SELF-TIMED RINGS AND THEIR APPLICATION TO DIVISION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF
ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By

Ted Eugene Williams

May 1991

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

[illegible]

Mark A. Horowitz
(Principal Advisor)

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

[illegible]

David L. Dill

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

[illegible]

Robert W. Dutton

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies

Abstract

Self-timed systems avoid the problems associated with the global clocks of synchronous systems. This thesis introduces a new type of structure called a self-timed ring that can pass data multiple times through the same function blocks without requiring any external control signals or clocking. The latency and throughput of self-timed rings are analyzed by a method that also determines the performance of asynchronous pipelines as a special case. By meeting certain constraints suggested by this analysis, a self-timed ring can completely hide its control logic delays and achieve operation with zero overhead. If, in addition, the ring is composed of a proposed domino stage configuration without latches, then the ring achieves, in much less area, the same minimal-latency operation as an unrolled combinational array implementing the same function.

A prime example of a problem for which a self-timed ring implementation achieves high performance is the iterative computation of the arithmetic function of division. This thesis compares two self-timed divider chips: a preliminary design and an improved one that adheres to the constraints determined by this research. Measurements showed these design techniques increased performance by a factor of 2.2 due to architecture alone. The self-timed ring of the new divider occupies 7mm^2 in a 1.2μ CMOS process and computes quotient bits in 2.9nS, requiring a total latency of 45nS to 160nS for a full 54-bit result, depending on the data operands.

Acknowledgements

My interests in self-timing began from the earliest days of my involvement with integrated circuits. Carver Mead's approach to VLSI design was lucidly presented by Chuck Seitz in the VLSI classes at Caltech. Based on his own extensive experience and founding work in asynchronous circuits, he suggested useful self-timed research projects using precharged domino circuits. I appreciate Chuck setting me enthusiastically on the path in this exciting area.

The work represented by this thesis was eased by the generous help of many people at Stanford. Most important have been the expertise, ideas, and skilled critiques from my primary research advisor, Mark Horowitz. His involvement with both the research and the text has provided both focus and clarity. The Stanford Asynchronous Seminar has been useful in providing a forum to discuss issues in my own research and in related areas. David Dill played an essential role in organizing these meetings and being my second advisor. Teresa Meng inspired me to analyze self-timed circuit performance in order to make precise comparisons of my work and the styles of other researchers.

The chip designs demonstrating the applications of this thesis were performed with CAD tools developed at UC Berkeley and Stanford. These tools were provided and maintained by the dedication and toil of several friends. Bob Mayo has taken on the role of supplying the world with

MAGIC, a first-class custom integrated circuit layout program. Arturo Salz improved and enhanced the switch-level simulator called IRSIM which played the primary role in design verification. Don Stark helped with appropriate hints in the operation of the GEMINI netlist comparator. Jim Burr provided access to and help with integrated circuit testing equipment.

Designing a custom integrated circuit takes a lot of work, and I am thankful to have received some assistance with the manual phases of the designs and the chip layout. Robert Alverson and Tsen-Shou Yang contributed to the first chip design. Yukihiko Shimazu and Leland Ho helped with the second chip design.

Thanks are also due to the colleagues and friends who have provided inspiration and moral support. Mark Greenstreet has been supportive and encouraging ever since we corresponded when I enrolled at Caltech. Mark Santoro has been a wise sage and model throughout our mutual and complementary sojourns at Stanford. Steve Burns and Alain Martin have provided fresh viewpoints and challenged me with their own work. Steve Nowick and Drew Wingard made careful and constructive comments on some of my publications prior to this thesis.

The text of this manuscript was improved by the generous assistance of Evelin Sutherland who scrutinized and clarified the language and sentence structure.

I especially thank Mark Horowitz, David Dill, Teresa Meng, and Robert Dutton for serving on my orals committee and reading this thesis.

My research has been made possible by a continuing succession of arrangements providing financial support with the freedom to pursue my own interests. I am grateful for the support of a Graduate Fellowship from the National Science Foundation. DARPA funded workstations and IC fabrication through MOSIS. Peter Møller-Nielsen and Jørgen Staunstrup arranged a position as a visiting professor at Aarhus University in Denmark. Karl Klingsheim and Einar Ås were my hosts during a brief stay at the national Electronics Laboratory (ELAB) in Trondheim, Norway. The US Government's Small Business Innovation Research Program supported my work through Silicon Engines, Inc. Most recently, HaL Computer Systems has been gracious in providing support for the writing of this thesis.

My parents have been loving and I will always appreciate and admire them. And finally, my new wife, Marianne, deserves deep thanks for her patience and care.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Traditional logic design separates data computation from storage. Computation takes place in combinational logic or "function blocks" and storage occurs in registers or latches. Clocks are distributed globally within a system to synchronize all the registers or latches. A clock tells each latch when its data inputs are to be considered valid. This approach to building **synchronous** systems is becoming increasingly more difficult at higher clock frequencies because of greater problems with clock skew and transmission line effects.

Other design approaches, collectively known as **self-timed** systems [SEIT80], avoid some of the problems of synchronously clocked systems. Instead of using an externally supplied global clock, self-timed systems either generate “private” local clocks with an on-chip clock generator [CHAPI84, SANT89], use carefully-crafted matched delays [CHAPP91], or use local handshaking control to communicate the presence of valid data [MENG88]. Local handshaking between blocks is also called asynchronous control [MULL63, MILL65] because events in blocks do not have a specific timing relationship except when they handshake [STAU87]. Unfortunately, the handshaking in previous asynchronous circuit implementations, such as in [JACO90], has degraded performance because the handshaking added overhead delays of up to 100% relative to the function block delays.

This thesis analyzes configurations and control structures for asynchronous circuits and introduces a new class of circuits called “self-timed rings,” which can avoid performance degradations from handshaking and control logic when the control meets certain conditions. Self-timed rings that are designed using the ideas in this thesis can completely hide handshaking delays and thus operate with zero overhead. Because of their repetitive structure, self-timed rings are particularly suitable for computing iterative functions.

Since the computation of the arithmetic operation of division is iterative, division is an example of an application that can achieve high performance by using a self-timed ring structure. I therefore chose division as a particular application problem for which to demonstrate integrated circuit implementations of self-timed rings.

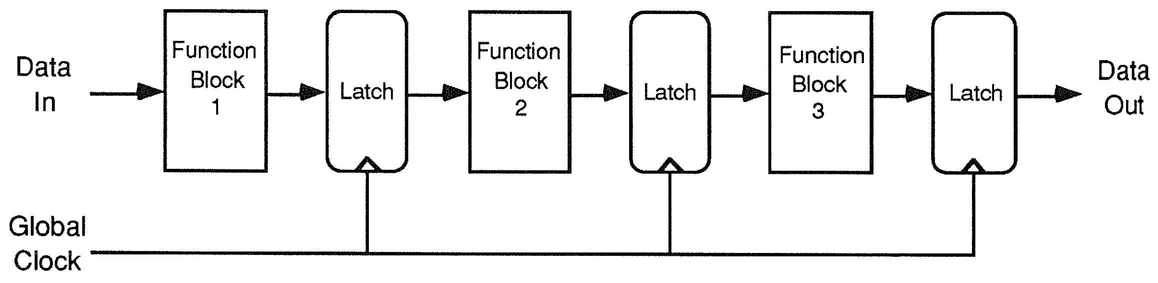


Figure 1.1: Traditional synchronous design partitions logic into stages placed between latches that are clocked with a global signal.

Section 1.1 of this chapter describes the potential advantages of zero-overhead self-timed systems over synchronous systems. The simplest model for data flow in self-timed systems is a linear pipeline, and Section 1.2 presents the self-timed ring structure as a generalization of self-timed pipelines. The top level of the specific self-timed ring structure for implementing division is introduced in Section 1.3. The remaining chapters are outlined in Section 1.4. They present the framework used to analyze self-timed rings, suggest specific methods for increasing performance, and provide a detailed comparison of the performance advantages achieved in the division application.

1.1 Synchronous versus Self-Timed Systems

Synchronous designs separate blocks of combinational logic by inserting explicit latches between them as illustrated in Figure 1.1. These latches and the distribution of a global clock to them show at least five issues for which self-timed systems may have an advantage either in the

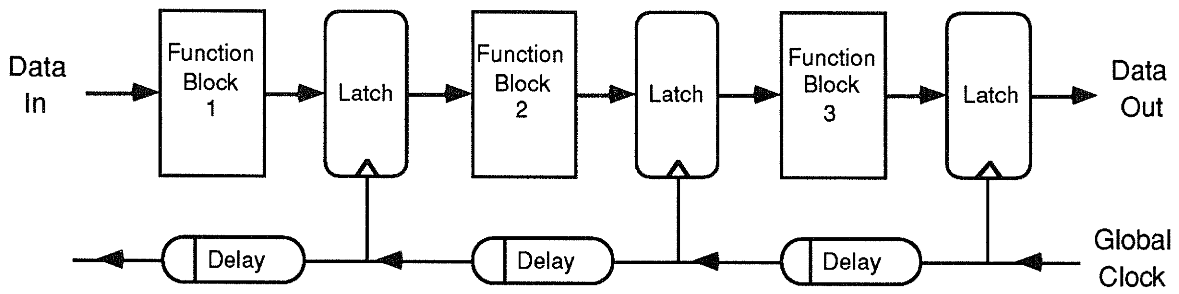


Figure 1.2: Delays in synchronous clock distribution can cause an apparent skew in the arrival time of the clock at individual latches.

performance achieved or in the total system cost. These issues are: 1) clock distribution and the margin added to tolerate clock skew, 2) propagation delay through the latches, 3) mismatched stage delays, 4) maximization of data-dependent delays, and 5) the assumption of environmental worst case timing of components. These issues are discussed in the five subsections below.

1.1.1 Clock Distribution and Skew

As the transistor technology for basic logic blocks improves, higher clock rates are required to take advantage of the higher logic speeds. Modern approaches to logic design [HENN90] may have less than ten gate delays in each clock cycle, and since integrated circuit fabrication technology can now provide basic gates that operate in a fraction of a nanosecond, clocks in the hundreds of MHz may be required to fully utilize the potential of the logic. However, the distribution of such high speed clocks uniformly throughout the levels of a system, circuit-board, and chip requires increased cost and special care because of transmission line effects and large capacitive loading. These effects and wire or driver delays create an apparent skew in the clocks observed by different latches, as

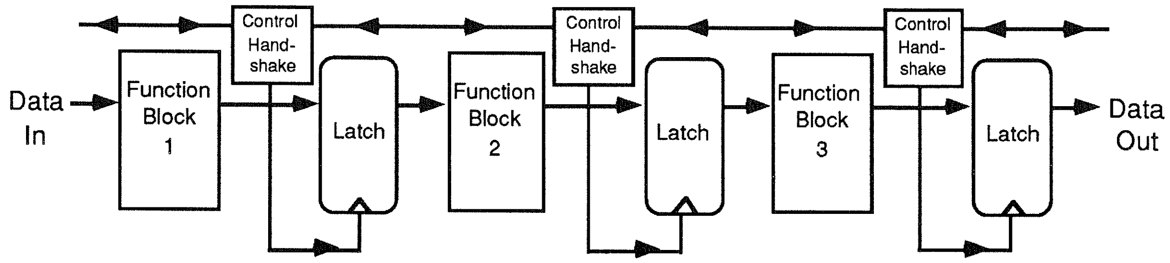


Figure 1.3: Asynchronous design methods can replace clocks with control signals supplied by handshaking blocks, but previous methods have always kept latches in place.

illustrated by the model in Figure 1.2. Clock skew can be somewhat mitigated by efforts to equalize clock loading and path lengths as suggested in [BAKO90], but the remaining skew must be accommodated by lengthening the clock period to provide an additional margin. Self-timed circuits can avoid the clock distribution costs and clock skew margins altogether by using local handshaking to control latches as shown in Figure 1.3. The handshaking allows data validity to be communicated locally rather than globally.

1.1.2 Propagation Delay through Latches

Registers and latches are a source of latency overhead because their setup time and propagation delay add additional delays. The minimum cycle time (clock period) of a synchronous circuit is the sum of the latch delays and the maximum combinational logic delay. An innovation presented in this thesis is a means to remove latch delays completely by forming self-timed pipelines and rings from stage configurations with no explicit latches. Previous self-timed circuit methods using the pattern of Figure 1.3 have concentrated on designing handshaking logic to generate internal latch control signals rather than simply removing the latches.

1.1.3 Mismatched Stage Delays

Another issue that may limit the performance of synchronous systems is the mismatching of the functional sections between the latches. Because the amount of time in a clock period is fixed, it must be set equal to the longest propagation delay of all of the different functional sections in the system. The difference between that maximum time and the actual time used by any functional section is wasted time and can be viewed as a contribution to the overhead of synchronous systems. A self-timed pipeline does not waste this time because it allows data to flow forward in response to data-driven local control, rather than waiting at each latch for resynchronization to the next clock edge. Although the throughput of a pipeline is still limited by its slowest stage, the latency *is* improved because each stage is allowed to progress as soon as data arrives at its inputs.

1.1.4 Maximization of Data-Dependent Delays

The fourth possible cause of a loss of performance in synchronous logic is the need for all timings and critical paths to be based on the worst-case data values. In contrast, self-timed systems communicating with "done signals" based on the actual data values do not need to wait for the worst-case data possibilities. If there is a large variance in delays, synchronous systems may have a performance loss due to the difference between the average and maximum values of delay. The goal in self-timed system design is to minimize the probabilistic expected value of the delay rather than to minimize the maximum delay. Known probability distributions may enable a self-timed system designer to rearrange logic and to size transistors to minimize the expected value of the total delay. This method does not help in a synchronous system because such a system is only concerned with the maximum delays.

1.1.5 Worst-case Environmental Conditions

The fifth possible loss of performance in synchronous circuits is the de-rating used to ensure performance over a range of temperature and voltage levels. Synchronous system designers always use conservative de-rated “worst-case” specifications because the system must work at the expected environmental extremes. But when the actual conditions are not at the extremes, the difference between the actual performance specified by the designer and the possible performance for the actual conditions is wasted performance. Self-timed components always run at their maximum speed for the existing conditions and deliver their outputs as soon as they are actually finished. By providing completion indication, they allow an enclosing system to make use of the output sooner than it can if it is always forced to wait for the worst case. Moreover, self-timed components are robust in the sense that they can continue to function correctly, and indicate their completion, even beyond the ranges of environmental conditions that are foreseen and specified at design time.

1.2 Self-Timed Rings

The general model of logic shown both in Figures 1.1 and 1.3 is a pipeline. If the desired application is a problem that is iterative in nature,

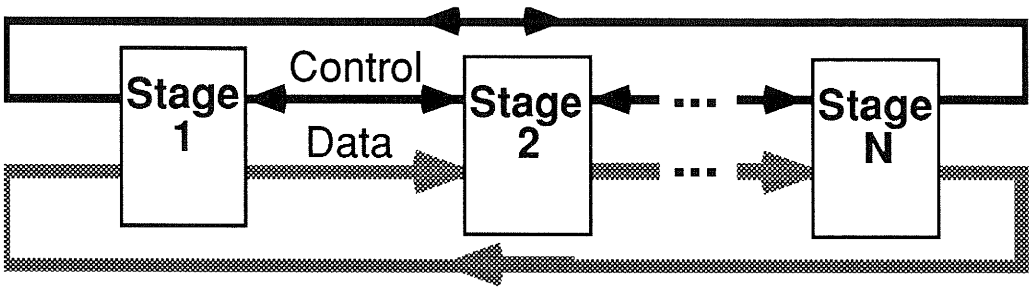


Figure 1.4: A pipeline with its outputs looped back around to the input forms a ring.

then one can turn a pipeline into a ring by looping data from its output back around to its input as shown in Figure 1.4. If the stages in the ring are all self-timed, then the ring will iterate wholly under self-timed control, once it is initialized with input data. Integrated circuit designs can benefit greatly from self-timed rings because the speed of the iterations is independent of any off-chip signals and pad delays.

Handshaking and control logic can potentially limit the performance of self-timed rings, but these problems can be avoided in rings satisfying certain constraints to be discussed in Chapter 4. If these constraints are met, the critical paths determining a ring's performance will go only through data elements. Furthermore, if a self-timed ring meeting these "zero-overhead" constraints is implemented with a stage configuration containing no latches, then the entire computation has the same latency as a combinational array performing the same operations in the computation. Since a combinational array defines the minimal latency by which a particular algorithm can solve a problem, a self-timed ring achieving this same latency is called a "minimal-latency" implementation. A minimal-latency ring effectively "rolls up" a repetitively structured combinational array onto itself. In an ordinary combinational array, only a small portion of the array is actually used at any instant as a data wavefront passes through it. A minimal-latency self-timed ring takes advantage of this fact and provides a structure that allows a traveling wavefront to reuse the logic elements without changing the total delay. By reusing the logic elements, the overall structure occupies only a fraction of the area of the full combinational array implementing the same function.

1.3 Example function: Division

An effective demonstration of self-timed rings requires an appropriate problem choice. Since self-timed rings repeatedly evaluate a series of function blocks, they are particularly well-suited to problems that require iterative computations. For a ring to be completely independent of external handshaking as data progresses around the stages, the computed function cannot require additional data values at each iteration step. The best example of self-timed rings is therefore when they are used to compute iterative functions that are fully specified by their initial input operands.

Elementary arithmetic functions like division, square-root, or trigonometric functions are examples of problems that can be efficiently evaluated with self-timed rings because they require an iterative procedure and only initial operands to determine their results. I chose division as the prime example application in my research because it is a problem amenable to a self-timed ring implementation and one for which computer system designers specifically desire to improve performance. This thesis uses

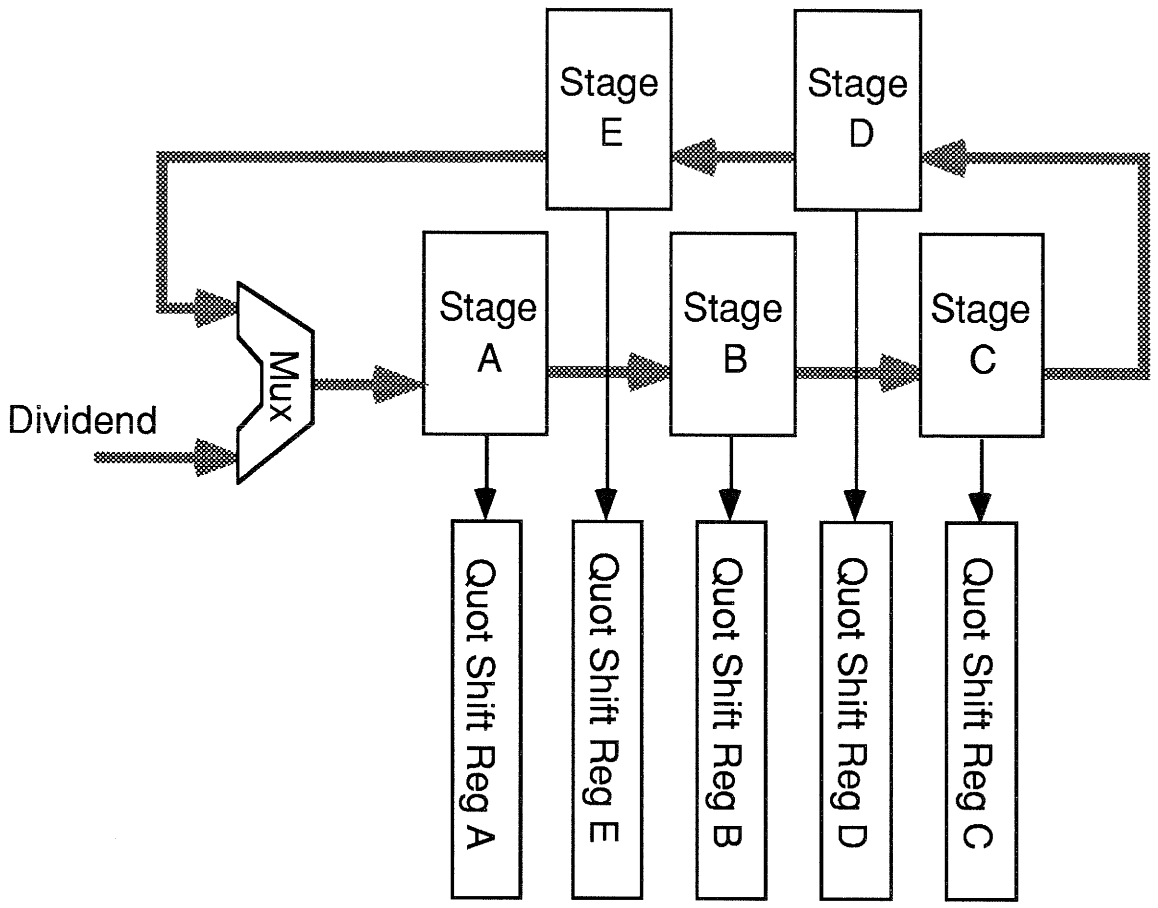


Figure 1.5: Block diagram for the division example using a ring of domino stages that iterates using self-timing.

division both as an example for the general self-timed ring structure and as the object for some specific optimizations.

The division algorithm chosen determines quotient digits sequentially. The basic structure of a self-timed ring for choosing and accumulating quotient digits is shown in Figure 1.5, where each stage determines one quotient digit. After the ring is initialized with the divisor and dividend, a "data token" flows around the ring. The data token is updated at each stage with new values for the partial remainder and the next quotient digit. The individual quotient digits from each stage are captured and stored in shift registers. The bits composing the final quotient are read out in parallel from these quotient shift registers.

1.4 Chapter Organization

This chapter discussed five issues for which self-timed systems can have an advantage over synchronous systems. Even though many previous self-timed design approaches suffered performance degradation from handshaking control overhead, my work has developed self-timed ring structures that can achieve zero-overhead operation.

Chapter 2 provides background information for the construction of self-timed rings by defining the general terms and issues involved in asynchronous pipelines. It defines the different strategies regarding delay assumptions and characterizes the variations in pipeline styles according to these definitions. The chapter names configurations for self-timed pipeline stages and separates them into families. The members of a family have different numbers of latches in each stage. The best latency in self-timed pipelines and rings is achieved by a suggested configuration that has no explicit latches.

Chapter 3 introduces a graph methodology that can be used for the analysis of both self-timed pipelines and rings. The method analyzes the dependencies that determine latency and throughput. This chapter applies the method to the pipeline configurations defined in Chapter 2 and presents tables summarizing and comparing their characteristics.

Chapter 4 extends the performance analysis of self-timed rings and introduces the idea of a zero-overhead ring. A zero-overhead ring constructed with no latches achieves the same total latency as a combinational array, but with much less area. This chapter defines the constraints for attaining zero-overhead operation and discusses their implications for designing minimal-latency rings.

Chapter 5 presents a variety of techniques that are helpful in improving performance and meeting the necessary constraints for zero-overhead operation in real implementations. These techniques are general ones and are applicable to any self-timed ring.

Chapter 6 describes the specific division algorithm chosen as an example application of a self-timed ring. It shows how the techniques of the previous two chapters can be applied to improve the performance of a division implementation. It also presents enhancements that are specific to division, such as an overlapped execution architecture. The chapter compares the measured performance of two fabricated versions of divider implementations in order to quantify the effectiveness of the methods described in this thesis.

Chapter 7 summarizes the analysis results for self-timed rings and reviews the principles for constructing minimal-latency self-timed rings. It suggests other likely application areas of self-timed rings and issues worthy of further research.

Chapter 2

Background: Self-Timed Pipeline Stage Configurations

This chapter defines the circuit styles and components used in the construction of self-timed pipelines and rings and the arrangements in which they may be configured. Pipelines pass a sequence of data tokens through a succession of stages as shown in Figure 2.1. Unlike a synchronous pipeline, which controls all its stages with a single global clock, a self-timed pipeline uses completion detectors along the datapath to generate separate local signals controlling the flow of tokens through its stages. Pipelines are useful where a sequence of data must pass through the same series of functional operations, as, for example, in microprocessors [HENN90] or digital signal processing [MENG88, JACO90]. A ring is a generalization of a pipeline that circulates tokens

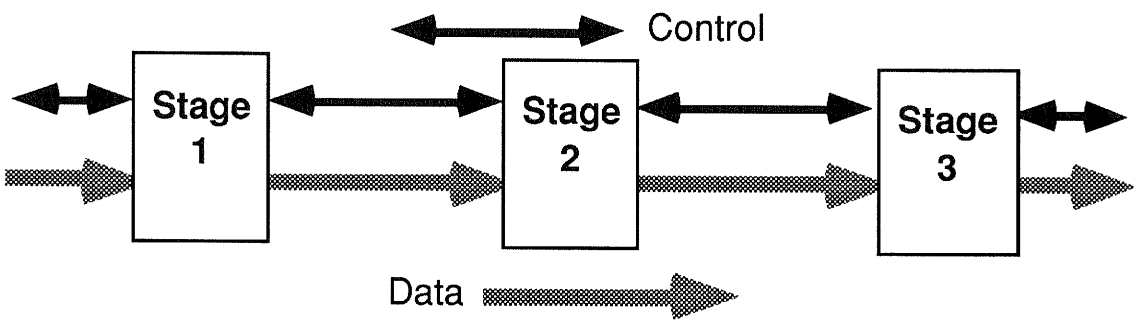


Figure 2.1: Overall structure of a pipeline is a linear sequence of stages.

from its output back to the input, thereby allowing the ring to iterate [GREE87] without needing its environment to externally supply more inputs or control signals.

In both synchronous and asynchronous pipelines each stage comprises a function block with some number of latches. The performance depends on the relative timings and ordering of the components. Previous works, such as [RAO86], analyzed the effects of timing and ordering for the synchronous case with registers. Good re-timing algorithms have been developed for increasing performance in synchronous systems by changing the number and location of registers [LEIS86]. This chapter characterizes the performance of a range of possible configurations for *asynchronous* pipelines with varying styles of function blocks and latches, varying numbers of latches per stage, and varying orders of connections to the completion detectors. Unlike the work in [MENG89], which synthesizes particular control arrangements under the assumption that function block evaluation dominates all other delays, the work reported in this thesis analyzes various configurations in terms of the delay parameters of each component.

The sections in this chapter consider the various options for hardware stages that can be composed into self-timed pipelines and rings. First, Section 2.1 describes the terminology used to describe classes of required delay calculations and assumptions that provide circuits with different degrees of insensitivity to wire and component delays. This section identifies the classes I chose for my research as those having more restricted circuit structures, which permit them to operate with fewer delay assumptions. These classes require a datapath signaling convention, described in Section 2.2, providing completion information along with the data. Since this convention requires special function blocks and latches, Section 2.3 gives a transistor-level description of the possible styles. These blocks can be arranged in several configurations to form self-timed pipelines, and Section 2.4 gives names to the configurations. Though this section introduces the configurations and their tradeoffs qualitatively, a quantitative performance comparison is saved for Chapter 3, which refers to the configurations by name. While linear dataflows define the basic pipeline stage configurations, more complex dataflows also can use these stage configurations with the addition of split and merge stages, which are discussed in Section 2.5. Finally, Section 2.6 summarizes the pipeline configurations of this chapter, in preparation for the analysis of their performance in Chapter 3.

2.1 Classes of Circuits and Delay Assumptions

Some digital logic design approaches require careful tuning of delay elements for the circuits to function correctly; whereas, other approaches can design circuits that operate correctly for any delays in gates or wires. Indeed, there is a progression of design approaches that successively lessen the dependence of correct logical operation on delay calculations or assumptions by placing additional restrictions on the styles of gates and their allowed connectivity.

Early asynchronous circuit designs were the least constrained in the allowable gate styles and connectivities but required the most careful analysis of path delays in order to achieve a desired switching sequence [UNGE69]. This approach was difficult, and terms like "controlled race" acquired a bad reputation. Although some self-timed styles still involve crafting matched delays [CHAP91], most modern asynchronous designs generally minimize hazards by requiring only a small number of local delay comparisons [SUTH89].

Synchronous design techniques achieved more reliability by restricting the allowable circuit connections between combinational logic and storage elements, and distinguishing between data signals and a clock signal. While a designer must still use information about delays in order to calculate the arrival time of data relative to the clock at every latch or register, the key feature of synchronous designs is that they do not require specific relative timings among the data signals. The restrictions of the synchronous design approach allow a designer to assure correct logical operation if the delays through all paths in the combinational logic between registers are individually less than a chosen fixed clock period.

More restrictive design approaches can be used to further reduce the number of delay calculations required to assure correct operation. The class of circuits that operates correctly for all gate delays but still might require delay calculations concerning interconnecting wires that branch is called "speed-independent." The issue of wire delay is only significant for wires that branch because a delay on a non-branching wire can always be considered equivalent to a gate with an increased output delay followed by an instantaneous wire. Local exceptions to an otherwise speed-independent circuit can be made when a particular wire is assumed to have equal branch delays, in which case that wire is called an "isochronic-fork" [MART86].

The most restrictive design approach [UDDI86, EBER88], which does not even require the assumption of isochronic forks, constructs circuits that work correctly for any arbitrary delays of gates or on wires. This class of circuits is called "delay-insensitive" because delays have no effect on logical operation.[1] Delay-insensitive circuits are the most conservative style because they require no delay assumptions or calculations by the designer.

---

[1] Strictly speaking, the class of circuits that are delay-insensitive down to the transistor level is minuscule [MART89], and therefore the term is usually only applied down to the level of some "primitive." I consider single stacks of transistors to be isochronic primitives, and hence never consider delays between the *n*-channel and *p*-channel transistors in the same stack. Considering arbitrary delays within single stacks (for example, on the wire connecting the gates of the two transistors in a CMOS inverter) would nearly preclude the ability to do digital analysis.

Since wires on any integrated circuit deliver nearly instantaneous transmissions, speed-independent circuits are almost equivalent to delay-insensitive circuits for most real integrated circuits. In larger domains, like printed-circuit boards or backplanes, the delays of wires may be substantial and these two classes need to be distinguished.

As is the philosophy in RISC processor design [HENN90], self-timed designs seek to design control circuits that are always correct but that are fast for common conditions. In general, the circuits discussed in this thesis assure correct operation by using circuits in the speed-independent and delay-insensitive classes. The philosophy for using these classes is that by constructing a circuit that is logically correct for any delay values, the resultant design is more robust and the designer need not acquire all the information and specifications affecting the actual delays. However, to the extent that delay information is available, the designer can use it to optimize performance by choosing between design alternatives, sizing transistors, and making local exceptions to a purely speed-independent design approach. Design recommendations and constraints can therefore be expressed in terms of nominal delay values even though the circuits continue to produce correct results for other actual delay values.

2.2 Data and Reset Signalling Conventions

Building a speed-independent circuit dictates certain restrictions in the method used to transmit data. This section describes the dual-rail signalling convention that speed-independent circuits use to send sequences of data tokens on a set of wires.

A synchronous circuit uses a clock signal to distinguish when the values on a set of wires should separately be considered valid tokens, but that approach requires assuming bounds on the skew between the data and the clock. Likewise, even asynchronous circuits that use a bundled data-valid wire alongside a data bus also require a delay assumption between the actual data wires and the signal indicating their validity. For a circuit to be truly speed-independent, the validity and separation of data tokens must be encoded within the data itself rather than bundled alongside it. An "embedded-completion" encoding transmits information about when each data token has completed on the *same* wires used for transmitting the data value itself.

The encoding of sequencing information within data can be accomplished with either transition or level encodings. Transition encoding, known as "two-phase" signalling [SUTH89], can transmit one bit of a data token on a pair of wires by flipping the logical level of one of the wires based on the value of the bit, where the presence of a transition on either wire denotes a new data token. Level encoding is ordinarily known as "four-phase" signalling on a "dual-rail" [SEIT80] or "dual-monotonic" pair. It can use the protocol in Table 2.1 to transmit each bit of a data token on two wires. The wires are initially both low, and either wire going high denotes a new data token bit, whose value is determined by the choice of which wire went high. A four-phase encoding must then return both wires to the initial reset condition in which both wires are low before

Wire A^T	Wire A^F	Signal A
0	0	Reset = Not Ready
0	1	Evaluated FALSE
1	0	Evaluated TRUE
1	1	Not used = Never occurs

Table 2.1: A dual-rail monotonic pair uses a simple encoding to convey both the value and completion-indication for a binary bit A on two wires, A^T and A^F.

transmitting a new data token. A two-phase encoding, on the other hand, can transmit another token without an intervening reset condition. Hybrid schemes that are two-phase and yet still use level encodings are also possible with the addition of the concept of function blocks maintaining an even state and an odd state [DEAN91]. In general, four-phase signalling can be implemented with function blocks that are simpler and faster than the function blocks required for two-phase signalling.

The pipeline styles in this chapter all use the normal four-phase level encoding shown in Table 2.1 for single bits. Usually, a separate dual-monotonic pair carries each bit of a wider bus, even though higher order group-encodings are possible with more complex function blocks [WILL87]. Another possible extension of monotonic set encodings for transmitting a value with n states, that is useful for small n, is a simple 1-of-n unary encoding on n wires.

Four-phase signalling has a natural symmetry between data and reset values. Events generally come in pairs because every data valid transition must correspond to some reset transition, which returns wires and circuit elements to their original states. Self-timed circuit elements that check for one transition should just as well verify the other transition in order to ensure correct operation. The easiest way to enforce this requirement is to consider every data token as composed of a separate data element and a "reset spacer," and to explicitly check for the progression of both parts. The reset spacers keep data tokens separate, and prevent bits in one token from racing ahead or lagging behind to corrupt an adjoining token.

Embedded completion signalling has the advantage that it allows the individual bits of a bus to begin evaluation individually as soon as their inputs have arrived without having to wait for all of the other bits in the bus. A specific advantage of dual-rail embedded completion signalling is that logic blocks can obtain either polarity of an input signal by using the appropriate wire of the pair; hence, logical signal inversions are free. A disadvantage of embedded completion signalling is the greater number of wires necessary, but in a structured custom layout the increase in the number of wires does not necessarily double the total cell areas because wires can be routed over transistors. Embedded completion signalling may also require more logic, but this depends on the specific logic function being implemented. Even if there is an area penalty, it does not necessarily imply a speed penalty because the individual wires are generated and driven in parallel.

2.3 Function Block and Latch Styles

Speed-independent circuits require function blocks that indicate the completion of evaluation in their output wires. The last section showed that dual-monotonic pairs transmit both value and completion information on the same wires. If therefore a function block generates its outputs as dual-monotonic pairs, then it is sufficient for use in a speed-independent circuit. This section shows several different ways of building appropriate function blocks and latches.

2.3.1 Function Block Styles

Figure 2.2 contrasts four styles of function blocks, all taking dual-monotonic input pairs and generating dual-monotonic pairs as outputs. The example circuit shown for all the styles implements a simple AND/NAND gate. I call these four styles "static logic," "direct logic," "semi-controlled precharge logic," and "full-controlled precharge logic."

The static logic and direct logic styles do not require a precharge control input. Both evaluate their outputs when their inputs get valid data, and reset their outputs when their inputs are reset. Static logic can change an output wire as soon as any input wire changes, but it does not enforce that all the signal transitions accompanying the input change actually arrive. Since arbitrary delays inserted on some of its input wires could cause the gate to malfunction by merging a stale delayed input transition with some new input, static logic cannot be delay-insensitive. However, if the input wires also branch to a separate completion detector that verifies the completion of their transitions, then a circuit using static logic can still attain speed-independence.

The direct logic style verifies all signal transitions have arrived on its input pairs before changing its outputs. It waits for all its inputs to reset before resetting its outputs, and it checks for one high input wire in all

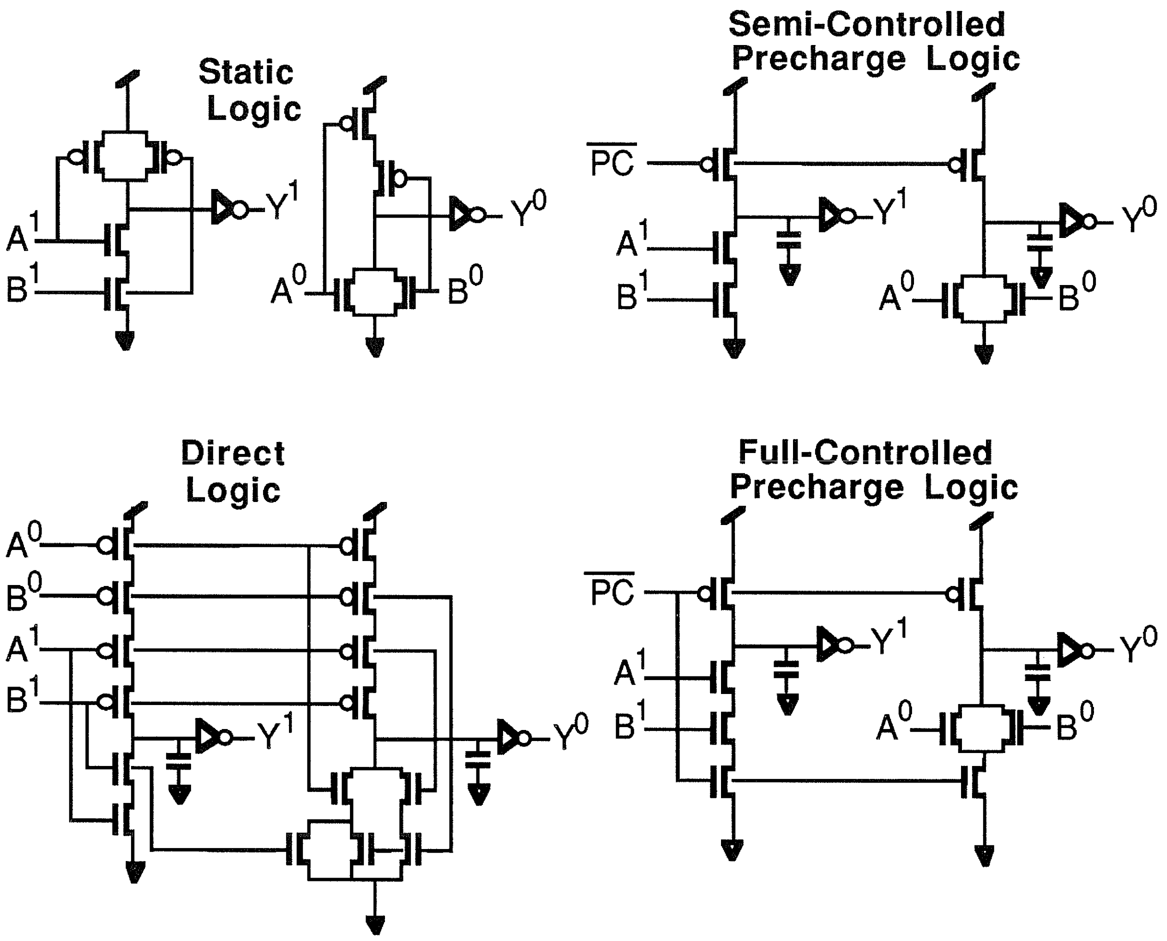


Figure 2.2: Four possible AND/NAND Function block styles to generate dual-monotonic outputs: Static Logic, Direct Logic, Semi-controlled Precharge Logic, and Full-Controlled Precharge Logic.

input pairs before evaluating an output pair. Configurations using direct logic function blocks can be delay-insensitive because even arbitrary delays on the input wires of a gate do not introduce any stale data values. Unfortunately, this verification makes the gates slower because of the taller transistor stacks in direct logic. Since in most real integrated circuits speed-independence is safe enough, static logic blocks can substitute for direct logic blocks in cases where its inputs are also checked by forks to completion detectors used in the control logic.

Both semi-controlled and full-controlled precharge logic styles take a precharge control input. The precharge input allows every block receiving it to avoid replicating the same transistor stack in its pull-up tree. The logic that generates the precharge signal effectively amortizes the cost of determining the status of the inputs for a group of blocks. The precharge input to a block is also the logical inverse of an enable signal because each block must have its precharge removed before the outputs can transition to an evaluated state. The only difference between the semi-controlled and full-controlled precharge styles is the presence of the bottom transistor in the full-controlled style, which prevents fighting if precharge is ever active concurrently with valid data inputs.

Fighting can occur, for example, in the later gates of a precharged stage internally composed of several serial semi-precharged gates whose precharge controls are tied together. Because the later gates in the chain do not have their data inputs reset until earlier gates finish resetting, semi-controlled precharge logic ripples reset data serially through the gates instead of allowing the gates to reset in parallel as they do in full-controlled precharge logic. Ratioing of the transistor stacks in semi-controlled precharge logic can achieve parallel resetting of internal domino chains, but at the expense of large precharge transistors if the n-channel stack is short. The advantages of semi-controlled precharge logic are that it has fewer transistors, faster evaluation transitions due to shorter pull-down stacks, and lower loading of the precharge control input. However, if semi-controlled precharge logic is not ratioed to have a well-defined output when both valid data and an active precharge are present, then the circuit must have control logic assuring that this condition never occurs.

Both semi-controlled and full-controlled precharge logic are at best speed-independent and not delay-insensitive. Full-controlled precharge logic or ratioed semi-controlled precharge logic are not delay-insensitive because the precharge input is overriding, and therefore does not allow verification of the arrival of reset edges on input wires. Non-ratioed semi-controlled precharge logic is not delay-insensitive because arbitrary delays on the precharge input or data inputs can cause incorrect operation when a transistor stack fights and outputs an intermediate voltage level.

Both controlled precharged function block styles have a significant advantage not possessed by direct or static logic: after the input data returns to the reset condition, but before the control asserts a precharge signal, the block will "hold" valid outputs and this feature can implicitly provide the function of a latch without additional transistors. In contrast, the next subsection describes latches that may be added explicitly in order to use direct or static logic in a pipeline at all, or to increase the number of latches in a stage.

2.3.2 Explicit Latch Styles

Latches in a pipeline are the elements in a datapath that control the flow of tokens in response to control signals. Latches keep tokens from incorrectly merging together, allowing a pipeline to contain multiple tokens. A designer can construct an explicit latch for each bit of a pipeline

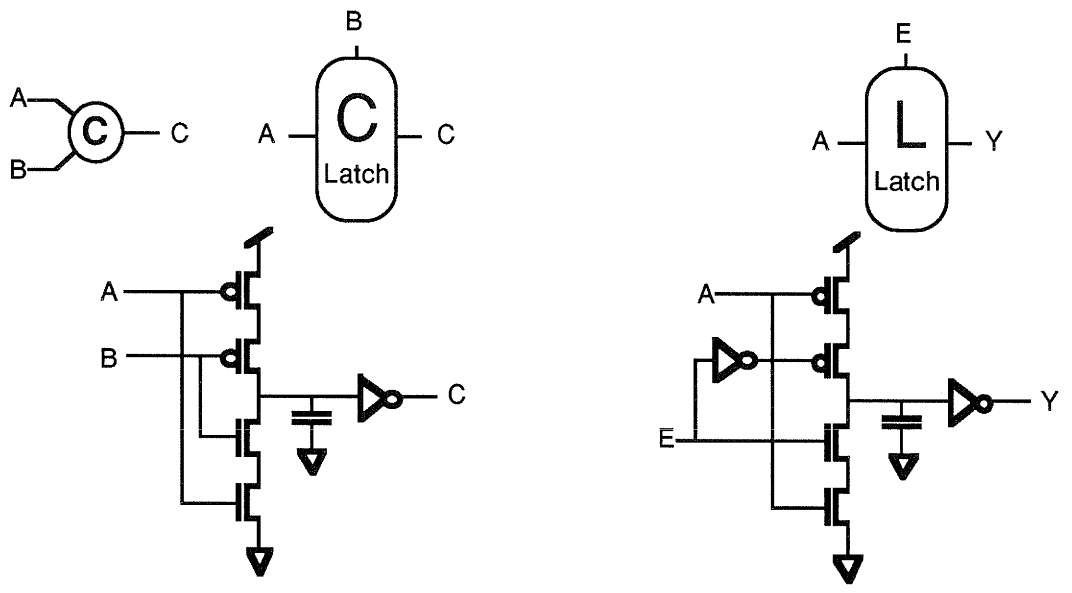


Figure 2.3: CMOS implementations of a C-element (or C-latch) and a standard flow latch.

either as a traditional flow-latch or by using a Muller C-element [SEIT80] to make a “C-latch” [GREE88] as illustrated in Figure 2.3. The flow-latch passes the value of its data signal, A, to its output, Y, when its enable signal, E, is high, and keeps the output unchanged when the enable is low. A C-element is a gate whose output is the state of the inputs when they were last the same. A C-element is like a flow-latch, but its control is more symmetric. If one input is viewed as data and the other as a control signal, a C-element passes a high data signal to its output when the control is high, passes a low data signal when the control is low, and otherwise leaves the output unchanged. The C-latch can therefore be used in a delay-insensitive circuit because it verifies that it has received new values on both data and control before changing its output. Conversely, the ordinary flow-latch is not delay-insensitive because there is no way to detect whether the latch has indeed changed to the “holding” condition when the enable is low.

Another disadvantage of using a flow-latch is that a self-timed circuit will need to use additional C-elements to generate the appropriate control signal for the flow-latch, whereas the simpler symmetric control of the C-latch allows it to be controlled directly from available signals, as will be shown in the next section. Hence, the C-latch style is the preferred latch style in my research.

2.4 Self-Timed Pipeline Stage Configurations

The function blocks and latches discussed in the previous section can be grouped together in stages to form self-timed pipelines. This section examines the possible configurations of components in each stage. I define a stage as a function block followed by zero or more explicit latches. Each stage has its own separate inputs for precharge or latch control.[2] Between the stages, a datapath conveys information on a unidirectional bus from each stage to the next, and control wires may traverse in both directions.

Pipelines process a sequence of tokens, composed of alternating data and reset elements. At any instant, the stages not occupied by data or reset elements can be said to contain a hole or "bubble." Like holes in a semiconductor, the bubbles flow backward as the data and reset spacers flow forward. The consumption of output tokens pushes new bubbles into the pipeline from the output end. Throughout the pipeline, data and reset

[2] A concatenation of function blocks with a *common* reset signal and no latches is a domino chain. For the purposes of this thesis, such a chain is referred to as if it were just one large function block because its characteristics as a block are independent of its internal partitioning.

elements can only move forward into a stage that has a bubble. Hence, in an asynchronous pipeline each token element moves independently when it can exchange position with a bubble. This is in contrast to a synchronous pipeline where all tokens move at the same time in lock-step.

As in the case of synchronous pipelines, asynchronous pipelines may tradeoff between latency and throughput. Since latches provide additional sites for tokens or bubbles, the presence of additional latches can increase the throughput. But latches increase the latency delays incurred by tokens as they flow through a pipeline. While Chapter 3 presents more precise definitions of these quantities and an analysis of these effects, this section describes them informally as it defines the configurations that contain additional latches.

Though pipelines could be composed of stages with different configurations, this thesis considers and analyzes pipelines composed of a series of stages having the same configuration. The name of each configuration type is a two-letter abbreviation designating the style of the function block and control logic, followed by a number indicating the number of additional explicit latches in each stage. The set of configurations all having the same two-letter abbreviation is called a family, and each specific configuration type is called a member of its family.

The next three subsections catalog all of the configurations mentioned in this thesis. The presentation starts with the precharged function block families because the other families can be obtained as transformations from them. The first subsection describes the two families, **PC** and **PS**, that use precharged function blocks. Subsection 2.4.2 presents the two families, **CF** and **FC**, that use combinational (direct or static) function blocks. Subsection 2.4.3 presents the **PL** family, which alters the latch style to pairs of latches forming an edge-triggered register. The **PL** family is discussed mainly for the purpose of comparison because it uses the control structure suggested in [MENG89].

2.4.1 Configurations using Precharged Function Blocks

The first configuration family to be considered is the speed-independent family **PC**, in which each stage has a precharged function block controlled by a C-element. A single stage of the **PC0** configuration, which has no explicit latches, is shown at the top of Figure 2.4. Below it appears a pipeline segment constructed from three of the stages. An instance "index" subscripts the components in each stage of the pipeline. The function block is either a full-controlled or a semi-controlled precharged block. The C-element in each stage merges the "go" or "request" signal from the preceding stage with the "done" or "acknowledge" signal from the following stage. The completion detector, labelled **D**, has an inverting bubble on its output, which means that it goes low when the bus has valid evaluated data and goes high when the bus is reset. A full completion detector uses a tree of C-elements to combine the outputs of a NOR gate on each dual-monotonic pair in the bus, but Section 5.4 will suggest simpler completion detectors.

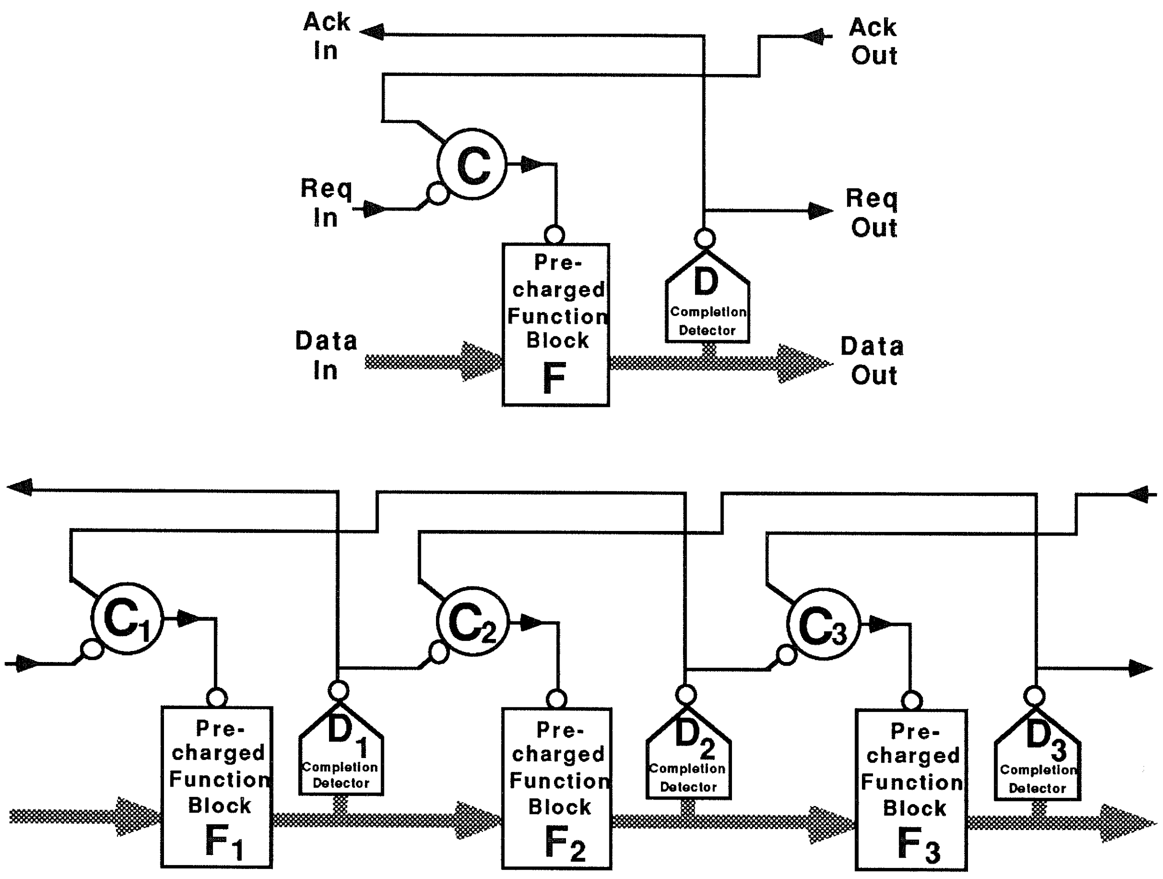


Figure 2.4: Schematic for stage configuration **PC0** and a short pipeline composed of that configuration.

The control in this configuration ensures that it functions correctly as a pipeline for processing a stream of tokens, composed of data elements separated by reset "spacers," and for keeping the tokens separated. The operation of the pipeline can best be understood by tracing the control wire connections through the segment example in Figure 2.4. In stage 2, the precharged function block is enabled for evaluation when its inputs have new valid data from stage 1 and when stage 3 is finished resetting. If stage 3 were not reset before stage 2 evaluated, the data token coming through stage 2 could corrupt the data token still remaining in stage 3.

Likewise, the precharged function block in stage 2 is reset when its inputs from stage 1 are reset and when stage 3 is finished evaluating. The symmetric pattern of the control sequencing serves to verify the completion of every rising and falling transition before the control removes the stimulus causing the transition. This configuration is therefore speed-independent for arbitrary component delays. In particular, the **PC** pipeline configuration family operates correctly even if the function blocks in different pipeline stages perform different functions or have different delays.

Even though the **PC** configurations are speed-independent, they are not fully delay-insensitive because the precharge input is overriding and each gate does not itself verify that its inputs have reset before resetting its outputs. Arbitrary delays inserted after the branch of an input wire to a preceding completion detector could produce incorrect operation. However, since such arbitrary wire delays do not usually occur in practice, speed-independence is sufficient and it is usually not important that the **PC** configuration is not completely delay-insensitive.

The sequencing of the precharge control logic has the feature of using each precharged function block both as a computational element and as a latch. This is possible because the dual-rail signalling convention allows the precharged function block to hold its outputs while waiting for a precharge signal or new active inputs. Even if its inputs reset, an evaluated stage, not yet precharged again, can hold valid data outputs to allow the next stage to have stable inputs during its evaluation. And even if some data inputs go active, a stage that has been reset can hold its outputs reset

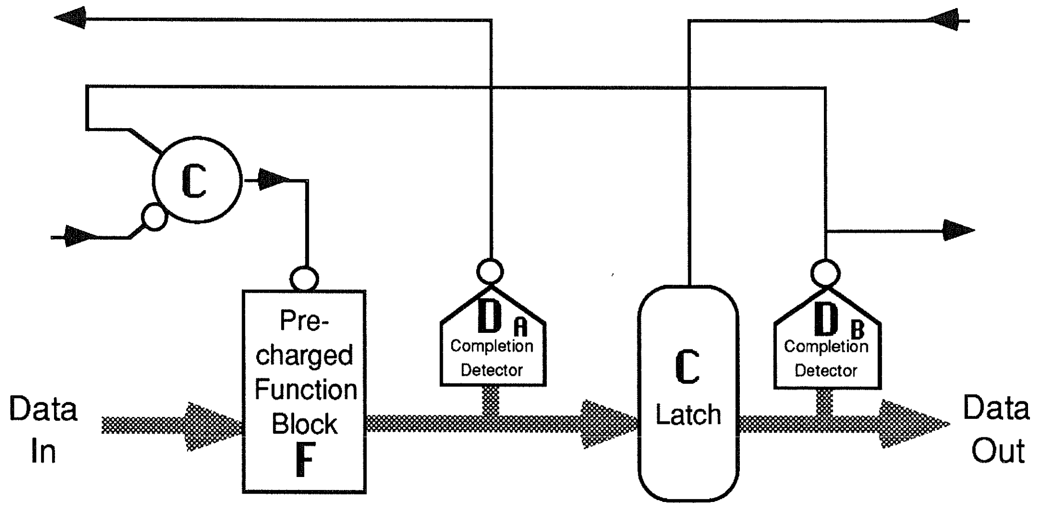


Figure 2.5: Schematic for stage configuration **PC1**.

until a sufficient number of inputs go active for the stage to make a correct evaluation. Thus, one can use a precharged function block as a latch "for free," with no additional transistors, merely by structuring the control as described in the previous paragraph to precharge each block only *after* its successor consumes the outputs. The **PC0** configuration, in particular, functions correctly as a pipeline without any explicit latches.

Adding latches to each stage can increase pipeline throughput by reducing the cycle time. Configuration **PC1** inserts one latch between the precharged function blocks as shown in the stage schematic in Figure 2.5. Each stage also has a second completion detector to detect the status at the output of the C-latch and supply an input to the control C-element. Configuration **PC2** contains a second latch and completion detector between the precharged function blocks as illustrated in Figure 2.6. Adding the explicit latches in configurations **PC1** and **PC2** and connecting them with the same control structure can also be viewed as adding dummy functional units that hold data instead of processing it. If the latches are

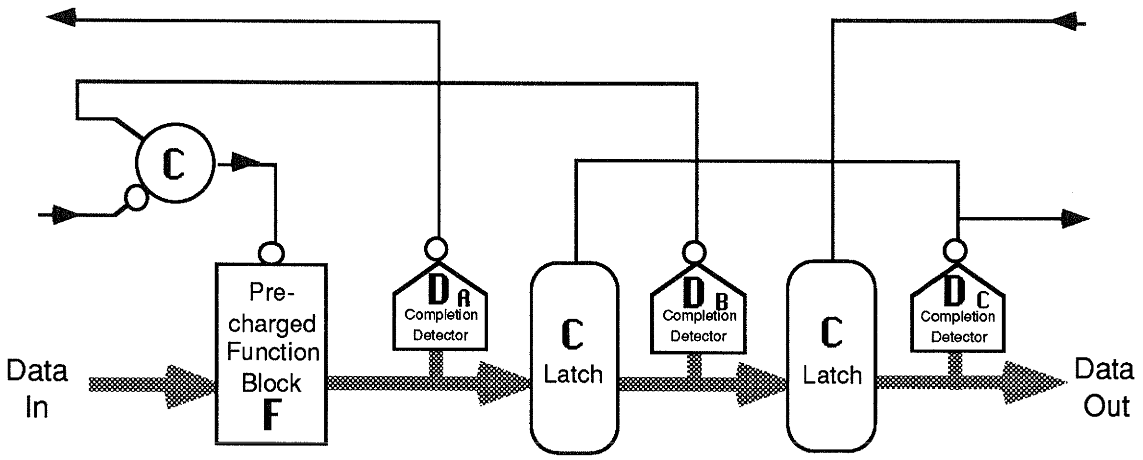


Figure 2.6: Schematic for stage configuration **PC2**.

small in area compared to the real function blocks, then this can be advantageous because it increases the utilization of the function blocks.

A problem with the **PC** configuration family is that the C-element in the control is in the critical path of forward flowing data tokens. Since the C-element transitions to remove the precharge signal only after it has received valid data inputs, the delay of the C-element adds to the delay of the function block evaluation. But if it can be assumed that stages reset faster than they evaluate, then another stage style is possible that removes the delay of the C-element in the control of the **PC** family. The modified family is called **PS** and it is formed by removing the C-elements in the control and connecting each precharge input directly to the completion detector following each block's successor. Figure 2.7 shows the simplest stage in this family, **PS0**, which has no explicit latches.

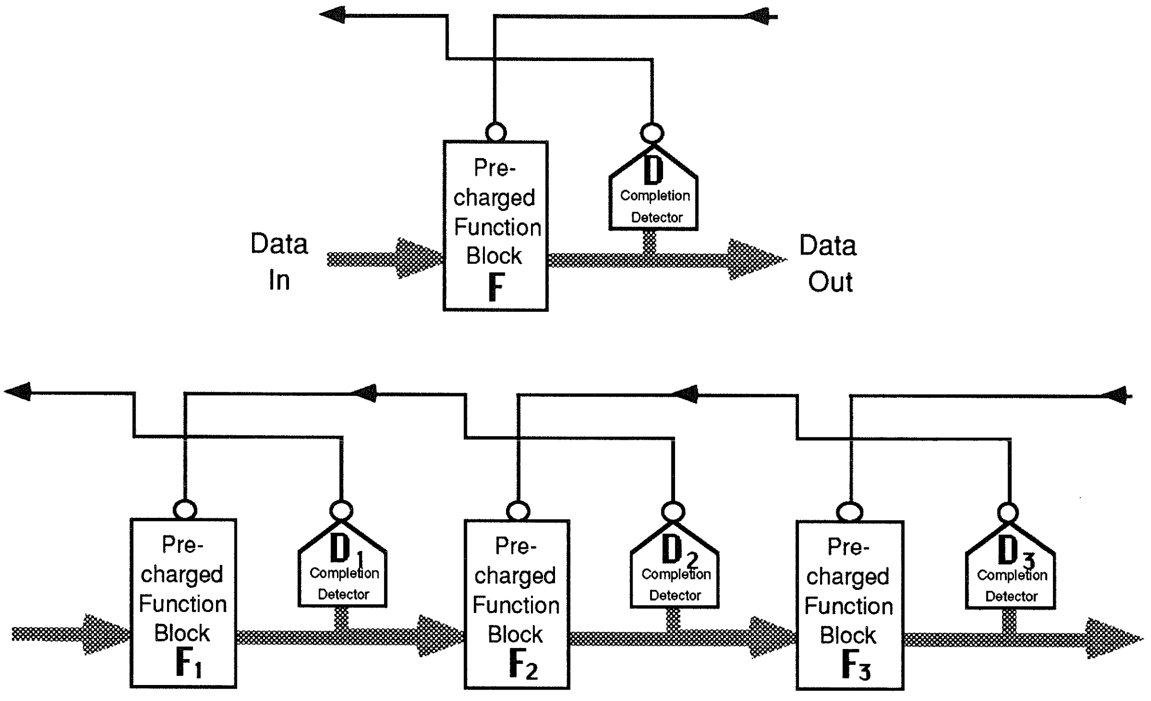

Pre-charged Function Block F
D Completion Detector
Data In
Data Out
Pre-charged Function Block F_1
D_1 Completion Detector
Pre-charged Function Block F_2
D_2 Completion Detector
Pre-charged Function Block F_3
D_3 Completion Detector

Figure 2.7: Schematic for stage configuration **PS0** and a short pipeline composed of that configuration.

The transformation from **PC** to **PS** stages is justified by showing, with the aid of the assumption about resetting, that the same operations occur in **PS** as occur for both the rising and falling transitions of the C-element in **PC**. The rising transition of the C-element is not important when dual-rail signalling is used for the datapath because this convention on the data inputs prevents each stage from evaluating until its inputs become valid. Under the assumption that each stage's predecessor resets no slower than the stage's successor evaluates, the falling transition of the C-element is always triggered by the falling of the signal from the successor's completion detector. Since the C-element is redundant on both transitions, the C-element can be removed and replaced with a wire from

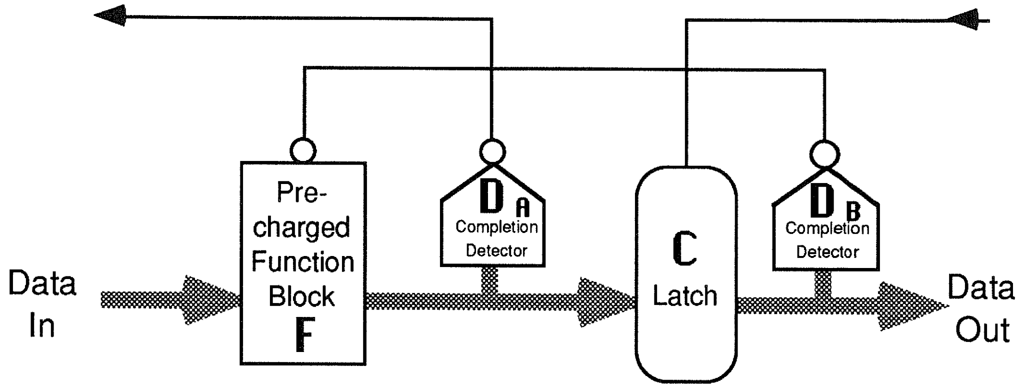


Figure 2.8: Schematic for stage configuration **PS1**.

the input that is assumed to fall last. As thus constructed, **PS** stages act equivalently to **PC** stages when it can be assumed that function blocks reset no slower than they evaluate.

The **PS** family, in which a forward flowing data token is not delayed by a C-element in the control logic, is both faster and smaller than the **PC** family. Pipelines composed of the **PS0** stage configuration are particularly simple since they directly concatenate precharged function blocks and have no C-elements at all. Additional latches can increase the throughput by allowing more token sites. Figure 2.8 shows configuration **PS1**, which adds one latch between the precharged function blocks, along with an additional completion detector. Likewise, configurations **PS2** and **PS3** contain, respectively, two and three latches between the precharged function blocks, with a completion detector after each latch.

Though the **PS** and **PC** families are generally similar, there is one other instructive difference between them. Since the C-element in the control of stages in the **PC** family does not enable evaluation until after valid data arrives, a **PC** stage using full-controlled precharged logic can accept ordinary single-rail data as well as embedded-completion dual-rail data. This feature was used in [MENG88] to halve the number of wires going through the latches, but the size of the function blocks remained unchanged because they still generated dual-rail signals internally in order to form the completion signals needed for the control logic. In contrast, this modification for using single-rail data inputs cannot be applied to the **PS** family because their stages require dual-rail signalling in order to allow the function blocks to be enabled for evaluation before valid data arrives.

2.4.2 Configurations using Combinational Function Blocks

Section 2.3 described both precharged and combinational function block styles that all generate outputs on dual-monotonic pairs. The precharged function block styles were used in the stage configurations of the previous subsection; this subsection defines the stage configurations using the combinational (static and direct) function block styles. The configurations can be derived by considering two transformations of the **PS** family that replace its precharged function block with a combinational function block and a C-latch. The transformation in which the C-latch precedes the function block is called the **CF** family, and the transformation in which the function block precedes the C-latch is called the **FC** family. The two transformations have similar properties of delay insensitivity, but differ significantly in performance.

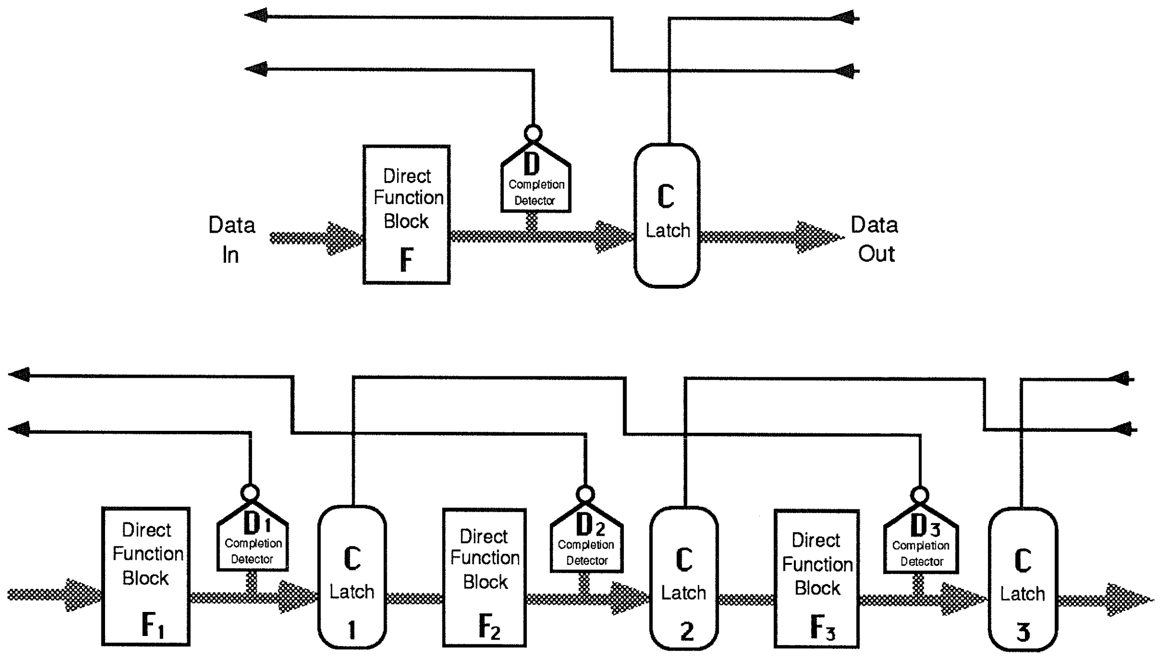


Figure 2.9: Schematic for stage configuration **CF0** and a short pipeline composed using that configuration.

The simplest member of the **CF** family is shown in Figure 2.9 (to be consistent with the other stage schematics, the figure is drawn with the function block to the left of the latch). Since, in order for a sequence of stages to act as a pipeline at all, each stage must have at least one latch, the configuration with just the one required latch is called **CF0** because it has no *added* latches. The operation of this configuration can be explained from the control wire connections of the pipeline segment example in Figure 2.9. Latch 1 resets when the data element of a token it held has passed down the pipe to function block 3 and no longer needs to be stored. The data element is no longer needed at latch 1 because its presence at the outputs of function block 3 verifies that it has been stored in latch 2. Observe that waiting only for the data element to pass through function block 2 would not verify that the token got stored into latch 2. Thus, the

CF0 configuration requires that each latch reset only after the token it holds has passed through the *second* function block following the latch. Because the combinational function blocks and C-elements are symmetric for rising and falling transitions, the control connections also handle the reset element portion of each token correctly: each latch is enabled to accept new valid data only after the second succeeding function block has reset.

As in the case of pipelines using precharged function blocks, additional latches can increase the throughput. Configuration **CF1** imposes one additional latch between the function blocks, for a total of two latches. Configurations **CF2** and **CF3** contain respectively three and four latches between the function blocks, with a completion detector before each latch.

The **FC** family is the transformation of the **PS** family that replaces each precharged function block by a combinational function block *followed* by a C-latch. An equivalent method of obtaining the same **FC** constructions is to transform the **CF** family by moving the latches and their control connections to the other side of the function blocks. Either transformation constructs configurations where each combinational function block is preceded by a completion detector and followed by a latch. As in the re-timing transformations of synchronous circuits [LEIS86], the transformations can change the performance without changing the total number of latches.

Figure 2.10 shows both a single **FC0** stage and a short pipeline composed using that stage configuration. Configurations **FC1**, **FC2**, and

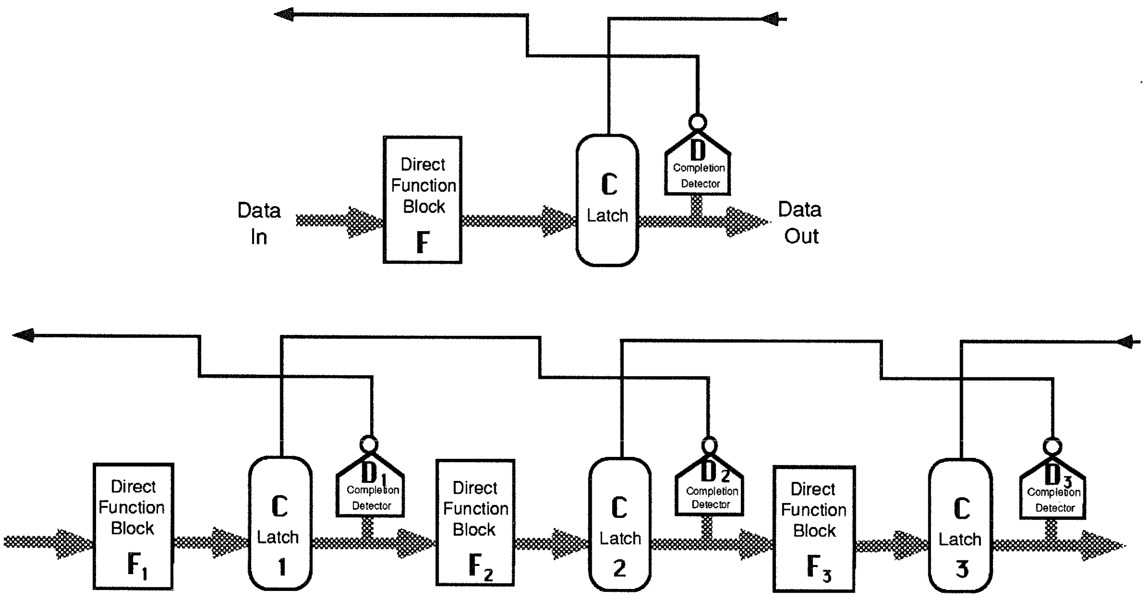


Figure 2.10: Schematic for stage configuration **FC0** and a short pipeline composed using that configuration.

FC3 contain respectively two, three, and four latches between the function blocks, with a completion detector after each latch. Their logical operation is similar to that of the **CF** configurations, but their performance is improved. The improvement occurs because each latch can reset after verification that the data element it holds has passed through only one function block following the latch, instead of two function blocks. Chapter 3 will show this improvement quantitatively in terms of the individual block delays.

The **CF** and **FC** pipeline configurations require different combinational logic block styles for different classes of delay assumptions. Since a direct function block, as defined in Section 2.3, checks within each gate to make sure all its input transitions arrive before changing any of its outputs, both the **CF** and **FC** families are delay-insensitive when a direct function block is used as the combinational block style. In contrast, the properties of the **CF** family are different from those of the **FC** family when a static function block is used as the combinational block style. A static function block may change an output prior to the arrival of *all* its input transitions. If some input wire changes particularly slowly and completion detectors do not verify its transition, then it may not finish changing before the removal of the stimulus. This could cause non-digital voltage values or leave a "stale" data value, which would erroneously merge with the next data token. Since the **FC** family has completion detectors that verify the transitions on the inputs to the function blocks, **FC** configurations using static function blocks are still speed-independent. However, the **CF** configurations, which have no completion detector between a function block and the preceding latch, do not verify the transitions on a static function block's inputs, and are hence not speed-independent if using static function blocks. In order to avoid this complication when discussing performance, I specify both configuration families with direct function blocks.

2.4.3 Configurations using Edge-Triggered Registers

The **PC** configuration family can also be transformed into a structure more similar to a synchronous circuit by replacing the latch in each stage with a register. Since registers do not actively reset their outputs, an asynchronous pipeline using completion detectors must rely on the function block for resetting. Since only the full-controlled precharged logic block style can reset its outputs with data still applied to its inputs, it

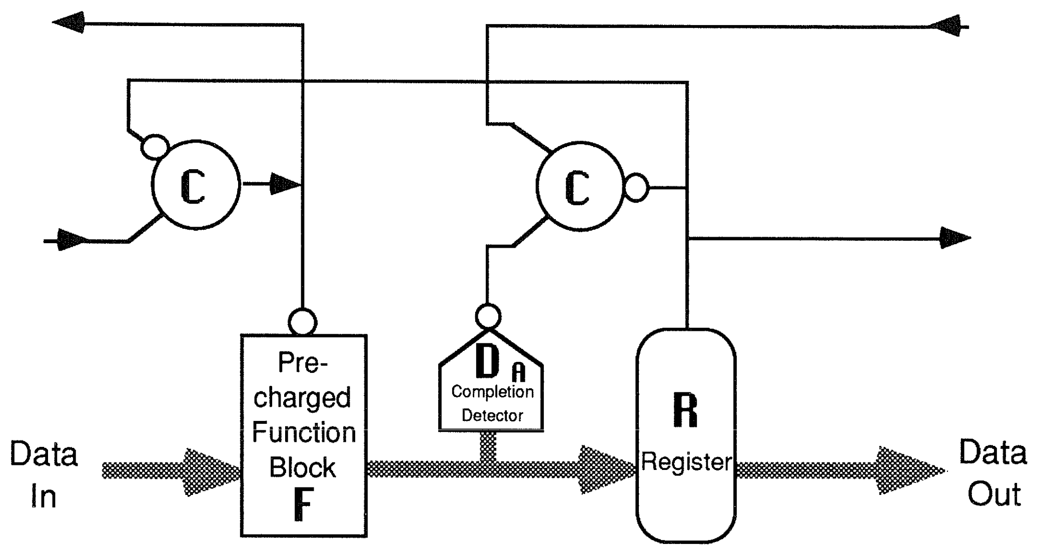


Figure 2.11: Schematic for stage configuration **PL1**.

is the only function block style that can be used in asynchronous pipelines with registers.

Replacing the C-latch in configuration **PC1** with a register controlled by a C-element yields the **PL1** configuration illustrated in Figure 2.11. This configuration has the control structure suggested in [MENG89] for a "full-handshake." Because the register is positive edge-triggered, its outputs do not explicitly change when its control signal goes low, and so there is no benefit from a completion detector on the register's output. Instead, timing assumptions need to be made concerning the register's delay. For example, when the register's control input rises, the delay before the register passes new valid data to its outputs must be less than the delay through the C-element enabling the evaluation of the following function block. If the register is too slow, then the function block might begin evaluating on the register's old data value. Thus, the **PL** family is not speed-independent like the **PC** family. However, since synchronous circuit designers have characterized register delays well, and the required assumption may be justifiable, the **PL** family is an alternative that offers good throughput with only one completion detector per stage. Unfortunately, like the **PC** family, the presence of the C-element in the critical path that enables function block evaluation increases the latency of the **PL** family relative to that of the **PS** family.

2.5 Extensions to more Complex Datapaths

The previous section cataloged the control variations for linear dataflows. For datapaths that branch, simple extensions to each configuration can generate the appropriate control circuits. The merging of datapaths coming together at a single stage requires sending the acknowledge from that stage to all its predecessors. Splitting the datapath output from a stage requires the addition of a C-element to collect the acknowledge signals from all successors of the stage. Figure 2.12 shows an example of such arrangements for the **PS0** configuration. Performance of dataflows that split or merge must be analyzed on a case-by-case basis.

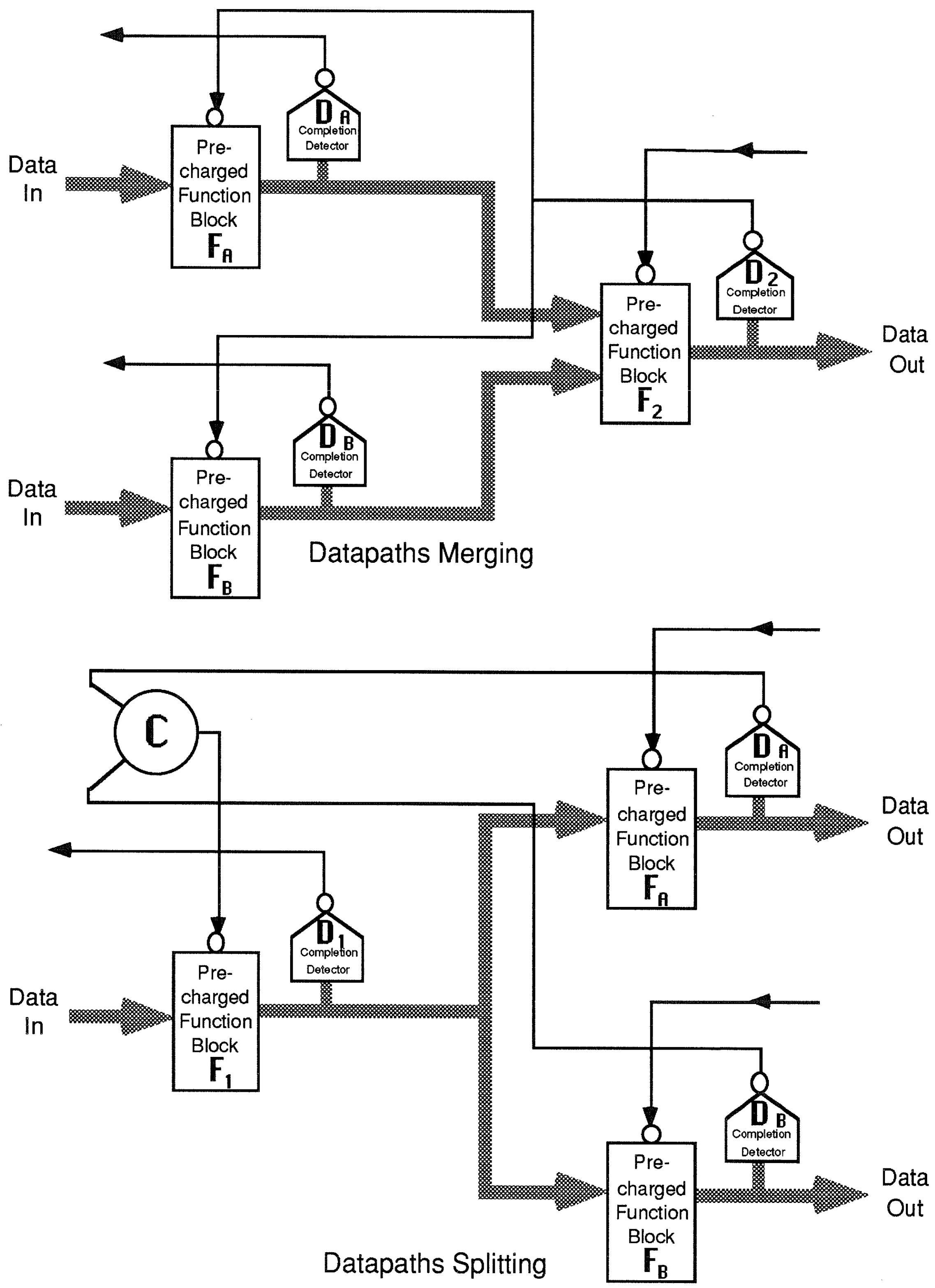


Figure 2.12: Simple extensions to the basic pipeline configurations can provide the appropriate control for merging and splitting datapaths. This figure shows extensions to the **PS0** configuration.

2.6 Summary

Self-timed pipelines and rings can be constructed from a range of possible stage configurations. Dual-rail signalling on monotonic wire pairs allows embedding completion information within the data itself and avoids having to rely on matched delays. The configurations vary in the style of function blocks used and the actual delay assumptions they require. Pipelines using full-controlled or semi-controlled precharged logic can provide speed-independent operation with the appropriate control structures. Ordinary static logic generating dual-rail outputs also can be speed-independent with one of the suggested configurations. Complete delay-insensitivity, which allows even arbitrary delays on different branches of a wire, can be attained with direct style function blocks.

This chapter cataloged families of configurations for the stages of self-timed pipelines. Latches added to the base configuration of each family form the members that allow the improvement of pipeline throughput at the expense of increased latency. The next chapter evaluates the performance tradeoffs of these pipeline configurations in detail.

Chapter 3

Analysis Methodology and Results

The previous chapter defined several different stage configurations with some qualitative statements about their latency and throughput tradeoffs. This chapter presents a framework to compare the performance of the stage configurations quantitatively when used in either asynchronous pipelines or rings. First, Section 3.1 defines the latency and throughput of an asynchronous pipeline more formally, and introduces the key parameters necessary to evaluate performance. These parameters can be expressed in terms of the delay variables of individual components. Section 3.2 defines these variables and then presents the Dependency Graph analysis method used to determine the latency and local cycle time of a stage configuration in terms of those delay variables. The method is applied in Section 3.3 to evaluate the performance of different pipeline stage configurations. This section gives two tables summarizing the results of applying the analysis method to all of the configurations introduced in Chapter 2. The first table gives the coefficients of the component delay variables in the equations expressing the latency and cycle times. The second table shows what these coefficients mean for extremes in the possible relative values of the component delays. The latency and throughput characteristics of the different pipeline stage configurations are summarized in Section 3.4 for self-timed pipeline design. The next chapter delves further into the implications of the same performance measures for self-timed rings.

3.1 Definition of Local Performance Parameters

Determining the performance of an asynchronous pipeline can be more complex than determining the performance of a synchronous pipeline. In an asynchronous pipeline, control signals govern token flow with local handshaking. Each four-phase token is composed of a data element and a reset spacer. At any instant, the stages not occupied by data elements or reset spacers can be described as containing a "hole" or "bubble." Control logic only allows an element to flow forward when the stage it will occupy is empty. When an element does flow forward, it leaves behind an empty slot. Thus, bubbles flow backward as they displace forward-flowing data elements and reset spacers. The performance can be limited by the supply of tokens, the supply of bubbles, or the local control handshaking. In a pipeline, the input supplies data tokens and the output supplies bubbles as a result of acknowledgement handshakes with the environment. Because the performance can be limited by several different effects, an asynchronous pipeline requires more variables to describe its performance than a synchronous pipeline does. This section defines the variables that characterize the local properties of pipeline stages.

In a synchronous pipeline, the delay from one stage to the next stage is simply equal to the clock period; in an asynchronous pipeline, the delay is an independent quantity called the per-stage latency. The forward latency, L_f, specifies the delay from new valid data outputs at one stage to new valid data outputs at the following stage. The reverse latency, L_r, specifies the delay from the acknowledgement of a stage's output to the acknowledgement of its predecessor's output. The reverse latency is usually greater than the forward latency in circuits using embedded-completion signalling because such circuits use some control signal transitions only for passing bubbles and have fewer required transitions in the critical path of forward flowing data. (The opposite is often true for circuits using delay-matching [SUTH89] instead of embedded-completion signalling because delay-matching can avoid some transitions in the bubble acknowledgement path, thereby lowering the reverse latency.) The forward latency, L_f, can be measured or analyzed independently by observing a data token flowing forward through an initially *empty* pipeline. Likewise, the reverse latency, L_r, can be measured or analyzed independently by observing the delays bubbles experience when flowing through a pipeline initially *packed* with data. Because any packing will consist of alternating data and reset elements, I define L_r to be the average delay of a bubble displacing a data element and displacing a reset spacer. Defining just the average is sufficient because bubbles must always displace equal numbers of data elements and reset spacers.

The minimum local cycle time, P, for an asynchronous pipeline is analogous to the minimum clock *P*eriod for a synchronous pipeline. Just as in the synchronous case, the slowest stage limits the minimum value for P. The *T*hroughput, T, is the reciprocal of P, and is the rate at which the input and output must deliver and consume tokens, respectively, to keep a pipeline flowing at its maximum capacity.

Since the different stage configurations in the preceding chapter have different numbers of latches, we need one more local variable to account for the different "capacity" of the stages for storing tokens. Let the variable S be the spread in stages between tokens packed statically in a pipeline. Since S is concerned with a static situation, it is dependent only on the connectivity of components and is independent of their delays. Because each token contains both a data element and a reset spacer, and each of these occupies the storage of one latch, S is always equal to $\frac{2}{H+1}$ where H is the number of extra latches in each stage, i.e., the number that follows the two-letter code designating each configuration.

3.2 Dependency Graph Construction from Component Delay Variables

In order to determine the latencies and cycle time of a pipeline built out of a particular configuration of components in each stage, it is necessary to analyze the dependencies of the required sequences of transitions. These dependencies can be drawn in a marked directed graph [COMM71, MURA77] in which the nodes of the graph correspond to specific rising or falling transitions of circuit components, and the edges depict the dependencies of each transition on the outputs of other components. A value attached to each node in the graph specifies the delay of the corresponding transition. I call these graphs "Dependency Graphs" because of the similarly named graphs used in analyzing the cycle time of synchronous systems [RAO86]. This section illustrates two types of Dependency Graphs. The first type is a "Flat" Dependency Graph, which is more intuitive and therefore used for qualitative descriptions of the salient aspects. Flat Dependency Graphs are functionally equivalent to the Signal Transition Graphs of [CHU86]. The second type of Dependency Graph is new and I call it the "Folded" Dependency Graph. This type is specifically for pipelines or rings with similar stages and provides a more compact representation suitable for easier quantitative analysis and more precise definitions.

The nodes in either type of Dependency Graph represent the delays of particular transitions. The delay of each transition is specified by a lower-case t, which denotes propagation time, subscripted with a capital letter abbreviating the block type as follows:

F	Function blocks
D	completion Detectors
C	C-elements, latches, or registers

If an up-arrow ($\uparrow$) or down-arrow ($\downarrow$) follows the block type, then the term refers specifically to the delay of the rising or falling transition.

When written with no arrow, the delay term refers to both transition directions. A delay term of a component specifies the propagation time from the last input to change until the output of the component changes. Each delay variable also includes the delay of driving a chain of buffers on a component's output if it is significantly loaded.

3.2.1 Flat Dependency Graphs

One constructs Dependency Graphs by inspecting the schematic for both the rising and falling transitions of each component and drawing each dependency. For a component like a C-element, which has symmetric dependencies for rising and falling transitions, the corresponding portion of the Dependency Graph is likewise symmetric. The node for each transition is annotated with an index indicating the particular pipeline stage that contains the component. As an example, a segment of the Flat Dependency Graph for the **PC0** pipeline configuration, which was defined in the previous chapter in Figure 2.4, is shown in Figure 3.1. Only a segment large enough to show the repeating pattern of the graph is drawn. The Dependency Graph is a simplification of the more general timed Petri-net description of asynchronous components [RAMC74]. Since the pipelines under consideration are deterministic, the Petri-net is decision-free and can therefore be fully represented by a Dependency Graph.

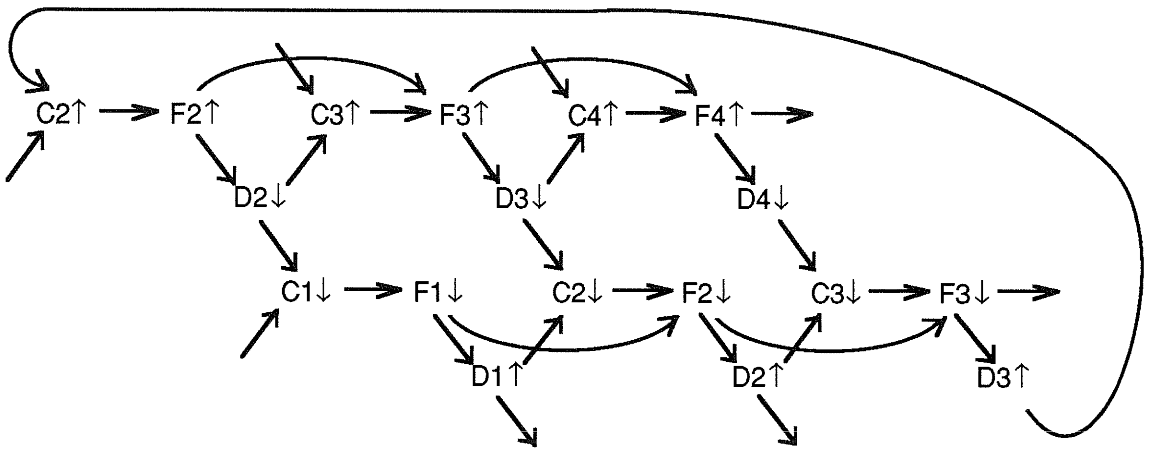


Figure 3.1: A portion of the Flat Dependency Graph for the **PC0** configuration pipeline.

Dependency Graphs can be used to determine both the per-stage latencies and the local cycle time. Simple *acyclic* paths in the flat graphs determine the latencies. The forward latency is the sum of delays along the longest path from some transition to the same transition in the stage with the next higher index. The reverse latency is one-half the longest sum of delays from some transition to the same transition in the stage with index two less. It is necessary to define the reverse latency in terms of two stages to account for bubbles displacing both data elements and reset spacers. More precise definitions of the latencies will be stated in the next subsection.

The local cycle time is determined by *cyclic* paths in the Dependency Graph. These cycles occur because a pipeline processes successive data tokens and the components in each stage go through a series of transitions. The transitions eventually return a stage to the same "state," where the state is defined by the output values of each component. Because the graph is "timed" by the delays at each node, cycles do not indicate undesirable feedback situations as they would in synchronous logic.[3] Rather, the sum of the node delay values around a cycle is a lower bound on the period required for the components to go through a sequence of transitions that return the stage to the same state.

Each transition in a Dependency Graph can fire only when all of its predecessors have executed their specified transitions, and cannot fire again until all of its predecessors have fired again. Since every cycle through the corresponding node in the dependency graph is a lower bound on the time before that transition can fire again, the actual minimum cycle time for a transition is given by the cycle with the longest sum of delay values. The correct speed-independent construction of each stage guarantees that all of the components in a stage will cycle at the same rate since every transition is part of some cycle in the Dependency Graph that verifies the transition's completion. Thus, the longest simple cycle in the complete Dependency Graph gives the minimum cycle time of the pipeline as a whole. These results were proved in [RAMA80] for decision-free Petri-nets, and more recently in [BURN91] specifically for analyzing self-timed circuits with a Flat Dependency Graph. In the present thesis, the method is applied specifically to pipelines and rings, where the cycle time is the time required for a stage to process each token.

[3] Only if the delay sum around a cycle were zero would there be a problematic "dependency loop" akin to the problem of a loop with no registers in the case of synchronous logic.

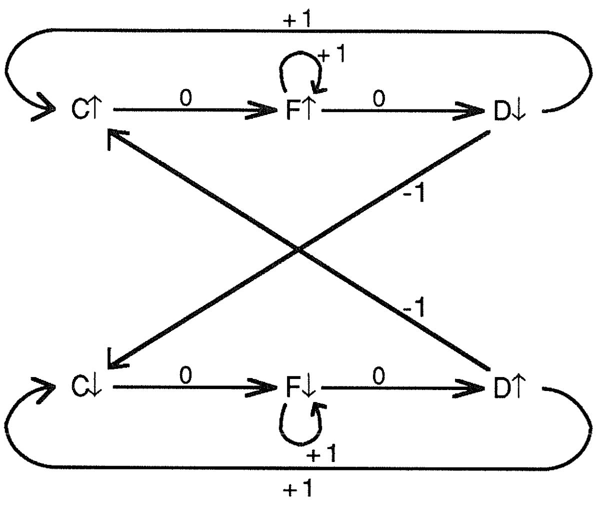


Figure 3.2: The Dependency Graph for the **PC0** configuration pipeline can be folded together to make a Folded Dependency Graph.

3.2.2 Folded Dependency Graphs

For a specific pipeline stage configuration, the structure of the Dependency Graph repeats after each stage, and this property can be used to make the representation more compact. When the stages are identical and thus the delay values also repeat, the Dependency Graph can be folded together. Figure 3.2 shows an example of a Folded Dependency Graph for the **PC0** pipeline configuration. As before, the nodes in the Folded Dependency Graph represent the transition delays, but it is not necessary to subscript them with a particular stage index since the node represents that same transition in all stages. Instead, each dependency *edge* in the Folded Dependency Graph is annotated with an integer label giving the offset in stage indices to which the dependency refers. Dependencies between components in the same stage thus have an offset label of zero.

The Folded Dependency Graph can be used to more precisely define the stage latencies and the local cycle time. These quantities can be written in terms of equations where t_i are the node transition delays and w_i are the stage index offset labels:

$$L_f = \underset{\substack{\text{non-repeating cyclic paths} \\ \text{with } 0 < \sum\limits_{i \in \text{path}} w_i}}{\text{maximum over all}} \left[\frac{\sum\limits_{i \in \text{path}} t_i}{\sum\limits_{i \in \text{path}} w_i} \right] \qquad (3.1)$$

$$L_r = \underset{\substack{\text{non-repeating cyclic paths} \\ \text{with } 0 > \sum\limits_{i \in \text{path}} w_i}}{\text{maximum over all}} \left[\frac{\sum\limits_{i \in \text{path}} t_i}{-\sum\limits_{i \in \text{path}} w_i} \right] \qquad (3.2)$$

$$P = \underset{\substack{\text{non-repeating cyclic paths} \\ \text{with } 0 = \sum\limits_{i \in \text{path}} w_i}}{\text{maximum over all}} \left[\sum_{i \in \text{path}} t_i \right] \qquad (3.3)$$

In all of equations (3.1)-(3.3), the set of paths considered in the Folded Dependency Graph corresponds to the set of paths enumerated to define the same quantities in the corresponding Flat Dependency Graph. Even though the latency paths were acyclic in the Flat Dependency Graph, they become cyclic in equations (3.1)-(3.2) when the graph is folded. The maximum of delays is taken specifically over those paths that do not repeat (a repeating path is one that passes through the same node more than once with the same accumulated stage index offset sum) because this path set directly corresponds to the examination of just the simple paths in the Flat Dependency Graph. Repeating paths do not need to be considered since they correspond in equations (3.1)-(3.2) to paths containing cycles in the Flat Dependency Graph, and in equation (3.3) to non-simple cyclic paths in the Flat Dependency Graph.

It should be emphasized that if the stages in a pipeline or ring are not identical, then the complete Flat Dependency Graph needs to be drawn because the delay terms from the different stages need to be distinguished. The longest cycles in the Flat Dependency Graph will be the ones through the slowest stages, and they will hence correctly determine the limiting cycle time for the entire pipeline.

For a pipeline or ring that has identical stages, the Folded Dependency Graph gives the same information as a Flat Dependency Graph but in a more compact form that also makes symmetry more apparent graphically. Either graph can be used for the analysis of a pipeline with identical stages. I wrote a computer program that examined paths in Folded Dependency Graphs to find the latencies and local cycle times according to equations (3.1)-(3.3).

3.3 Results of Analysis of the Configurations

Using the program implementing the analysis method, I determined equations for the latencies and cycle times of all the configurations in the previous chapter. This section gives details only for the application of the method to the **PC0** configuration; [WILL90] lists equations for the other configurations. In order to simplify the equations, this section defines a set of "standard assumptions" and uses them to construct two tables that summarize the equations for all the configurations. The first table gives the coefficients of the equations as simplified by the standard assumptions, and the second table shows examples of the possible ranges of actual latency and throughput expressed by the equations.

Equations (3.1)-(3.3) are easily applied to the particular example for configuration **PC0** drawn in Figure 3.2. Equation (3.1) gives an equation for the forward latency,

$$L_f = \max \left[t_{F\uparrow}+t_{D\downarrow}+t_{C\uparrow}\,,\, t_{F\downarrow}+t_{D\uparrow}+t_{C\downarrow}\right], \qquad (3.4)$$

and equation (3.2) gives the reverse latency,

$$L_r = \frac{1}{2}\left[t_{F\uparrow}+t_{D\downarrow}+t_{C\downarrow}+t_{F\downarrow}+t_{D\uparrow}+t_{C\uparrow}\right]. \qquad (3.5)$$

There are several possibilities for the cycles enumerated by equation (3.3) used to determine the local cycle time. Any cycle with zero offset sum must be the concatenation of the sequence $F_\uparrow.D_\downarrow.C_\downarrow.F_\downarrow.D_\uparrow.C_\uparrow$, which has offset sum -2, with two more trips through adjoining loops that have offset sum +1. Since the longest cycles need to be found, the self-cycles on the $F_\uparrow$ and $F_\downarrow$ nodes can be ignored because they are always shorter than the $F_\uparrow.D_\downarrow.C_\uparrow$ and $F_\downarrow.D_\uparrow.C_\downarrow$ cycles. Therefore the following cycles are the possibilities for the longest zero offset sum cycles:

$$F_\uparrow.D_\downarrow.C_\downarrow.F_\downarrow.D_\uparrow.C_\uparrow.F_\uparrow.D_\downarrow.C_\uparrow.F_\uparrow.D_\downarrow.C_\uparrow$$

$$F_\uparrow.D_\downarrow.C_\downarrow.F_\downarrow.D_\uparrow.C_\downarrow.F_\downarrow.D_\uparrow.C_\uparrow.F_\uparrow.D_\downarrow.C_\uparrow$$

$$F_\uparrow.D_\downarrow.C_\downarrow.F_\downarrow.D_\uparrow.C_\downarrow.F_\downarrow.D_\uparrow.C_\downarrow.F_\downarrow.D_\uparrow.C_\uparrow$$

and the equation for the cycle time is

$$P = t_{F\uparrow}+t_{D\downarrow}+t_{C\downarrow}+t_{F\downarrow}+t_{D\uparrow}+t_{C\uparrow} + 2 \max\left[t_{F\uparrow}+t_{D\downarrow}+t_{C\uparrow},\ t_{F\downarrow}+t_{D\uparrow}+t_{C\downarrow}\right] \quad (3.6)$$

Since the equations for the latencies and cycle times contain many terms, they are easier to compare when simplified by some reasonable assumptions. The following relationships are defined as the standard assumptions:

$t_D = t_{D\uparrow} = t_{D\downarrow}$ (Completion detector delays are equal and symmetric) (3.7)

$t_C = t_{C\uparrow} = t_{C\downarrow}$ (C-element delays are equal and symmetric) (3.8)

$t_{F\downarrow} <= t_{F\uparrow}$ (Function resetting is no worse than evaluation) (3.9)

$t_C <= t_{F\downarrow}$ (C-element delays are no worse than function resetting) (3.10)

Equations simplified by the standard assumptions give latency and cycle time in terms of four variables: $t_{F\uparrow}$, $t_{F\downarrow}$, t_C, and t_D. For example, the standard assumptions reduce the **PC0** equations (3.4)-(3.6) to:

$$L_f = t_{F\uparrow} + t_D + t_C \quad (3.11)$$

$$L_r = t_D + t_C + \frac{1}{2}(t_{F\uparrow} + t_{F\downarrow}) \quad (3.12)$$

$$P = 3t_{F\uparrow} + t_{F\downarrow} + 4t_D + 4t_C \quad (3.13)$$

3.3.1 Table of Characteristic Coefficients

To compare the different configurations, Table 3.1 summarizes the equations for the cycle time and per-stage latencies by showing the coefficients of the four variables: $t_{F\uparrow}$, $t_{F\downarrow}$, t_C, and t_D. For example, the

Config	Class	Cycle Time Coefficients, P				Forward Latency Coefficients, L_f			Reverse Latency Coefficients, L_r			
		$t_{F\uparrow}$	$t_{F\downarrow}$	t_C	t_D	$t_{F\uparrow}$	$t_{C\uparrow}$	$t_{D\downarrow}$	$t_{F\uparrow}$	$t_{F\downarrow}$	t_C	t_D
PC0	SI	3	1	4	4	1	1	1	0.5	0.5	1	1
PC1	SI	2	0	4	4	1	2	1	0.5	0.5	2	2
PC2	SI	1	1	4	4	1	3	1	0.5	0.5	3	3
PC3	SI	1	1	4	4	1	4	1	0.5	0.5	4	4
PS0		3	1	0	2	1	0	0	0.5	0.5	0	1
PS1		2	0	2	2	1	1	0	0.5	0.5	1	2
PS2		1	1	2	2	1	2	0	0.5	0.5	2	3
PS3		1	1	2	2	1	3	0	0.5	0.5	3	4
CF0	DI	3	1	4	2	1	1	0	0.5	0.5	1	1
CF1	DI	2	0	4	2	1	2	0	0.5	0.5	2	2
CF2	DI	1	1	4	2	1	3	0	0.5	0.5	3	3
CF3	DI	1	1	4	2	1	4	0	0.5	0.5	4	4
FC0	DI	2	0	4	2	1	1	0	0	0	1	1
FC1	DI	1	1	4	2	1	2	0	0	0	2	2
FC2	DI	1	1	4	2	1	3	0	0	0	3	3
FC3	DI	1	1	4	2	1	4	0	0	0	4	4
PL1		1	1	4	2	1	2	1	0	0	2	0
PL2		1	1	4	2	1	3	1	0	0	3	0
PL3		1	1	4	2	1	4	1	0	0	4	0

Table 3.1: Coefficients of equations for pipeline stage cycle time and latency. The class column specifies those families that are speed-independent and delay-insensitive.

top line of the table labeled **PC0** shows the coefficients in equations (3.11)-(3.13).

Comparing the terms in Table 3.1 for the different stage configurations provides many interesting results. One observation is that the forward latency of the **PC** configuration family is clearly worse than the forward latency of the other families because of its completion detector in the critical path determining the forward latency. This dependency causes the function block in a **PC** stage not to get enabled until after valid input data has already arrived. In the other families, which require data signals with embedded completion, the control enables the function block earlier, in anticipation of the data. The **PC** family does, however, have one advantage over the **PS** family: the **PC** family is speed-independent and works correctly for any composition of individual stage reset or evaluation delays, but the **PS** family requires assuming that neighboring stages reset no slower than each stage itself evaluates. On the other hand, since pipelines composed of a sequence of similar stages can reasonably make this assumption, the complete speed-independence of the **PC** family is usually not an important requirement and **PS** is preferable to **PC** because of its lower latency.

Another observation from Table 3.1 is the tradeoff between latency and cycle time, which is exhibited by all the families and is dependent on the number of latches. Adding explicit latches to the "**0**" member in each family decreases the cycle time to an extent. However, different families' cycle times "saturate" with differing numbers of latches. The **PC**, **PS**, and **CF** families saturate with two extra latches; the **FC** family saturates with only one extra latch. This means that there is no improvement in choosing **FC2** instead of **FC1**, but there may be an improvement in using **PC2**, **PS2**, or **CF2** instead of **PC1**, **PS1**, or **CF1**, respectively. Actually, whether there is any difference between these latter sets is dependent on the relative sizes of $t_{F\uparrow}$ and $t_{F\downarrow}$. If the reset time is about equal to the evaluation time of the function blocks, then adding a second latch does not help, but if $t_{F\downarrow} << t_{F\uparrow}$ then adding a second latch helps significantly. This kind of dependence is a specific property of asynchronous pipelines that has no analogy in synchronous pipelines.

Yet another observation is that the **FC** configurations have the same forward latency as the equivalent members of the **CF** family, but the cycle times are better for the **FC** family members. The **FC0** cycle time has, in fact, the same coefficients as the **CF1** cycle time even though **FC0** has one less latch per stage. Likewise, **FC1** and **FC2** have cycle times corresponding to those found for **CF2** and **CF3**. The reverse latency of the **FC** family is also better because bubbles can flow through the control logic without going through the function blocks. Therefore, the **FC** family members are always better than the corresponding members in the **CF** family.

Finally, the comparison between the **FC** family and the **PS** family is not as clear cut as the previous observations for a couple reasons. First, Table 3.1 does not express the increased function block delay (because of taller transistor stacks) of the combinational function blocks in the **FC** family over the precharged function blocks of **PS**. And although the **PS** family has better forward latency, the **FC** family has better reverse latency. So, while the **PS** family will generally be the correct choice for the absolute lowest forward latency, the **FC** family may be preferred for simple function blocks because of its delay-insensitivity or better throughput.

3.3.2 Table of Extreme Cases

In order to illustrate the range of values that the latencies and cycle times can have, Table 3.2 shows the results of applying the simplifications listed at the tops of its columns to the coefficients previously summarized in Table 3.1. The numbers for both latency and throughput are normalized by $t_{F\uparrow}$, the function block evaluation time. Since all the numbers represent delays, the smaller numbers are always better. The first three columns of Table 3.2 are based on the assumption that $t_{F\downarrow} = t_C = t_D = 0$, which is nearly true in the extreme case of large function blocks composed internally of several precharged domino gates. Since the stages evaluate in series but reset in parallel, the reset time will be much less than the evaluate time, and can justifiably be approximated as zero. The table is not filled in in these columns for the combinational logic cases since these assumptions could never apply in those cases because they will reset serially even if composed internally of several gates. The middle three columns in Table 3.2 are for the case of $t_C = t_D = 0$ while $t_{F\downarrow} = t_{F\uparrow}$. This case is an appropriate assumption for large function blocks that both evaluate and reset serially, making the reset time comparable to the evaluate time. The last three columns are for the case of $t_C = t_D = t_{F\downarrow} = t_{F\uparrow}$, an appropriate assumption for very small function blocks, in which both the function blocks and the latches are merely a gate delay. Real applications would, of course, have latency and cycle times somewhere between the extremes listed in this table.

Pipeline Config	Big Composite F Block			Big Serial F Block			Small F Block		
	$t_{F\downarrow}=0$			$t_{F\downarrow}=t_{F\uparrow}$					
	$t_C=t_D=0$						$t_C=t_D=t_{F\uparrow}$		
	L_f	$S(L_f+L_r)$	P	L_f	$S(L_f+L_r)$	P	L_f	$S(L_f+L_r)$	P
PC0	1	3.00	3	1	4.00	4	3	12.00	12
PC1	1	1.50	2	1	2.00	2	4	9.00	10
PC2	1	1.00	1	1	1.33	2	5	8.00	10
PC3	1	0.75	1	1	1.00	2	6	7.50	10
PS0	1	3.00	3	1	4.00	4	1	6.00	6
PS1	1	1.50	2	1	2.00	2	2	6.00	6
PS2	1	1.00	1	1	1.33	2	3	6.00	6
PS3	1	0.75	1	1	1.00	2	4	6.00	6
CF0				1	4.00	4	2	10.00	10
CF1				1	2.00	2	3	8.00	8
CF2				1	1.33	2	4	7.33	8
CF3				1	1.00	2	5	7.00	8
FC0				1	2.00	2	2	8.00	8
FC1				1	1.00	2	3	7.00	8
FC2				1	0.66	2	4	6.66	8
FC3				1	1.00	2	5	6.50	8
PL1	1	1.00	1	1	1.00	2	4	6.00	8
PL2	1	0.67	1	1	0.67	2	5	5.33	8
PL3	1	0.50	1	1	0.50	2	6	5.00	8

Table 3.2: Performance parameters under extreme case conditions. (Normalized to $t_{F\uparrow}$, the function block evaluate delay)

Each section of Table 3.2 gives the forward latency, L_f, the cycle time, P, and the quantity $S(L_f+L_r)$, which is a quickly determined lower bound on the cycle time for all of the configurations presented. The quantity $S(L_f+L_r)$ will also be used in the next chapter to distinguish when the cycle time handshaking can limit performance.

3.4 Summary and Conclusions

Using Dependency Graphs, a method has been developed to determine the latencies and local cycle times of deterministic self-timed pipelines. Applying this method to examples has led to useful tables for comparison of self-timed pipeline configurations. These comparisons could be used by synthesis tools to allow choosing from a wider range of possible circuits based on specific delay considerations.

For ordinary pipelines, throughput is usually the dominant consideration. Table 3.1 shows that the best choice for a high-throughput pipeline may vary depending on the actual ratios of t_C and $t_{F\downarrow}$ to $t_{F\uparrow}$. Likely good choices for pipeline configurations are **PS2**, **PL1**, and **FC1**. Configuration **FC1** uses a direct or static function block style rather than a precharged function block and can save the cost of one completion detector per stage while achieving similar throughput, but it is important that the remaining completion detectors follow the latches and not the function blocks.

The next chapter extends the analysis of latency and throughput characteristics to self-timed rings. It shows that by meeting certain constraints, it is possible to construct rings whose performance is limited solely by the forward latency of the stages. For rings, configuration **PS0**, which this chapter showed to have the lowest forward latency, is therefore particularly attractive.

Chapter 4

Performance Analysis of Self-Timed Rings

Self-timed pipelines process a series of data tokens. If an application uses a pipeline to solve an iterative problem requiring a series of internal computation operations, then, after an initializing input step, the pipeline can proceed to take its inputs from its own output. A looped pipeline forms a ring, whose stages can implement partial steps of the computation operation or can repeat the entire operation multiple times around the ring. If a given problem is fully specified by its initial input operands, then the ring's performance is not limited by a need for additional external data inputs during its iterations. And since the ring is composed of self-timed pipeline stages, which communicate locally with their neighbors, its performance is also independent of external clock and control signals. The performance of a self-timed ring is therefore determined solely by the circuit configuration of its stages. The goal of this chapter is to express the overall total performance of a ring in terms of the local stage parameters found from the analysis of stage configurations presented in the previous chapter.

Although, in general, rings may contain fork or join stages, which introduce or consume additional tokens as the tokens flow around the ring, this thesis examines only simple rings. Complex rings, which have different functions in each stage or forking or joining dataflows, and thus have different stage delays, need individual case-by-case analysis. Evaluating the actual performance of a particular complex ring requires specific information about the individual relative delays of the function blocks and control elements. To analyze specific cases, one can use the Dependency Graph analysis presented in the previous chapter by drawing the graph for the whole ring rather than for just a pipeline segment. The graph analysis finds the critical path through the ring as a whole, which can then be adjusted to achieve the best operation.

This chapter applies the results of the performance analysis methodology presented in the previous chapter specifically to self-timed rings composed of a simple loop of identical stages. Section 4.1 describes the terminology for rings, qualitatively describes data and bubble flow, and defines the variables characterizing each stage. Section 4.2 determines the overall performance of self-timed rings in terms of the variables describing the stages. The performance is expressed on graphs showing regions defined by the number of stages and tokens. Section 4.3 provides more explanations of the boundaries and edges of the performance regions, including the observation that a self-timed pipeline is simply the special case of an unrolled self-timed ring. The performance equations are examined further in Section 4.4, which presents graphs of latency and throughput slices versus the number of stages and tokens. A key boundary line in the graphs has a coefficient that can be interpreted as a “wavelength” for flowing tokens. This wavelength and related occupancy measures are characterized in Section 4.5 for ranges of self-timed ring stage configurations. Section 4.6 mentions, as an extension, how the performance measures are changed for self-timing using two-phase signalling. Section 4.7 summarizes this chapter by discussing the design implications suggested by this analysis of self-timed rings.

4.1 Ring Terminology

The previous chapter defined a variety of stage configurations. Since all of these configurations contain one function block in each stage, a self-timed pipeline requires G stages for a given problem that requires G function block evaluations. If the structure of the problem is cyclic, with a period of N or some factor of N, then the pipeline can be made into a ring containing N stages. Data tokens circulating around the ring complete the given problem in $\frac{G}{N}$ iterations. A design must include appropriate multiplexing to introduce data tokens into the ring and to remove each token after it loops $\frac{G}{N}$ times. More than one problem may be computed concurrently if the logic that controls the multiplexors allows the introduction of multiple tokens into the ring. Let the number of tokens kept in the ring be denoted by K. The input multiplexors and output connections can be ignored in discussing the fundamental properties and performance of the ring.

Each stage of a self-timed ring can use one of the configurations suggested in the previous chapter for pipeline stages. The stage configuration chosen for the ring will have particular parameters characterizing its local performance, and these parameters can be determined from the Dependency Graph analysis of the previous chapter. The forward latency, L_f, is the delay through each stage of data tokens flowing forward around a ring. The reverse latency, L_r, is the delay through each stage of bubbles flowing backward. The local cycle time P is the minimum interval between tokens passing through the same stage. Since the stage configurations may contain a varying number of latches, the parameter S characterizes the number of stages required to contain a pair of data and reset tokens held statically back to back.

As tokens and bubbles flow around a ring they displace each other, but the total number of tokens and bubbles remains fixed after the ring is initialized. Since every data token consists of a pair of one data element and one reset spacer, the pipeline space occupied by a data token can contain two bubbles. A ring with N stages and K tokens therefore contains $2\left(\frac{N}{S} - K\right)$ bubbles.

When self-timed stages form a ring, the ensemble may be viewed as a whole to define a total latency and total cycle time. The total latency, λ, is the delay between the introduction of a new data token into the ring and the removal of the corresponding processed token after the number of loops necessary for the token to have passed through G function evaluation stages in all. The total cycle time, Φ, is the delay between the input or output of successive tokens while K tokens circulate in the ring. Since all the other tokens in a ring are replaced with new data during the time it takes for one token to loop to completion, the relation $\lambda = K\Phi$ holds in the steady state. This relation for rings is a generalization of the similar relation for pipelines, which gives the total latency of a pipeline containing K tokens as K times the delay between successive tokens. Observe, however, that the relation $\lambda = K\Phi$ depends on neither the number of stages in the ring nor the number of times tokens must loop. The overall throughput, T, of the ring is of course given by $T = 1/\Phi$. Therefore, the latency and throughput are related by the simple equation

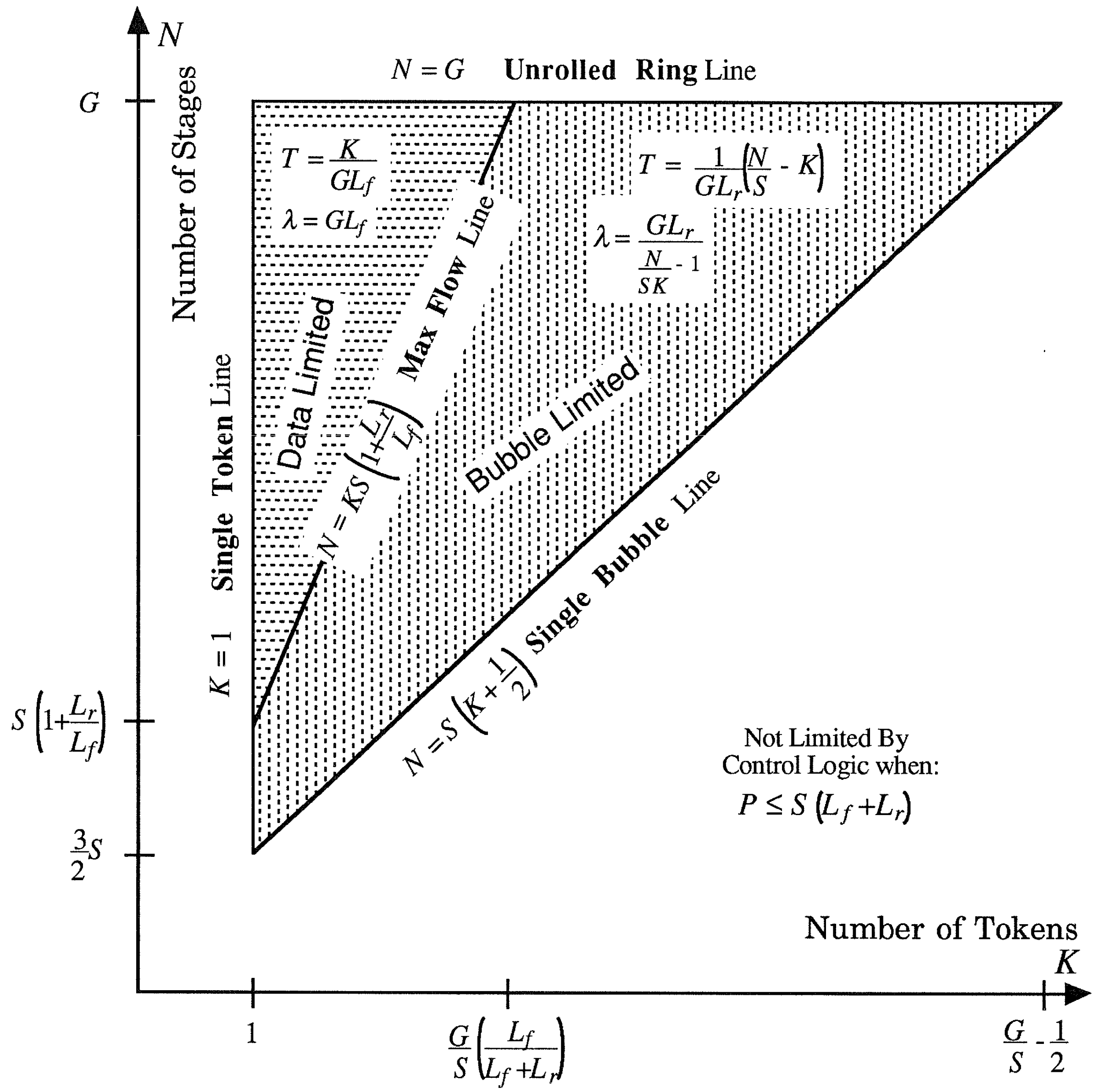
$$\lambda = \frac{K}{T}. \tag{4.1}$$

In a ring, latency and throughput do not fundamentally trade off with each other, but both latency and throughput do trade off with the ring area.

Analyzing the performance of a ring means determining the total latency, λ, and total cycle time, Φ, for the computation of a problem with G function evaluations as functions of the local parameters L_f, L_r, P, and S, which describe the particular stage configuration chosen.

4.2 Ring Performance Graphs

The performance of a self-timed ring can be limited by different causes. The number of stages and number of tokens in a ring determine which of these causes predominates. The possible limiting considerations



N
Number of Stages
G
$N = G$ Unrolled Ring Line
$T = \frac{K}{GL_f}$
$\lambda = GL_f$
$T = \frac{1}{GL_r}\left(\frac{N}{S} - K\right)$
$\lambda = \frac{GL_r}{\frac{N}{SK} - 1}$
Max Flow Line
Data Limited
$N = KS\left(1+\frac{L_r}{L_f}\right)$
Bubble Limited
$K = 1$ Single Token Line
Single Bubble Line
$N = S\left(K + \frac{1}{2}\right)$
$S\left(1+\frac{L_r}{L_f}\right)$
$\frac{3}{2}S$
Not Limited By Control Logic when:
$P \le S\left(L_f + L_r\right)$
Number of Tokens
K
1
$\frac{G}{S}\left(\frac{L_f}{L_f+L_r}\right)$
$\frac{G}{S} - \frac{1}{2}$

Figure 4.1 Performance in rings not limited by handshaking control logic.

define regions of values for N and K in which different relations expressing ring performance apply. Therefore, the fundamental diagrams showing performance are graphs having N and K as the axes that define the applicable regions for particular equations giving total latency and throughput. Figure 4.1 shows a graph for stages whose cycle time satisfies

$P \leq S(L_f + L_r)$, which means the control logic is fast enough that it does not limit the ring's performance.

Three lines enclosing the entire valid region of ring operation bound the possible values for N and K in a self-timed ring. The top line for which $N = G$ is the "**Unrolled Ring**" line because it represents the degenerate case of a ring completely unrolled into a pipeline for accomplishing the needed G function evaluations. The left edge of the valid region in Figure 4.1 is the K=1 "**Single-Token**" line. The minimum value for K is, of course, one because a ring with no data tokens is not useful. The right edge of the region is the diagonal "**Single-Bubble**" line where $N = S\left(K + \frac{1}{2}\right)$. Values of $K > \frac{N}{S} - \frac{1}{2}$ are not possible because a ring must have at least one bubble for data to circulate at all. There are two regions of operation within the triangle formed by the three boundary lines. One region, marked "**Data-Limited**," is where the token flow rate is limited by the forward latency; the other, marked "**Bubble-Limited**," is where the token flow rate is limited by the reverse latency. These names apply because limitation by forward latency occurs when stages wait for new data, and limitation by reverse latency occurs when stages must wait for new bubbles.

Within the **Data-Limited** region, so few data tokens are in the ring that there are plenty of bubbles, and therefore the performance is determined entirely by the forward latency of the tokens. The total latency within this region is

$$\lambda = GL_f, \tag{4.2}$$

and from equation (4.1) the throughput in this region is thus

$$T = \frac{K}{GL_f} \, . \tag{4.3}$$

Within the **Bubble-Limited** region, so many data tokens are in the ring that the low supply of bubbles limits the rate at which data can flow. Since the backward flow rate of bubbles is specified by the reverse latency of the stages, the total throughput is

$$T = \frac{1}{GL_r}\left(\frac{N}{S} - \mathrm{K}\right), \tag{4.4}$$

which is just the number of bubbles in the ring times the rate at which they flow. From (4.1) and (4.4), the total latency in the **Bubble-Limited** region is

$$\lambda = \frac{GL_r}{\frac{N}{SK} - 1} \, . \tag{4.5}$$

The boundary between these regions is the "**Max Flow**" line, on which the flow rates of tokens and bubbles are matched for maximum performance. The intersection of (4.2) and (4.5) determines this line has equation

$$N = KS\left(1 + \frac{L_r}{L_f}\right). \tag{4.6}$$

Figure 4.1 showed the regions of performance when the local cycle time is sufficiently fast for it never to be the limiting factor. If, however, $P > S(L_f + L_r)$, then the cycle time can limit the total performance. Figure 4.2 shows this condition, which introduces a new region of operation between the **Data-Limited** and **Bubble-Limited** regions.

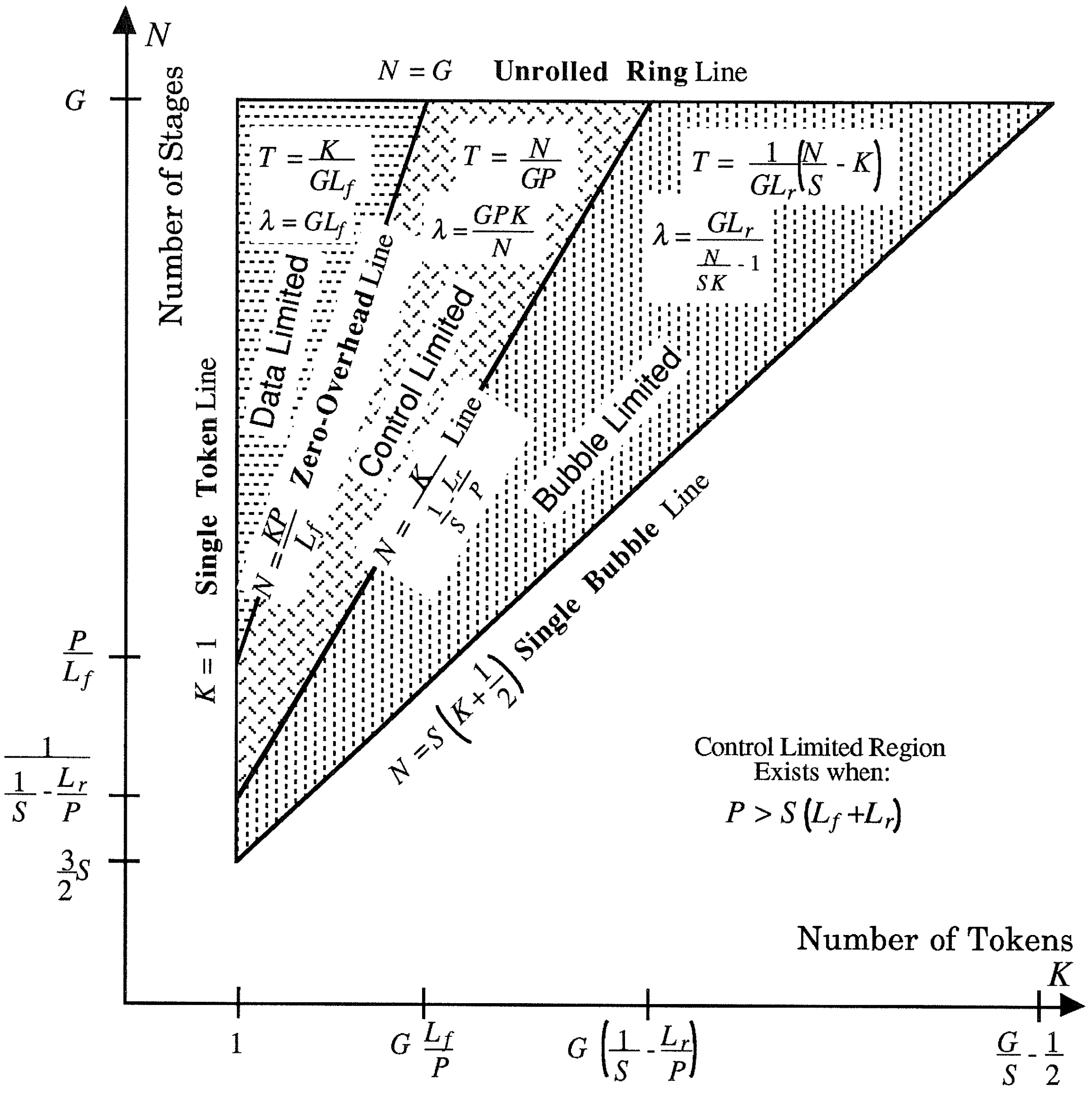

N
Number of Stages
N = G Unrolled Ring Line
$T = \frac{K}{GL_f}$
$\lambda = GL_f$
$T = \frac{N}{GP}$
$\lambda = \frac{GPK}{N}$
$T = \frac{1}{GL_r}\left(\frac{N}{S} - K\right)$
$\lambda = \frac{GL_r}{\frac{N}{SK} - 1}$
Data Limited
Zero-Overhead Line
Control Limited
Line
Bubble Limited
Single Bubble Line
K = 1 Single Token Line
$N = \frac{KP}{L_f}$
$N = \frac{K}{\frac{1}{S} - \frac{L_r}{P}}$
$N = S\left(K + \frac{1}{2}\right)$
Control Limited Region Exists when:
$P > S(L_f + L_r)$
G
$\frac{P}{L_f}$
$\frac{1}{\frac{1}{S} - \frac{L_r}{P}}$
$\frac{3}{2}S$
Number of Tokens
K
1
$G\frac{L_f}{P}$
$G\left(\frac{1}{S} - \frac{L_r}{P}\right)$
$\frac{G}{S} - \frac{1}{2}$

Figure 4.2 Performance in rings possibly limited by handshaking control logic.

This region is "**Control-Limited**" because the local cycle time constrains the performance within it. Within this region the throughput is

$$T = \frac{N}{GP}, \tag{4.7}$$

and, by using (4.1), the total latency within this region is

$$\lambda = \frac{GPK}{N}. \tag{4.8}$$

The **Control-Limited** region's boundary with the **Data-Limited** region is where (4.2) and (4.8) intersect, which is the line with equation

$$N = K\frac{P}{L_f}. \tag{4.9}$$

This line can be called the "**Zero-Overhead**" line because only rings in the region to the left of it operate without any overhead caused by control logic. The boundary between the **Control-Limited** region and the **Bubble-Limited** region is at the intersection of (4.5) and (4.8), which is the line given by the equation

$$N = \frac{K}{\frac{1}{S} - \frac{L_r}{P}}. \tag{4.10}$$

Note that Figure 4.1 is really just a special case of Figure 4.2 for $P = S(L_f + L_r)$ where the **Control-Limited** region collapses into the **Max Flow** line. This degeneracy occurs when the local cycle time ceases to play a constraining role because it is reduced enough to be less than the constraints from the forward and reverse latency.

4.3 Performance Region Edges

The edges of the performance graphs deserve special attention because they specify both desirable and undesirable limiting cases. The points where the diagonal lines in Figure 4.2 cross the vertical $K=1$ **Single-Token** line are particularly significant. The lowest point at $\left[K=1\ ,\ N=\frac{3}{2}S\right]$ is the point giving the absolute minimum number of stages, $\frac{3}{2}S$, in a ring. A ring with fewer stages cannot be self-timed with embedded-completion signaling, since it would not provide sufficient space for a single data element, reset spacer, and bubble to circulate.

The desired region of operation for the lowest latency is the **Data-Limited** region, where $\lambda = GL_f$. The best throughput, or the smallest area for a given throughput, occurs on the boundary between the **Data-Limited** region and the **Control-Limited** region, which the **Zero-Overhead** line specifies. The best design goal is therefore the lowest point on this line able to achieve the desired throughput. If there is not a specific constraint on the throughput, then *the* unique desired operating point is where the **Zero-Overhead** line intersects the **Single-Token** line at $\left[K=1\ ,\ N=\frac{P}{L_f}\right]$ because it achieves zero overhead with the fewest stages. Rings with a single token are therefore the preferred case when latency or area are the most important considerations. If there is a specific need for higher throughput, then other points on the **Zero-Overhead** line can satisfy it with the additional area cost of more stages and tokens. Multiple token rings may also be called for if there is a specific need for buffering or delaying more data tokens in the first-in-first-out (FIFO) queue formed by a self-timed ring.

The top edge of Figure 4.2 is the "**Unrolled Ring**" line, on which there are G stages to accomplish the G function evaluations, and iteration is not necessary. This line, therefore, simply describes a self-timed *pipeline* as the special case of an unrolled ring. A pipeline's total latency and throughput will be characterized by the same three data-, control-, and bubble-limited regions of operation. A pipeline that is limited by its input rate operates in the **Data-Limited** region. A pipeline that is limited by its output rate operates in the **Bubble-Limited** region since bubbles are introduced at the output. A pipeline limited by local cycle time constraints along its length operates in the **Control-Limited** region. Since $N{=}G$, the maximum throughput of a self-timed pipeline is $\frac{1}{P}$, and the pipeline achieves this rate when the number of tokens it contains in the steady state is within the range given by

$$N\frac{L_f}{P} \leq K \leq N\left(\frac{1}{S} - \frac{L_r}{P}\right). \tag{4.11}$$

The lower right **Single-Bubble** diagonal edge of Figure 4.2 is the least desirable condition for a self-timed ring. Along this edge, the total throughput is $T = \frac{1}{2GL_r}$ and the total latency is $\lambda = 2GL_rK$. The factor of 2 occurs because the single bubble must make two cycles around the ring for all of the tokens to advance once: one cycle for the data portion, and one cycle for the reset spacer portion of each token.

4.4 Latency and Throughput Slices through the Performance Graphs

Figures 4.1 and 4.2 illustrate the values of N and K that dictate the different regions of performance. Slices through these diagrams can more clearly illustrate the resultant latency and throughput characteristics for specific values of N or K. The endpoints of the slices correspond to the edges discussed in the previous section. Figure 4.3 shows the total latency, λ, plotted versus K for a fixed value of N. Figure 4.4 shows λ plotted versus N for a fixed value of K. The corresponding graphs for throughput are shown in Figures 4.5 and 4.6. All the graphs have three

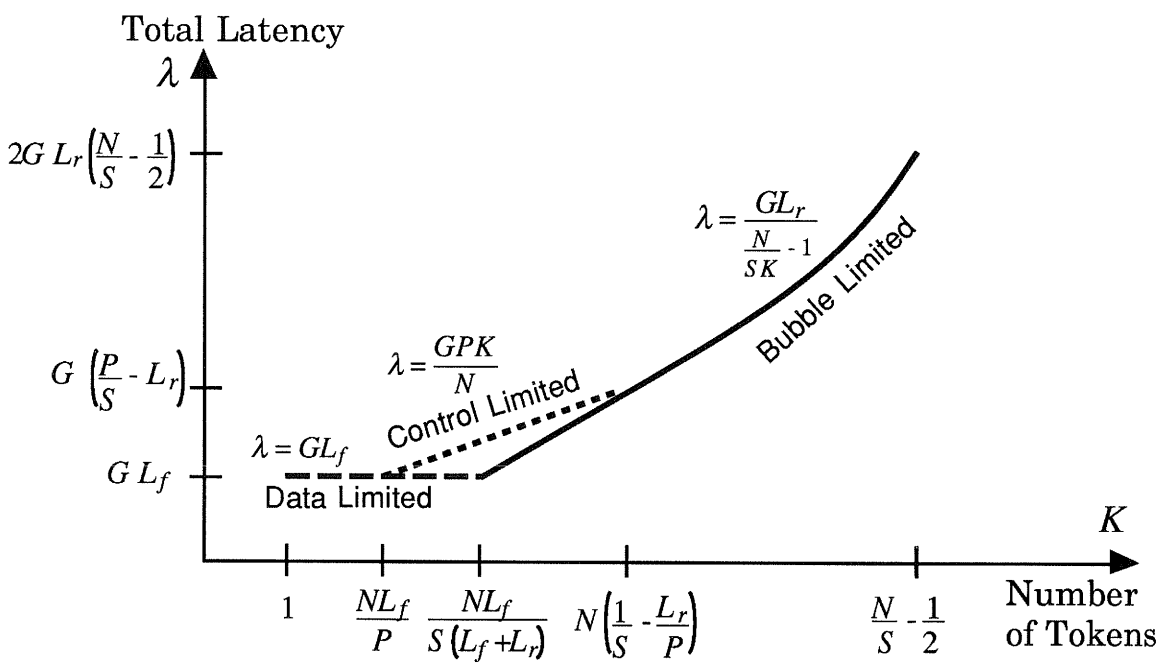


Figure 4.3 Latency versus K (tokens) while N (stages) is constant.

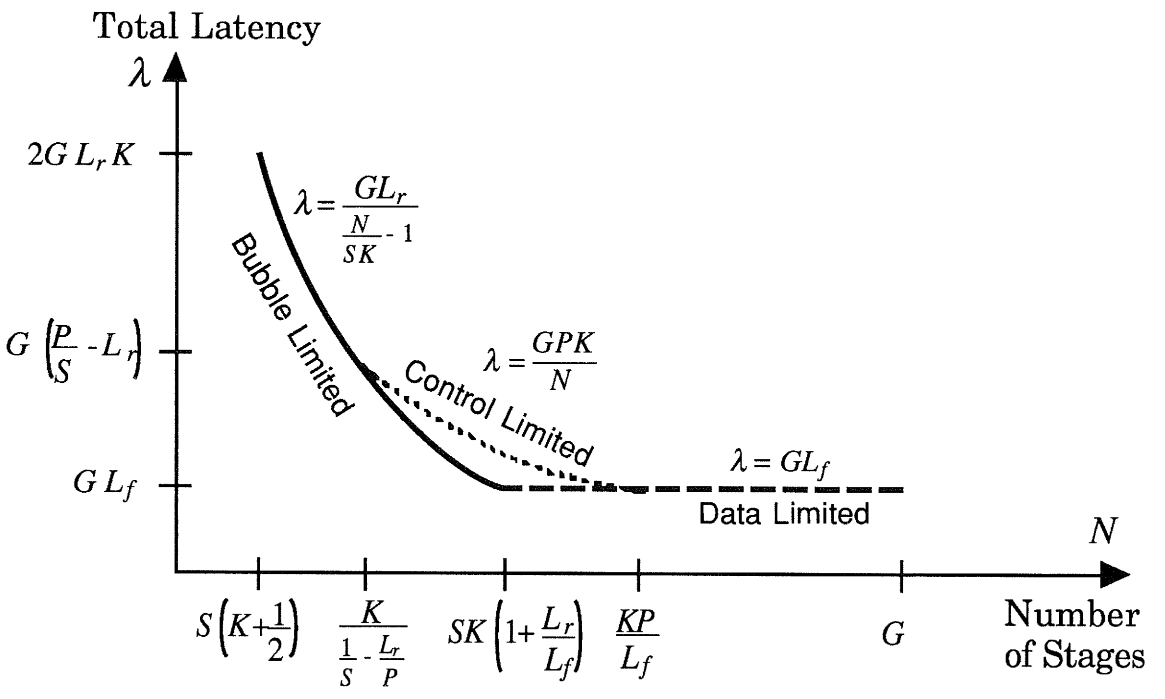


Figure 4.4 Latency versus N (stages) while K (tokens) is constant.

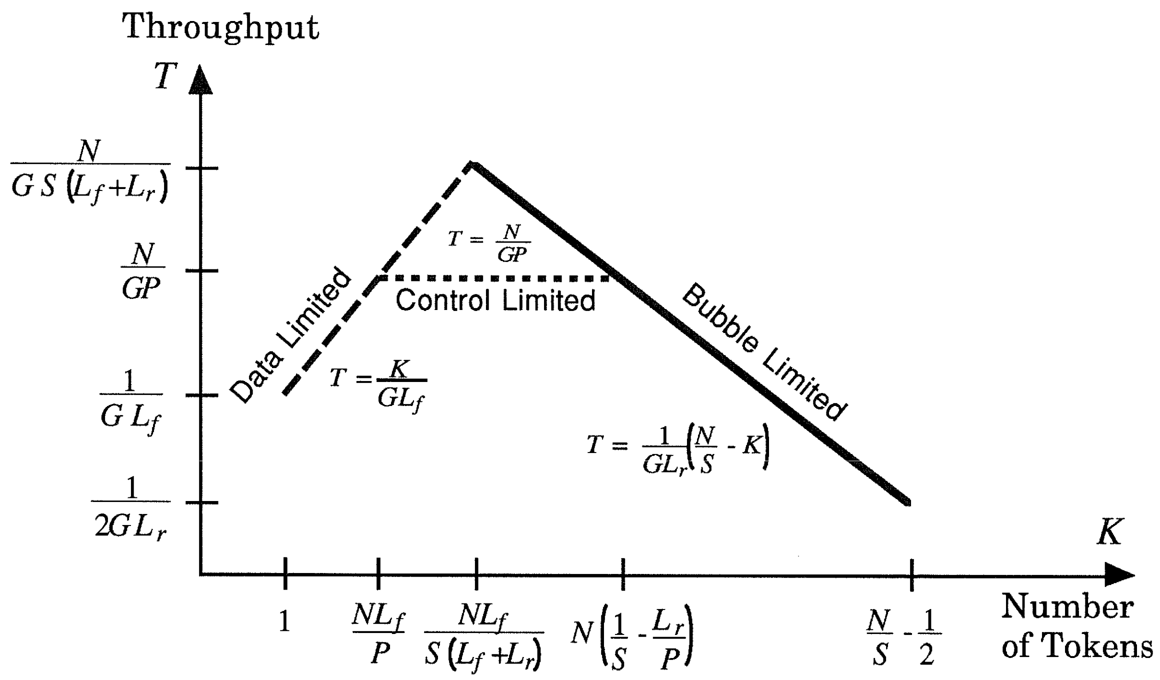


Figure 4.5 Throughput versus K (tokens) while N (stages) is constant.

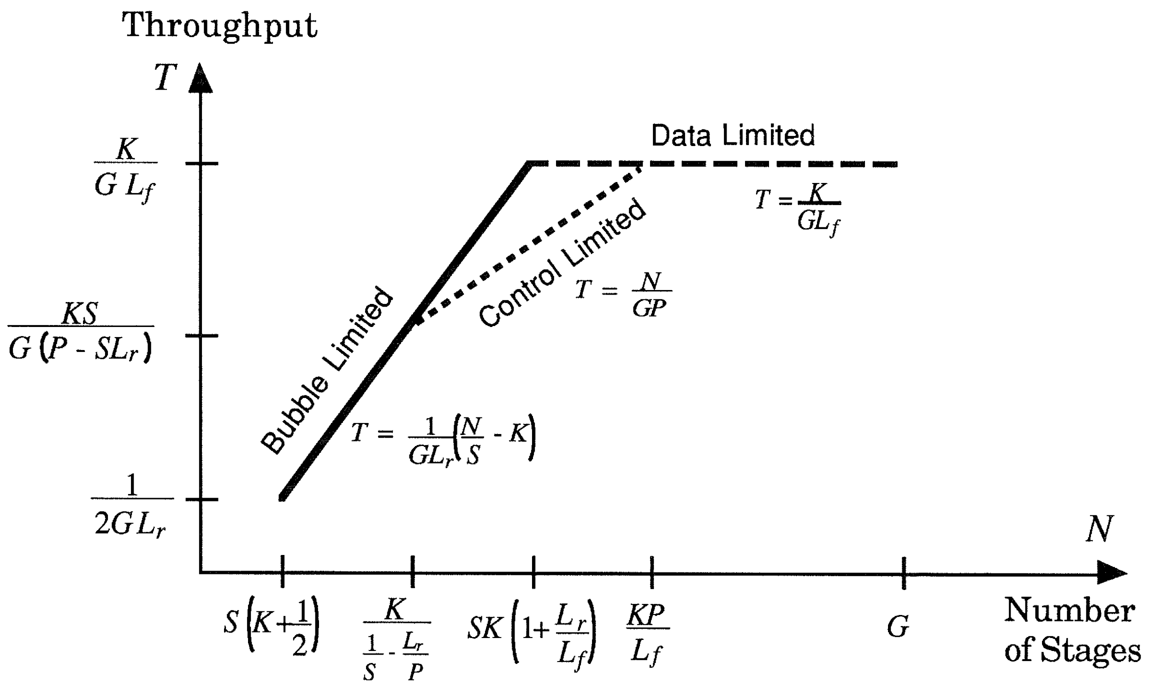


Figure 4.6 Throughput versus N (stages) while K (tokens) is constant.

line segments corresponding to the three regions in Figure 4.2. The dashed line segment shows the **Data-Limited** region, in which the latency and throughput are given by equations (4.2) and (4.3). The solid line shows the **Bubble-Limited** region, in which the latency and throughput are given by equations (4.5) and (4.4). Where there is a dotted line from the **Control-Limited** region, it clips the attained values of latency and throughput to the values given by equations (4.8) and (4.7), which are worse. The specific instances of Figures 4.3 and 4.5 with $N=G$ are useful in describing the performance of a self-timed pipeline versus the number of tokens.

Figure 4.3 shows that the **Data-Limited** region attains the best latency, but that a ring will not operate in this region if there are more than $N\frac{Lf}{P}$ tokens packed into it. For a given number of tokens, Figure 4.4 shows that a ring's latency can be improved by increasing the number of stages up to $K\frac{P}{Lf}$, but that greater values for N do not decrease the latency any further.

The throughput graph in Figure 4.5 illustrates how increasing the number of tokens increases the throughput while the ring is still within the **Data-Limited** region, but adding too many tokens causes the ring to enter the **Bubble-Limited** region. When there are so many tokens that there is room for just a single bubble in the ring, the throughput degrades to a level even lower than the throughput for just a single token because the bubble must circulate twice in order for all the data elements and all the reset spacers to advance once. Figure 4.6 shows that throughput increases as one increases N up to $K\frac{P}{L_f}$, but stays constant for higher values of N; more stages cost more in area without improving performance if no more tokens are added to make use of the added stages.

For both latency and throughput considerations, Figures 4.3-4.6 show that the desired point of operation is at the intersection of the dashed and dotted lines. This intersection corresponds to the entire **Zero-Overhead** line in Figure 4.2. Since $\lambda = GL_f$ everywhere on this line, any point on or above this line will be a zero-overhead self-timed ring.

4.5 Dynamic Wavelength, Occupancy, and Static Spread

Because the operation of a ring or pipeline is cyclic, the model of a travelling wave is a good analogy. Every data token is like a cycle of the wave. The data and reset spacer portions of each token correspond to the two extremes of a wave's period, its "crest" and "trough." The wavelength is the average distance between like points of two data tokens and is hence equal to $\frac{N}{K}$. The previous section observed that a zero-overhead pipeline or ring has $\frac{N}{K} \geq \frac{P}{L_f}$. The minimum desirable wavelength, denoted by W, is therefore $\frac{P}{L_f}$, and is simply the coefficient of the **Zero-Overhead** line in Figure 4.2. The wavelength can also be viewed as the "dynamic spread" between tokens as they are flowing, and should not be confused with the value S, which is the static spread of tokens when they are fully packed together. The dynamic spread specifies the average number of stages occupied by a data element and its accompanying reset spacer, along with bubble space sufficient to allow it to flow unimpeded by the other tokens.

The reciprocal of the wavelength is the dynamic occupancy or "utilization" of the stages in the ring or pipeline. The utilization tells how effectively the stages are being used in parallel. For example, if $W = 2$, then only every other stage in the pipeline can be simultaneously evaluating. Likewise, the reciprocal of S is the maximum static occupancy or "packing density," but is determined by the connectivity of the components and not their delays. This quantity is important if the application requires the pipeline or ring to provide a buffer queue for a specified number of tokens during brief periods of input/output rate mismatch.

Both the dynamic and static spreads for the various pipeline configurations are shown in Table 4.1. The numbers are in units of stages/token (the smaller numbers being of course better); however, the stages are not necessarily of constant area since the stages with more latches will be larger. The numbers provide a fair indication of the relative areas of the different configurations if the function block areas dominate the latch areas.

The value of the wavelength, W, in Table 4.1 is a quick reference to the number of stages necessary to achieve zero-overhead in a single token ring. Table 4.1 evaluates W under the same three conditions used in Table 3.2 (Big Composite Function Block, Big Serial Function Block, and Small Function Block) representing the extremes of practical cases. Actual implementations of the configurations will have values somewhere between the extremes in the table. The necessary number of stages is the larger of W and $\frac{3}{2}S$. For example, any function implemented with the **PS0**

Pipeline Config	Big Composite F Block	Big Serial F Block	Small F Block	Static Spread S
	$t_{F\downarrow}=0$	$t_{F\downarrow}=t_{F\uparrow}$		
	$t_C=t_D=0$		$t_C=t_D=t_{F\uparrow}$	
	$W=P/L_f$	$W=P/L_f$	$W=P/L_f$	
PC0	3.00	4.00	4.00	2.00
PC1	2.00	2.00	2.50	1.00
PC2	1.00	2.00	2.00	0.67
PC3	1.00	2.00	1.67	0.50
PS0	3.00	4.00	6.00	2.00
PS1	2.00	2.00	3.00	1.00
PS2	1.00	2.00	2.00	0.67
PS3	1.00	2.00	1.50	0.50
CF0		4.00	5.00	2.00
CF1		2.00	2.67	1.00
CF2		2.00	2.00	0.67
CF3		2.00	1.60	0.50
FC0		2.00	4.00	2.00
FC1		2.00	2.67	1.00
FC2		2.00	2.00	0.67
FC3		2.00	1.60	0.50
PL1	1.00	2.00	2.00	1.00
PL2	1.00	2.00	1.60	0.67
PL3	1.00	2.00	1.33	0.50

Table 4.1: Wavelength and Static spreads in units of stages/token.

configuration will require three to six stages to achieve **Zero-Overhead** operation, depending on the style and size of the function blocks.

4.6 Extension to Two-Phase Signalling

This thesis considers primarily level-encoded four-phase signalling, which requires the separation of data elements with reset spacers.

However, many of the results and principles also apply to two-phase transition-encoded systems [SUTH89] with appropriate modifications to the function blocks, such as those we suggested in [DEAN90]. In particular, the last columns of Tables 3.2 and 4.1 with the assumption $t_{F\downarrow}=t_{F\uparrow}$ are appropriate for two-phase configurations. Since the values they enumerate for the wavelength, cycle time and static spread include both the data and reset elements of a token, these values can just be halved when used to refer to a two-phase case in which both the rising and falling transitions convey a separate useful data token.

The analysis in this chapter is fundamentally unchanged when describing two-phase pipelines and rings. The same equations describing performance apply, but the value of K is doubled, and the values of S and P are halved, relative to their values in the four-phase case. This means that a two-phase pipeline or ring has the same latency but twice the throughput of the corresponding four-phase case with the same number of stages. Likewise, since the wavelength is halved, it is possible to achieve **Zero-Overhead** operation with only half the number of stages as are required in the four-phase case. However, these statements about two-phase performance are overly optimistic because they are true only if the local parameters of each stage remain constant. In reality, the values of the per-stage latencies and local cycle time become worse because a two-phase function block for a given function is usually more complex than the corresponding four-phase function block.

4.7 Summary of Implications for Design of Self-Timed Rings

The analysis of ring performance shows that a self-timed ring can achieve a total latency equal to just the sum of the forward latencies of the stages through which data must pass. This minimal latency is $\lambda = GL_f$, which is possible when the values of N and K are in the **Data-Limited** region. This observation yields three important rules for the design of minimal latency self-timed iterative rings. First, since the total latency of a ring is proportional to the latency, L_f, of the individual stages composing the ring, it is important a designer choose a stage configuration minimizing L_f. The direct concatenation of precharged function blocks, configuration **PS0**, has the lowest L_f because it has no latches, and its latency therefore comes solely from the delay of the raw functional blocks themselves. Furthermore, the configuration is compact because of the absence of control logic. Configuration **PS0** is therefore the best configuration for low latency rings. Rings that have **Zero-Overhead** and that are built using configuration **PS0** are "**Minimal-Latency**" because the entire total latency is just the sum of the raw function block delays.

The second rule is to minimize the value of the local cycle time, P, which results in the **Zero-Overhead** line of Figure 4.2 approaching the **Max-Flow** line of Figure 4.1. This rule allows a **Zero-Overhead** ring to be achieved in the least area. The value of P can be reduced by methods presented in the next chapter for the composition and connection of completion detectors.

The third, and most important, rule for self-timed ring design is to use at least $K\frac{P}{L_f}$ stages in the ring to achieve **Zero-Overhead** and avoid degradations due to control or bubble limitations. The next chapter will further discuss the optimal size for function blocks and the value of $\frac{P}{L_f}$.

Chapter 5

Methods for Increasing Performance

For a self-timed iterative ring to achieve zero-overhead performance, it must operate in the **Data-Limited** region where the critical path goes solely through data elements. When it operates in this region, control logic is not in the critical path determining the ring cycle time, and data flows around the ring at the same speed at which it would flow through the same functional data blocks if they were "unwrapped" into an array. Wrapping the blocks up into a ring thus can achieve the performance of the array without requiring the area of the full array.

Chapter 2 defined a range of stage configurations that can be used in self-timed pipelines and rings. For the **PS0** configuration in particular, which has no explicit latches, a **Zero-Overhead** ring is also

Minimal-Latency because the total latency is the same as that of an unwrapped array and that array is purely combinational, i.e., no (latch or control) delays add overhead that would increase the total latency beyond the sum of the pure combinational function block delays.

If the input and output control of a ring keep K tokens flowing, then, as the analysis of the previous chapter determined, at least $K\frac{P}{L_f}$ stages are required in the ring to achieve **Zero-Overhead** operation where $\lambda = GL_f$. I therefore call the inequality

$$N \geq K\frac{P}{L_f} \tag{5.1}$$

the "**Zero-Overhead constraint.**" This chapter suggests approaches and ideas for hardware design that can help to achieve this constraint, improve performance, and minimize the total area.

The easiest way to meet the **Zero-Overhead constraint** is to adjust the number of stages and tokens, and Section 5.1 mentions some issues involved in these choices. Since these choices alone may not achieve good tradeoffs within a reasonable area, Section 5.2 calculates how to minimize the area by adjusting the number of gates in each function block, called the grain size. Another way to ease satisfying the constraint is to change the stages to reduce their local cycle time. Section 5.3 presents a modification to the standard placement of completion detectors that can reduce the local cycle time by overlapping the completion detector delay with part of the function block delay. Section 5.4 points out how the completion detectors can be simplified by making reasonable delay assumptions. The cycle time also can be reduced by the introduction of asymmetric control logic, as suggested in Section 5.5. Another possible method of improving performance, discussed in Section 5.6, is to minimize the expected value of the forward latency by using known probabilistic information to reorder logic or adjust transistor sizing.

5.1 Number of Stages and Tokens

The parameters most easily varied in a self-timed ring are the number of tokens and stages. Many applications derive no benefit from having multiple tokens and, for these cases, a single token ring will satisfy the **Zero-Overhead constraint** with the least area. Applications that benefit from multiple tokens are those specifically requiring higher throughput. Because $T = \frac{K}{\lambda}$, increasing the number of tokens by some factor increases the throughput by the same factor if (5.1) continues to be satisfied. Although the same throughput increase could also be achieved by multiplexing tokens among multiple rings operating in parallel, that approach would have additional datapath routing and signal loading without achieving any reduction in area. Introducing multiple tokens in a ring has the benefit of raising the throughput in a single structure without increasing loading on external bus connections.

It is possible to satisfy (5.1) for any desired value of throughput up to $\frac{1}{P}$ by using sufficiently high values for N and K. The maximum throughput is always $\frac{1}{P}$, because no ring can accept tokens faster than the local constraints of its stages. It does not make sense to increase N to a value greater than G, the number of stage executions necessary to accomplish the desired function. When $N = G$, the ring is fully unrolled into a pipeline, as discussed in Section 4.4. At this extreme, the maximum useful value for K is $G\frac{Lf}{P}$.

While increasing the number of stages is a primary means of satisfying the **Zero-Overhead constraint**, it does cost additional silicon area for each stage. Hence, it is relevant to consider other ways that also can help to satisfy the constraint and can do so with a lesser value for N.

5.2 Grain Size

One degree of freedom in designing a self-timed pipeline or ring is choosing the size of the function performed in each stage. This choice is often called adjusting the "grain size." The number of stage evaluations, G, required for a given problem always changes inversely to the grain size. Since the total area is proportional to the number of stages times the complexity of each stage, adjusting the grain size can also change the total area. In order to keep the total area fixed, the number of stages in the pipeline or ring must be changed inversely to a change in the grain size. That is, if the grain size is increased by some factor, then N and G must be decreased by that same factor; if the grain size is decreased by some factor, then N and G must be increased by that factor. The most important effects of changing the grain size while keeping the total area constant can be understood in terms of changes in the performance figures from the previous chapter. If each stage's delay is dominated by the function block delay, then to first-order, L_f, L_r, and P all scale linearly with the grain size, as N and G scale inversely. The parameter S remains constant, and K

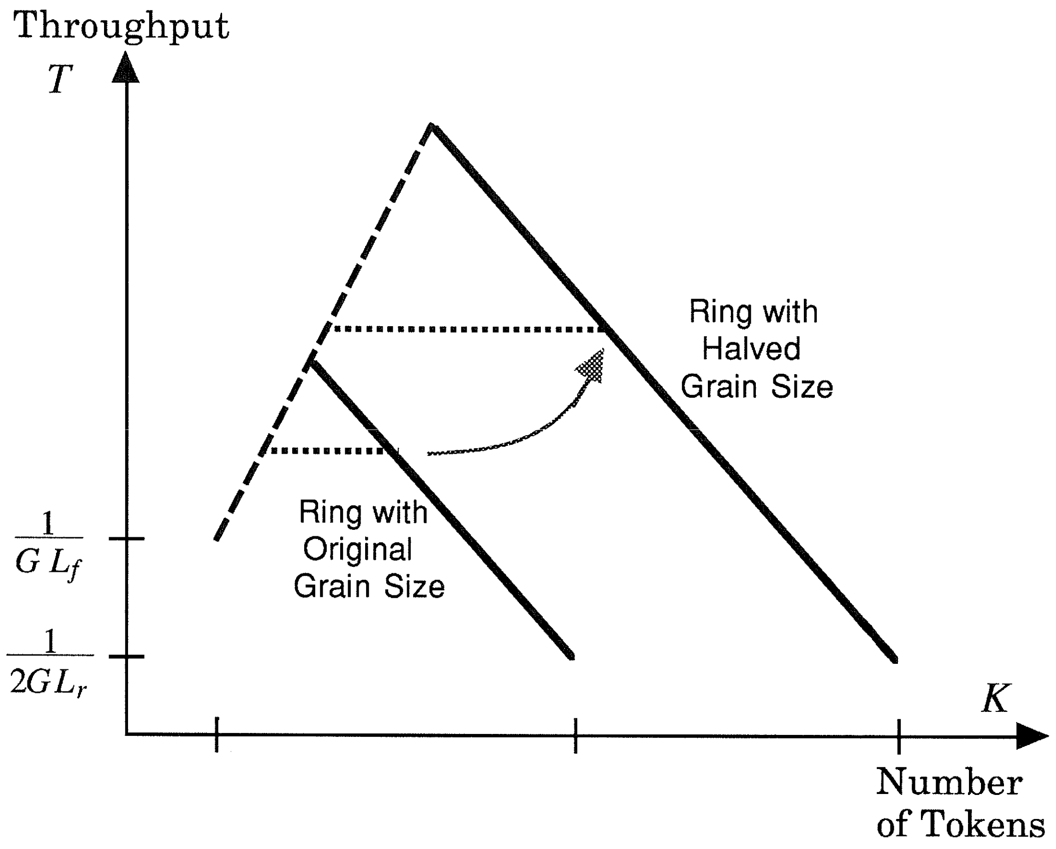


Figure 5.1: Decreasing the grain size effectively changes Figure 4.5 by extending the Data-Limited line and moving the Control-Limited and Bubble-Limited lines.

is still independent. For example, Figure 5.1 illustrates how Figure 4.5 changes when the grain size is halved. Values along the horizontal axis double, and so do the upper values on the vertical axis. But the lower endpoints on the vertical axis, $\frac{1}{GL_f}$ and $\frac{1}{2GL_r}$, remain approximately the same. Although the figure shows that the modified ring with half the grain size can contain up to twice as many tokens as the original ring, the *effect* of halving the grain size is indicated by the new value of the throughput for the original value of K. The effect is strongly dependent on the region in which the original operating point lay. If the ring were originally in the **Data-Limited** region, then the change in grain size has no first-order effect on the throughput or latency. In contrast, if the ring were originally in the **Control-Limited** or **Bubble-Limited** regions, then decreasing the grain size both increases the throughput and decreases the total latency. Because the decrease in grain size increases the extent of the **Data-Limited** region, the **Zero-Overhead constraint** is made easier to satisfy.

Second-order effects occur if the grain size is decreased enough so that the values of L_f, L_r, and P are no longer dominated by the function block delays. These second-order effects occur only for the stage configurations that contain latches and are more significant for the stage configurations that contain more latches. Whereas the first-order effects of the previous paragraph can simultaneously improve both the latency and throughput, it is the second-order effects for small grain sizes in which latency and throughput trade off against each other. The result is the same as for synchronous pipelines, in which decreasing the grain size increases the throughput but also increases the total latency. But because this is a second-order effect for self-timed rings, it is relevant only within the **Data-Limited** region in which a change in the grain size has no first-order effect.

Within the **Data-Limited** region, where the **Zero-Overhead constraint** is satisfied, it is also reasonable to consider changes in the number of tokens or stages while keeping the grain size constant. Increasing the number of tokens increases the throughput, as long as the ring stays within the **Data-Limited** region. Likewise, decreasing the number of stages decreases the total ring area without changing performance, as long as the ring stays within the **Data-Limited** region.

When the grain size and the number of stages are both allowed to change without constraining the total area to be constant, the effects discussed in the above paragraphs combine. The minimum total area that allows satisfying the **Zero-Overhead constraint** occurs at some intermediate tradeoff point. For a ring of identical stages, a procedure can be followed to find the best grain size, which satisfies (5.1) in the least area. First, the expressions for P and L_f can be written in terms of $t_{F\uparrow}$, $t_{F\downarrow}$, t_C, and t_D for a particular stage configuration with the aid of Table 3.1. These expressions can then be substituted into (5.1). Next, the values for $t_{F\uparrow}$ and $t_{F\downarrow}$ can be further decomposed in terms of a measure of the grain size such as g_F, the number of gates in the function. An expression for the total area can then be written in terms of g_F, by substituting in the expressions for N and K that are required in order to satisfy constraint (5.1). This expression for area can finally be minimized with respect to g_F to find the best value for the number of gates in each stage and the number of stages required.

The procedure to optimize grain size can be demonstrated for the example of the **PS0** stage configuration. This configuration is important because it can achieve the lowest possible total latency. Substituting in the expressions from Table 3.1 reformulates (5.1) as

$$N \geq K\left(3 + \frac{t_{F\downarrow} + 2t_D}{t_{F\uparrow}}\right). \qquad (5.2)$$

If the function blocks use full-controlled precharged logic, then the function block will be able to reset in a single gate delay, independent of the number of gates in the function, and the substitution $t_{F\uparrow} = (g_F)(t_{F\downarrow})$ is reasonable. Substituting this expression into (5.2) yields

$$N \geq K \left[3 + \frac{1}{g_F} \left(1 + \frac{2t_D}{t_{F\downarrow}} \right) \right]. \tag{5.3}$$

We can approximate the total area consumed by the ring as

$$A \propto (g_F + a_D)\, N \tag{5.4}$$

where a_D is the size of the completion detector relative to the size of gates in the function block. This equation can be minimized with respect to g_F and combined with (5.3) to find the optimal values for both the grain size and the wavelength when using configuration **PS0**:

$$g_F = \sqrt{\frac{a_D}{3} \left(1 + \frac{2t_D}{t_{F\downarrow}} \right)} \tag{5.5}$$

$$\frac{N}{K} = 3 + \sqrt{\frac{3 \left(1 + \frac{2t_D}{t_{F\downarrow}} \right)}{a_D}} \tag{5.6}$$

The optimal grain size for minimal area in a **PS0** configuration thus depends on the ratios of the speed and size of the completion detector (including its buffers for driving loads) to the precharged function block gates. Real implementations will, of course, require integral values for N and K that satisfy (5.3). If the completion detector's delay is small compared to that of a precharged gate in the function block, then the optimal grain size is likely just a single gate level per stage, with four to five stages in a single-token ring. However, since the term for the delay of the completion detector includes the delay through any buffers required to drive the precharge control signals, it is more likely for this stage configuration that the completion detector's delay is large compared to that of a precharged gate in the function block. In this case, equations (5.5) and (5.6) indicate the best performance is attained with more stages and a larger grain size. For example, when the completion detector and buffer delay is twice that of a gate in the function block, then the optimal grain size is probably near two gate levels per stage, with six stages in a single-token ring. In general though, these guidelines suggest that self-timed rings should have smaller stages than ordinary pipelines because a ring can save area by iterating more times around a loop of small stages.

5.3 Completion Detector Placement

Since the completion detector delay, t_D, directly affects the local cycle time, another means to reduce the cost of satisfying the **Zero-Overhead constraint** is to adjust the completion detector placement. Completion detectors tap off of the datapath at various points in a pipeline or ring. The two key issues affecting the choice of points at which to attach completion detectors are the datapath width and the overlapping of completion detector delay with part of the function block delay.

If the datapath width varies, one means of reducing t_D is to compute the completion status at a narrower cross-section of the datapath. A narrower datapath tap point allows the completion detector to be simpler because it can observe fewer bits. This principle is especially significant if function blocks are internally composed of smaller domino sections in chains with a common reset, since there is a choice in how to group these domino sections together into stages. For example, I chose to separate the stages in the division chip in [WILL87] at a point where the relevant datapath width was the number of quotient digits (3) rather than the number of remainder digits (55).

Another way in which a designer can optimize the completion detector placement is by overlapping the operation of the completion detector with part of the function block delay. This reduces both the reverse latency, L_r, and the cycle time, P. If the function blocks are internally composed of a domino chain of smaller sections, one can achieve the desired overlap by moving the point at which a completion detector taps the datapath to an earlier internal point within the function block, as suggested in Figure 5.2. The completion detector is thereby "shadowed" and gets a head-start because it overlaps with the rest of the function block after the point where it taps the datapath. The effect can be evaluated by separating the function block delays into the parts before and after the completion detector tap: $t_{F_A}\uparrow$ and $t_{F_B}\uparrow$ for the evaluation, and $t_{F_A}\downarrow$ and $t_{F_B}\downarrow$ for the reset. The change in completion detector placement causes the removal of some of the t_{F_B} terms from the equations for L_r and P. One can determine the exact changes by reexamining the Dependency Graphs with the function delays split into the two portions. For all of the stage configurations considered in the Chapter 2, the reverse latency is reduced by t_{F_B}, the delay of the portion of the function block bypassed by moving

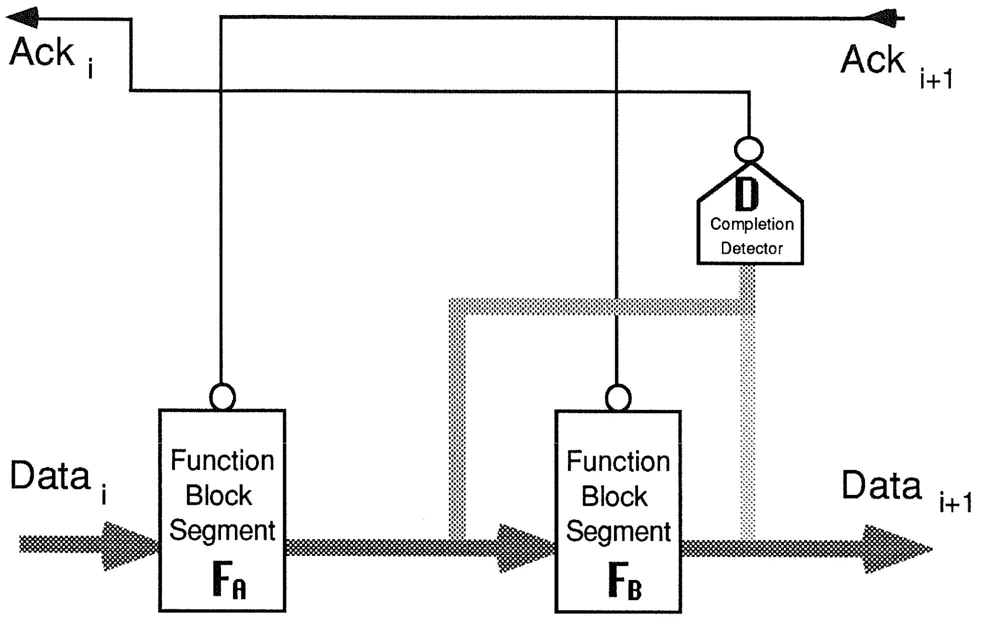


Figure 5.2: When a precharged Function block is internally composed of two or more domino segments with a common reset, the completion detector can be moved to tap off an internal node rather than the final output. This schematic illustrates the change in a **PS0** configuration.

the completion detector. More precisely, since L_r is defined to be the average of the reverse latency through a stage containing valid data and one containing a reset spacer, the new L_r will be lessened by avg($t_{F_B}\downarrow$, $t_{F_B}\uparrow$). The effect on P of moving the completion detector in each stage varies for the different configurations. P will be reduced by $t_{F_B}\downarrow + t_{F_B}\uparrow$ for the **PC0**, **PS0**, and **CF0** configurations, P will be reduced by $t_{F_B}\uparrow$ for the **PC1**, **PS1**, and **CF1** configurations, and P hardly changes at all for the **PC2**, **PS2**, and **CF2** configurations. (An exact analysis shows the reduction for these latter configurations to be $\min\left[t_{F_B}\downarrow, t_{F_B}\uparrow, \left|t_C\downarrow - t_C\uparrow\right|\right]$, which is zero for symmetric C-elements). The overlapping of the completion detector delay with part of the function block does not apply to the **FC** family of configurations because it has its completion detectors after the C-elements and not after the function blocks.

Moving the completion detector tap to an earlier interior point within the internal domino chain of each function block does not invalidate the self-timing of the rings or pipelines. An analysis of the new dependency graphs resulting from the relocation determined that no additional delay assumptions are required for each of the configurations of Chapter 2 to maintain its same speed-independent or delay-insensitive classification. I compared all paths from the point of the completion detector tap through each successor of the function block with the paths through the completion detector and around to the transition of the function block in the opposite direction. The comparison shows that the latter paths already include dependencies that verify the completion of the portion of the function block after the completion detector tap. Moving the completion detector to tap the outputs of an internal section of a domino chain within each stage therefore requires no additional assumptions or restrictions. However, the completion detector tap cannot be moved all the way to the inputs because there must be at least one precharged gate storing each input before the completion detector. The reason for this requirement is that the completion detector indicates when the inputs may be removed, and this signalling should occur only when all of the inputs have been consumed by the first section in the domino chain.

The two issues regarding completion detector placement may be at odds if a desired internal datapath point from which to tap a completion detector is not narrow. This difficulty may sometimes be fixed by redefining the stage boundaries. For example, if the output of the first internal domino section in a function block is wide and the second is narrow, then moving the wide domino section to the output of the preceding stage will allow the new arrangement to have the narrow section as its first section. A simple completion detector can then placed after the narrow section, and it will correctly overlap with the delays of the rest of the function block.

5.4 Completion Detector Composition

The cycle time can be lessened not only by moving the completion detector as discussed in the previous section, but also by reducing the completion detector's internal delay. This section discusses how the completion detector delay can be reduced by changing its composition.

An ordinary NOR gate can serve as a completion detector for each unary-encoded set of monotonically transitioning wires within a datapath. For the standard case of a bit encoded on two wires, the bit done signal can be generated by just a 2-input NOR gate. Since a speed-independent completion detector for an entire datapath width needs to detect both when a function block or latch has finished evaluating all of the bits in a datapath *and* when it has finished resetting them, a full completion detector is formed by combining the bit done signals with a tree of standard C-elements as shown in Figure 5.3. For monotonic inputs, a tree of C-elements with n inputs is equivalent to an n-input C-element. The tree of C-elements is required in order to verify that the total done signal always waits for the slowest of the bit done signals for both rising and falling

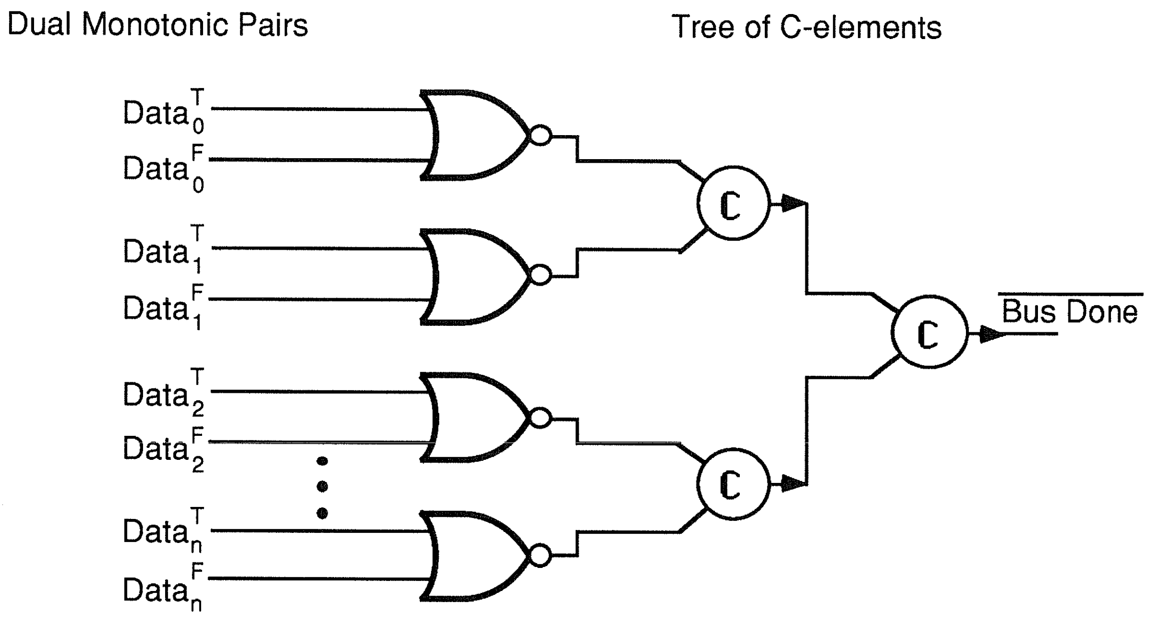


Figure 5.3: A standard Completion Detector uses a tree of C-elements to compute the bus done signal from the individual bit done signals.

transitions. Although the delay of the completion detector tree grows only logarithmically with the width of the datapath, the delay can still be appreciable compared to other delays. This delay can be lessened by making certain reasonable timing assumptions, which are discussed below.

One possible assumption is that the delays of the bits in a bus are bounded by the delays of the bits on the ends. This assumption is applicable if the bits are all driven in parallel by similar circuits that do not have a serial dependency. This assumption would therefore be quite reasonable for a carry-save adder, but invalid for a carry-propagate adder. Though the delay of each bit could be dependent on its polarity, many circuits that generate dual-monotonic pairs (such as for a sum bit in a carry-save adder) exhibit little bitwise data dependence because they are symmetric. When the assumption is justified, it is therefore reasonable to form a bus completion detector that examines only the two ends of a datapath, namely, the signals from the bit slices farthest apart physically on a chip. Such a completion detector reduces the tree of C-elements across all the bits to just a simple two-input C-element combining the two end bit-done signals. The delay of the completion detector itself usually makes the assumption quite safe because the sum of the delays through the NOR gates, the single two-input C-element, and any buffers that follow it in the control logic are usually much greater than the difference in delay between the end bits and the unexamined bits, even if some are of the opposite polarity.

Going one step further, the tree of C-elements can be eliminated entirely by just using one of the bit-done signals directly as the total bus done signal. This requires the stricter assumption that none of the other bits nor the wire delay across the width of the bus are slower than the observed bit by more than the delay through the NOR gate generating the bit done signal and any buffer chain following it in the control logic. In practice, even this assumption is very reasonable for bit-parallel datapaths because it takes so long for the control logic to drive any signal in response to the completion detector. Indeed, incorrect operation could occur only if the control logic could change some signal that removed the driving signals for one of the unexamined bits before it had finished transitioning to its correct value.

If the cross-section of the datapath tapped by the completion detector is not entirely composed of dual-monotonic pairs generated in parallel in a similar manner, then the above assumptions should be applied only to the groups within which the pairs are similar. For example, if the datapath contains three separate fields that are computed differently or get their data from different sources, then the assumptions allowing just one bit pair to be examined should be applied to each field separately and the group done signals from the three fields then combined with a three-input C-element to form the total done signal.

5.5 Asymmetric Control Connections

Another reasonable way to reduce the local cycle time constraint is to allow control logic to be asymmetric with respect to evaluation and reset. The symmetric control in the stage configurations suggested in Chapter 2 had the advantage of automatically enforcing the completion of both transition directions, but this section discusses cases in which it can be appropriate to use other configurations.

When precharged function blocks use the full-controlled precharge logic style (defined in Figure 2.2), the precharge signal takes precedence over the data inputs. This means that when control logic applies the precharge signal to a block, the block resets, even if its data *inputs* become active. The control logic therefore needs not wait for the block's outputs to reset before allowing the *preceding* block to evaluate; rather, it is sufficient to allow the preceding block to evaluate as soon as the precharge signal is *applied* to a block. When the **PC** stage configuration family is modified to enact this change, it forms a new family of configurations, called the **PG** family. The control logic in Figure 5.4 shows a **PG0** stage configuration, which applies the modification to the control logic of the

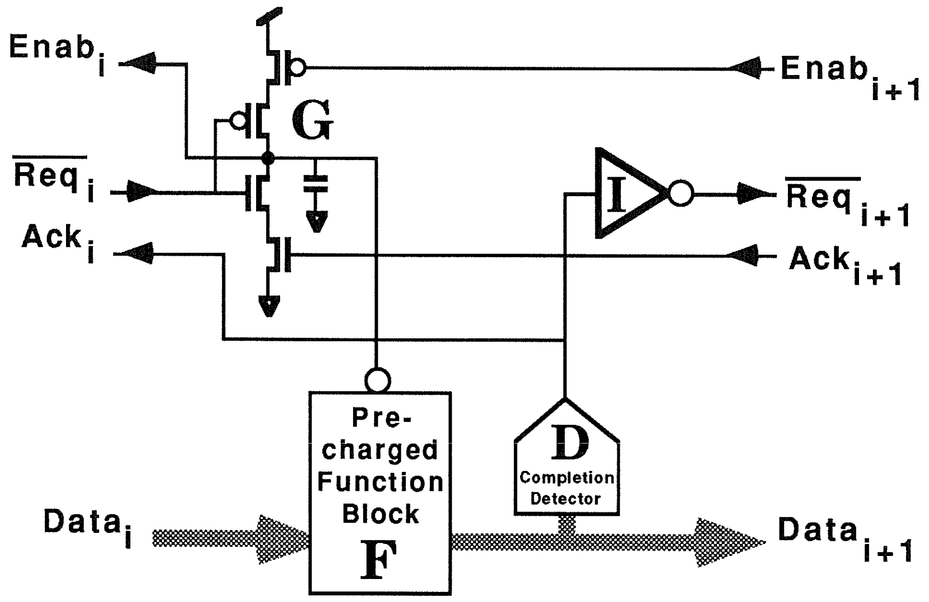


Figure 5.4: The **PG0** stage configuration style is an allowable modification of the **PC0** style when the **F** function block uses full-controlled precharge logic.

PC0 configuration. The four explicitly drawn transistors form an asymmetric or **G**eneralized C-element [MART86] that generates the precharge signal for each block. The function block in each stage is enabled for evaluation when its active-low precharge signal is high. The modification does not affect the forward latency but removes the function block resetting delay terms from both the reverse latency and cycle time. The lessening of the reverse latency can significantly improve the performance of a ring operating in the **Bubble-Limited region**. Like the **PC** family on which it is based, the **PG** family of configurations is still speed-independent.

The same modification idea can be applied to the **PS** family. Since the **PS** family already requires the assumption that each stage's predecessor

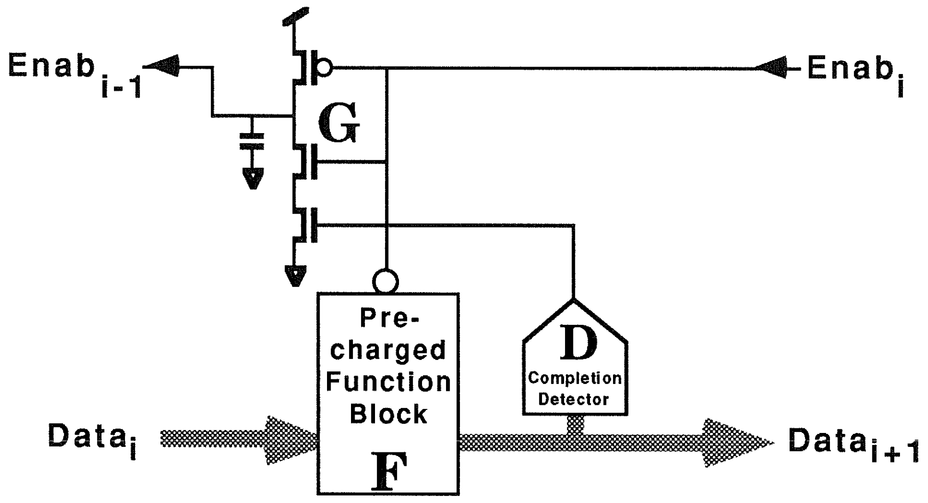


Figure 5.5: The **PA0** configuration modifies the **PS0** configuration to enable the preceding block's evaluation as soon as the reset signal is applied to a function block.

resets no slower than the successor evaluates, it is reasonable to assume that each stage's successor will also reset no slower than the stage itself can evaluate. For a self-timed ring of identical stages, these assumptions about a stage's neighbors are, in fact, the same assumption. Thus, having the control logic explicitly check for the completion of resetting in each stage's successor is unnecessary since it does not make the configuration any more insensitive to delays. The modification of the control logic to remove this reset checking in the **PS** family forms the **PA** configuration family. Figure 5.5 shows the **PA0** configuration, which requires adding a Generalized C-element to the control logic of each stage. This gate enables each function block for evaluation as soon as the precharge signal is *applied* to the successor instead of waiting for the completion detector to detect the reset outputs from the successor's function block. Since the Generalized C-element is not in the forward path, the forward latency remains

Config	Class	Cycle Time Coefficients, P				Forward Latency Coefficients, L_f			Reverse Latency Coefficients, L_r			
		$t_{F\uparrow}$	$t_{F\downarrow}$	t_C	t_D	$t_{F\uparrow}$	$t_{C\uparrow}$	$t_{D\downarrow}$	$t_{F\uparrow}$	$t_{F\downarrow}$	t_C	t_D
PG0	SI	3	0	4	3	1	1	1	0.5	0	1	0.5
PG1	SI	2	0	4	3	1	2	1	0.5	0	1.5	1
PG2	SI	1	0	4	3	1	3	1	0.5	0	2.5	2
PA0		3	0	2	1	1	0	0	0.5	0	1	0.5
PA1		2	0	2	1	1	1	0	0.5	0	1.5	1
PA2		1	0	2	1	1	2	0	0.5	0	2.5	2

Table 5.1: Coefficients of equations for the precharged stage configurations modified to use asymmetric control that avoids delays from waiting for function block resetting.

unchanged, but the delay of that gate does add new terms to the reverse latency and cycle time. However, the predominant effect of the modification is the removal of the delay terms that came from the resetting of the function block and the detection of this event's completion.

The new coefficients of the delay equations for both the **PG** and **PA** families are summarized in Table 5.1, which includes the extension of the modification to the family members that have explicit latches. The table assumes that the delays of the asymmetric Generalized C-elements are close to ordinary C-element delays and therefore counts their delays in the t_C columns. The most significant change from Table 3.1 is the removal of all of the delay terms from the resetting of function blocks.

Each member of the **PG** family has better performance than the corresponding member of the **PC** family, and hence the **PG** member should always replace the **PC** member in an implementation that uses full-controlled precharge blocks. The **PA1** and **PA2** configurations have better reverse latency and cycle time than **PS1** and **PS2** and therefore would likely be preferred in any implementation where it can be assumed that stages reset faster than their neighbors evaluate. However, the cycle time of **PA0** may be worse than that of **PS0**, and therefore **PA0** will only ensure a performance improvement for a ring in the **Bubble-Limited** region. Because it is operating in the **Bubble-Limited** region, the use of **PA0** in the self-timed ring in [WILL87] was a key feature enabling it to achieve some parallelism with only three stages.

5.6 Moving Logic and Sizing Transistors Based on Data Probabilities

In addition to changes in the completion detectors or control logic, self-timed ring performance can also be increased by speeding up the forward evaluation of the function block itself. One method for improving the function block speed is the adjustment of logic or transistor sizes according to expected data probabilities. Because dual-monotonic encodings contain embedded completion information, subsequent computations may begin as soon as each data token arrives. Since processing times for different tokens may differ and the tokens may have a non-uniform distribution, the real desired goal is to minimize the total expected value of delay. In cases where data values come distributed with equal probability, the "expected value" is, of course, minimized when the average delay is minimized. However, in cases where a designer knows that data values will have a particular distribution, this information can be

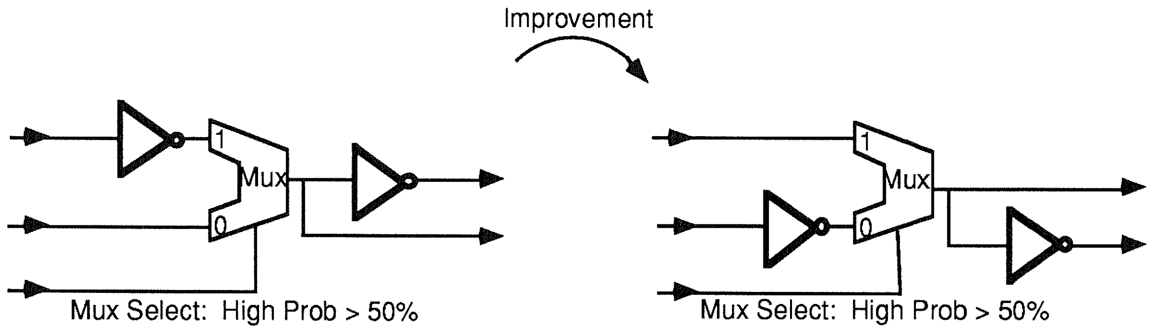


Figure 5.6: The expected value of total delay will be improved by positioning inversions according to known probabilities about choosing alternative branches.

used to minimize the total expected value of delay, making it less than the simple average of all data value delays.

The most important case in which known statistics may improve performance is when branching paths are chosen with different probabilities. Paths known to have higher-than-average use can be made faster by shortening the number of logic blocks they contain or by widening their transistors to make the blocks evaluate faster. An example of the first method for paths that merge is shown in Figure 5.6, where a net improvement in the expected value of delay results when a designer inverts both arms of a multiplexor. The expected value of delay improves because the change removes an inverter from the arm known to be more frequently chosen, even though it adds an inverter to the other arm. The second method applies to paths that fork. When some output of a block must be loaded with transistors branching into two different paths, narrowing the transistors in the infrequently chosen path will slow transitions along that path, but might result in an overall improvement in expected delay because the output node that was also part of the frequently chosen path will be faster due to less loading.

Particular arithmetic problems may have numerical characteristics that produce a non-uniform probabilistic distribution of values on internal signals even when the inputs to the overall problem are uniform. Taking advantage of these statistics will speed the overall result because the longest total delay can be made less than the sum of all the internal longest delays. Unlike in synchronously clocked circuit designs, it is not an objective in self-timed design to equalize all path lengths within a block because it is possible to take advantage of data-dependent variances in delay.

5.7 Summary

Several methods can be used to increase the performance of self-timed rings. These methods can make the **Zero-Overhead constraint** easier to meet, or can reduce the area of a circuit already meeting the constraint. They also can provide additional delay-variance margin to ensure that differences between the nominal delay or environmental parameters and their actual values do not violate the constraint and decrease performance.

Many self-timed rings could apply all of the ideas suggested in this chapter. Adjusting the function block grain size is a fundamental method resulting from the analysis of the previous chapters. The ideas in Sections 5.3, 5.4, and 5.5 are important individual improvements with wide applicability. Any function blocks composed of an internal domino chain should have the completion detector immediately after the first block in the chain. Completion detectors can tap off just a single bit, if it is possible to determine the single slowest bit at design time. The **PG** and **PA** configurations allow precharged stage configurations to improve performance by avoiding delays from the resetting of function blocks. Known data probabilities should be used to move logic or adjust transistor sizing to minimize the total expected delay.

To use these methods successfully requires careful thought and sometimes verification of specific assumptions. A designer can apply the ideas to achieve the desired constraints and to attain the best performance in self-timed pipelines or rings. The next chapter gives examples of how real implementations used each of the improvements.

Chapter 6

Division Chip Implementations

Floating-point division is an application that can benefit from the self-timed ring structures analyzed in the previous chapters. Division is generally a hard problem, and the algorithms that are exact take time linear with respect to the number of quotient bits. These algorithms require the iterative solution of repeating steps. The steps can be mapped onto the stages of a self-timed ring, which allows computation to iterate without limitations from external control or clock signals. I designed two chips to demonstrate self-timed ring implementations of division. The first chip [WILL87] was designed early in my research, before overhead and performance issues were well understood. The second chip [WILL91a] was designed with the benefit of the analysis methods presented in this thesis, and it demonstrates a **Zero-Overhead Minimal-Latency** self-timed ring.

This chapter discusses the features of both chip designs and compares their performance. Section 6.1 gives background on a division algorithm that uses redundant arithmetic and shows a block diagram of a self-timed ring structure to implement that algorithm. I modified the dataflow of the algorithm by overlapping the execution of neighboring stages to improve performance in the second chip design. Section 6.2 explains how the self-timed approach allows this overlapping to choose dynamically the shortest critical paths and analyzes the performance benefit gained. The performance analysis is extended in Section 6.3 by comparing different radix choices, with and without the overlapped execution feature. Section 6.4 then presents test results and measurements from fabricated chips implementing the two designs. These results are studied in Section 6.5 in terms of the ring performance regions in Chapter 4 and the performance enhancement methods in Chapter 5. Another performance enhancement, described in Section 6.6, is the quotient done-detection in the second chip design that can terminate iterations early for repeating fractions. Finally, Section 6.7 summarizes the comparisons between the two designs and shows the benefits of applying the methodology of **Minimal-Latency** self-timed ring design reported in this thesis.

6.1 Redundant SRT Algorithm

To meet the IEEE floating-point standard for double-precision operands, a divider must take operands and generate a quotient, each with a mantissa precision of 53 bits. A division algorithm finds a quotient either by a straightforward successive determination of quotient digits, or by other schemes, such as a Newton-Raphson recurrence, that use large multipliers to successively refine approximations to the final quotient. Hybrid schemes are also possible [MATU90]. Since Section 5.1 suggested that a small grain size is best for self-timed rings, the straightforward division methods are best suited for implementation in a self-timed ring.

The straightforward methods for division determine quotient bits sequentially, starting with the most significant digits and progressing to the least significant. The algorithm is thus broken into steps, where each step chooses a quotient digit and computes the next partial remainder based on the previous partial remainder and quotient digit. Between each step, the partial remainder is shifted left to scale it by r, the base or radix of the algorithm. Each step thus implements

$$R_{i+1} = rR_i - Dq_i \tag{6.1}$$

where R_i is the partial remainder output from stage i, r is the radix, q_i is the quotient digit determined from stage i, and D is the divisor. The sequence is initialized by setting R_0 so that rR_0 is equal to the dividend.

In ordinary division, like the method learned in "grade-school," the quotient digits q_i are in the set $\{0, ..., r\text{-}1\}$, and the full quotient has only a single valid representation since each digit position in the quotient has only a unique correct value. Determining the correct digit at each position requires comparison of the exact partial remainder, and this means the entire partial remainder must be computed before each quotient digit can

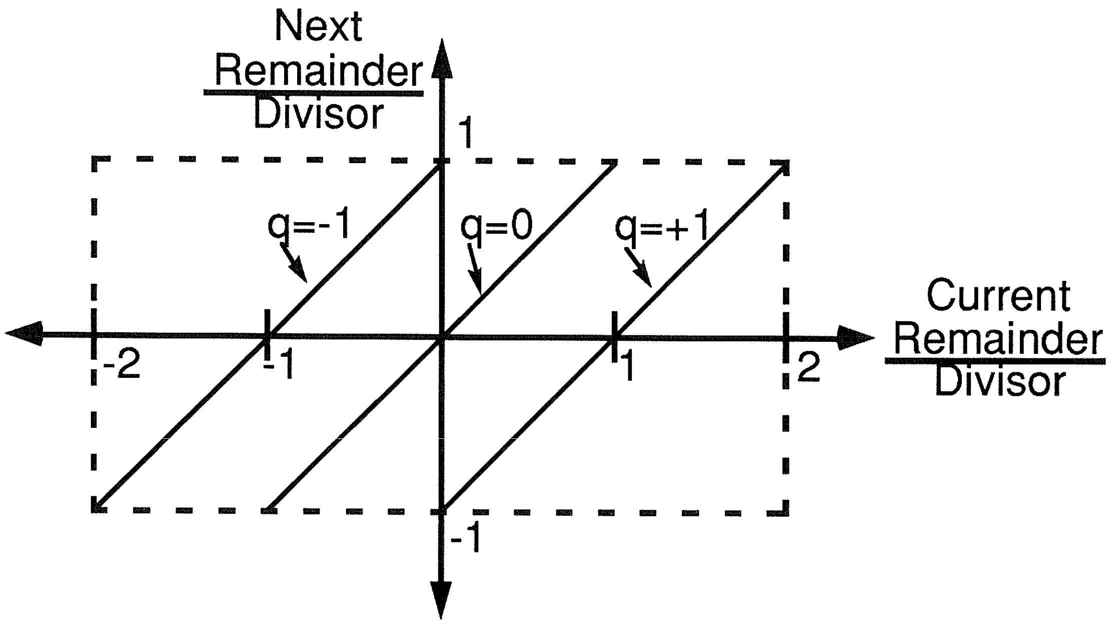


Figure 6.1: The division diagram for Radix 2 SRT division shows that, for every possible remainder value, a quotient digit can be chosen that is valid for a range around that remainder value.

be determined. In hardware, this computation requires a complete carry-propagate subtract, which is a slow operation for the long operand lengths used in floating-point arithmetic.

An alternate division algorithm, called SRT (Sweeny, Robertson, Tochner) division [ROBE58], avoids the complete carry propagation in each step by making the set of valid quotient digits redundant by including both positive and negative integers. The valid quotient digit set is $\{-\rho, ..., 0, ..., \rho\}$, where ρ is an integer chosen in the range $\frac{r}{2} \le \rho \le r-1$. With redundant quotient digit sets, the final quotient result can be represented in several different ways, which means that there is a choice of quotient digits for each position. This choice is illustrated graphically by the overlapping of the quotient lines in the division diagram of Figure 6.1.

A circuit can convert the redundant quotient representation to an ordinary binary irredundant representation by subtracting the positionally weighted negative digits from the positionally weighted positive digits. This subtraction requires a carry propagation, but it is a single operation, which needs to be performed only once for the whole division operation rather than once per quotient digit step. Furthermore, in an integrated floating-point chip, this full-width carry-propagate operation could be performed by shipping the redundant quotient results to a separate part of the chip that implements fast carry-look-ahead addition.

Since in SRT division the quotient set contains digits of both signs, the quotient selection logic for a given position need only use an approximation of the divisor and partial remainder. This is because small errors may be corrected at a later stage with quotient digits of less significance and opposite sign. Because the algorithm requires only an approximation of the partial remainder at each stage for the selection of a quotient digit, only a small number of the most significant bits of the partial remainder need to be examined. Choosing the number of bits to examine involves some tradeoffs, which are fully discussed in [WILL86]. The redundant representation of the examined bits needs to be resolved with a short carry-propagate adder before the remainder approximation is used by the quotient selection logic. Though this adder would normally need to be 4 bits wide for radix 2 division, 3 bits are sufficient when the quotient selection logic also gets information from the previous quotient digit selection [WILL91b]. All of the other bits in the remainder need only

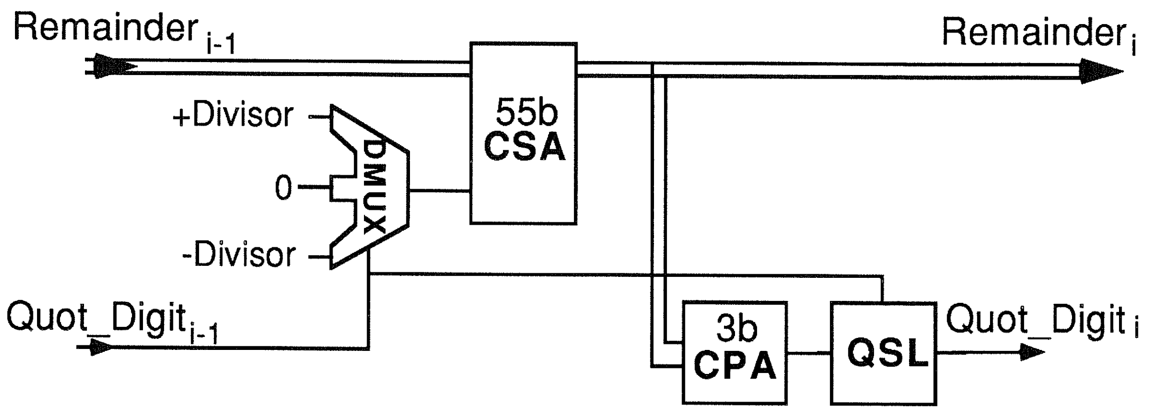


Figure 6.2: Ordinary dataflow of a radix 2 SRT division step with the required bit widths for IEEE double-precision operands.

be computed with a carry-save adder. Together, the hardware dataflow of a step of the SRT division algorithm is shown in Figure 6.2.

Because the steps of SRT division need a repetitive structure, a self-timed ring can implement the steps directly by mapping a small number of steps onto the stages of a ring. As was hinted in Figure 1.5, one can achieve a **Minimal-Latency** division by directly concatenating **PS0** style configuration stages into a self-timed ring using precharged logic blocks with no explicit latches.

6.2 Dynamic-path-ordering Overlapped Execution

It is possible for the computation within each stage of the division algorithm to be partially overlapped with the computation in the preceding stage by modifying the dataflow of each step as shown in Figure 6.3. The overlapped execution speeds up the overall computation by allowing additional parallelism. The carry-propagate adders are replicated for each possible quotient digit so they can begin operation before the actual quotient digit arrives and chooses the correct branch. Two of the three

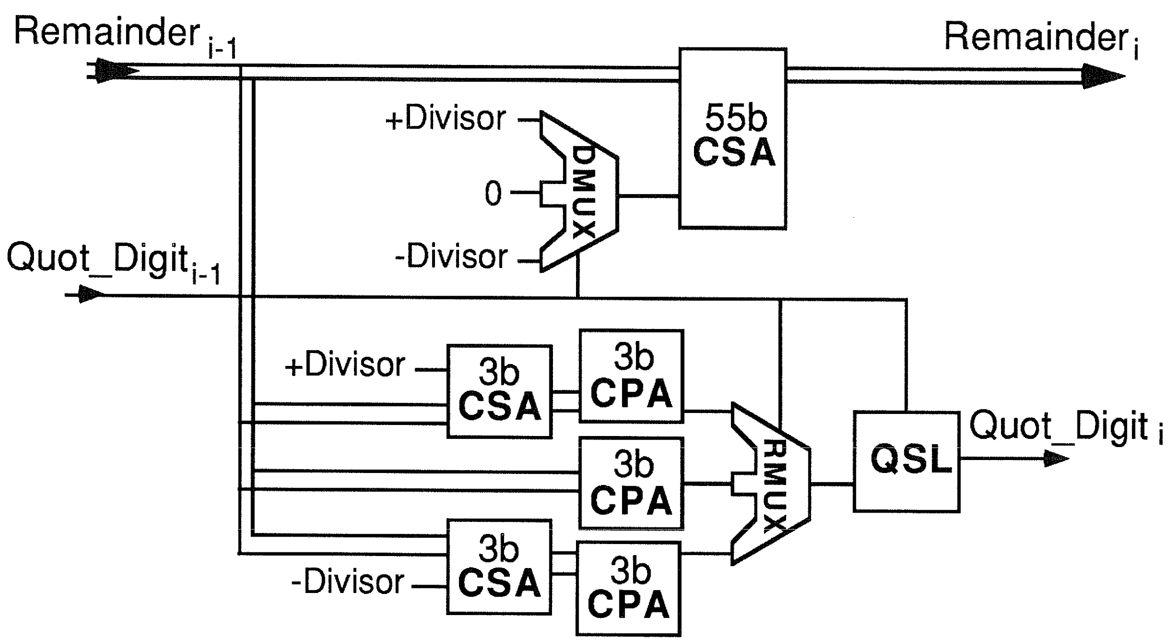


Figure 6.3: Dataflow of a stage with overlapped execution.

carry-propagate adders are preceded by short carry-save adders: one to combine the remainder with the divisor, and the other to combine the remainder with the negation of the divisor. Actually, these two carry-save adders share the circuit generating the sum terms because the dual-monotonic data convention already provides both the true and complement of each bit. The carry terms cannot be shared.

Overlapped execution allows the partial 3-bit carry-save and carry-propagate adders for the remainder formation in each stage to operate in parallel with the previous stage's quotient digit selection and the stage's own divisor multiplexor and full 55-bit carry-save adder. The dataflow for a pair of stages with symmetric overlapped execution is shown in Figure 6.4. The overlapping of execution allows the average delay through a stage to be the average instead of the sum of the propagation

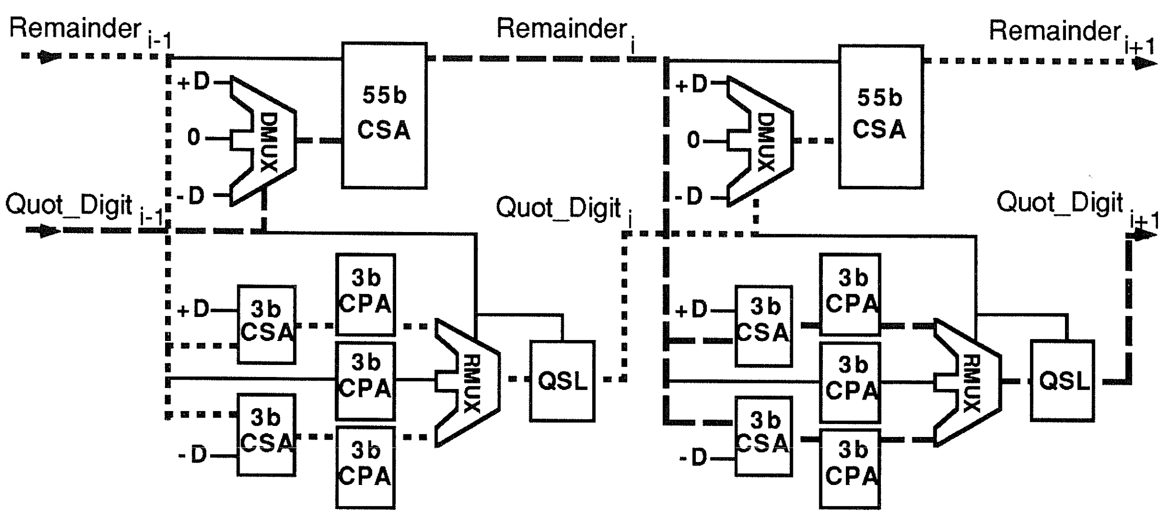


Figure 6.4: The dataflow through a pair of stages in the symmetric overlapped execution scheme has two possible critical paths, highlighted by the dashed and dotted lines.

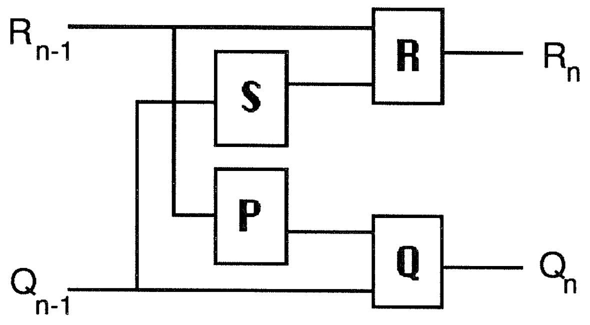


Figure 6.5: Model for the overlapped execution dataflow in each stage. The variables P, Q, R, and S are the delays of the abstracted blocks.

delays through the remainder and quotient digit selection paths. The overlapping of these paths can be abstracted to the arrangement of overlapping blocks shown in Figure 6.5. When these blocks are self-timed and therefore operate as soon as their required operands arrive, the average delay per stage[4] in a chain of identical stages of the overlapped arrangement can be simplified to:

$$\frac{1}{2}\left\{ P+Q+R+S + \max[0, \mathrm{abs}(R-Q)-(P+S)] \right\}.$$

The last term is usually negative and drops out, giving a performance increase due to the factor of $\frac{1}{2}$ in front. In the overlapping of the stages for SRT division, the delay of block P in the quotient selection path is the largest of the delays because it contains the carry-propagate adders. The overlapping reduces by one-half the effect of the delay in block P on the total delay. As will be shown in Section 6.3, when the added multiplexor delays are taken into account, the overlapping of the stages in a radix 2 design increases performance by 40% over a standard sequential arrangement of the same blocks in the same technology.

The structure of this overlapping scheme results in a data wavefront that leapfrogs down the succession of stages. If the critical path goes

[4] An exact analysis using induction on the delays of Figure 6.5 shows that the time the nth R block finishes in a chain of identical stages is

$$R_n = \frac{n}{2}(P+Q+R+S) +$$
$$\max\left[S+\frac{n}{2}(R-S-P-Q),\ \left(\frac{n}{2} - 1\right)(Q-P-R-S),\ \frac{e}{2}(Q-P-R-S),\ R-P-Q+\frac{e}{2}(Q+P-R+S) \right]$$

when the chain's inputs start at $R_0 = Q_0 = 0$ and where $e \equiv \left\{ \begin{array}{l} 1 \text{ if n odd} \\ 0 \text{ if n even} \end{array} \right\}$

Symmetrically, the nth Q block finishes at time

$$Q_n = \frac{n}{2}(P+Q+R+S) +$$
$$\max\left[P+\frac{n}{2}(Q-P-R-S),\ \left(\frac{n}{2} - 1\right)(R-S-P-Q),\ \frac{e}{2}(R-S-P-Q),\ Q-R-S+\frac{e}{2}(R+S+P-Q) \right].$$

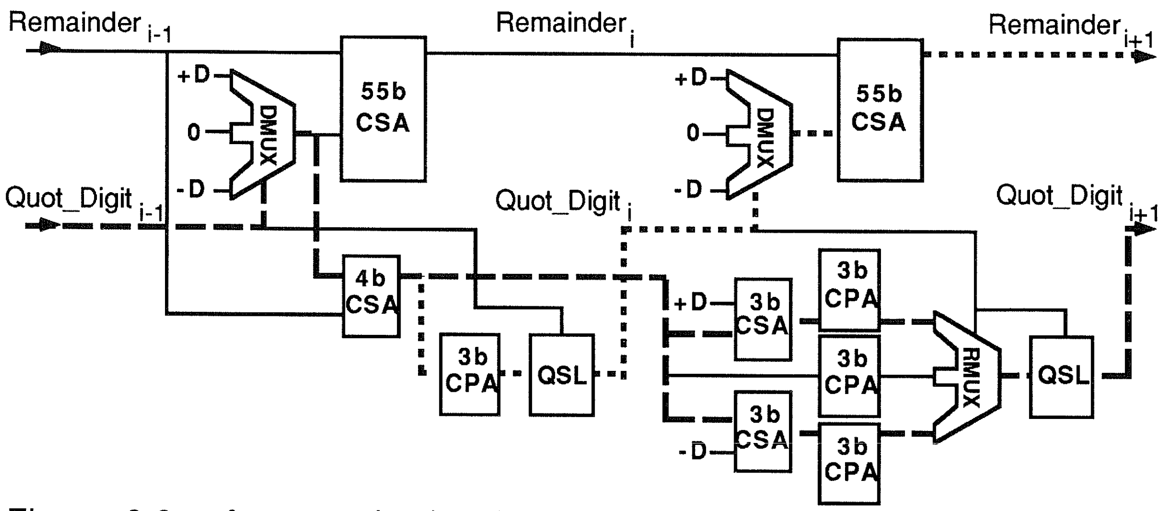


Figure 6.6: Asymmetric dataflow through a pair of stages in previous overlapped execution schemes enforced a specific grouping of the stages into pairs.

through the quotient selection path in one stage, it will likely go through the partial remainder path in the next stage, and vice-versa. However, data-dependent variances in delays make it sometimes possible for the overall minimal critical path to go through the same path in two adjacent stages. Delay variances arise because of the varying number of bits propagated in the carry chains, the occurrence of some zero quotient digits, and the cases in which a negative quotient digit can be selected in advance when a single stage can determine two quotient digits. The self-timing of the datapath always ensures data flows through the minimal critical path. Previous overlapped execution schemes, such as the one from [TAYL85] shown in Figure 6.6, have pre-grouped stages into pairs. Whereas the present approach makes all the stages symmetric, the scheme in Figure 6.6 does not replicate the first carry-propagate adder. This lack of symmetry makes the critical path go through the same blocks every time. Such a grouping adds delay because it enforces extra waiting in some cases and does not achieve the additional minimization possible with self-timed overlapped execution, which allows a “dynamic” adjustment of the execution order.

The leapfrog progression down the series of stage pairs in the symmetric overlapped execution scheme also makes specific use of the self-timed nature of each stage because the function blocks can begin operation as soon as the input they need arrives, without waiting for the other input. If clocked latches were placed at the edges of the blocks shown in Figure 6.4, then the clock period would have to be slow enough so each pair of stages could always wait for both inputs to arrive.

6.3 Choice of Radix

Redundant SRT algorithms can be formulated for radix 2, radix 4, or even radix 8 [ATKI68, FADR89]. Compositions of SRT steps can be grouped together and collectively denoted by even higher radices [TAYL85]. In synchronously clocked circuits, higher radix arithmetic can achieve more computation in a fixed clock cycle, allowing performance to be improved without the use of faster clocks. Self-timed circuits, however, are not constrained by fixed clock cycles, and the lower radices can be computed more quickly. Self-timing therefore provides a useful means of comparing the performance of different radix approaches without clocking constraints, or the delays from latches or input/output considerations. To make fair relative comparisons, I use the delay of a gate with unity fan-in and unity fan-out as a basic speed unit. Real gates

Radix & Style (OverExec means with Overlapped Execution)	Average Critical Path per Pair of Stages in Unity fan-in, Unity fan-out gate delays	Unity fan in/out Gate Delays per Quot Bit	Latency for 54 bits with 250pS unit gate delays	Silicon Area in 1.2μ Process
Radix 2	2 (CPA3+QSL3+DMUX3+CSA55) 2 (5.7 + 3.8 + 2.8 + 4.5) = 33.6	16.8	225 nS	7 mm^2
Radix 2, OverExec	CSA3+CPA3+RMUX3+QSL3+ DMUX3+CSA55 3.9 + 4.9 + 3.5 + 3.8 + 2.8 + 4.7 = 23.6	11.8	160 nS	10 mm^2
Radix 4	2 (CPA7+QSL5+DMUX5+CSA56) 2 (11 + 16 + 3.5 + 4.5) = 70.0	17.5	235 nS	12 mm^2
Radix 4, OverExec	CSA7+CPA7+RMUX5+QSL5+ DMUX5+CSA56 3.9 + 10 + 5.0+ 16 + 3.5 + 6.2 = 44.6	11.2	150 nS	18 mm^2

Table 6.1: Tradeoffs in speed and area for different implementation algorithms mapped onto self-timed rings.

have delays many times this basic unit because of stacked transistors (fan-in) and loading (fan-out). The driving of control lines and buffers associated with the various functional units is included in their delays.

During the chip designs, I estimated parameters for SRT radix 2 and radix 4 choices, with and without overlapped execution. The estimates were based on simulations, and then updated and calibrated with measurements from the fabricated chips. Table 6.1 summarizes the comparisons of the implementation choices. All of the parameters are for an implementation with a **Minimal-Latency** self-timed ring. The radix 2 designs require five stages in the ring to meet the **Zero-Overhead constraint**, while the radix 4 designs require only four stages since their $\frac{L_f}{P}$ ratio is larger. The radix 4 function blocks are larger because the carry-propagate adders must be 7 bits long instead of 3 bits, and the quotient digit selection logic must include the delay for a 39 product-term PLA [TAYL81]. None of the figures include the final 54-bit carry-look-ahead adder required for rounding and converting the redundant representations back to a standard binary format. The right two columns contain numbers specific to the CMOS fabrication technology available at the time of fabrication of the second chip.

Table 6.1 shows that radix 2 is slightly better than radix 4 when neither have overlapped execution. This is because the additional complexity of the radix 4 quotient selection does not quite justify the use of radix 4 when clocking does not need to be considered. However, if the design were clocked, the difficulty of supplying a clock at twice the frequency might make radix 4 preferable. Overlapped execution in either radix 2 or radix 4 gives a significant performance increase, about 40% for radix 2 and 55% for radix 4. The key advantage of the self-timed overlapped execution style is that the average critical path per stage has only a factor of $\frac{1}{2}$ times the delay from the carry-propagate adders and quotient selection logic. Since, for higher radices, these components occupy bigger proportions of the total delay, the effect of overlapped execution is more significant for radix 4. To summarize, with overlapped execution, radix 4 will be faster than radix 2, but the area cost is much higher because of the replication of the carry-propagate adders. Not only are five adders required instead of only three, but they are also larger. Still higher radices, such as radix 8, would accentuate these tradeoff effects overlapped execution would have an even greater percentage performance increase, but at a formidable cost in area.

6.4 Test Results and Measurements

The tradeoffs discussed in the previous section led to the choice of implementing radix 2 division. The first chip design used only three stages and did not have overlapped execution. The second design used five stages with overlapped execution. Both designs were laid out using MAGIC in full-custom CMOS technology, were verified using the Stanford IRSIM simulator [SALZ89], and were fabricated through MOSIS. The first design was fabricated in 2.0μ technology in two versions: a short version shown in Figure 6.7 implementing only the top 7-bit portion containing all the critical paths, and a tall version implementing the full array with an active area of 10.7 mm^2. The second design was fabricated in 1.2μ technology and is pictured in Figure 6.8. It has a core iterating ring in the central 6.8mm^2 surrounded by test registers for a total active area of 9.7mm^2. The ring's five stages are columns that are mirrored appropriately to weave the datapath and achieve equal path lengths.

Both the first and the second designs were functionally correct on first silicon. Since the designs are self-timed, their performance varies with power supply voltage and temperature. Unlike the testing of synchronous designs, where voltage and temperature testing requires repeated readjustments of a clock to determine the exact point of failure, self-timed designs are easy to test for varying voltage and temperature. For each voltage and temperature point desired, the speed of the output is simply *measured* since the chips always produce outputs as fast as possible

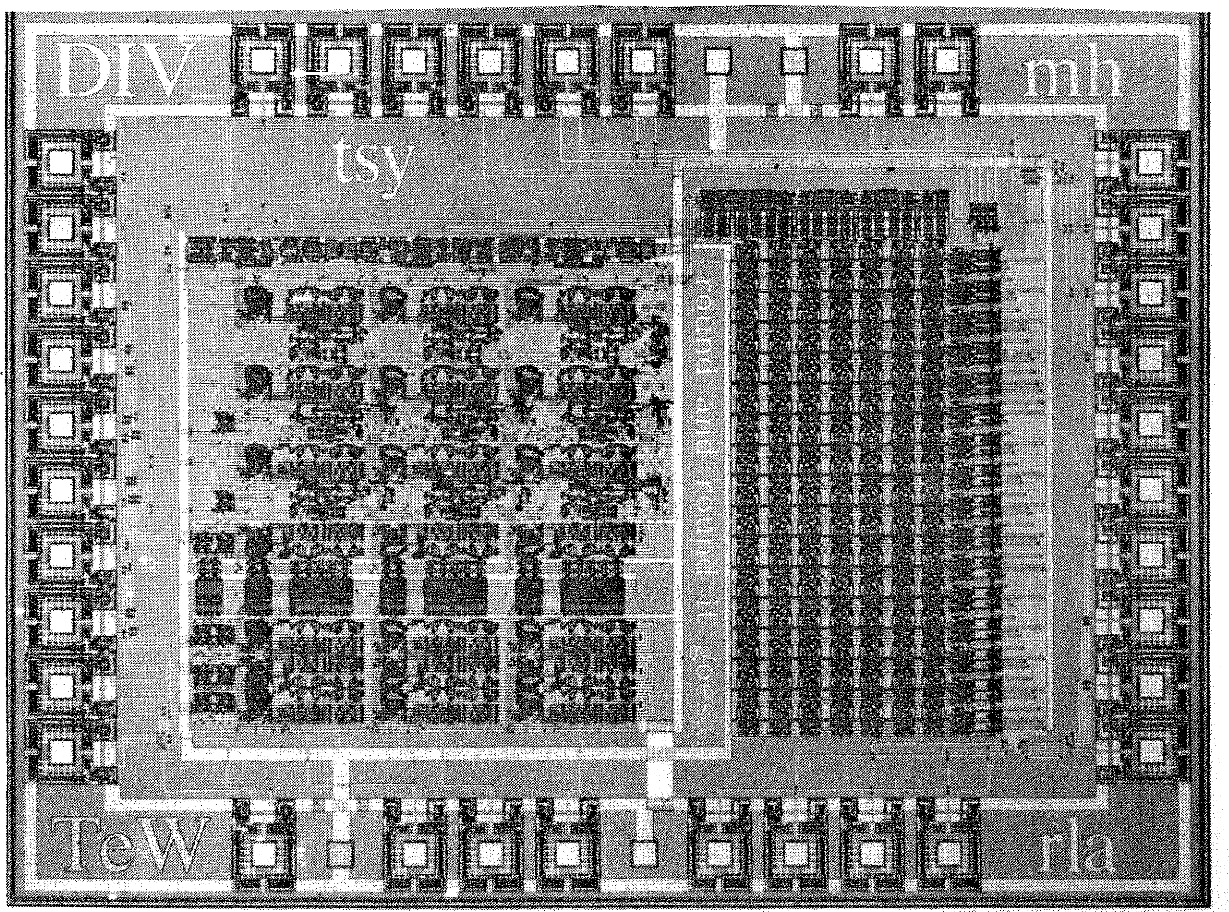


Figure 6.7: Micrograph of the first self-timed Divider in 2.0μ technology.

for the actual operating conditions. Figure 6.9 shows measured speeds for the second chip design over large ranges of voltage and temperature.

At the nominal conditions of 5V and 35°C ambient temperature, the first design had a measured performance of 13nS per quotient digit and the second design had a measured performance of 2.9nS per quotient digit. Because of a factor of two difference in the speed of the fabrication technologies, the performance improvement due to design and architecture improvements was really a factor of 2.2. The sources of this improvement are examined further in the next section.

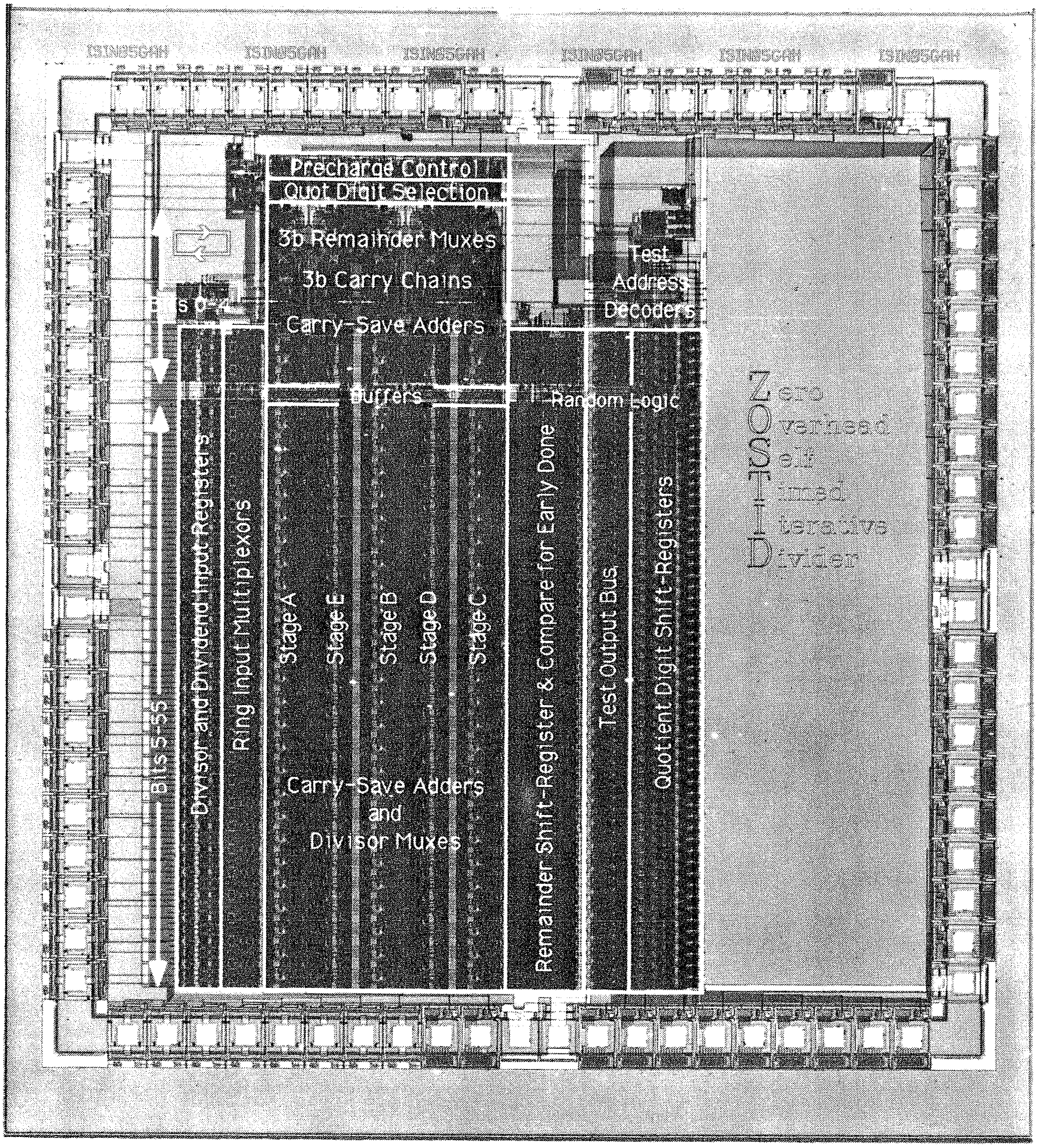


Figure 6.8: Micrograph of the Zero-Overhead Self-Timed Divider in 1.2μ technology

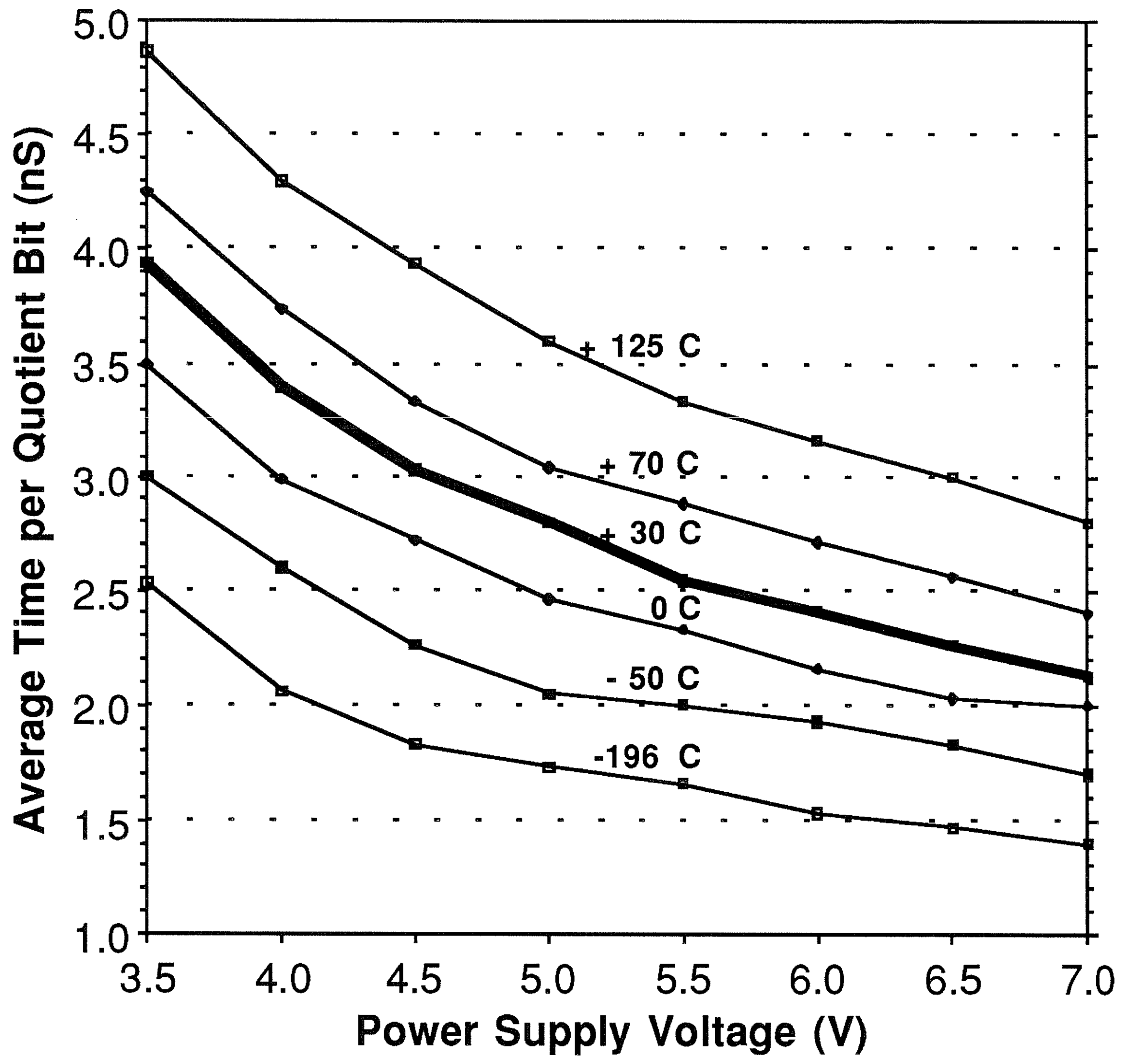


Figure 6.9: Measured performance per quotient digit in 1.2μ technology at various voltages and temperatures.

6.5 Ring Performance Comparisons

Both of the chip designs used some of the performance enhancement ideas discussed in Chapter 5. The first design used the ideas from Sections 5.3 and 5.5. It rearranged the grouping of the domino chains within each stage by moving the divisor multiplexor from the input of a function block to the output of the preceding stage. This enabled using the "shadowing" suggestion of Section 5.3 to place the completion detectors at a narrow point in the datapath, in front of the divisor multiplexors. The control logic used asymmetric Generalized C-elements to reduce the reverse latency by forming the **PA0** stage configuration suggested in Section 5.5, which removes the delay terms of the function block reset transitions. This reduction in reverse latency was essential to the first design since it operated in the **Bubble-Limited region** where the reverse latency was the limiting factor of the total performance.

The second chip design used the methods from Sections 5.1, 5.2, 5.4, and 5.6 to enhance its performance. The number of stages was increased to five, based on Section 5.1. Following the suggestions of Section 5.2, the grain size was decreased by the introduction of the two C-elements shown in Figure 6.10 that partition each stage in sections. By separately generating the control signals for the carry-propagate adders and the remainder multiplexor, the finer grain size effectively increases the total number of stages and the ring operates in the **Data-Limited region**. The completion detectors use the reasonable assumptions suggested in Section 5.4 to simplify the completion detector for the full remainder by

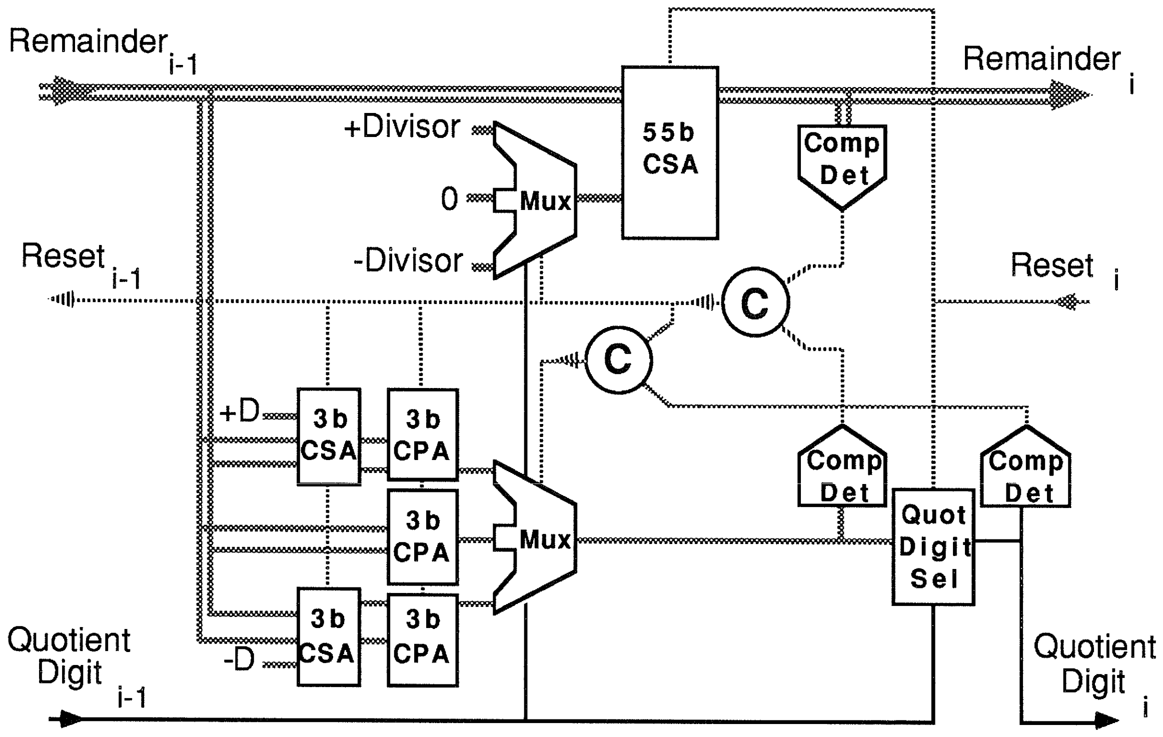


Figure 6.10: The structure within each stage has control logic that independently resets internal sections of the stage. Dotted lines are control signals; shaded lines are dual-rail monotonic datapaths.

examining just a single bit in the datapath. Finally, as suggested in Section 5.6, the transistor sizes on the arms of the replicated short carry-save adders were adjusted to take advantage of a non-uniform known distribution of quotient digits [WILL91b].

Both of the self-timed divider designs use only a single token because low total latency, not high throughput, was the important goal. The most fundamental difference between the two self-timed divider chips is the points at which they operate in terms of the regions of analysis in Section 4.2. Since the first design operates in the **Bubble-Limited** region, it has performance limited to $\lambda = 2GL_r$. Because that design uses completion detector shadowing, the forward latency is composed of two parts, $L_f = t_{F_A}\uparrow + t_{F_B}\uparrow$. The second part, containing the divisor multiple multiplexors, does not contribute to the reverse latency since its execution overlaps with the completion detector and control logic. The **PA0** configuration therefore has $L_r = \frac{t_{F_A}\uparrow + t_D\downarrow}{2} + t_A$, where t_A is the delay of the Generalized C-element used in the control of each stage. The total measured latency is thus $\lambda = G(t_{F_A}\uparrow + t_D\downarrow + 2t_A)$, where G is the number of quotient bits accumulated. The $t_D\downarrow + 2t_A$ terms are overhead over the combinational latency. The fraction $\frac{t_{F_A}\uparrow + t_D\downarrow + 2t_A}{t_{F_A}\uparrow + t_{F_B}\uparrow}$ is the overhead factor by which the actual latency per stage exceeds the minimal combinational latency. The measured values of the delays for the 2.0μ chips are $t_{F_A}\uparrow = 8.4\text{nS}$, $t_{F_B}\uparrow = 1.4\text{nS}$, and $t_D\downarrow + 2t_A = 4.6\text{nS}$, so the control overhead degrades performance by a factor of $\frac{13}{9.8} = 1.33$.

The second design uses **PS0** stages and operates in the **Data Limited** region. Figure 6.11 is an oscilloscope trace illustrating how the control logic removes the precharge signal for each block, which enables the block's evaluation, 2nS before the data inputs arrive at the block. This measurement shows that the $\frac{P}{L_f}$ wavelength is equal to 4.2 and that therefore five stages are sufficient to meet the **Zero-Overhead constraint** of equation (5.1). The design is thus able to achieve a total performance of $\lambda = GL_f$, which is the same rate at which data would flow through a combinational array of the stages if the ring were unrolled. Further, since the **PS0** configuration has $L_f = t_F\uparrow$, each stage's forward latency is equal to just the combinational function block delay, and the total latency is $\lambda = Gt_F\uparrow$. So, whereas the first chip had a substantial overhead

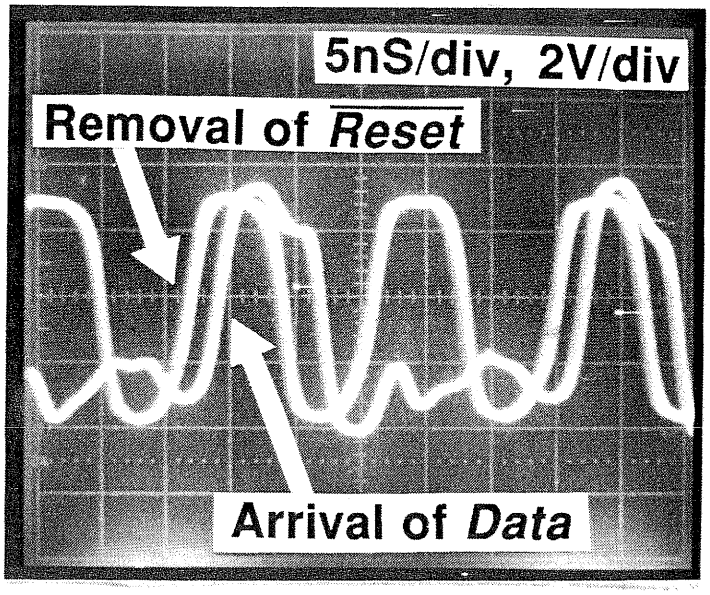


Figure 6.11: Self-timed Control logic removes each stage's Reset signal 2nS before data arrives, verifying that the control introduces zero overhead.

from the control logic, this second design achieves **Zero-Overhead** operation, for which the total latency is equal to the sum of only the combinational function block delays.

Additional enhancements in the second design reduce the value of $t_F\uparrow$ itself. The most important enhancement is the overlapped execution of the blocks within each stage. The dataflow for this structure was analyzed in Section 6.2. The extensions discussed in Section 2.6 allow the **PS0** control configuration to be modified to account for the repetitive splitting and rejoining of the datapaths, as was shown in Figure 6.10. The overlapping of execution reduces the forward latency by 40%, and the self-timing allows this entire benefit to be reflected in the total performance. The second design is also improved by better VLSI layout techniques. In particular, careful sharing of transistor drain contacts reduces the

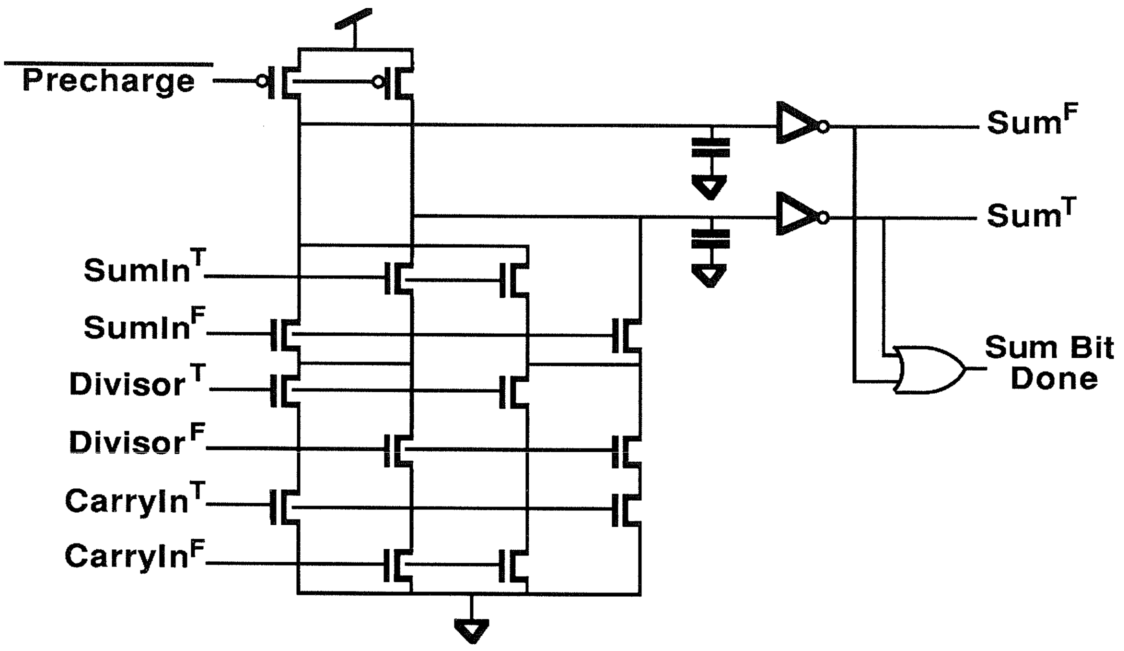


Figure 6.12: The sum logic in a bit slice of a precharged carry-save adder was improved by a layout minimizing the capacitance on the internal nodes of the pull-down network.

capacitance on the internal nodes of transistor stacks. This effect is particularly significant for the precharged blocks with complex pull-down networks, such as the basic carry-save adder sell shown in Figure 6.12. The changes to the layout and other small logic enhancements discussed in [WILL91b] provide an additional 20% improvement in the per-stage forward latency of the second chip design over the first design for worst-case data. The second design can also provide its result significantly faster for some data inputs because of the quotient digit shift register structure discussed in the next section.

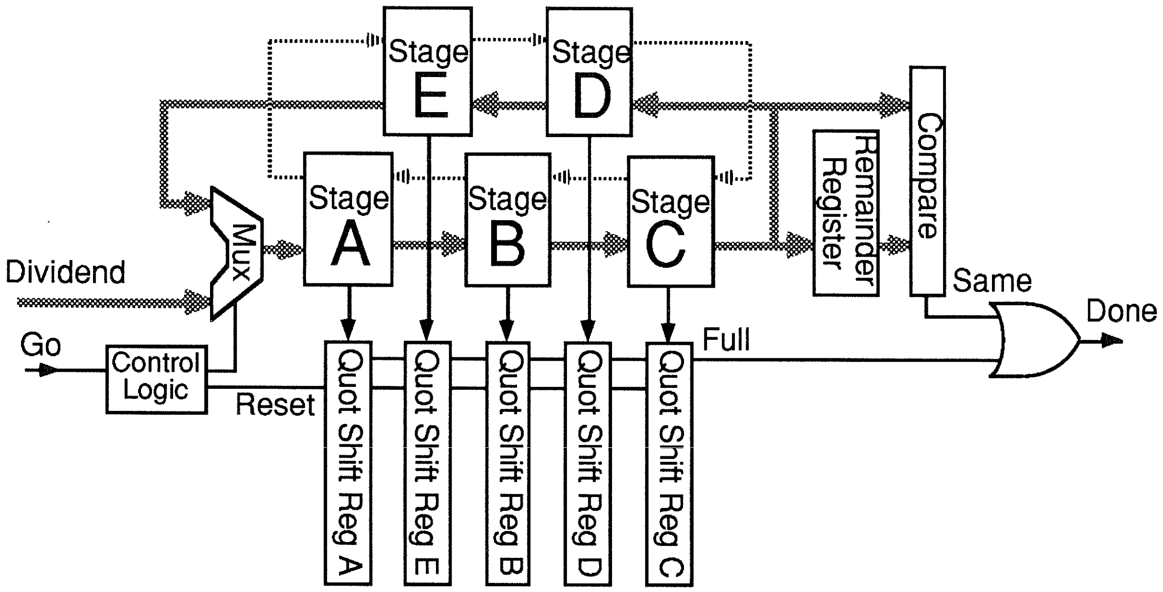
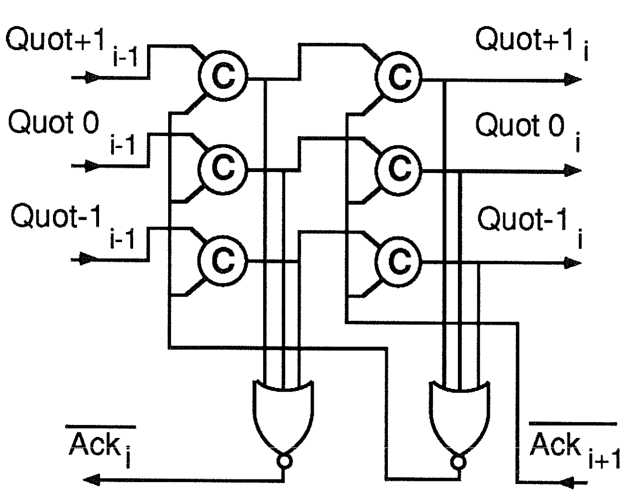


Figure 6.13: Complete block diagram of a division implementation showing the self-timed ring and quotient digit shift registers with early done detection.

6.6 Quotient Shift Registers and Done Detection

In both designs, the critical paths determining the performance are entirely within the self-timed rings. Signals generated in the ring also control shift registers that collect the quotient digits output from the ring's stages. There is one shift register for each stage, as shown in Figure 6.13 for the second design. The maximum number of loops around the ring is the number required to fill the shift registers with the desired number of quotient digits. For an IEEE double-precision result, the first (three-stage) design requires 18 iterations to generate 54 quotient bits, and the second (five-stage) design requires 11 iterations to generate 55 quotient bits.



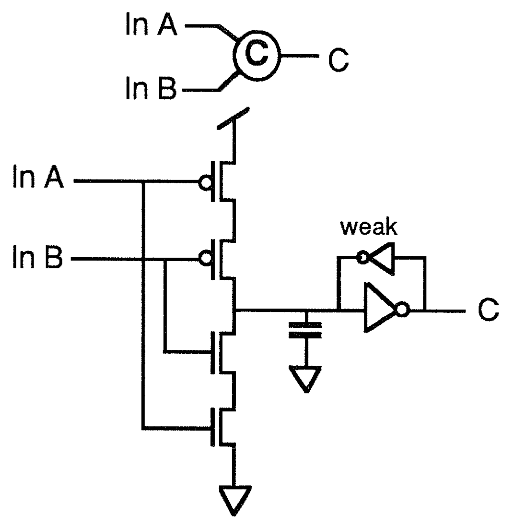


Figure 6.14: A cell of the asynchronous shift registers for capturing quotient digits on a triple-monotonic wire set. Each quotient digit arrives when one of the three input wires is set high, and is followed by a "spacer," where all three are again reset low. The C-elements are static, as defined at right.

The second design has the additional feature of detecting when it is possible to terminate the iterations early. If the remainder comparison on the right side of Figure 6.13 determines that the partial remainder has remained unchanged during the last iteration, the remainder is repeating. If the remainder repeats, then subsequent quotient digits also will repeat; hence, there is no need to compute them again, and the division-done signal can be generated early. Even when iterations terminate early, the full quotient is immediately available from the shift registers because of their asynchronous design. Each shift register cell shown in Figure 6.14 consists of two **PS0** style stages. Since this configuration has a static spread, S, equal to two, each cell needs the two stages in order to store both a single quotient digit and its corresponding reset spacer. After the

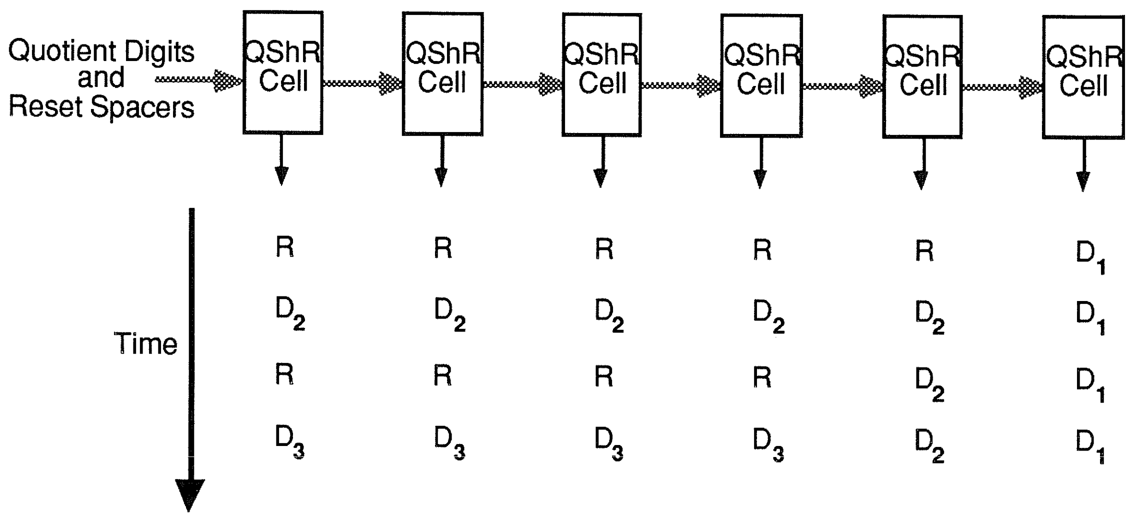


Figure 6.15: Quotient digits and reset spacers propagate through the cells of the asynchronous quotient digit shift register.

ring sends each valid quotient digit into a shift register, a reset spacer normally follows. Both the quotient digit tokens and intervening spacers ripple from the least-significant cell towards the most-significant cell in the shift register. The asynchronous handshaking allows the quotient digits to ripple to their final positions as they arrive, as illustrated in Figure 6.15, rather than waiting for a fixed number of clocks as they would in the case of a synchronous shift register.

Normally, the data and reset elements continue to pack alternately into the C-elements in the shift register cells. However, an innovation in the design of the shift registers interlocks the sending of the reset spacer elements with the remainder comparison on the right side of Figure 6.13. In the interval between when the ring sends a quotient digit into a shift register and when the ring sends the corresponding reset spacer, the cells of lesser significance (through which the most recent quotient digit token has traversed) are normally all filled with copies of the most recent digit's value. If the remainder comparison determines that the quotient will be repeating, then the reset spacer is never sent, and all the quotient digits are immediately available with their correct values from their respective cells. Only if the remainder comparison determines more iterations will be needed are the spacers sent into the shift registers to wipe out the repeated digits and prepare each shift registers to accept its next digit. The interlocking does not add any delay to the overall computation because a one-cell buffer allows it to operate in parallel with the evaluation of other quotient digits. Even if the quotient is repeating, the correct final *remainder* is immediately available because it is the same as the remainder that triggers the early termination of the iterations.

The early done detection substantially reduces the overall division latency for some data inputs. Early done detection allows computing full results for best-case data in only 45nS, requiring only 2 ring iterations, as compared with 160nS for worst-case data, requiring 11 ring iterations. Figure 6.16 shows the range of performance when only the data inputs are changed.

The effect of detecting repeating quotients and finishing early is dependent upon the distribution of input operands. Data from some algorithmic applications may be likely to have more round numbers, while data from an external input, like a sensor, may be uniformly distributed only within a limited precision. The early done detection, for example, will speedup 12% of the cases in a uniform distribution of 8-bit input operands, for a total performance improvement of 9%.

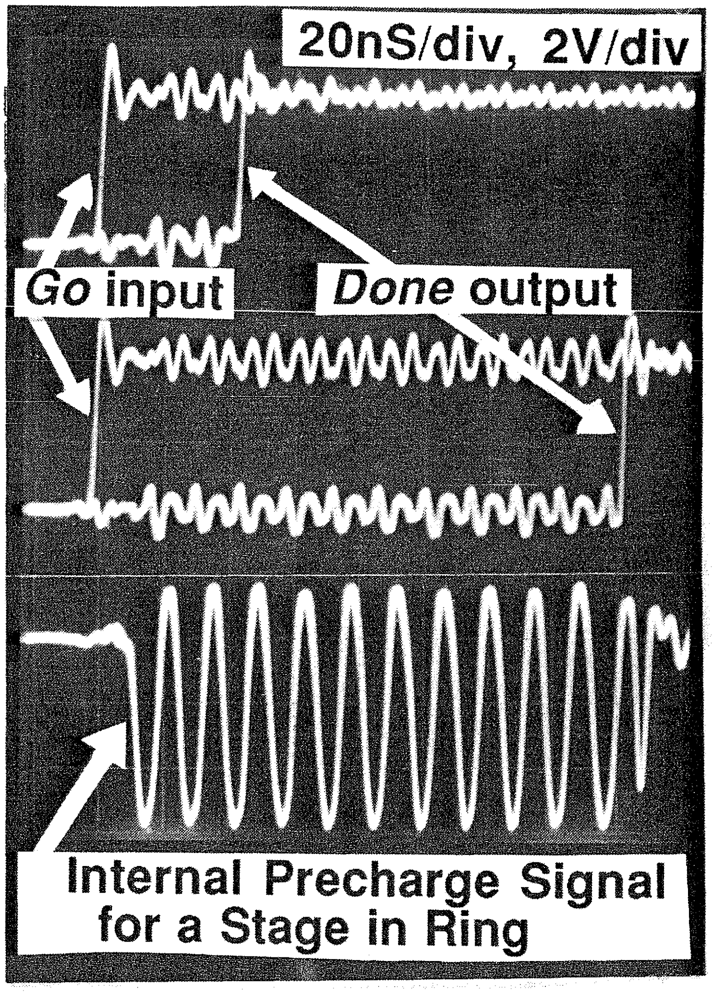


Figure 6.16: The top traces show that the total latency in the 1.2μ design varies from 45nS to 160nS depending on the data. Below is the self-timed precharge signal for one of the internal stages.

The final quotient can be rounded correctly even when the iterations terminate early. In both the early done and the normal case requiring all of the iterations, the remainder at the stage where the iterations stopped can be sent through a carry-look-ahead adder (CLA) to determine its sign. If the remainder is negative, the quotient must be decremented at the least significant bit position to which the remainder corresponds. This operation and the conversion of the redundant quotient into a standard binary form can be performed by a carry-select-adder with multiple carry chains operating in parallel. The different rounding possibilities and the remainder sign select the correct CLA output. Thus, after the iterations terminate, only a single CLA delay is required to resolve both the final remainder and rounded quotient.

6.7 Summary of Comparisons

The two self-timed divider designs provide a fair demonstration of the principles in this thesis because the designs solve the same problem with the same general structure. The most important difference in the designs is that they operate in different performance regions of the self-timed ring analysis in Chapter 4. The first design has a control overhead factor of 1.33 because it does not meet the **Zero-Overhead constraint**. The second design meets this constraint and therefore operates in the **Data-Limited region** instead of in the **Bubble-Limited region**. The second design also improved performance by a factor of 1.40 because of the overlapped execution feature, and an additional factor of 1.20 from other enhancements. Thus, there is a total improvement by a factor of (1.33)(1.40)(1.20) = 2.2 due solely to architecture improvements between the first and the second chip designs. An additional factor of two improvement due to the change from 2.0μ to 1.2μ technology explains the measured performance improvement from 13nS to 2.9nS per quotient bit between the two designs.

Chapter 7

Conclusions

7.1 Summary

In contrast to synchronous systems, self-timed components avoid the need of distributing global clocks and avoid the delay overheads associated with clock-skew. Variances and data dependencies in delays can be used advantageously because each component or section can begin operation as soon as its required operands actually arrive rather than always waiting for worst-case timing. The performance will also be the best possible for the actual environmental conditions, without the need to de-rate specifications to allow additional margins for the ranges of power supply voltage, die temperature, and fabrication spread. Because they can operate under a wide range of environmental conditions, self-timed components also are robust in the sense of not failing when conditions exceed arbitrary specifications set at design time.

Maximum benefit is attained when a self-timed component is combined with other self-timed components in an asynchronous system by using the *done* outputs of each component as the *go* inputs to succeeding components. One can also embed self-timed components within synchronous systems by using the *done* signal to stretch clock cycles as in [WOLR84], or to indicate on which clock cycle the system may take the outputs from the self-timed component.

This thesis has presented a family of basic configurations for the stages of self-timed pipelines or rings. An analysis methodology was shown that can determine local variables characterizing each stage and then use these to find the total overall performance of a pipeline or ring. For rings in particular, this analysis led to a new categorization of three possible regions of operation called **Data-Limited**, **Control-Limited**, and **Bubble-Limited**. These regions are defined in terms of the number of stages and tokens in the ring. The distinctions give direct guidelines for global design choices that satisfy the **Zero-Overhead constraint** for the best ring performance. This constraint is satisfied when a ring has at least as many stages as the number of tokens it contains times the wavelength of tokens for its particular stage configuration.

The **PS0** and **PA0** stage configurations have a forward latency equal to the function block evaluation delay alone because they contain no explicit latches. A ring meeting the **Zero-Overhead constraint** that is composed of these types of stages will have performance precisely equal to an "unrolled" combinational array of the same functional blocks while occupying only a fraction of its silicon area. This performance is possible because the control logic never enters into the critical path of the data. Rather, the self-timed control logic removes the precharge signal at each stage, enabling its evaluation, before data arrives at its input on each ring iteration. The ring thereby achieves operation with **Minimal-Latency** as defined by the latency of the pure combinational function delay.

The design of high-performance chips for floating-point division was a motivation for examining self-timed structures. The comparison of two chip designs demonstrated the performance possible with self-timing and the benefits of applying the ideas and analysis presented in this thesis. The second chip design attained **Zero-Overhead** operation and, together with overlapped execution, gained a factor of 2.2 in performance solely from architectural improvements over the first design.

7.2 Future Work

The performance analysis in this thesis was based on fixed block delays. An interesting extension would be to consider stochastic delays with specified probabilistic distributions. Such an analysis has been performed in [GREE88] for a very abstract model, closest to **PS0** in this thesis, but could be extended to the other pipeline configurations suggested in Chapter 2. Such stochastic models would more accurately reflect situations where delays are more variable or unknown. The results of a stochastic delay model will always be worse than those of a fixed delay model because the former can only introduce additional waiting and pipeline stalling in a self-timed circuit. However, except for data-dependencies, most real circuits are probably quite sufficiently modeled by fixed delays because similar blocks on the same chip have very similar characteristics.

The circuits discussed as examples in this thesis use ordinary CMOS technology, but self-timed circuit elements can also be built in other technologies, such as GaAs [WILL88] or bipolar [WILL90a]. Moreover, because self-timed circuits are not limited by the speed of external clocks, these technologies that are faster than CMOS can gain an even greater performance advantage from self-timing. Though small circuits have already been fabricated and tested, many opportunities exist for implementing larger self-timed circuits in faster technologies.

Self-timed rings could be used for many applications that require iterative computation. Within the area of elementary arithmetic functions, not only division, but also square-root, and CORDIC algorithms for computing transcendental functions, could attain high-performance with a self-timed ring implementation. Other applications for self-timed rings are likely to be found in digital signal processing, computer graphics, or microprocessor design.

References

[ATKI68] Atkins D., "Higher-Radix Division Using Estimates of the Divisor and Partial Remainder," *IEEE Trans. on Computers*, vol. 17, no. 10, pp. 925-934, Oct. 1968.

[BAKO90] Bakoglu H.B., *Circuits, Interconnections, and Packaging for VLSI*, Reading MA: Addison-Wesley, 1990.

[BURN91] Burns S., "Performance Analysis and Optimization of Asynchronous Circuits," Ph.D. Thesis, Caltech, 1991.

[CHAPI84] Chapiro D.M., "Globally-Asynchronous, Locally-Synchronous Systems," Ph.D. Thesis, Stanford Univ., 1984.

[CHAPP91] Chappell T., et.al., "A 2nS Cycle, 4nS Access 512kb CMOS ECL SRAM," *IEEE International Solid-State Circuits Conference* Digest of Technical Papers, pp. 50-51, Feb. 1991.

[CHU86] Chu T.A., "Synthesis of Self-timed Control circuits from Graphs: An example," Proceedings of *ICCD,* pp. 565-571, Oct. 1986.

[COMM71] Commoner F.G., Holt A.N., et.al., "Marked directed graphs," *Journal of Computer and System Sciences*, vol. 5, pp. 511-523, Oct. 1971.

[DEAN91] Dean M., Williams T., Dill D., "Efficient Self-Timing with Level-Encoded 2-Phase Dual-Rail (LEDR)," Proceedings of the Santa Cruz Conference on *Advanced Research in VLSI*, pp. 55-70, March 1991.

[EBER88] Ebergen J.C., "Transforming Programs into Delay-Insensitive Circuits," Ph.D. Thesis, Eindhoven Tech. Univ., 1988.

[FAND89] Fandrianto J., "Algorithm for High Speed Shared Radix 8 Division and Radix 8 Square Root," Proceedings of the *9th IEEE Symposium on Computer Arithmetic*, pp. 68-75, Sept. 1989.

[GREE87] Greenstreet M., Williams T., Staunstrup J., "Self-Timed Iteration," Proceedings of *VLSI-87*, Vancouver Canada, Elsevier Science Publishers: Amsterdam, pp. 309-322, Aug. 1987.

[GREE88] Greenstreet M., Steiglitz K., "Throughput of Long Self-Timed Pipelines," CS-TR-190-88, Princeton Univ., Nov. 1988.

[HENN90] Hennessy J.L., Patterson D.A., *Computer Architecture: A Quantitative Approach.* Palo Alto CA: Morgan Kaufmann, 1990.

[JACO90] Jacobs G., "A Fully Asynchronous Digital Signal Processor," *ISSCC Digest of Technical Papers*, pp. 150-151, Feb. 1990.

[LEIS86] Leiserson C., Saxe J., "Retiming Synchronous Circuitry," DEC Systems Research Center TR-13, Palo Alto CA, 1986.

[MART86] Martin A., "Compiling Communicating Processes into Delay-Insensitive VLSI Circuits," *Distributed Computing*, vol. 1, no. 4., pp. 226-234, 1986.

[MART89] Martin A., "The Limitations to Delay-Insensitivity in Asynchronous Circuits," Proceedings of the 6th MIT Conference on *Advanced Research in VLSI*, ed. Dally W.J., pp. 263-278, March 1989.

[MATU90] Matula D., "Highly Parallel Divide and Square Root," SMU Technical Article, Dallas TX, Oct. 1989.

[MCAL86] McAllister W., Zuras D., "An nMOS 64b Floating-Point Chip Set," *IEEE International Solid-State Circuits Conference* Digest of Technical Papers, pp. 34-35, Feb. 1986.

[MENG88] Meng T., "Asynchronous Design for Programmable Digital Signal Processors," Ph.D. Thesis, UC Berkeley, 1988.

[MENG89] Meng T., Brodersen R., Messerschmitt D., "Automatic Synthesis of Asynchronous Circuits from High-Level Specifications," *IEEE Trans. on CAD*, vol. 8, no. 11, pp. 1185-1205, Nov. 1989.

[MILL65] Miller R.E., *Switching Theory.* New York: Wiley, 1965.

[MULL63] Muller D.E., "Asynchronous logics and applications to information processing," Proceedings of *Symposium on Applications Switching Theory Space Technology*, pp. 289-297, 1963.

[MURA77] Murata T., "Petri Nets, Marked Graphs, and Circuit-system Theory," *Circuits and Systems*, vol 11 no. 3, pp. 2-12, June 1977.

[RAMA80] Ramamoorthy C.V., Ho G.S., "Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets," *IEEE Trans. on Software Engineering*, vol. SE-6 no. 5, pp. 440-449, Sept. 1980.

[RAMC74] Ramchandani C., "Analysis of Asynchronous Concurrent Systems by Petri nets," Project MAC, TR-120, MIT, Cambridge MA, Jan. 1974.

[RAO86] Rao S.K., "Regular Iterative Algorithms and their Implementation on Processor Arrays," Ph.D. Thesis, Stanford Univ., 1986.

[ROBE58] Robertson J., "A New Class of Digital Division Methods," *IRE Trans. Electronic Computers*, vol. EC-7, pp. 218-222, 1958.

[SALZ89] Salz A., Horowitz M., "IRSIM: An Incremental MOS Switch-Level Simulator," Proceedings of the *26th Design Automation Conference*, pp. 173-178, June 1989.

[SANT89] Santoro M., Horowitz M., "SPIM: A Pipelined 64x64-bit Iterative Multiplier," *IEEE Journal of Solid-State Circuits*, vol. 24, no. 2, pp. 487-493, Apr. 1989.

[SEIT80] Seitz C., "System Timing," Chapter 7 in *Introduction to VLSI Systems,* eds. Mead C. & Conway L., Addison-Wesley: Reading MA, 1980.

[SUTH89] Sutherland I., "Micropipelines," *Communications of the ACM*, vol. 32 no. 6, pp. 720-738, June 1989.

[STAU87] Staunstrup J., Ravn A.P., "Synchronized Transitions," DAIMI Tech Report PB-219, Computer Science Department, Aarhus University, Denmark, Jan. 1987.

[TAYL81] Taylor G., "Compatible Hardware for Division and Square Root," Proceedings of the *5th IEEE Symposium on Computer Arithmetic*, pp. 127-134, May 1981.

[TAYL85] Taylor G., "Radix 16 SRT Dividers with Overlapped Quotient Selection Stages," Proceedings of the *7th IEEE Symposium on Computer Arithmetic*, pp. 64-71, June 1985.

[UDDI86] Udding J.T., "Classification and Composition of Delay-Insensitive Circuits," Ph.D. Thesis, Eindhoven Tech. Univ., 1986.

[UNGE69] Unger S.H., *Asynchronous Sequential Switching Circuits.* Malabar, Florida: Krieger Publishing, reprinted 1983.

[WILL86] Williams T., Horowitz M., "SRT Division Diagrams and Their Usage in Designing Custom Integrated Circuits for Division," Stanford Tech Report CSL-TR-87-326, Nov. 1986.

[WILL87] Williams T., Horowitz M., et.al., "A Self-Timed Chip for Division," Proceedings of the Stanford Conference on *Advanced Research in VLSI,* pp. 75-96, March 1987.

[WILL88] Williams T., Klingsheim K., "Self-Timed Circuit Elements in GaAs," Proceedings of the *13th Nordic Semiconductor Conference*, Stockholm Sweden, pp. 344-347, June 1988.

[WILL90a] Williams T., Horowitz M., "Bipolar Circuit Elements Providing Self-Completion-Indication," *IEEE Journal of Solid-State Circuits*, vol. 25, no. 1, pp. 309-312, Feb. 1990.

[WILL90b] Williams T., "Latency and Throughput Tradeoffs in Self-Timed Asynchronous Pipelines and Rings," Stanford Tech Report CSL-TR-90-431, Aug. 1990.

[WILL91a] Williams T., Horowitz M., "A Zero-Overhead Self-Timed 54b 160nS CMOS Divider," *IEEE International Solid-State Circuits Conference*, Digest of Technical Papers, pp. 98-99, Feb. 1991.

[WILL91b] Williams T., Horowitz M., "A 160nS CMOS Division Implementation using Self-Timing and Overlapped SRT Stages," Proceedings of the *10th IEEE Symposium on Computer Arithmetic*, June 1991.

[WOLR84] Wolrich G., McLellan E., et.al., "A High Performance Floating-Point Coprocessor," *IEEE Journal of Solid-State Circuits*, vol. 19, no. 5, pp. 690-696, Oct. 1984.