FHP86] U. Frish, B. Hasslacher, and Y. Pomeau. A lattice gas automaton for the Navier-Stokes equation. *Physics Review Letters*, 56:1505–1508, 1986.

[FK85] Allan L. Fisher and H.T. Kung. Synchronizing large vlsi processor arrays. *IEEE Transactions on Computers*, C-34(8):734–740, August 1985.

[FLS65] Richard P. Feynman, Robert B. Leighton, and Matthew L. Sands. *The Feynman Lectures on Physics*, volume 3. Addison-Wesley, Reading, MA, 1965.

[GD85] Lance A. Glasser and Daniel W. Dobberpuhl. *The Design and Analysis of VLSI Circuits*. Addison-Wesley, 1985.

[GG89] Stephen J. Garland and John V. Guttag. An overview of LP: the Larch Prover. In *Proceedings of the Third International Conference on Rewriting Techniques and Applications*. Springer-Verlag, 1989. LNCS 385.

[GH85] Michael J.C. Gordon and John Herbert. A formal hardware verification methodology and its application to a network interface chip. Technical Report 66, Computing Laboratory, Cambridge University, 1985.

[GL+90] Kourosh Gharachorloo, Daniel Lenoski, et al. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Seventeenth Annual International Symposium on Computer Architecture, Seattle.*, pages 15–26. IEEE, May 1990.

[Gor85] Michael J.C. Gordon. HOL: a machine oriented formulation of higher order logic. Technical Report 68, Computing Laboratory, Cambridge University, 1985.

[Gor86] Michael J.C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G.J. Milne and P.A. Subrahmanyan, editors, *Formal aspects of VLSI design*, pages 153–177. Elsevier, 1986.

[Gra90] Brian T. Graham. SECD: The design and verification of a functional microprocessor. Master's thesis, Dept. of Computer Science, University of Calgary, Calgary, Alberta, Canada, 1990.

[Gra92] Brian T. Graham. *The SECD Microprocessor: A Verification Case Study*. Kluwer, Norwell, MA, 1992. An earlier version appeared as [Gra90].

[Gre92] Mark R. Greenstreet. Using Synchronized Transitions for simulation and timing verification. In Jørgen Staunstrup and Robin Sharp, editors, *1992 Workshop on Designing Correct Circuits*, pages 215–236, Lyngby, Denmark, January 1992. Elsevier. An earlier version published as Matsushita Information Technology Laboratory technical report MITL-TR-01-91.

[GS88a] Mark R. Greenstreet and Jørgen Staunstrup. Atomicity, programs, and hardware. Technical Report ID-TR: 1988-46, Department of Computer Science, Technical University of Denmark, Lyngby, Denmark, November 1988.

[GS88b] Mark R. Greenstreet and Kenneth Steiglitz. Throughput of long self-timed pipelines. Technical Report CS-TR-190-88, Department of Computer Science, Princeton University, Princeton, NJ, November 1988. appeared in 1989 Conference on Information Sciences and systems.

[GS90] Mark R. Greenstreet and Ken Steiglitz. Bubbles can make self-timed pipelines fast. *Journal of VLSI and Signal Processing*, 2(3):139–148, November 1990.

[GSW87] Mark R. Greenstreet, Jørgen Staunstrup, and Ted E. Williams. Self-timed iteration. In Carlo H. Séquin, editor, *Proceedings of VLSI '87*, pages 269–282. IFIP, August 1987.

[HC90] Henrik Hulgaard and Per Henrik Christensen. Automated synthesis of delay insensitive circuits. Master's thesis, Department of Computer Science, Technical University of Denmark, August 1990.

[HE+86] Mark D. Hill, Susan J. Eggers, et al. SPUR: A VLSI multiprocessor workstation. *IEEE Computer*, 19(11):8–22, November 1986.

[HH+90] William Hardell, Jr., Dwain A. Hicks, et al. Data cache and storage control units. In Mamata Misra, editor, *IBM RISC System/6000 Technology*, pages 44–50. International Business Machines, Inc., 1990.

[Hil85] W. Daniel Hillis. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.

[HN86] Andy Hopper and Roger M. Needham. The Cambridge Fast Ring networking system. Technical Report 90, Computing Laboratory, Cambridge University, 1986.

[Hor83] Mark Horowitz. *Performance Modelling for MOS Integrated Circuits.* PhD thesis, Stanford University, October 1983.

[HP90a] John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[HP90b] John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*, chapter 9.7. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[Hun85] Warren A. Hunt, Jr. FM8501: A verified microprocessor. Technical Report ICSCA-CMP-47, University of Texas at Austin, 1985.

[HW89] Paul Horowitz and Paul Winfield. *The Art of Electronics*, chapter 4. Cambridge University Press, second edition, 1989.

[Int89] Intel Corp. *i860^{TM} 64-Bit Microprocessor*, 1989.

[Ive62] Kenneth Iverson. *A Programming Language.* Wiley, 1962.

[JH88] Mark G. Johnson and Edwin L. Hudson. A variable delay line PLL for CPU-coprocessor synchronization. *IEEE Journal of Solid-State Circuits*, 23(5):1218–1223, October 1988.

[JKP89] N. Jagadish, J.M̃ohan Kumar, and L.M. Patnaik. An efficient scheme for interprocessor communication using dual-ported RAMs. *IEEE Micro*, pages 10–19, October 1989.

[Jou89] Norman P. Jouppi. Architectural and organizational tradeoffs in the design of the MultiTitan CPU. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 281–289, June 1989.

[JW74] Kathleen Jensen and Niklaus Wirth. *PASCAL: user manual and report.* Springer-Verlang, 1974.

[Kao90] John S. Kao. *Birth and Death on a Flow: a study of a random particle system and its statistical equilibrium.* PhD thesis, Department of Civil Engineering and Operations Research, Princeton University, December 1990.

[KB69] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, England, 1969.

[KE+85] R.H. Katz, S.J. Eggers, et al. Implementing a cache consistency protocol. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 276–283, June 1985.

[Kin70] J.F.C. Kingman. Inequalities in the theory of queues. *Journal of the Royal Statistical Society, Series B*, 32:102–110, 1970.

[KK88] Thomas F. Knight and Alexander Krymm. A self-terminating low-voltage swing CMOS output driver. *IEEE Journal of Solid-State Circuits*, 23(2):457–464, April 1988.

[Kle75a] Leonard Kleinrock. *Queueing Systems, volume 1: theory.* John Wiley and Sons, 1975.

[Kle75b] Leonard Kleinrock. *Queueing Systems, volume 1: theory*, chapter 5. John Wiley and Sons, 1975.

[Kle76a] Leonard Kleinrock. *Queueing Systems, volume 2: applications*, chapter 2.4. John Wiley and Sons, 1976.

[Kle76b] Leonard Kleinrock. *Queueing Systems, volume 2: applications*. John Wiley and Sons, 1976.

[Kni89] Thomas F. Knight. Personal communication, 1989.

[KRY81] H.T. Kung, L.M. Ruane, and D.W.L. Yen. A systolic array for rapid string comparison. In *Proceedings of the CMU Conference on VLSI Systems and Computation*, Rockeville, MD, 1981. Computer Science Press.

[KS89] Steven D. Kugelmass and Kenneth Steiglitz. A scalable architecture for lattice-gas simulations. *Jounal of Computational Physics*, 84:311–325, 1989.

[KS90] Steven D. Kugelmass and Kenneth Steiglitz. An upper bound on expected clock skew in synchronous systems. *IEEE Transactions on Computers*, 39(12):1475–1477, December 1990. (correspondence).

[Lam86] Leslie Lamport. A simple approach to specifying concurrent systems. Technical Report 15, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, December 1986.

[Lam87] Leslie Lamport. *win* and *sin*: Predicate transformers for concurrency. Technical Report 17, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, May 1987.

[LE90] Richard P. LaRowe Jr. and Carla S. Ellis. Experimental comparison of memory management policies for NUMA multiprocessors. Technical Report CS-1990-10, Department of Computer Science, Duke University, Durham, NC, April 1990.

[LH86] Kai Li and Paul Hudak. Memory coherence in Shared Virtual Memory systems. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, August 1986.

[LH89] Kai Li and Paul Hudak. Memory coherence in Shared Virtual Memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Li86] Kai Li. *Shared Virtual Memory on Loosely-coupled Multiprocessors*. PhD thesis, Department of Computer Science, Yale University, October 1986. Tech Report YALEU-RR-492.

[LL85] Richard J. Lipton and Daniel Lopresti. A systolic array for rapid string comparison. In *Proceedings of the 1985 Chapel Hill Conference on VLSI*, Rockeville, MD, 1985. Computer Science Press.

[LM84] Tzu-Mu Lin and Carver Mead. Signal delay in general RC networks with application to timing simulation of digital integrated circuits. In *Advanced Research in VLSI: Proceedings of the Third MIT Conference*, pages 93–99, January 1984.

[LS84] Leslie Lamport and Fred B. Schneider. The 'Hoare logic' of CSP, and all that. *ACM Transactions on Programming Languages*, 6(2), April 1984.

[LS86] Charles E. Leiserson and James B. Saxe. Retiming synchronous circuitry. Technical Report 13, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, August 1986.

[Mar90] Alain J. Martin. Synthesis of asynchronous VLSI circuits. In Jørgen Staunstrup, editor, *Formal Methods for VLSI Design*, chapter 6, pages 237–283. North-Holland, 1990.

[MB59] David E. Muller and W.S. Bartky. A theory of asynchronous circuits. In *Proc. Int. Symp. Theory Switching*, pages 204–243, 1959.

[MB76] R.M. Metcalfe and D.R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[MBM88] Teresa H.-Y. Meng, Robert W. Brodersen, and David G. Messerschmitt. Asynchronous processor design for digital signal processing. In *Proc. IEEE ICASP 1988*, April 1988.

[MC79] Carver Mead and Lynn Conway. *Introduction to VLSI Systems.* Addison Wesley, 1979.

[McC84] Edward M. McCreight. The Dragon computer system: An early overview. Technical report, Xerox PARC, Palo Alto, CA, June 1984.

[MFR85] C.E. Molnar, T.P. Fang, and F.U. Rosenberger. Synthesis of delay-insensitive modules. In Henry Fuchs, editor, *Proc. 1985 Chapel Hill Conference on VLSI.* Computer Science Press, 1985.

[Mil65] R.E. Miller. *Switching Theory.* Wiley, New York, 1965.

[Mil91] George J. Milne. The formal description and verification of hardware timing. *IEEE Transactions on Computers*, 40(7):811–826, July 1991.

[Mot88] Motorola Inc. *MC88200: 16-kilobyte Cache/Memory Management Unit*, 1988. Rev. 1.

[Nag75] L.W. Nagel. SPICE2: a computer program to simulate semiconductor circuits. Technical Report ERL-M520, Electronics Research Laboratory, University of California, Berkeley, CA, May 1975.

[Nic88] Jean-Daniel Nicoud. Video RAMs: Structure and applications. *IEEE Micro*, pages 8–27, February 1988.

[Nie90] Christian D. Nielsen. Design of delay insensitive circuits using `Synchronized Transitions`. Master's thesis, Department of Computer Science, Technical University of Denmark, 1990.

[Now89] Andreas Nowatzyk. *A Communication Architecture for Multiprocessro Networks.* PhD thesis, School of Computer Science, Carnegie Mellon University, December 1989. Technical report CMU-CS-89-181.

[OA89] Susan Owicki and Anant Agarwal. Evaluating the performance of software cache coherence. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 230–242, April 1989.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(1), 1976.

[Par87] Arvin Park. *A Shared Memory Architecture for Distributed Computing.* PhD thesis, Department of Computer Science, Princeton University, 1987.

[PL92] Karin Petersen and Kai Li. A comparative evaluation of cache coherence schemes based on virtual memory support. Technical Report CS-TR-373-92, Department of Computer Science, Princeton University, 1992.

[Pnu81] Amir Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[PR+84] Raymond Pinkham, Donald J. Redwine, et al. A high speed dual port memory with simultaneous serial and random mode access for video applications. *IEEE Journal of Solid-State Circuits*, SC-19(6):999–1007, December 1984.

[PS82] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice Hall, Englewood Cliffs, NJ, 1982.

[Rei68] Raymond Reiter. Scheduling parallel computations. *Journal of the ACM*, 15(4):590–599, October 1968.

[RG85] Randall D. Rettberg and Lance A. Glasser. Digital phase adjustment. U.S. Patent no. 4,700,347, October 1985.

[RS87] Anders P. Ravn and Jørgen Staunstrup. Synchronized Transitions. Technical Report DAIMI PM-219, Computer Science Department, Aarhus University, Århus, Denmark, January 1987.

[RT+88] R.F. Rashid, A. Tevanian, et al. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers*, 37(8):896–908, August 1988.

[SA88] Richard L. Sites and Anant Agarwal. Multiprocessor cache analysis using ATUM. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 186–195, May 1988.

[Sau89] Jon R. Sauer. An optoelectronic multi-gb/s packet switching network. Technical Report OCS TR 89-06, Optical Computing Systems Laboratory, University of Colorado, Boulder, CO, 1989.

[Sax92] James B. Saxe. Personal communication, January 1992.

[SB+90] Michael D. Schroeder, Andrew D. Birrell, et al. Autonet: a high-speed, self-configuring local area network using point-to-point links. Technical Report 59, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, April 1990.

[Sei79] Charles L. Seitz. System timing. In *Introduction to VLSI Systems*, chapter (Carver Mead and Lynn Conway) 7, pages 218–262. Addison Wesley, 1979.

[Ser90] Dimitrios Serpanos. *Scalable Shared Memory Interconnections.* PhD thesis, Department of Computer Science, Princeton University, July 1990.

[SFC91] Pradeep S. Sindhu, Jean-Marc Frailong, and Michel Cekleov. Formal specification of memory models. Technical Report CSL-91-11, Xerox PARC, December 1991.

[SG88] Jørgen Staunstrup and Mark R. Greenstreet. From high-level descriptions to VLSI circuits. *BIT*, 28(3):620–638, 1988.

[SG89] Jørgen Staunstrup and Mark R. Greenstreet. Designing delay insensitive circuits using "Synchronized Transitions". In *Proceedings of the IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, vol. 2*, pages 741–758, Leuven, Belgium, November 1989. Published by Elsevier as "Formal VLSI Specification and Synthesis-I".

[SG90] Jørgen Staunstrup and Mark R. Greenstreet. Synchronized Transitions. In Jørgen Staunstrup, editor, *Formal Methods for VLSI Design*, chapter 2, pages 71–128. North-Holland, 1990.

[Sin81] N.P. Singh. A design methodology for self-timed systems. Master's thesis, Laboratory for Computer Science, MIT, Cambridge, MA, 1981. Tech. Report TR-258.

[Smi82] Alan Jay Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[Smi85a] Alan Jay Smith. CPU cache consistency with software support and using 'One Time Identifiers'. In *Proceedings of the Pacific Computer Communications Symposium*, pages 142–150, Seoul, Korea, October 1985.

[Smi85b] Alan Jay Smith. Disk cache—miss ratio analysis and design considerations. *ACM Transactions on Computer Systems*, 3(3):161–203, August 1985.

[Smi88] Kenneth C. Smith. Multiple-valued logic: A tutorial and appreciation. *IEEE Computer*, pages 17–27, April 1988.

[Su89] Wen-King Su. Reactive-process programming and distributed discrete event-simulation. Master's thesis, Computer Science Department, California Institute of Technology, 1989.

[Sut89] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989. Turing Award lecture.

[Sze81] S.M. Sze. *Physics of Semiconductor Devices.* Wiley, New York, second edition, 1981.

[Tan76] C.K. Tang. Cache design in the tightly coupled multiprocessor system. In *AFIPS Conference Proceedings, National Computer Conference*, pages 749–753, New York, NY, June 1976.

[Thi87] *Connection Machine Model CM-2 Technical Summary*, April 1987.

[Thi91a] Lothar Thiele. On the analysis and optimization of selftimed processor arrays. *INTEGRATION*, 12(2):167–187, December 1991.

[Thi91b] *The Connection Machine CM-5 Technical Summary*, October 1991.

[TS87] Charles P. Thacker and Lawrence C. Stewart. Firefly: a multiprocessor workstation. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–172, October 1987.

[Tuc86] Stuart G. Tucker. The IBM 3090 system: An overview. *IBM Systems Journal*, 25(1):4–19, November 1986.

[Udd84] Jan T. Udding. *Classification and Composition of Delay-Insensitive Circuits*. PhD thesis, Eindhoven University of Technology, 1984.

[Ver88] Tom Verhoeff. Delay-insensitive codes – an overview. *Distributed Computing*, 3:1–8, 1988.

[WH81] Patrick H. Winston and Berthold K.P. Horn. *Lisp*. Addison-Wesley, 1981.

[WH+87] T.E. Williams, M. Horowitz, et al. A self-timed chip for division. In *Proceedings of the Conference on Advanced Research in VLSI*. Computer Science Press, 1987.

[Wil85] Brian C. Williams. Qualitative analysis of MOS circuits. In Daniel G. Bobrow, editor, *Qualitative Reasoning About Physical Systems*, pages 281–346. MIT Press, Cambridge, MA, 1985.

[Wil91] Ted E. Williams. *Self-Timed Rings and their Application to Division*. PhD thesis, Stanford University, May 1991.

[Wir85] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlang, third edition, 1985.

[YS89] Jiren Yuan and Christer Svensson. High-speed CMOS circuit technique. *IEEE Journal of Solid-State Circuits*, 24(1):62–70, February 1989.

Appendix A

Properties of Self–Timed Pipelines

This appendix presents three theorems for pipelines. First we present a sufficient condition to exclude deadlock. Then we give a proof that the waiting time cannot be decreased by increasing the delay of one or more operations in a realization. Finally, we show sufficient conditions under which the average waiting time T is well defined.

Let n be the number of processors (and therefore the number of storage elements) in the pipeline, b be the number of bubbles, and $s = n/b$ be the average spacing of bubbles. All operations on processor, storage element, and loop indices are implicitly modulo n. Each loop (as described in section 3) can be in one of the three states *down*, *left*, or *right*. Assume that processing times are identically and independently distributed, and that t_f is a random variable with this distribution. Likewise, assume that storage times are identically and independently distributed, and that t_c is a random variable with this distribution.

Theorem 1 *For any pipeline with $1 < s \leq n$, and for any distributions for t_c and t_f, at least one processor or C-element is active at any given time.*

Proof: By the definition of s, $1 < s \leq n$ is equivalent to $1 \leq b < n$, which means that there is at least one loop that is a bubble and at least one loop that is not a bubble. Since the loops are arranged in a ring, we can find a i such that loop_i is in state *down* and loop_{i-1} is in state *left* or *right*. If loop_{i-1} is in state *left*, then processor f_{i-1} is active. Otherwise, loop_{i-1} is in state *right* and C-element C_{i-1} is active. In both

cases the claim is established.

□

The next theorem shows that the throughput of a self–timed pipeline cannot be increased by slowing down any one or more operations. In general, the times for computation and storage are random variables. A **realization** is a tuple (q, delay) where "q" specifies the initial state of the pipeline and "delay" gives the delay for each operation. In particular, $q(i)$ is the initial state (*down*, *left*, or *right*) of loop i. Delays are specified by a real valued function, for example, delay(j, i, w), where j is the iteration, i is the position in the ring, and w ("what") is either "processor" or "C-element." For example, delay(42, 17, "C-element") = 1.23 means that C_{17} takes 1.23 time units (after its inputs are both available) to perform its 42^{nd} storage operation. Likewise, we define start(j, i, w) to be the time at which the inputs become available for w_i to start its j^{th} operation. We write $\text{delay}_1 \leq \text{delay}_2$ to denote that for all j, i, and w, $\text{delay}_1(j, i, w) \leq \text{delay}_2(j, i, w)$. We define $\text{start}_1 \leq \text{start}_2$ in the same manner.

Theorem 2 *Let* $R_1 = (q_1, \text{delay}_1)$ *and* $R_2 = (q_2, \text{delay}_2)$ *be two realizations of a pipeline. If* $q_1 = q_2$ *and* $\text{delay}_1 \leq \text{delay}_2$, *then* $\text{start}_1 \leq \text{start}_2$.

Proof: Assume otherwise. Then, of all operations that start earlier in R_2 than in R_1, let (j, i, w) be the first.

If w = "processor", this means that the output of the preceding C-element changed sooner in R_2 than in R_1. By the choice of (j, i, w) to be the first violation of the claim, this C-element started its operation no earlier in R_2 than in R_1. Furthermore, the C-element in R_2 took at least as long to complete its operation as the one in R_2 by the hypothesis $\text{delay}_1 \leq \text{delay}_2$. Thus, the input to f_i became available no earlier in R_2 than in R_1. A contradiction.

Otherwise, w = "C-element", and a similar argument leads to the required contradiction.

□

The following corollary shows that the behavior of a self–timed pipeline (as described by starting times of operations) is completely determined by the initial state and the delay function.

Corollary 2.1 *Let* $R_1 = (q_1, \text{delay}_1)$ *and* $R_2 = (q_2, \text{delay}_2)$ *be two realizations of a pipeline. If* $q_1 = q_2$ *and* $\text{delay}_1 = \text{delay}_2$, *then* $\text{start}_1 = \text{start}_2$.

Proof: Since $\text{delay}_1 \leq \text{delay}_2$, Theorem 2 implies $\text{start}_1 \leq \text{start}_2$. Likewise, $\text{delay}_2 \leq \text{delay}_1$ implies $\text{start}_2 \leq \text{start}_1$. Therefore, $\text{start}_1 = \text{start}_2$ as claimed.
□

We now give a condition that guarantees that utilization depends only on the distributions of processing and storage times and the number of bubbles in the pipeline.

Definition: The Restart Condition
A delay function, d, satisfies the restart condition if

1. *For all* j_1, j_2, i_1, i_2, w_1, *and* w_2, *if* $j_1 \neq j_2$, $i_1 \neq i_2$, *or* $w_1 \neq w_2$, *then* $d(j_1, i_1, w_1)$ *is independent of* $d(j_2, i_2, w_2)$.

2. *For all* j_1, j_2, i, *and* w, $d(j_1, i, w)$ *and* $d(j_2, i, w)$ *are identically distributed.*

3. *For all* j, i, *and* w, *the expectation* $E[d(j_1, i, w)]$ *exists and is finite.*

We now introduce the idea of "restarting" that will allow us to analyze aggregate properties of (almost) all realizations. For any integer k, after each processor and C-element has completed exactly k operations, the state of the pipeline is identical to the initial state. The **restart–at–k** of a realization $R = (q, \text{delay})$ is a realization $R' = (q, \text{delay}')$ such that $\text{delay}' \leq \text{delay}$ and all processors and C-elements complete their *ksupth* operation at the same time. In particular, we consider the first processor or C-element to complete its k^{th} operation; let this occur at time τ_k. We reduce delays (in "delay′" relative to "delay") for operations in progress at this instant so that they will also complete at time τ_k. Finally, we set the delays of all remaining operations of iterations at or before k to zero to guarantee that each processor and C-element completes its k^{th} operation at time τ_k. By construction $\text{delay}' \leq \text{delay}$ and by theorem 2, $\text{start}' \leq \text{start}$. Since the pipeline is restarted to a fixed state (independent

of the state before the restart operation), and the delays of computation and storage operations are independent random variables (assuming the restart condition), the behaviors of the pipeline before and after the restart are independent. The **restart–every–k** of a realization $R = (q, \text{delay})$ is a realization $R' = (q, \text{delay}')$ such that $\text{delay}' \leq \text{delay}$ and all processors and C-elements complete their mk^{th} operation at the same time for all integer $m \geq 1$.

Theorem 3 *If a pipeline satisfies the restart condition, then the limit*

$$T \;=\; \lim_{m\to\infty} \frac{\text{start}(m,\, i,\, w)}{m}$$

exists almost surely and has the same value for all i and w, and all initial states with the same number of bubbles.

Proof: In the following, we assume $1 \leq b < n$ to guarantee that the pipeline is deadlock free (see theorem 1). Otherwise, the pipeline deadlocks, and the limit is infinity for all realizations. We first show that the limit exists for any fixed initial state and for any integer k, when each realization is replaced by its restart–every–k version. We then show that in the limit as k goes to infinity, the limit for the restart–every–k version is the limit for the original pipeline. Finally, we show that this limit is independent of the initial state, i, and w.

Let start_k be the starting times of a restart–every–k version of a realization. We consider

$$T_k \;=\; \lim_{m\to\infty} \frac{\text{start}_k(m,\, i,\, w)}{m}\,.$$

Let $\delta_k(1) = \text{start}_k(1,\, i,\, w)$ and $\delta_k(m) = \text{start}_k(m,\, i,\, w) - \text{start}_k(m-1,\, i,\, w)$. Then we can write the above limit as

$$T_k \;=\; \lim_{m\to\infty} \frac{1}{m} \sum_{j=1}^{m} \delta_k(j) \;=\; \lim_{m\to\infty} \frac{1}{m} \sum_{j=0}^{m-1} \frac{1}{k} \sum_{h=1}^{k} \delta_k(ik+h)\,.$$

Because the operation of the pipeline before any restart is independent of the operation after the restart, the sums over h are independent random variables. Thus, by the strong law of large numbers [Fel50b], the limit exists almost surely. Since $\text{start}_k \leq \text{start}$, we have

$$T_k \;=\; \liminf_{m\to\infty} \frac{\text{start}(m,\, i,\, w)}{m}\,.$$

We now derive an upper bound, and show that the two bounds are equal in the limit as k goes to infinity. In a pipeline with b bubbles, the difference at any time in the number of operations performed by any two processors or C-elements is at most $b+1$. Therefore, the number of processor delays that are decreased by each restart is at most $n(b+1)$. Likewise, at most $n(b+1)$ C-element delays are decreased in the construction of delay$'$. The largest difference in starting times between the original realization and its restarted counterpart would occur if these operations were performed sequentially in the original. Let

$$\text{gap}_{j,k} = \sum_{h=(j+1)k-b}^{(j+1)k} \sum_{i=0}^{n-1} (\text{delay}(h,\, i,\, \text{“processor”}) + \text{delay}(h,\, i,\, \text{“C-element”}))\,.$$

For each j, $\text{gap}_{j,k}$ is an upper bound on the decrease of starting times due to the j^{th} restart. This yields

$$\limsup_{m\to\infty} \frac{\text{start}(m,\, i,\, w)}{m} \leq \lim_{m\to\infty} \frac{1}{m} \sum_{j=0}^{m-1} \frac{1}{k} \left(\text{gap}_{j,k} + \sum_{h=1}^{k} \delta_k(jk+h) \right) .$$

Applying the strong law of large numbers again, the right side of the inequality becomes

$$\lim_{m\to\infty} \frac{1}{m} \sum_{j=0}^{m-1} \frac{1}{k} (E[\text{gap}] + T_k)\,.$$

Taking limits as k goes to infinity yields

$$\begin{aligned} \limsup_{k\to\infty} T_k &\leq \liminf_{m\to\infty} \frac{\text{start}(m,i,w)}{m} \\ &\leq \limsup_{m\to\infty} \frac{\text{start}(m,i,w)}{m} \\ &\leq \liminf_{k\to\infty} \left(\frac{E[\text{gap}]}{k} + T_k \right) \end{aligned}$$

almost surely. $E[\text{gap}]$ is bounded by $n(b+1)(E[t_f + t_c])$ which must be finite by the third requirement of the restart condition. Therefore, $\lim_{k\to\infty} E[gap]/k = 0$, and we conclude

$$\liminf_{m\to\infty} \frac{\text{start}(m,i,w)}{m} = \limsup_{m\to\infty} \frac{\text{start}(m,i,w)}{m}$$

almost surely. Therefore for any fixed initial state, the limit exists almost surely as claimed.

The final step is to show that this limit is the same for all initial states with the same number of bubbles, and for all i and w. Let q_1 and q_2 be two states with the same number of bubbles. Let T_1 and T_2 be their respective values of above limit. We note that a pipeline initially in state q_1 can be brought into state q_2 by setting the delays of a bounded, $O(nb)$, number of operations to zero. Thus, $T_2 \leq T_1$. By symmetry, we have $T_1 \leq T_2$ and therefore $T_2 = T_1$. As noted above, the difference in the number of operations performed by any two processors or C-elements is at most $b+1$. Because each loop completes any finite number of operations in a finite amount of time, the difference between starting times of different loops and processors is insignificant in the limit. Thus, the limit exists and is the same for all initial states, and for all i and w.
□

Corollary 3.1 *If a pipeline satisfies the restart condition, then its utilization is defined.*

Proof: The average waiting time is

$$T \;=\; \lim_{m\to\infty} \frac{\mathrm{start}(m,i,w)}{m}\,.$$

By theorem 3, this limit exists almost surely. By the third condition of the restart condition, $E[t_f]$ is well defined. Therefore, the utilization $U = E[t_f]/T$ is defined.
□

Corollary 3.2 *Let P_1 and P_2 be two pipelines that satisfy the restart condition and have the same number of processors and bubbles. Let F_1 and C_1 be the distribution functions for processing and storage times respectively in P_1, and F_2 and C_2 be the same for P_2. If $F_1 \geq F_2$ and $C_1 \geq C_2$, then $T_1 \leq T_2$.*

Proof: By theorem 3, T_1 and T_2 are defined. By theorem 2 and the monotonicity of averages, it is sufficient to exhibit a mapping from realizations of P_1 to realizations of P_2 with the following two properties: (1) the realization in P_2 has a delay function that is greater than or equal to that of the realization in P_1, and (2) the mapping

preserves the probabilities of measurable sets. If F_1, C_1, F_2, and C_2 are continuous this can be achieved for each realization by mapping $\text{delay}_1(j, i, \text{“processor”})$ to $F_2^{-1}(F_1(\text{delay}_1(j, i, \text{“processor”})))$ and likewise for C-element delays. Since $F_1 \geq F_2$ and $C_1 \geq C_2$,

$$\begin{aligned} F_2^{-1}(F_1(\text{delay}_1(j, i, \text{“processor”}))) &\geq \text{delay}_1(j, i, \text{“processor”}) \\ \text{and} \quad C_2^{-1}(C_1(\text{delay}_1(j, i, \text{“C-element”}))) &\geq \text{delay}_1(j, i, \text{“C-element”}) \end{aligned}$$

as required. If one or more of the distributions are not continuous, suitable variations of this simple mapping produces the required result.

□

Appendix B

Timing Verification of STARI

Formal methods can be used for the specification, verification, simulation, and synthesis of digital hardware. A formal method consists of a notation for writing expressions, and rules for manipulating these expressions. The application of formal methods to hardware verification has been an active area of research for several years [Bar84, Hun85, Gor86, Coh88, Dil88, Coh89, Dil89, Mil91, SFC91, Gra92]. Most of this work has focused on establishing the functional correctness of designs.

In this appendix, the `Synchronized Transitions` [RS87] notation is used to model a STARI interface and verify its timing properties. In `Synchronized Transitions`, digital circuits (such as STARI) are modeled as concurrent programs. The correctness condition is formulated as an invariant. Proof rules are presented for verifying invariants, and these are used to show that STARI maintains its timing specification.

Section B.1 describes the `Synchronized Transitions` language and gives proof rules for verifying invariants. Section B.2 presents a `Synchronized Transitions` program for a self–timed FIFO, states an invariant the expresses the schedule of operations, and proves the invariant.

B.1 Synchronized Transitions

B.1.1 Notation

A `Synchronized Transitions` program is a collection of state variables and a **transition valued expression**. This expression defines the set of allowed state transitions. An execution is a possibly infinite sequence of states.

The simplest expression is a **transition**. A transition is composed of a **precondition** and an **action**. When the precondition is satisfied, the action may be performed. Syntactically, transitions are written with the form:

$$\ll \textit{precondition} \rightarrow \textit{action} \gg$$

The precondition is a Boolean valued expression and the action is a multi–assignment. To avoid conflicting assignments, the variables appearing on the left side of the multi–assignment must be disjoint. For example

$$\ll \texttt{a < b} \rightarrow \texttt{a, b := b, a} \gg$$

is a transition that exchanges `a` and `b` if they are not in descending order.
Transition valued expressions are defined by two productions:

1. A transition is a transition valued expression.

2. $e_1 \| e_2$ is a transition valued expression, where e_1 and e_2 are transition valued expressions,

The **asynchronous combinator**, $\|$, specifies that two expressions are executed independently; there is no global thread of control determining the order of execution. If either precondition is satisfied, the corresponding action may be performed. If both preconditions are satisfied, one of the transitions is selected *non–deterministically.* This combinator is an associative, commutative operator. Furthermore, transitions combined with $\|$ are executed atomically; the sequence of states of any execution corresponds to some sequential execution of transitions[1]. For example

[1] In a hardware implementation, the levels (e.g. voltages) of many signals throughout the circuit may change simultaneously. A formal definition of mappings from signal levels to program states is outside the scope of this presentation (see [GS88a, SG88] for examples).

$$\ll \texttt{a < b} \rightarrow \texttt{a, b := b, a} \gg \ \| \ \ll \texttt{b < c} \rightarrow \texttt{b, c := c, b} \gg$$

is an expression that when repeatedly executed sorts `a`, `b`, and `c` into descending order.

B.1.2 Proof rules

A `Synchronized Transitions` program is a collection of state variables and a transition valued expression. Let V be a set of state variables and e be a transition valued expression such that all variables appearing in e are members of V. Then, (e, V) denotes a program; let $P = (e, V)$. A state of P is an assignment of values to the elements of V. Accordingly, a state variable is a function from states to values of the underlying type of the variable. If `x` is a state variable and s is a state, we will write $\texttt{x}(s)$ to denote the value of `x` in state s. If E is an expression of state variables, then $E(s)$ has the obvious meaning.

The verification of STARI presented in this appendix is based on invariants [LS84]. A property, I, is an invariant of a program, P, if I holding in one state guarantees that I will hold in all future states. The following two proof rules are used in the verification of STARI in section B.2:

Property 1 *Let* (e_1, V) *and* (e_2, V) *be programs. A predicate* I *is an invariant of* $(e_1 \| e_2, V)$ *if and only if* I *is an invariant of both* (e_1, V) *and* (e_2, V).

Given a program P and a predicate I on state of P, to show that I is an invariant of P, property 1 shows that each transition of P can be considered separately. The next property shows how to establish that I is an invariant of a single transition.

Property 2
Let $P = (\ll C \rightarrow l := r \gg, V)$. *Let* I *be a predicate.* I *is an invariant of* P *if and only if for every pair of states,* s_1 *and* s_2, *of* P:

$$\begin{array}{ll} & I(s_1) \wedge C(s_1) \wedge (l(s_2) = r(s_1)) \wedge (\forall v \in V - L : v(s_2) = v(s_1)) \\ \Rightarrow & I(s_2) \end{array}$$

where L *is the set of variables appearing in* l.

B.1.3 Miscellany

When writing a `Synchronized Transitions` program, the set of state variables is defined by declaring them in a style adopted from Pascal [JW74], or Modula-2 [Wir85]. The declarations are followed by a `BEGIN` — `END` pair. The transition valued expression for the program is written between the `BEGIN` and `END` keywords.

To describe regular structures, such as the FIFO of a STARI interface, expressions may be quantified. Quantified expressions correspond to reduction operators in functional languages (e.g. the `\` operator of APL [Ive62] and Crystal [Che86], or the `mapcar` function in Lisp [WH81]). A quantified expression applies an operator to a set of expressions to produce a single value. Quantified expressions are written as shown below:

$$\{op\ declarations\ \texttt{|}\ expression\text{–}list\}$$

Each variable declared in *declarations* must be of a bounded discrete type (e.g. an integer subrange). The declarations section may be empty. The value of the quantified expression is obtained by evaluating each expression in *expression–list* for each value of each variable in *declarations* and combining the results with *op*. If the set is empty, the value is the identity element for the operator. For example, the value of

```
{AND i: 1..n | x(i)}
```

is true if `x(1)` through `x(n)` are true. The expression list of a quantified expression may contain several expressions, separated by commas. For example, `{+ | 1, 2, 3}` is a quantified expression with the value of 6.

The operator `ANY` is only used to reduce quantified expressions. Applying the `ANY` operator to a non–empty set produces the value of some element of the set; the choice of which element is not specified. Applying the `ANY` operator to an empty set is an error. For example, `{ANY i: 1..n | i}` is an expression whose value can be any integer in the range of 1 to `n` inclusive. In typical usage, the `ANY` operator is only used in specifications. For example, in section B.2, `ANY` is used to express a model of time.

B.2 Verification of STARI

The novelty of STARI is that it provides a communication path between two circuits operating at the same clock frequency; it allows the transmitter's and receiver's clock to be skewed; and it is free from synchronization failure. This section describes the verification of these properties.

To verify STARI, a communication link including the transmitter, the FIFO, and the receiver must be described. This description includes both the logical behavior of these components and their timing behavior. To describe real–time properties of the implementation, time is introduced using auxiliary variables [OG76]. For the purposes of verification, auxiliary variables can be regarded as state variables. The current time is represented by the real valued variable τ. Furthermore, the value at each stage in the FIFO is represented by a record: one field of this record, `.v` is the logical value output by the stage; the other, $.\tau$, represents the time at which the `.v` field was assigned its current value. In general, there are two kinds of timing properties:

Timing lower bounds: a transition is not performed until after a specified time.

Timing upper bounds: a transition is guaranteed to be performed by a specified time.

Timing lower bounds can be expressed by strengthening the transition's precondition. Upper bounds are expressed by the transition that updates the current time, τ.

This section presents a program for STARI. Then, an invariant is given that expresses the schedule for operations that was described in section 3.1.2. Finally, this invariant is verified.

B.2.1 The program

Figure 34 shows a program for a STARI interface. As described above, signals in the STARI interface are described with a record type that has two fields: `.v` represents the value of the signal, and $.\tau$ represents the time at which the signal acquired its current value. In the program, the value of the signal is represented with a Boolean valued variable. This is an abstraction of self–timed encoding. The essential feature

```
TYPE Signal = RECORD
                v: BOOLEAN; (* current value is "valid" *)
                τ: ℛ        (* time at which .v was assigned *)
              END;

STATIC (* constants used to describe the FIFO *)
  n: INTEGER = ...; (* how many stages in the FIFO, n ≥ 0 *)
  δ: ℛ = ...;       (* maximum FIFO stage delay: δ > 0 *)
  π: ℛ = ...;       (* global clock period: π > 2δ *)

STATE (* state variables *)
  y: ARRAY 0..n+1 OF Signal; (* FIFO inputs and value for each stage *)
  τ: ℛ; (* the current time *)

BEGIN (* the transitions that describe STARI *)
  (* the transmitter *)
    ≪ τ≥ y(0).τ + π → y(0).v, y(0).τ := NOT y(0).v, τ ≫

  (* the FIFO *)
  ‖ {‖ i: 1..n |
      ≪    (y(i−1).v ≠ y(i).v) AND (y(i).v = y(i+1).v)
         → y(i).v, y(i).τ:= y(i−1).v, τ
      ≫
    }

  (* the receiver *)
  ‖ ≪ τ≥ y(n+1).τ+ π → y(n+1).v, y(n+1).τ:= NOT y(n+1).v, τ ≫

  (* modeling time *)
  ‖ ≪ TRUE → τ:=
      {ANY ν: ℛ
              (ν > τ)
          AND (ν ≤ y(0).τ + π)
          AND {AND i: 1..n |
                        ((y(i−1).v≠y(i).v) AND (y(i).v=y(i+1).v))
                IMPLIES (ν < max(y(i−1).v, y(i+1).v) + δ)
              }
          AND (ν ≤ y(n+1).τ + π)
        ? ν
      }
    ≫
END;
```

Figure 34: `Synchronized Transitions` program for STARI

of a self–timed encoding is that it allows successive values in the sequence sent by the transmitter to be distinguished. A Boolean variable whose value alternates between true and false is a simple abstraction of this behavior. By analogy with the dual–rail code described in chapter 4.1.1, we will say that a false value of the `.v` field represents an empty value of the dual–rail code, and that a true value represents a non–empty value. The assignment time field, $.\tau$ is real valued (reals are denoted by $\mathcal{R}$ in the program). This and the other real valued quantities that appear in this program are auxiliary variables; they are included to facilitate reasoning about the program, but they are not implemented by the circuit. Thus, issues of how to represent real numbers in a digital circuit are not relevant.

Following the definition of the `Signal` type are the declarations of static variables (constants) used in the program. The integer valued constant `n` specifies the number of stages of the FIFO. The real constant δ is the maximum delay between when a stage receives new data and an acknowledge and when the stage transfers the new data value to its output. The real constant π is the period of the global clock.

Next, state variables are declared. The array `y` represents the output of the transmitter and input of the FIFO, `y(0)`; the outputs of the FIFO stages, `y(1)` through `y(n)`; and the acknowledge signal from the receiver, `y(n+1)`. The auxiliary variable τ represents the current time.
The transmitter is modeled by the transition

$$\ll \tau \geq \texttt{y(0)}.\tau + \pi \rightarrow \texttt{y(0).v, y(0)}.\tau \texttt{ := NOT y(0).v, } \tau \gg$$

The multi–assignment part of this transition states that the value output by the transmitter alternates between empty and non–empty values (`y(0).v := NOT y(0).v`), and records the time of the last assignment (`y(0)`$.\tau$ `:=` τ). The precondition $\tau \geq$ `y(0)`$.\tau + \pi$ enforces a delay of at least π time units between the output of successive values by the transmitter. In the expression that gives values with which to update the current time τ, the clause $\nu \leq$ `y(0)`$.\tau + \pi$ states the current time cannot exceed `y(0)`$.\tau$ by more than π time units[2]. Therefore, exactly π time units elapse between

[2] An operational interpretation of the program could suggest that time "stops" when τ attains a value of `y(0)`$.\tau + \pi$ and "waits" for the transmitter to output a new value. This issue is addressed in section B.3

the output of successive values by the transmitter. Note that the condition for the transmitter to output a new value depends only on the amount of time that has elapsed since the current value was output. The program does not explicitly ensure that the FIFO has input the current value before the next one is output. Instead, an invariant that implies this is stated in section B.2.2 and proven in section B.2.3.

Stage `i` of the FIFO is modeled by the transition

$$\begin{array}{ll} \ll & \texttt{(y(i-1).v} \neq \texttt{y(i).v) AND (y(i).v = y(i+1).v)} \\ & \rightarrow \texttt{y(i).v, y(i).}\tau\texttt{:= y(i-1).v, } \tau \\ \gg & \end{array}$$

This transition states that stage `i` may transfer a new value to its output, `y(i)` when its input has a new value (`y(i-1).v` $\neq$ `y(i).v`), and its output has its current value (`y(i).v` = `y(i+1).v`). There is no lower bound given for the delay between when this condition is satisfied and when the transition may be performed. On the other hand, the clause

$$\begin{array}{ll} & \texttt{((y(i-1).v} \neq \texttt{y(i).v) AND (y(i).v=y(i+1).v))} \\ \texttt{IMPLIES} & (\nu < \max(\texttt{y(i-1).v, y(i+1).v}) + \delta) \end{array}$$

in the expression that gives values with which to update the current time τ, states that at most δ time units elapse between enabling this transition and its performance.

The transition that describes the operation of the reciever is analogous to the one that describes the transmittter. The transition that models time states that time advances ($\nu > \tau$), and that the timing upper bounds of the transmitter, FIFO stages, and receiver are satisfied.

B.2.2 The invariant

Let I be defined as shown in figure 35. The definitions of k, Δ and Ψ are as in equations 8, 9, and 11 respectively. The "knee" in the piecewise linear timing bound, Δ is at k, and by construction, $-1 \leq k \leq n$. Furthermore, the definition of Δ ensures that for any integer i, $\delta \leq \Delta(i) \leq \pi - \delta$. By definition, $\Psi(\texttt{n+1}) = \lambda$.

Each clause of this invariant has a simple interpretation. The first clause,

$$\forall \texttt{i} \in \{\texttt{0} \ldots \texttt{n} + 1\} : \texttt{y(i).}\tau \leq \tau$$

$$
\begin{aligned}
\mathtt{n} &= \text{number of FIFO stages, } \mathtt{n} \geq 0 \\
\delta &= \text{maximum stage delay, } \delta > 0 \\
\pi &= \text{global clock period, } \pi > 2\delta \\
\lambda &= \text{the FIFO latency, } (n+1)\delta < \lambda < (n+1)(\pi - \delta) \\
\mathtt{k} &= \mathtt{n} - \left\lfloor \frac{\lambda - (\mathtt{n}+1)\delta}{\pi - 2\delta} \right\rfloor \\
\Delta(\mathtt{i}) &= \begin{cases} \delta & \text{, if } \mathtt{i} < \mathtt{k} \\ \lambda - k\delta - (n-k)(\pi - \delta) & \text{, if } \mathtt{i} = \mathtt{k} \\ \pi - \delta & \text{, if } \mathtt{i} > \mathtt{k} \end{cases} \\
\Psi(\mathtt{j}) &= \sum_{\mathtt{j}=\mathtt{i}}^{\mathtt{i}} \Delta(\mathtt{j}) \\
\text{insert}(\mathtt{i}) &= \begin{cases} \mathtt{y(i)}.\tau & \text{, if } \mathtt{i} = \mathtt{0} \\ \text{insert}(\mathtt{i}) & \text{, if } \mathtt{i} > \mathtt{0} \text{ and } \mathtt{y(i)}.\mathtt{v} = \mathtt{y(i\text{-}1)}.\mathtt{v} \\ \text{insert}(\mathtt{i}) - \pi & \text{, if } \mathtt{i} > \mathtt{0} \text{ and } \mathtt{y(i)}.\mathtt{v} \neq \mathtt{y(i\text{-}1)}.\mathtt{v} \end{cases} \\
I &= \quad \forall \mathtt{i} \in \{\mathtt{0} \ldots \mathtt{n}+1\} : \mathtt{y(i)}.\tau \leq \tau \\
&\quad \wedge \ \tau \leq \mathtt{y(0)}.\tau + \pi \\
&\quad \wedge \ \forall \mathtt{i} \in \{\mathtt{1} \ldots \mathtt{n}\} : \\
&\qquad ((\mathtt{y(i{-}1)}.\mathtt{v} \neq \mathtt{y(i)}.\mathtt{v}) \wedge (\mathtt{y(i)}.\mathtt{v} = \mathtt{y(i{+}1)}.\mathtt{v})) \\
&\qquad \Rightarrow \ (\tau < \max(\mathtt{y(i{-}1)}.\tau, \mathtt{y(i{+}1)}.\tau) + \delta) \\
&\quad \wedge \ \tau \leq \mathtt{y(n{+}1)}.\tau + \pi \\
&\quad \wedge \ \forall \mathtt{i} \in \{\mathtt{1} \ldots \mathtt{n}\} : \mathtt{y(i)}.\tau < \text{insert}(\mathtt{i}) + \Psi(\mathtt{i}) \\
&\quad \wedge \ \mathtt{y(n{+}1)}.\tau = \text{insert}(\mathtt{n{+}1}) + \lambda
\end{aligned}
$$

Figure 35: An invariant for the program of figure 34

states that no signals acquired their current values in the future. The next three clauses

$$\begin{array}{lll} & \tau \leq \mathtt{y(0)}.\tau + \pi & \\ \wedge & \forall \mathtt{i} \in \{\mathtt{1}\ldots\mathtt{n}\} : & \\ & \quad ((\mathtt{y(i-1)}.\mathtt{v} \neq \mathtt{y(i)}.\mathtt{v}) \wedge (\mathtt{y(i)}.\mathtt{v} = \mathtt{y(i+1)}.\mathtt{v})) & \\ & \quad \Rightarrow \quad (\tau < \max(\mathtt{y(i-1)}.\tau, \mathtt{y(i+1)}.\tau) + \delta) & \\ \wedge & \tau \leq \mathtt{y(n+1)}.\tau + \pi & \end{array}$$

state that the timing upper bounds for the transmitter, FIFO stages, and receiver are satisfied. The fifth clause

$$\forall \mathtt{i} \in \{\mathtt{1}\ldots\mathtt{n}\} : \mathtt{y(i)}.\tau < \mathrm{insert}(\mathtt{i}) + \Psi(\mathtt{i})$$

is the schedule for the operations of the FIFO stages. In particular, it states that each value reaches stage `i` at most $\Psi(\mathtt{i})$ time units after it was output by the transmitter. The last clause of I

$$\mathtt{y(n+1)}.\tau = \mathrm{insert}(\mathtt{n+1}) + \lambda$$

states that the receiver outputs acknowledges a data token exactly λ time units after it was output by the transmitter.

B.2.3 The proof

This section presents a proof that I, (from figure 35) is an invariant of P, the program shown in figure 34. If you already believe it, skip to page 142. The arguments are given in a simple "statement–reason" form as illustrated below:

$$\begin{array}{ll} \langle \mathit{line}\ \# \rangle & \mathit{statement} \\ & \mathit{reason} \end{array}$$

The invariant I is formulated in a way that separates issues of timing from issues of logical function. In particular, the current time, τ, does not appear in clauses that refer explicitly to the values (the `.v` field) of signals. This separation of concerns simplifies the formulation of the invariant and its verification. However, a global schedule of operations is helpful. The following two lemmas define a such a schedule.

The first lemma shows that the invariant guarantees that the each data token reaches stage i within $\Psi(\mathtt{i})$ time units of being output by the transmitter.

Lemma 1

$$\forall \mathtt{i} \in \{\mathtt{0}\ldots\mathtt{n}\}:$$
$$(I \wedge (\tau > \text{insert}(\mathtt{i}) + \Psi(\mathtt{i+1}))) \Rightarrow (\mathtt{y(i+1)\,.\,v} = \mathtt{y(i)\,.\,v})$$

Proof: By induction on i.

Base case: $\mathtt{i} = \mathtt{n}$

Note that we are considering $0 \leq i < n$ and $0 < k \leq n$. The argument is by contradiction.

$\langle 1.1 \rangle$ $I \wedge (\text{insert}(\mathtt{n}) + \Psi(\mathtt{n+1})) \wedge (\mathtt{y(n+1)\,.\,v} \neq \mathtt{y(n)\,.\,v})$
assumption of negated conjecture

$\langle 1.2 \rangle$ $\text{insert}(\mathtt{n}) = \text{insert}(\mathtt{n+1}) + \pi$
$\langle 1.1 \rangle$ and the definition of "insert"

$\langle 1.3 \rangle$ $\tau > \text{insert}(\mathtt{n+1}) + \Psi(\mathtt{n+1}) + \pi$
$\langle 1.1 \rangle$ and $\langle 1.2 \rangle$

$\langle 1.4 \rangle$ $\tau > \mathtt{y(n+1)}.\tau + \pi$
$\langle 1.1 \rangle$, $\langle 1.3 \rangle$, $\lambda = \Psi(n+1)$, and
$I \Rightarrow (\mathtt{y(n+1)\,.\,}\tau = \text{insert}(\mathtt{n+1}) + \lambda)$

$\langle 1.5 \rangle$ $\tau \leq \mathtt{y(n+1)}.\tau + \pi$
$\langle 1.1 \rangle$ and $I \Rightarrow (\tau \leq \mathtt{y(n+1)\,.\,}\tau + \pi$

) Statements $\langle 1.4 \rangle$ and $\langle 1.5 \rangle$ provide the desired contradiction.

Induction step: assume for $\mathtt{i} = \mathtt{m}$, prove for $\mathtt{i} = \mathtt{m} - \mathtt{1}$.

The argument is by contradiction.

$\langle 2.1 \rangle$ $I \wedge (\tau > \text{insert}(\mathtt{i}) + \Psi(\mathtt{i+1})) \wedge (\mathtt{y(i+1)\,.\,v} \neq \mathtt{y(i)\,.\,v})$
assumption of negated conjecture

$\langle 2.2 \rangle$ $\text{insert}(\mathtt{i+1}) = \text{insert}(\mathtt{i}) - \pi$
$\langle 2.1 \rangle$ and the definition of "insert"

$\langle 2.3 \rangle$ $\mathtt{y(i)\,.\,}\tau < \tau - \Delta(\mathtt{i+1})$
$\langle 2.1 \rangle$, $\langle 2.2 \rangle$, the definition of Ψ, and
$I \Rightarrow (\mathtt{y(i)\,.\,}\tau \leq \text{insert}(\mathtt{i}) + \Psi(\mathtt{i}))$

$\langle 2.4 \rangle$ $\mathtt{y(i)}.\tau < \tau - \delta$

$\langle 2.3 \rangle$ and $\Delta(\mathtt{i}) \geq \delta$

$\langle 2.5 \rangle$ $\tau > \text{insert}(\mathtt{i+1}) + \Psi(\mathtt{i+2})$

$\langle 2.1 \rangle$, $\langle 2.2 \rangle$, $\Delta(\mathtt{i+2}) < \pi$, and the definition of "insert"

$\langle 2.6 \rangle$ $\mathtt{y(i+2).v} = \mathtt{y(i+1).v}$

$\langle 2.5 \rangle$ and the induction hypothesis

$\langle 2.7 \rangle$ $\text{insert}(\mathtt{i+2}) = \text{insert}(\mathtt{i}) - \pi$

$\langle 2.2 \rangle$, $\langle 2.6 \rangle$, and the definition of "insert"

$\langle 2.8 \rangle$ $\mathtt{y(i+2)}.\tau < \text{insert}(\mathtt{i}) + \Psi(\mathtt{i+1}) + \Delta(\mathtt{i+2}) - \pi$

$\langle 2.7 \rangle$, the definition of Ψ, and

$I \Rightarrow (\mathtt{y(i+2)}.\tau \leq \text{insert}(\mathtt{i+2}) + \Psi(\mathtt{i+2}))$

$\langle 2.9 \rangle$ $\mathtt{y(i+2)}.\tau < \tau - \delta$

$\langle 2.1 \rangle$, $\langle 2.8 \rangle$ and $\Delta(\mathtt{i+2}) \leq \pi - \delta$

$\langle 2.10 \rangle$ $\tau < \tau$

$\langle 2.1 \rangle$, $\langle 2.4 \rangle$, $\langle 2.6 \rangle$, $\langle 2.9 \rangle$, and

$((\mathtt{y(i).v} \neq \mathtt{y(i+1).v}) \wedge (\mathtt{y(i+1).v} = \mathtt{y(i+2).v}))$

$\Rightarrow \quad (\tau < \max(\mathtt{y(i)}.\tau, \mathtt{y(i+2)}.\tau) + \delta)$

Statement $\langle 2.10 \rangle$ provides the desired contradiction.

□

The next lemma defines a global schedule for bubbles. This schedule is analogous to the schedule for data tokens defined by lemma 1.

Lemma 2

$$\forall \mathtt{i} \in \{\mathtt{1} \ldots \mathtt{n+1}\} :$$
$$(I \wedge (\tau > \text{insert}(\mathtt{i}) + \Psi(\mathtt{i-1})) + \pi) \Rightarrow (\mathtt{y(i-1).v} \neq \mathtt{y(i).v})$$

Proof: The proof is by induction on i and contradiction. The base case is $\mathtt{i} = \mathtt{1}$, and the induction step assumes the claim for $\mathtt{i} = \mathtt{k}$ with $\mathtt{k} < \mathtt{n}$ and establishes the claim for $\mathtt{i} = \mathtt{k+1}$. The arguments are analogous to those given for lemma 1 and are omitted.

□

The proof that I is an invariant of P considers each transition of P separately according to property 1 in section B.1.2 (page 125).

The transmitter

The transition is

$$\ll \tau \geq \texttt{y(0).}\tau + \pi \rightarrow \texttt{y(0).v, y(0).}\tau \texttt{ := NOT y(0).v, } \tau \gg$$

Let s_1 and s_2 be any two states of P and let

$$\begin{array}{lcll}
I_1 & = & I(s_1) & \text{, the invariant} \\
C & = & (\tau \geq \texttt{y(0).}\tau)(s_1) & \text{, the precondition} \\
A & = & \quad (\texttt{y(0).v}(s_2) = \neg\texttt{y(0).v}(s_1)) & \text{, the multi–assignment} \\
 & & \wedge \quad (\texttt{y(0).}\tau(s_2) = \tau(s_1)) & \\
U & = & \quad \forall \texttt{i} \in \{\texttt{1...n+1}\} : \texttt{y(i)}(s_2) = \texttt{y(i)}(s_1) & \text{, unchanged variables} \\
 & & \wedge \quad \tau(s_2) = \tau(s_1) &
\end{array}$$

By property 2 in section B.1.2, the proof obligation for this transition is

$$I_1 \wedge C \wedge A \wedge U \quad \Rightarrow \quad I(s_2)\,.$$

We assume the left side of the implication and show that this establishes $I(s_2)$.

$\langle 1 \rangle$ $(\texttt{y(0).}\tau = \tau)(s_2)$
A and U

$\langle 2 \rangle$ $\forall \texttt{i} \in \{\texttt{1...n+1}\} : \texttt{y(i).}\tau(s_2) = \texttt{y(i).}\tau(s_1)$
U

$\langle 3 \rangle$ $\forall \texttt{i} \in \{\texttt{1...n+1}\} : \texttt{y(i).}\tau(s_2) \leq \tau(s_1)$
$\langle 2 \rangle$, I_1, and U

$\langle 4 \rangle$ $\forall \texttt{i} \in \{\texttt{1...n+1}\} : (\texttt{y(i).}\tau \leq \tau)(s_2)$
$\langle 3 \rangle$ and U

$\langle 5 \rangle$ $\forall \texttt{i} \in \{\texttt{0...n+1}\} : (\texttt{y(i).}\tau \leq \tau)(s_2)$
$\langle 1 \rangle$ and 4

$\langle 6 \rangle$ $(\tau \leq \texttt{y(0).}\tau + \pi)(s_2)$
$\langle 1 \rangle$ and $\pi > 0$

$\langle 7 \rangle$ $($ $\forall \texttt{i} \in \{\texttt{1...n}\}$:
$((\texttt{y(i-1).v} \neq \texttt{y(i).v}) \wedge (\texttt{y(i).v} = \texttt{y(i+1).v}))$
$\Rightarrow$ $(\tau < \max(\texttt{y(i-1).}\tau, \texttt{y(i+1).}\tau) + \delta)$
$)(s_2)$
U

⟨8⟩ $(\tau \leq \mathtt{y(n+1)}.\tau + \pi)(s_2)$
U

⟨9⟩ $(\tau < \text{insert}(\mathtt{0}) + \Psi(\mathtt{1}))(s_1)$
C, and $\Psi(\mathtt{1}) = \Delta(\mathtt{1}) < \pi$

⟨10⟩ $(\mathtt{y(1).v} = \mathtt{y(0).v})(s_1)$
I, ⟨9⟩, and lemma 1

⟨11⟩ $(\text{insert}(\mathtt{1}) = \tau - \pi)(s_1)$
⟨10⟩, C, and the definition of "insert"

⟨12⟩ $(\mathtt{y(1).v} \neq \mathtt{y(0).v})(s_2)$
⟨10⟩, A, and U

⟨13⟩ $(\text{insert}(\mathtt{1}) = \tau - \pi)(s_2)$
⟨12⟩, A, and the definition of "insert"

⟨14⟩ $\forall \mathtt{i} \in \{\mathtt{1} \ldots \mathtt{n}+\mathtt{1}\} : \text{insert}(\mathtt{i})(s_2) = \text{insert}(\mathtt{i})(s_1)$
⟨11⟩, ⟨13⟩, and the definition of "insert"

⟨15⟩ $\forall \mathtt{i} \in \{\mathtt{1} \ldots \mathtt{n}\} : (\mathtt{y(i)}.\tau < \text{insert}(\mathtt{i}) + \Psi(\mathtt{i}))(s_2)$
⟨14⟩, I_1, and U

⟨16⟩ $(\mathtt{y(n+1)}.\tau = \text{insert}(\mathtt{n+1}) + \lambda)(s_2)$
⟨14⟩, I_1, and U

⟨17⟩ $I(s_2)$
⟨5⟩, ⟨6⟩, ⟨7⟩, ⟨8⟩, ⟨15⟩, and ⟨16⟩

Thus, $(I_1 \wedge C \wedge A \wedge U) \Rightarrow I(s_2)$ as required.

The FIFO stages

The transition is (for $\mathtt{j} \in \{\mathtt{1} \ldots \mathtt{n}\}$)

```
≪    (y(j−1).v ≠ y(j).v) AND (y(j).v = y(j+1).v)
   → y(j).v, y(j).τ:= y(j−1).v, τ
≫
```

Let s_1 and s_2 be any two states of P and let

$$\begin{array}{lcll}
I_1 & = & I(s_1) & \text{, the invariant} \\
C & = & \quad (\texttt{y(j-1).v} \neq \texttt{y(j).v}) & \text{, the precondition} \\
 & & \wedge \quad (\texttt{y(j).v} = \texttt{y(j+1).v}) & \\
A & = & \quad (\texttt{y(j).v}(s_2) = \texttt{y(j-1).v}(s_1)) & \text{, the multi–assignment} \\
 & & \wedge \quad (\texttt{y(j).}\tau(s_2) = \tau(s_1)) & \\
U & = & \quad \forall \texttt{i} \in \{\texttt{0}\ldots\texttt{j-1},\texttt{j+1}\ldots\texttt{n+1}\} : & \text{, unchanged variables} \\
 & & \qquad \texttt{y(i)}(s_2) = \texttt{y(i)}(s_1) & \\
 & & \wedge \quad \tau(s_2) = \tau(s_1) &
\end{array}$$

By property 2 in section B.1.2, the proof obligation for this transition is

$$I_1 \wedge C \wedge A \wedge U \quad \Rightarrow \quad I(s_2)\,.$$

We assume the left side of the implication and show that this establishes $I(s_2)$.

$\langle 1 \rangle$ $(\texttt{y(j).}\tau = \tau)(s_2)$
A and U

$\langle 2 \rangle$ $\forall \texttt{i} \in \{\texttt{0}\ldots\texttt{j-1},\texttt{j+1}\ldots\texttt{n+1}\} : (\texttt{y(i).}\tau(s_2) = \texttt{y(i).}\tau(s_1))$
U

$\langle 3 \rangle$ $\forall \texttt{i} \in \{\texttt{0}\ldots\texttt{j-1},\texttt{j+1}\ldots\texttt{n+1}\} : (\texttt{y(i).}\tau \leq \tau)(s_2)$
$\langle 2 \rangle$, I_1, and U

$\langle 4 \rangle$ $\forall \texttt{i} \in \{\texttt{0}\ldots\texttt{n+1}\} : (\texttt{y(i).}\tau \leq \tau)(s_2)$
$\langle 1 \rangle$ and $\langle 3 \rangle$

$\langle 5 \rangle$ $(\tau \leq \texttt{y(0).}\tau + \pi)(s_2)$
I_1 and U

$\forall \texttt{i} \in \{\texttt{1}\ldots\texttt{j-2},\texttt{j+2}\ldots\texttt{n}\} :$

$\langle 6 \rangle$ $((\texttt{y(i-1).v} \neq \texttt{y(i).v}) \wedge (\texttt{y(i).v} = \texttt{y(i+1).v}))$
$\Rightarrow$ $(\tau < \max(\texttt{y(i-1).}\tau, \texttt{y(i+1).}\tau) + \delta)$
I_1, and U

$\langle 7 \rangle$ $(\texttt{j} > 1) \Rightarrow (\tau < \max(\texttt{y(j-2).}\tau, \texttt{y(j).}\tau) + \delta)(s_2)$
A and $\delta > 0$

$\langle 8 \rangle$ $(\texttt{y(j-1).v} = \texttt{y(j).v})(s_2)$
A

$\langle 9 \rangle$ $(\mathtt{j} < \mathtt{n}) \Rightarrow (\tau < \max(\mathtt{y(j)}.\tau, \mathtt{y(j{+}2)}.\tau) + \delta)(s_2)$

A and $\delta > 0$

$\forall \mathtt{i} \in \{\mathtt{1} \ldots \mathtt{n}\}$:

$\langle 10 \rangle$ $((\mathtt{y(i{-}1)}.\mathtt{v} \neq \mathtt{y(i)}.\mathtt{v}) \wedge (\mathtt{y(i)}.\mathtt{v} = \mathtt{y(i{+}1)}.\mathtt{v}))$

$\Rightarrow$ $(\tau < \max(\mathtt{y(i{-}1)}.\tau, \mathtt{y(i{+}1)}.\tau) + \delta)$

$\langle 6 \rangle$, $\langle 7 \rangle$, $\langle 8 \rangle$, and $\langle 9 \rangle$

$\langle 11 \rangle$ $(\tau \leq \mathtt{y(n{+}1)}.\tau + \pi)(s_2)$

I_1 and U

$\langle 12 \rangle$ $\forall \mathtt{i} \in \{\mathtt{0} \ldots \mathtt{j{-}1}\} : \mathrm{insert}(\mathtt{i})(s_2) = \mathrm{insert}(\mathtt{i})(s_1)$

U

$\langle 13 \rangle$ $((\mathrm{insert}(\mathtt{j}) = \mathrm{insert}(\mathtt{j{-}1}) - \pi) \wedge (\mathrm{insert}(\mathtt{j{+}1}) = \mathrm{insert}(\mathtt{j{-}1}) - \pi))(s_1)$

C and the definition of "insert"

$\langle 14 \rangle$ $((\mathrm{insert}(\mathtt{j}) = \mathrm{insert}(\mathtt{j{-}1})) \wedge (\mathrm{insert}(\mathtt{j{+}1}) = \mathrm{insert}(\mathtt{j{-}1}) - \pi))(s_2)$

C, A, U, and the definition of "insert"

$\langle 15 \rangle$ $\forall \mathtt{i} \in \{\mathtt{j{+}1} \ldots \mathtt{n}\} : \mathrm{insert}(\mathtt{i})(s_2) = \mathrm{insert}(\mathtt{i})(s_1)$

$\langle 12 \rangle$, $\langle 13 \rangle$, $\langle 14 \rangle$, U, and the definition of "insert"

$\langle 16 \rangle$ $\forall \mathtt{i} \in \{\mathtt{1} \ldots \mathtt{j{-}2}, \mathtt{j{+}2} \ldots \mathtt{n}\} : \mathtt{y(i)}.\tau < \mathrm{insert}(\mathtt{i}) + \Psi(\mathtt{i})$

$\langle 12 \rangle$, $\langle 15 \rangle$, I_1, and U

$\langle 17 \rangle$ $(\mathtt{y(j{-}1)}.\tau < \mathrm{insert}(\mathtt{j{-}1}) + \Psi(\mathtt{j{-}1}))(s_1)$

I_1

$\langle 18 \rangle$ $\mathtt{y(j{-}1)}.\tau(s_1) < \mathrm{insert}(\mathtt{j})(s_2) + \Psi(\mathtt{j}) - \Delta(\mathtt{j})$

$\langle 13 \rangle$, $\langle 14 \rangle$, $\langle 17 \rangle$, and the definition of Ψ

$\langle 19 \rangle$ $\mathtt{y(j{-}1)}.\tau(s_1) < \mathrm{insert}(\mathtt{j})(s_2) + \Psi(\mathtt{j}) - \delta$

$\langle 18 \rangle$ and $\Delta(\mathtt{j}) \geq \delta$

$\langle 20 \rangle$ $(\mathtt{y(j{+}1)}.\tau < \mathrm{insert}(\mathtt{j{+}1}) + \Psi(\mathtt{j{+}1}))(s_1)$

I_1

$\langle 21 \rangle$ $\mathtt{y(j{+}1)}.\tau(s_1) < \mathrm{insert}(\mathtt{j})(s_2) + \Psi(\mathtt{j}) + \Delta(\mathtt{j{+}1}) - \pi$

$\langle 13 \rangle$, $\langle 14 \rangle$, $\langle 20 \rangle$, and the definition of Ψ

$\langle 22 \rangle$ $\mathtt{y(j{+}1)}.\tau(s_1) < \mathrm{insert}(\mathtt{j})(s_2) + \Psi(\mathtt{j}) - \delta$

$\langle 21 \rangle$ and $\Delta(\mathtt{j{+}1}) \leq \pi - \delta$

$\langle 23 \rangle$ $\tau(s_1) < \text{insert}(j)(s_2) + \Psi(\texttt{j})$
$\langle 19 \rangle$, $\langle 22 \rangle$, C, and I_1

$\langle 24 \rangle$ $(\texttt{y(j)}.\tau < \text{insert}(\texttt{j}) + \Psi(\texttt{j}))(s_2)$
$\langle 23 \rangle$ and A

$\langle 25 \rangle$ $\forall \texttt{i} \in \{\texttt{1} \ldots \texttt{n}\} : \texttt{y(i)}.\tau < \text{insert}(\texttt{i}) + \Psi(\texttt{i})$
$\langle 16 \rangle$ and $\langle 24 \rangle$

$\langle 26 \rangle$ $\texttt{y(n+1)}.\tau = \text{insert}(\texttt{n+1}) + \lambda$
U

$\langle 27 \rangle$ $I(s_2)$
$\langle 4 \rangle$, $\langle 5 \rangle$, $\langle 10 \rangle$, $\langle 11 \rangle$, $\langle 25 \rangle$, and $\langle 26 \rangle$

Thus, $(I_1 \wedge C \wedge A \wedge U) \Rightarrow I(s_2)$ as required.

The receiver

The transition is

$$\ll \tau \geq \texttt{y(n+1)}.\tau + \pi \rightarrow \texttt{y(n+1).v, y(n+1)}.\tau \texttt{ := NOT y(n+1).v, } \tau \gg$$

Let s_1 and s_2 be any two states of P and let

$$\begin{array}{lcll} I_1 & = & I(s_1) & \text{, the invariant} \\ C & = & (\tau \geq \texttt{y(n+1)}.\tau)(s_1) & \text{, the precondition} \\ A & = & \quad (\texttt{y(n+1)}.\texttt{v}(s_2) = \neg \texttt{y(n+1)}.\texttt{v}(s_1)) & \text{, the multi–assignment} \\ & & \wedge \quad (\texttt{y(n+1)}.\tau(s_2) = \tau(s_1)) & \\ U & = & \quad \forall \texttt{i} \in \{\texttt{0} \ldots \texttt{n}\} : \texttt{y(i)}(s_2) = \texttt{y(i)}(s_1) & \text{, unchanged variables} \\ & & \wedge \quad \tau(s_2) = \tau(s_1) & \end{array}$$

By property 2 in section B.1.2, the proof obligation for this transition is

$$I_1 \wedge C \wedge A \wedge U \quad \Rightarrow \quad I(s_2)\,.$$

We assume the left side of the implication and show that this establishes $I(s_2)$.

$\langle 1 \rangle$ $(\texttt{y(n+1)}.\tau = \tau)(s_2)$
A

$\langle 2 \rangle$ $\forall \texttt{i} \in \{\texttt{1} \ldots \texttt{n+1}\} : (\texttt{y(i)}.\tau \leq \tau)(s_2)$
I_1 and U

$\langle 3 \rangle$ $\forall \mathtt{i} \in \{\mathtt{0} \ldots \mathtt{n{+}1}\} : (\mathtt{y(i)}.\tau \leq \tau)(s_2)$

$\langle 1 \rangle$ and $\langle 2 \rangle$

$\langle 4 \rangle$ $(\tau \leq \mathtt{y(0)}.\tau + \pi)(s_2)$

I_1 and U

$\langle 5 \rangle$ $($ $\forall \mathtt{i} \in \{\mathtt{1} \ldots \mathtt{n{-}1}\} :$

$((\mathtt{y(i{-}1)}.\mathtt{v} \neq \mathtt{y(i)}.\mathtt{v}) \wedge (\mathtt{y(i)}.\mathtt{v} = \mathtt{y(i{+}1)}.\mathtt{v}))$

$\Rightarrow$ $(\tau < \max(\mathtt{y(i{-}1)}.\tau, \mathtt{y(i{+}1)}.\tau) + \delta)$

$)(s_2)$

I_1 and U

$\langle 6 \rangle$ $(\tau \leq \mathtt{y(n{+}1)}.\tau + \pi)(s_2)$

$\langle 1 \rangle$ and $\pi > 0$

$\langle 7 \rangle$ $(\forall \mathtt{i} \in \{\mathtt{1} \ldots \mathtt{n}\} : \mathtt{y(i)}.\tau < \mathrm{insert}(\mathtt{i}) + \Psi(\mathtt{i}))(s_2)$

I_1 and U

$\langle 8 \rangle$ $(\tau = \mathrm{insert}(\mathtt{n{+}1}) + \lambda + \pi)(s_1)$

C, I_1, and $I_1 \Rightarrow (\mathtt{y(n{+}1)}.\tau = \mathrm{insert}(\mathtt{n{+}1}) + \lambda)$

$\langle 9 \rangle$ $(\tau < \mathrm{insert}(\mathtt{n{+}1}) + \Psi(\mathtt{n}) + \pi)(s_1)$

$\langle 8 \rangle$, the definition of Ψ and $\Delta(\mathtt{n{+}1}) > 0$

$\langle 10 \rangle$ $(\mathtt{y(n)}.\mathtt{v} \neq \mathtt{y(n{+}1)}.\mathtt{v})(s_1)$

$\langle 9 \rangle$ and lemma 2

$\langle 11 \rangle$ $(\mathrm{insert}(\mathtt{n{+}1}) = \mathrm{insert}(\mathtt{n}) - \pi)(s_1)$

$\langle 10 \rangle$ and the definition of "insert"

$\langle 12 \rangle$ $(\mathrm{insert}(\mathtt{n}) + \lambda = \tau)(s_1)$

$\langle 11 \rangle$, C, and I_1

$\langle 13 \rangle$ $\mathrm{insert}(\mathtt{n})(s_1) = \mathrm{insert}(\mathtt{n})(s_2)$

U

$\langle 14 \rangle$ $(\mathtt{y(n)}.\mathtt{v} = \mathtt{y(n{+}1)}.\mathtt{v})(s_1)$

$\langle 10 \rangle$ and A

$\langle 15 \rangle$ $(\mathrm{insert}(\mathtt{n{+}1}) = \mathrm{insert}(\mathtt{n}))(s_2)$

$\langle 14 \rangle$ and the definition of "insert"

$\langle 16 \rangle$ $\mathrm{insert}(\mathtt{n{+}1})(s_2) + \lambda = \tau(s_1)$

$\langle 12 \rangle$, $\langle 13 \rangle$, and $\langle 15 \rangle$

$\langle 17 \rangle$ $(\texttt{y(n+1)}.\tau = \text{insert}(\texttt{n+1}) + \lambda)(s_2)$
$\langle 16 \rangle$, A, and U

$\langle 18 \rangle$ $I(s_2)$
$\langle 3 \rangle$, $\langle 4 \rangle$, $\langle 5 \rangle$, $\langle 6 \rangle$, $\langle 7 \rangle$, and $\langle 17 \rangle$

Thus, $(I_1 \wedge C \wedge A \wedge U) \Rightarrow I(s_2)$ as required.

The transition that models time

The transition is

```
≪ TRUE → τ:=
  {ANY ν: ℛ
          (ν > τ)
      AND (ν ≤ y(0).τ + π)
      AND {AND i: 1..n |
                   ((y(i−1).v≠y(i).v) AND (y(i).v=y(i+1).v))
             IMPLIES (ν < max(y(i−1).v, y(i+1).v) + δ)
          }
      AND (ν ≤ y(n+1).τ + π)
    ? ν
  }
≫
```

Let s_1 and s_2 be any two states of P and let

$$
\begin{array}{lcll}
I_1 & = & I(s_1) & \text{, the invariant} \\
A & = & \quad \tau(s_2) > \tau(s_1) & \text{, the assignment} \\
 & & \wedge \quad \tau(s_2) \leq \texttt{y(0)}.\tau(s_2) + \pi & \\
 & & \wedge \quad \forall i \in \{\texttt{1} \ldots \texttt{n}\} : & \\
 & & \qquad ((\texttt{y(i−1).v} \neq \texttt{y(i).v}) \wedge (\texttt{y(i).v} = \texttt{y(i+1).v}))(s_2) & \\
 & & \qquad \Rightarrow \quad (\tau < \max(\texttt{y(i−1).v}, \texttt{y(i+1).v}) + \delta)(s_2) & \\
U & = & \forall \texttt{i} \in \{\texttt{0} \ldots \texttt{n+1}\} : \texttt{y(i)}(s_2) = \texttt{y(i)}(s_1) & \text{, unchanged variables}
\end{array}
$$

By property 2 in section B.1.2, the proof obligation for this transition is

$$I_1 \wedge A \wedge U \quad \Rightarrow \quad I(s_2)\,.$$

We assume the left side of the implication and show that this establishes $I(s_2)$.

$\langle 1 \rangle$ $\forall \mathtt{i} \in \{\mathtt{0} \ldots \mathtt{n{+}1}\} : (\mathtt{y(i)}.\tau = \mathtt{y(i)}.\tau)(s_2)$
U and $A \Rightarrow (\tau(s_2) > \tau(s_1))$

$\langle 2 \rangle$ $(\tau \leq \mathtt{y(0)}.\tau + \pi)(s_2)$
A

$\langle 3 \rangle$ $($ $\forall \mathtt{i} \in \{\mathtt{1} \ldots \mathtt{n}\}$:
$((\mathtt{y(i{-}1)}.\mathtt{v} \neq \mathtt{y(i)}.\mathtt{v}) \wedge (\mathtt{y(i)}.\mathtt{v} = \mathtt{y(i{+}1)}.\mathtt{v}))$
$\Rightarrow$ $(\tau < \max(\mathtt{y(i{-}1)}.\tau, \mathtt{y(i{+}1)}.\tau) + \delta)$
$)(s_2)$
A

$\langle 4 \rangle$ $(\tau \leq \mathtt{y(n{+}1)}.\tau + \pi)(s_2)$
A

$\langle 5 \rangle$ $\forall \mathtt{i} \in \{\mathtt{1} \ldots \mathtt{n}\} : (\mathtt{y(i)}.\tau < \mathrm{insert}(\mathtt{i}) + \Psi(\mathtt{i}))(s_2)$
U

$\langle 6 \rangle$ $(\mathtt{y(n{+}1)}.\tau = \mathrm{insert}(\mathtt{n{+}1}) + \lambda)(s_2)$
U

$\langle 7 \rangle$ $I(s_2)$
$\langle 1 \rangle$, $\langle 2 \rangle$, $\langle 3 \rangle$, $\langle 4 \rangle$, $\langle 5 \rangle$, and $\langle 6 \rangle$

Thus, $(I_1 \wedge C \wedge A \wedge U) \Rightarrow I(s_2)$ as required.

Thus, every transition maintains I. By property 1, I is an invariant of P as claimed.

□

B.2.4 The correctness of STARI

It has just been established that I is an invariant of P. For a STARI interface to operate correctly, the transmitter and receiver must observe the self–timed protocol of the FIFO. In particular, the current value output by the transmitter must be inserted into the FIFO (i.e. transferred to `y(1)`) before the transmitter is enabled to output a new value. Likewise, between each pair of acknowledge tokens output by the receiver, the FIFO must output a new value. These two conditions are expressed by T as

defined below:

$$\begin{aligned} T \quad = \quad & (\tau \geq \mathtt{y(0).}\tau + \pi) \Rightarrow (\mathtt{y(1).v} = \mathtt{y(0).v}) \\ \wedge \quad & (\tau \geq \mathtt{y(n+1).}\tau + \pi) \Rightarrow (\mathtt{y(n).v} \neq \mathtt{y(n+1).v}) . \end{aligned}$$

Lemma 1 shows that I implies the first clause of T, and lemma 2 shows that I implies the second clause. Thus, the invariant I guarantees the timing of a STARI interface is correct.

B.3 Remarks on the Verification

The previous section established an invariant for the STARI interface and showed that this invariant ensures that the synchronous transmitter and receiver observe the protocol of the self–timed FIFO. There are several issues that were not addressed in the formal proof. In particular:

Liveness:

Invariants are **safety properties**; they assert the "bad" things *don't* happen. In this case, the invariant asserts that the interface does not reach a state where the transmitter is enabled to output a new data value and the FIFO has not yet accepted the current one; and likewise for acknowledgements from the receiver. Invariants do not assert that "good" things *do* happen; for example, that the transmitter is guaranteed to eventually output another value. Such assertions are called **liveness properties** [Pnu81].

In the approach taken here, real–time properties are formulated as safety properties [Lam86]. The invariant guarantees that if the current time advances without bound, then an infinite sequence of data values will be conveyed through the STARI interface. However, the invariant does not require time to advance without bound.

There are valid executions of the program shown in figure 34 in which the current time, τ, converges to a finite limit. The transition that updates τ is intended to be a *conservative* model of time. If it allows behaviors that are

not physically realistic, this only means that the circuit has been verified for a wider range of operating conditions than necessary, and no harm has been done. On the other hand, this model should admit all possible behaviors of time; in particular, it should be possible for time to advance without bound. Abadi and Lamport [AL92] have shown that if a program is "non–Zeno" (a safety property), then a assumption of weak fairness ensures that time can advance without bound. The program shown in figure 34 is non–Zeno.

More safety properties:

The verification in the previous section shows that the signaling protocol of the self–timed FIFO is observed; other correctness criteria were not considered. In particular, it was not shown that the sequence of values output by the FIFO is identical to the sequence of values output by the transmitter. In fact, the abstraction for signals only distinguished between states of the self–timed protocol; different data values were not distinguished. We chose to focus on timing issues because this is the novel aspect of STARI. Self–timed FIFOs like the one described by the program of figure 34 are common in the literature of self–timed design [DNS89, Sut89, Bur91, Wil91] and are well understood. On the other hand, it is a straightforward exercise to formalize the property that the FIFO outputs the sequence of values that it received from the transmitter. To do this, a less abstract representation of signals that distinguishes data values must be used. Then, sequences must be defined (e.g. using typical list operations). Finally, the invariant is strengthened to include clauses that imply that the concatenation of the sequence of values input by the receiver with the sequence of values held in the FIFO is equal to the sequence of values output by the transmitter.

Initialization:

Showing that I (as defined in figure 35 is an invariant of the program shown in figure 34 establishes that if the program ever enters a state in which I is satisfied, then all future states will satisfy I as well. It does not guarantee that I ever enters such a state. In general, it must be assumed that a circuit can

start in any state, and a reset signal must be applied to ensure that the circuit enters a state which satisfies the desired invariant. Such initialization is, in general, performed as a synchronous operation, even in a self–timed circuit. In particular, the reset signal is asserted long enough to ensure that the circuit reaches a stable state. `Synchronized Transitions` provides combinators for describing synchronous operations (see [SG90]), and a program can be written that includes initialization operations. However, further proof rules are required in addition to the two presented in section B.1.2. The generalization of the proof techniques presented here to this case is beyond the scope of this dissertation.

Composition:

The program shown in figure 34 describes how a synchronous transmitter and receiver can interact with a self–timed FIFO without timing hazards. To produce a real implementation, this specification can be used as the starting point to derive a more complete design for the FIFO (e.g. the one described in chapter 4.1) and more comprehensive descriptions of the transmitter and receiver (e.g. nodes of a multiprocessor, instead of circuit that only toggle input signals of the FIFO). To show that these more detailed implementations interact correctly, a new proof must be constructed for the more detailed system (such a proof would be much more complicated than the one of the previous section). Alternatively, compositional proof techniques can be applied. Such techniques require more formalism than can be presented in this dissertation (see [AL89] for an example of this approach).

Timing Models:

The program shown in figure 34 employs an "inertial model" for delays. This is a very simplistic model, and one could criticize the analysis on this basis. On the other hand, it is straightforward to show that once an input to a FIFO stage changes, it retains its value until the stage performs a transition. This satisfies the requirements of most popular timing models.

Another limitation of the delay model for the FIFO stages is that the a stage forwards data to its successor and an acknowledge to its predecessor as a single

atomic action. In practice (e.g. in the design described in chapter 4.1), these two operations may involve different circuits. The program of 34 can be refined to include such details. The timing verification of the resulting program is analogous to that presented in the previous section; of course, more details must be addressed.

The synchronous operations of the transmitter and of the receiver are assumed to occur at times separated by exactly π. As described in chapter 3.1.3, a similar analysis applies when the transmitter and receiver are assumed to have bounded delays. This requires introducing an auxiliary variable that records the the sequence of times of the transmitter's operations, and another for the times of the receiver's operations. There is an analogous invariant and analogous proof for the resulting program.

Finally, although time is modeled as a real number, operations are assumed to be atomic; the underlying physical (i.e. analog) circuitry is ignored. As shown in chapter 4.4, these issues can be significant. This is an area for future research.

B.4 Related Work

The `Synchronized Transitions` notation used in this appendix has been used extensively for formal hardware verification. `Synchronized Transitions` was originally proposed by Ravn and Staunstrup [RS87]. Since then, `Synchronized Transitions` has been used extensively for the design and verification of both synchronous [SG88] and asynchronous [SG89] circuits. An overview of prior work using `Synchronized Transitions` is presented in [SG90]. Students at the Danish Technical University have implemented several syntax directed translators [Chr90, HC90, Nie90] that accept `Synchronized Transitions` programs and produce structural circuit descriptions in the MODEL [Eur] hardware description language.

There are many striking similarities between `Synchronized Transitions` and UNITY, developed by independently Chandy and Misra[CM88]. Programs written in UNITY consist of guarded multiassignments that are composed with operators

that correspond to the synchronous and asynchronous combinators. Other similarities between `Synchronized Transitions` and UNITY include the use of shared variables for communication and the absence of sequential composition. Chandy and Misra propose UNITY as a general programming paradigm, and present solutions to many concurrent programming problems. Most of the proof techniques developed for UNITY can be applied to `Synchronized Transitions` programs as well.

The expression of real–time properties as safety properties is described in [Lam86]. An extensive development of this approach is given in [AL92]. The latter work includes verifying real time properties of a simple FIFO queue. The FIFO considered in [AL92] does not include acknowledge signals: it is shown to accept data from a transmitter that is not too fast and output data at a bounded rate. As the times at which this FIFO outputs data are not correlated to the operations of the consumer of these values. Accordingly, this FIFO cannot be used to compensate for clock skew like the self–timed FIFO of a STARI interface. The unique feature of the FIFO in STARI is that each stage waits both for data and for acknowledgements. This leads to a schedule for the propagation of both data and acknowledgements that ensures proper operation with both the transmitter and the receiver.

Appendix C

Analysis of Cache Consistency Schemes

C.1 Write–Through Overhead

In this appendix, we derive upper bounds for overhead due the write–through policy for shared data of our consistency schemes. Our analysis models the write buffer as a queuing system. We use the following definitions:

p_{sw} the probability that an instruction is a write to shared data (i.e. the dual–port memory).

p_{ms} the probability that an instruction is a read to shared data that misses in the cache.

T_c the fixed cycle time of the dual–port memory. For simplicity, we assume T_c is fixed.

$C_{w,k}$ overhead due to the write–through caching policy for shared data for a write–buffer with capacity k. Reads cannot bypass pending writes.

$C_{wb,k}$ overhead due to the write–through caching policy for shared data for a write–buffer with capacity k. Reads bypass pending writes.

$C_{ww,k}$ the overhead due to stalled writes when using a write–through caching policy for shared data and a write–buffer with capacity k.

C.1.1 The single stage write buffer

A write can cause the processor to stall if it is followed by another read or write to the dual–port memory within T_c cycles. Let p_s denote the probability that an instruction requires access to the shared memory (i.e. it is a shared–write, or it is a cache miss for shared data). For a single stage write buffer,

$$C_{w,1} = p_{sw} \sum_{i=1}^{T_c} (T_c - i)(1 - p_{sw} - p_{ms})^{i-1} p_s \,. \tag{14}$$

The summation gives the expected overhead per shared write. In particular, $(1 - p_{sw} - p_{ms})^{i-1}(p_{sw} + p_{ms})$ is the probability of not blocking on the first $i-1$ cycles after a shared write and blocking on the i^{th}, and $(T_c - i)$ is the cost of blocking on the i^{th} cycle. Standard combinatorial methods yield:

$$C_{w,1} = p_{sw} \left(T_c - \frac{1 - (1 - p_{sw} - p_{ms})^{T_c}}{p_{sw} + p_{ms}} \right) . \tag{15}$$

Noting that

$$(1 - x)^k \leq 1 - kx + \frac{k(k-1)}{2} x^2$$

for $0 \leq x \leq 1$ and $k > 0$, and that $p_s \leq p_{sw} + p_{ms}$, we conclude

$$C_{w,1} \leq \frac{T_c(T_c - 1)}{2} p_{sw}(p_{sw} + p_{ms}) \,. \tag{16}$$

C.1.2 Unbounded buffer without read bypassing

With an unbounded buffer, writes never block. All overhead is due to reads waiting for the write buffer to empty. Thus the overhead is the product of the frequency of reads to the dual–port memory and the expected waiting time in the write buffer. If $p_{sw} T_c < 1.0$, we can analyze the buffer as a M/D/1 queue, where writes into the write buffer are arrivals, and the completing writes in the dual–port memory are services. This analysis ignores the emptying of the write buffer with each read; thus,

it provides an upper bound on the overhead. Using the Pollaczek–Khinchin mean value formula [Kle75b], we obtain

$$w \leq \frac{T_c(T_c-1)}{2(1-T_c p_{sw})} p_{sw}$$

where w is the expected time that a read has to wait for all pending writes to complete. It follows immediately that

$$C_{w,\infty} \leq \frac{T_c(T_c-1)}{2(1-T_c p_{sw})} p_{sw} p_{ms} \,. \tag{17}$$

C.1.3 Unbounded buffer with read bypassing

When reads can bypass writes, overhead is only incurred when a read to the dual–port memory has to wait for the current write to complete. For M/D/1 queues (such as this one) the expected residual service time is half the mean service time [Kle75b]. Therefore, the expected time that a read blocks is

$$w = (1-p_0)\frac{T_c}{2}$$

where w is the expected time that a read must wait for the current write (if any) to complete, and p_0 is the probability that the write buffer is empty. For any stable queueing system, the probability that the queue is not empty times the service rate must equal the arrival rate. Thus, $(1-p_0) = T_c p_{sw}$, and we conclude,

$$C_{wb,\infty} = \frac{{T_c}^2}{2} p_{sw} p_{ms} \,. \tag{18}$$

C.1.4 How big is infinity?

The results presented in A.2 and A.3 assumed an unbounded buffer, but real designs have buffers of fixed capacity. Fortunately, the behavior of a rather small buffer can closely approximate the unbounded one. When a finite buffer is used, writes can block and contribute to the overhead. The overhead due to blocking reads will be lowered slightly because the expected queue length will decrease due to a lower frequency of writes. We derive an upper bound on the overhead due to blocking

writes for bounded buffers, this provides an upper bound on the difference between the overhead of a bounded write buffer and the unbounded ones analyzed above.

We now derive the following inequalities:

$$\begin{aligned} C_{ww,k} &< P[N_k = k] p_{sw} T_c \\ P[N_k = k] &\leq P[N_\infty \geq k] = P[W_\infty \geq \frac{k}{p_{sw}} - T_c] \leq \zeta^{T_c - \frac{k}{p_{sw}}} \end{aligned}$$

where $C_{ww,k}$ is the overhead due to blocking writes, $P[N_k = k]$ is the probability that a buffer of capacity k is full, $P[N_\infty \geq k]$ is the probability that an unbounded buffer contains at least k items, $P[W_\infty \geq y]$ is the probability that the waiting time in an unbounded buffer is at least y, and ζ is a function of T_c and p_{sw}; ζ is always greater than 1. Combining these inequalities yields

$$C_{ww,k} < p_{sw} T_c \zeta^{(T_c - \frac{k}{p_{sw}})} \tag{19}$$

The first inequality, $C_{ww,k} < P[N_k = k] p_{sw} T_c$ follows from the definition of $C_{ww,k}$ and the observation that the residual service time of the first item in a full buffer can be no greater than $T_c - 1$ when a blocking write is performed.

To establish the second inequality, we consider a bounded buffer and an unbounded one with the same workload, and assume that both buffers are initially empty. Induction on the sequence of arrivals to both buffers establishes that the number of items in the bounded buffer at the instant of a particular arrival is always less than or equal to the number in the unbounded buffer at the instant of the same arrival. The second inequality, $P[N_k = k] \leq P[N_\infty \geq k]$ follows immediately.

$P[N_\infty \geq k] = P[W_\infty \geq \frac{k}{p_{sw}} - T_c]$ is Little's result.

Let the variable $u_n = T_c - t_{n+1}$ where t_n is time between issuing the n^{th} and $(n+1)^{th}$ writes. The discrete probability density function for u_n is

$$P[u_n = m] = \begin{cases} p_{sw}(1 - p_{sw})^{T_c - 1 - m} & , m < T_c - 1 \\ 0 & , m \geq T_c - 1 \, . \end{cases}$$

Let

$$\zeta = \sup\{z > 1 : E[z^{u_n}] \leq 1\} \, .$$

Then [Kin70, Kle76a],

$$P[W_\infty \geq y] \;\leq\; \zeta^{-y}\,.$$

By definition,

$$E[z^{u_n}] \;=\; \sum_{m=-\infty}^{T_c-1} z^m p_{sw}(1-p_{sw})^{T_c-1-m} \;=\; \frac{p_{sw}z^{T_c}}{z-1+p_{sw}}\,,$$

and it follows that ζ is the only real root of the polynomial

$$S(z) \;=\; p_{sw}z^{T_c} - z + 1 - p_{sw} \tag{20}$$

that is greater than 1. $S(z)$ has a minimum at $\eta = (p_{sw}T_c)^{-1/(T_c-1)}$, and at this point $S(z) < 0$. Since $\eta > 1$, we conclude $\zeta > \eta$,

$$\zeta \;>\; (p_{sw}T_c)^{-1/(T_c-1)}\,. \tag{21}$$

(This estimate corresponds to modeling the queue as an M/M/1 queue). The plots in this paper were obtained by finding the roots of $S(z)$ using Newton's method.

C.2 Workload Statistics

The analysis of the two–list and physical alias schemes presented in section 5.4 and Appendix A, include the following workload statistics:

p_{ms} the probability that an instruction causes a cache miss for shared data.

p_{sw} the probability that an instruction is a write to shared data, (the dual–port memory).

α the fraction of the physical page frames in the dual–port memory held free by the virtual memory management software.

I_p the average number of instructions executed between receptions of pages from the network.

Workload statistics shared memory programs are reported in [EK88] and [OA89]. Cache miss statistics are given in [OA89] but not [EK88]; therefore, we use the data from [OA89] to estimate p_{ms}. Conversely, only [EK88] reports statistics on the frequency of writes to shared data; so, we use the [EK88] data to estimate p_{sw}. We note that to the extent that these two studies can be compared, the reported workloads are very similar. A slightly higher write/read ratio (by about 20%) is reported in [OA89], and the data in [EK88] shows a slightly higher fraction of the references being to shared data (by about 30%). These differences are small; overall, the two studies seem reasonably compatible. However, by combining the results of two different studies, we derive workloads that do not correspond to any specific architecture or application. In section 5.4, we showed that the low overhead of our consistency schemes is robust for wide variations of the workload statistics; therefore, we believe that our conclusions are qualitatively valid in spite of the limitations of our workload estimates.

We derive p_{ms} using the following conditional probabilities:

$$\begin{aligned} p_{ms} \;=\; & P\{\text{load or store}\} \\ & \times\; P\{\text{cache miss} \mid \text{load or store}\} \\ & \times\; P\{\text{shared reference} \mid \text{cache miss}\}\,. \end{aligned}$$

We assume that the cache miss rate is the same for shared and private data:

$$P\{\text{shared reference} \mid \text{cache miss}\} = P\{\text{shared reference} \mid \text{load or store}\}\,.$$

This is probably slightly optimistic as misses to shared data may arise after another processor has had exclusive access, whereas this does not occur for private data. Estimates for each of these probabilities are given in [OA89]:

min		what		max	source
0.2	$\leq$	$P\{\text{load or store}\}$	$\leq$	0.4	*ls* in [OA89]
0.004	$\leq$	$P\{\text{cache miss} \mid \text{load or store}\}$	$\leq$	0.024	*msdat* in [OA89]
0.08	$\leq$	$P\{\text{shared reference} \mid \text{load or store}\}$	$\leq$	0.42	*shd* in [OA89]

Accordingly, $0.000064 \leq p_{ms} \leq 0.004$. Having used the worst–case value for each parameter, this estimate is conservative (pessimistic). Few applications would exhibit

worst–case behavior with respect to all three parameters. We note that the cache miss rate depends strongly on the cache size. The value above is for a 16Kbyte data cache.

To derive a value for p_{sw}, we need an estimate of the rate of writes to shared data. For the programs analyzed in [EK88], the fraction of data references that were shared writes ranged from 0.01 to 0.05. These values were for CISC processors: a Sequent Balance 8000, and an ELXSI 6400. Our CPI based analysis assumes a one–instruction–per–cycle architecture (a RISC). If we consider the same data accesses but assume a maximum load–store ratio for a RISC processor of 0.4 (as in [OA89]), we obtain $0.004 \leq p_{sw} \leq 0.02$.

A significant feature of these programs reported in [EK88] is that the fraction of references to shared data that were writes was substantially lower than the corresponding fraction for private data. In particular about 22% of all references to private data were writes, but only 8% of all references to shared data were writes (geometric means). With this access behavior, the policy of only using write–through for shared data significantly reduces the memory bandwidth required compared to a policy of write–through for all data.

We assume that the virtual memory management software can hold 5% of the dual–port memory's page frames free. Most applications that need more than 95% of the dual–port memory need a processor with a larger dual–port memory. This yields a value for α of 0.05.

We now derive a value for I_p. We note that

$$I_p = \frac{1}{P\{\text{read fault to shared data}\}} .$$

Using conditional probabilities, we obtain,

$$\begin{aligned} P\{\text{read fault to shared data}\} = & \\ & P\{\text{load or store}\} \\ \times \quad & P\{\text{shared reference} \mid \text{load or store}\} \\ \times \quad & P\{\text{remote reference} \mid \text{shared reference}\} . \end{aligned}$$

We use the values of *ls* and *shd* from [OA89] for $P\{\text{load or store}\}$ and $P\{\text{shared reference} \mid \text{load or store}\}$ respectively as we did earlier to estimate p_{ms}.

name	worst	best	description
α	0.05	—	fraction of pages unallocated
I_p	15,000	60,000	# instruction / remote page–fault
p_{ms}	0.004	0.000064	per instruction probability of cache miss, shared data
p_{sw}	0.02	0.004	per instruction probability of write to shared data

Figure 36: Work–load statistics

In [Li86], $P\{\text{remote reference} \mid \text{shared reference}\}$ was reported to be between 0.0001 and 0.0004 for the parallel applications studied. This yields a worst–case value for I_p of 15,000. Having used worst–case values for each parameter, this estimate is conservative (pessimistic).

The worst–case values of p_{ms}, p_{sw}, α and I_p derived in this section are shown in figure 36