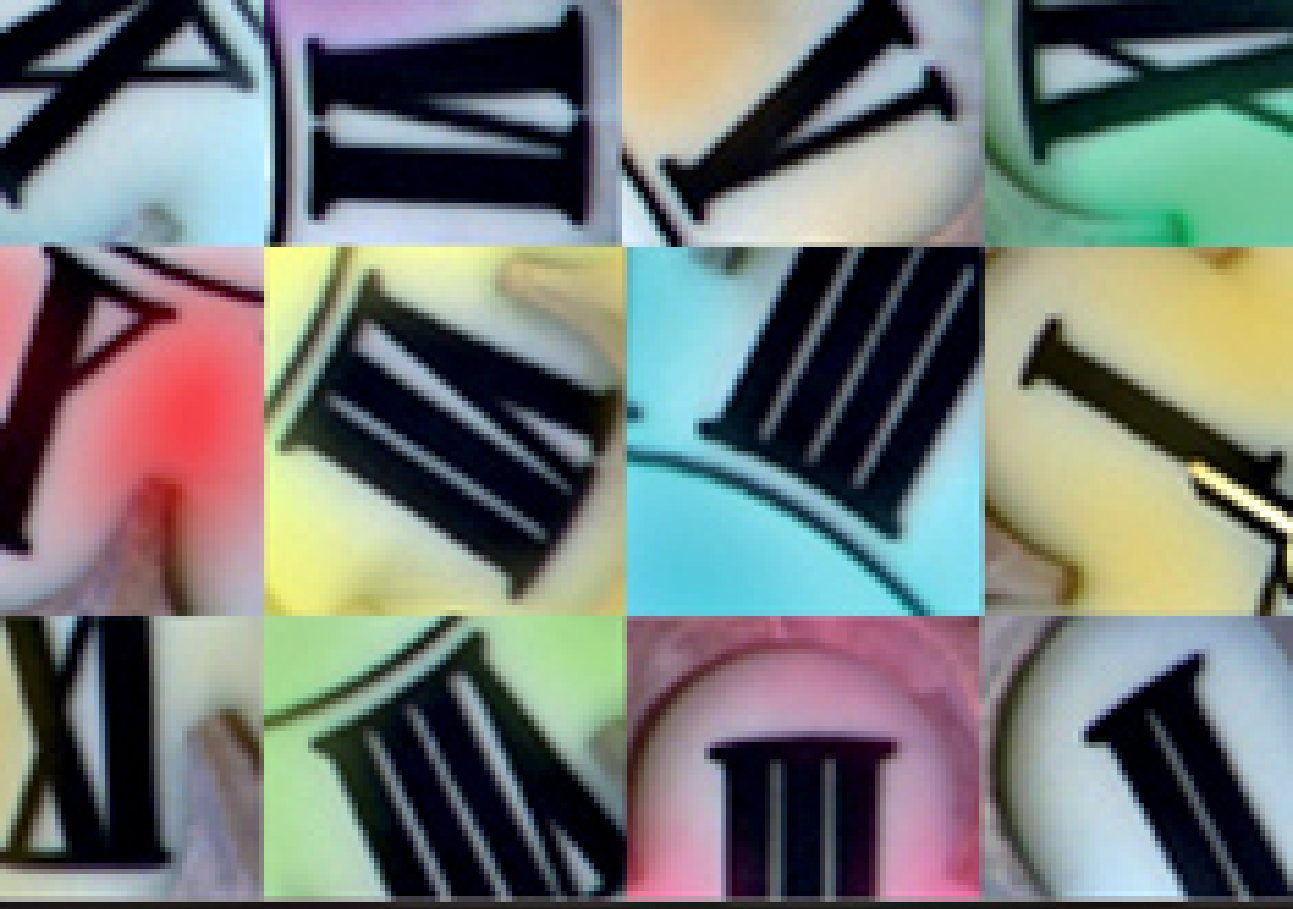

A Designer's Guide to

Asynchronous VLSI

Peter A. Beerel, Recep O. Ozdag
and Marcos Ferretti

CAMBRIDGE

CAMBRIDGE www.cambridge.org/9780521872447

This page intentionally left blank

A Designer's Guide to Asynchronous VLSI

Create low power, higher performance circuits with shorter design times using this practical guide to asynchronous design. This practical alternative to conventional synchronous design enables performance close to full-custom designs with design times that approach commercially available ASIC standard cell flows. It includes design trade-offs, specific design examples, and end-of-chapter exercises. Emphasis throughout is placed on practical techniques and real-world applications, making this ideal for circuit design students interested in alternative design styles and system-on-chip circuits, as well as for circuit designers in industry who need new solutions to old problems.

Peter A. Beerel is CEO of TimeLess Design Automation – his own company commercializing asynchronous VLSI tools and libraries – and an Associate Professor in the Electrical Engineering Department at the University of Southern California (USC). Dr. Beerel has 15 years' experience of research and teaching in asynchronous VLSI and has received numerous awards including the VSoE Outstanding Teaching Award in 1997 and the 2008 IEEE Region 6 Outstanding Engineer Award for significantly advancing the application of asynchronous circuits to modern VLSI chips.

Recep O. Ozdag is IC Design Manager at Fulcrum Microsystems and a part-time Lecturer at USC, where he received his Ph.D. in 2004.

Marcos Ferretti is one of the founders of PST Electrônica S. A. (Positron), Brazil – an automotive electronic systems manufacturing company – where he is currently Vice-President. He received his Ph.D. from USC in 2004 and was co-recipient of the USC Electrical Engineering-Systems Best Paper Award in the same year.

A Designer's Guide to Asynchronous VLSI

PETER A. BEEREL
University of Southern California / Timeless Design Automation

RECEP O. OZDAG
Fulcrum Microsystems Inc.

MARCOS FERRETTI
PST Electrônica S. A. (Positron)

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore,
São Paulo, Delhi, Dubai, Tokyo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 8RU, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521872447

First published in print format 2010

ISBN-13 978-0-511-67549-2 eBook (NetLibrary)

ISBN-13 978-0-521-87244-7 Hardback

To Janet, Kira, and Kim
– P. A. B.

To Jemelle and Charlotte
– R. O. O.

To Lilione
– M. F.

Contents

Acknowledgments

There are many people that helped make this book a reality that deserve our thanks and recognition. First, we'd like to thank many colleagues that we had the pleasure of working with and whose research has shaped our understanding of asynchronous design. This included Dr. Peter Beerel's advisors Professors Teresa Meng and David Dill at Stanford University, Professor Kenneth Yun of University of San Diego, Professors Chris Myers and Kenneth Stevens of University of Utah, Professor Steven Nowick at Columbia University, and Professor Steve Furber at Manchester University. Special thanks goes to Dr. Ivan Sutherland and Marly Roncken, now at Portland State University, for their leadership, vision, support, and encouragement.

Moreover, we'd like to thank other members of the USC asynchronous VLSI/CAD research group. This included Dr. Sunan Tugsinavisut who drafted an early version of Chapter 9 on Micropipelines and Dr. Sangyun Kim who helped write and early version of Chapters 4 and 5 on performance analysis. In addition, we'd like to acknowledge Arash Saifhashemi for his help in drafting Chatper 3 and in particular developing the VerilogCSP modeling support for asynchronous designs. Other former students that also supported this work include Dr. Aiguo Xie, Mallika Prakash, Gokul Govindu, Amit Bandlish, Prasad Joshi, and Pankaj Golani. Special thanks go to Dr. Georgios Dimou who is helping take the culmination of much of this work to market by co-founding TimeLess Design Automation with Dr. Peter Beerel.

We would also like to recognize the various industrial and government funding sources that made much of our research possible, including Intel, NSF, and Semiconductor Research Corporation. In particular, much of the research on asynchronous pipelines was supported via a NSF Large Scale ITR research grant that funded both Dr. Recep Ozdeg and Dr. Marcos Ferretti's thesis work on advanced asynchronous pipelines and the developed back-end ASIC flow. Also, we would like to recognize the support from the MOSIS Education Program and Fulcrum Microsystems in the fabrication and test of the demonstration design presented in Chapter 13. We also thank all our colleagues at USC for their continued support and guidance and in particular Professors Mel Breuer, Massoud Pedram, Alice Parker, and Sandeep Gupta.

We also acknowledge the competence and professionalism of the Cambridge University Press personnel, especially Susan Parkinson for the great help

reviewing the book and the reviewers for their excellent feedback. In particular, Marly Roncken provided invaluable feedback on Chapter 10, Syntax-Directed Translation. Moreover, the students of EE 552, Asynchronous VLSI at USC identified and helped fix many bugs in early drafts of the book.

Lastly, we'd like to thank the support of our significant others without whose support and understanding this three-year effort could never have been completed, including Janet A. Martin, Jemelle A. Pulido-Ozdag, and Lilione Sousa Ferretti.

1 Introduction

Innovations in mobile communication, e-commerce, entertainment, and medicine all have roots in advances in semiconductor processing, which continually reduce the minimum feature size of transistors and wires, the basic building blocks of chips. This continual reduction supports ever-increasing numbers of transistors on a single chip, enabling them to perform increasingly sophisticated tasks. In addition, smaller transistors and wires have less resistance and capacitance, enabling both the higher performance and lower power that the integrated circuit market continually demands.

These manufacturing advances, however, also change the design challenges faced by circuit designers and the computer-aided-design (CAD) tools that support their design efforts. Beginning in the 1980s, wire resistance became an important factor to consider in performance and is now also important in analyzing voltage drops in power grids and long wires. Starting in the 1990s, higher mutual capacitance exacerbated the impact of cross-talk, which is now addressed by a new range of timing and noise analysis tools. And today, as the transistor's feature size approaches fundamental atomic limits, transistors act less like ideal switches and wires act less like ideal electrical connections. In addition, the increased variations both within a single chip and between chips can be substantial, making precise estimates of their timing and power characteristics virtually impossible [1]. Consequently, modern CAD tools must conservatively account for these new non-ideal transistor characteristics as well as their variability.

As part of this ever-changing technological backdrop, the relative merits of different circuit design styles change. The predominant circuit design style is synchronous design with complementary metal-oxide–semiconductor (CMOS) static logic gates and a global clock to regulate state changes. However, as process variability increases and the challenges of routing a global clock across a large chip become increasingly problematic, radically different design styles such as asynchronous design have become an increasingly interesting alternative. In its most general form, asynchronous design removes the global clock in favor of distributed local handshaking to control data transfer and changes of state. While academic research in this area can be traced back to the 1950s [2], it has taken until the late 1990s and 2000s for this technology to mature. Several start-up companies have begun to commercialize asynchronous design as a competitive advantage for

a wide variety of applications [45][46][47][48]. However, the mass application of asynchronous design has been an elusive goal for academic researchers and, while recent advances are promising, only time will tell whether this technology will take a larger foothold in the very-large-scale integration (VLSI) world.

There are many different types of integrated circuits (ICs) and the design style choice for a particular application depends on the relative performance, power, volume, and other market demands of the device. For example, traditionally, low-volume specialized products with only moderate power and performance requirements can use field programmable gate arrays (FPGAs), which provide reduced time to market and low design risk, primarily because of their reprogrammable nature. Higher-volume products with more aggressive power and performance requirements often require application specific integrated circuits (ASICs), which come at the cost of the increased design and verification effort associated with the finalizing of the manufacturing process.

Products that require significant programmability may also contain some type of microprocessor to enable software support. Products which require significant storage will contain large banks of on-chip memories. Both micro-processors and memory blocks are available on modern FPGAs and can be integrated into an ASIC in the form of intellectual property cores. In addition, dedicated chips for memory are critical in complex system design and can store billions of bits of data either in volatile or non-volatile forms.

Chips with high volumes, such as microprocessors, memory chips, and FPGAs, may be able to support *full-custom* techniques with advanced circuit styles, such as asynchronous design. In fact, asynchronous techniques have been used in memory for years and a recent start-up is the commercializing of high-speed FPGAs, which has been enabled by high-speed asynchronous circuits [42][43][48].

Most ASICs, however, rely on *semi-custom* techniques in which more constrained design styles are used. The relative simplicity of the constrained design style enables the development of CAD tools that automate large portions of the design process, significantly reducing design time. For asynchronous design to be adopted for ASICs, existing CAD tool suites must be enhanced with scripts and new tools to support asynchronous circuits. This chapter provides an overview of the general issues that guide this design choice. In doing so, it identifies the potential advantages of asynchronous design and the remaining challenges for its widespread adoption.

1.1 Synchronous design basics

Synchronous design has been the dominant methodology since the 1960s. In synchronous design, the system consists of sub-systems controlled by one or more clocks that synchronize tasks and the communication between blocks. In traditional semi-custom synchronous ASIC flows, combinational logic is placed in between banks of flip-flops (FFs) that store the data, as shown in Figure 1.1.

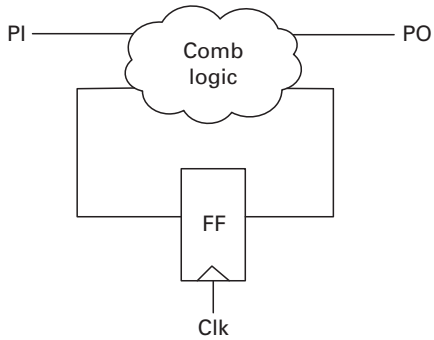

Figure 1.1. Traditional synchronous ASIC, showing the combinational logic and flip-flop. The primary inputs and outputs are denoted PI and PO.

The combinational block must complete its operation within one clock cycle under all possible input combinations, from all reachable states, in the worst-case operating environment. In this way, the clock ensures synchronization among combinational blocks by guaranteeing that the output of every combinational block is valid and ready to be stored before the next clock period begins. In fact, the data at the inputs of the FFs may exhibit glitches or hazards, as long as they are guaranteed to settle before the sampling clock edge arrives. In order to guarantee that the data is stable when sampled, the clock period should account for the worst-case delay including clock skew and all process variations.

Typical semi-custom design flows use a fixed set of library cells that have been carefully designed, verified, and characterized to support synthesis, placement, routing, and post-layout verification tasks. This library is generally limited to static CMOS gates, which, compared with more advanced dynamic logic families, have higher noise margins and thus require far less analog verification. The cells have accurate table-based characterization to support *static* timing and noise analysis rather than more computationally intensive analog simulation. In particular, timing constraints are reduced to the setup and hold times on FFs, which static timing analysis tools can verify reliably with minimal user input. This constrained methodology has facilitated the development of mature suites of CAD tools that yield relatively short, 12-month, design times.

Full-custom flows use more labor-intensive advanced design styles and less automated tools to obtain higher performance and lower power. In particular, full-custom ICs have clock frequencies that are three to eight times higher than those for semi-custom flows, owing, to their advanced architectures, micro-architectures, circuit styles, and manufacturing processes [4]. The differences include: the use of advanced pipelining, clock-tree design, registers, dynamic logic, and time borrowing; more careful logic design, cell design, wire sizing, floor-planning, placement, and management of wires; the better management of process variation; and, finally, accessibility to faster manufacturing processes.

In particular, full-custom designs can use a variety of forms of dynamic logic which offer significant improvements in performance and power consumption [1][3][6]–[8]. For example, the application of dynamic logic in the IBM 1.0 GHz design yields 50% to 100% higher performance compared with its static logic equivalent [8]. In addition, advanced FFs and latches can reduce the overhead associated with clock skew and latch delays [7]. Moreover, latch delays can even be removed by means of multiple overlapping clocks and dynamic logic, in a widely used technique recently named skew-tolerant domino logic [6].

However, dynamic logic has lower noise margins than its static counterpart. Dynamic logic is also more difficult to characterize using table-based methods, and static timing analysis tools tend to have less accurate results for such circuits [3]. Consequently, more careful analog-level noise and timing verification is required. Managing this verification, along with the other more manual aspects of full-custom design, typically results in a significant increase in design time.

1.2 Challenges in synchronous design

Synchronous design has been the predominant design methodology largely because of the simplicity and efficiency provided by the global clock. The registers decompose the design into acyclic islands of combinational logic which facilitate efficient design, synthesis, and analysis algorithms. However, the global nature of the clock also leads to increasing design and automation challenges. In particular, the time-to-market advantage of standard-cell-based ASIC designs is being subverted by the increasingly difficult design challenges posed by modern semiconductor processes [4]. These challenges affect both high-performance and low-power ASICs, as described below.

1.2.1 Computer-aided design for high-performance

In earlier submicron designs, architecture, logic, and technology-mapping design could proceed before and somewhat independently from the placement and routing of the cells, power grid, and clocks because wire delays were negligible compared with gate delays. In deep-submicron design, however, interconnect has not scaled to the same degree as gates, and cross-talk between wires leads to substantial changes in wire delay. Consequently, wire delay increasingly accounts for a larger fraction of the critical path. In particular, the delays of long-range wires may account for up to 70% of the critical path of some high-performance designs [36]. This change in relative importance has caused the traditional separation of logic synthesis and physical design tasks to break down, because synthesis cannot now properly account for the actual wire delays and other geometrical effects. Since the late 1990s, this timing-closure problem and disconnect between synthesis and physical design has forced numerous shipment schedules to slip.

As a consequence, tighter integration of these CAD tools has been developed and a wide range of post-placement optimization, including gate resizing, buffer insertion, and wire width optimization, have become an essential part of CAD tool suites. In addition, sophisticated cross-talk analysis techniques that account for the switching window of various wires have been developed to determine the impact in delay associated with increasingly large cross-coupling capacitances. Understanding the impact of cross-talk during post-placement optimization is computationally challenging owing to the inherent feedback between accurate cross-talk analysis and post-placement optimizations. For example, switching windows can shift dramatically owing to buffer insertion, resulting in significant changes in the delay of neighboring wires. This computational challenge forces CAD tools to rely on approximations that yield non-optimal designs.

Despite the continued advances of modern CAD tools, the overall impact of these challenges has been an increasingly large performance gap between full-custom integrated circuits (ICs) and semi-custom ASICs. Moreover, it is predicted that high-performance semi-custom designs will remain three times slower than their full-custom counterparts despite all the potential advances in CAD tools [4]. In particular, conventional wisdom suggests that semi-custom CAD tools may never support dynamic logic because of the added complexity of noise and timing constraints and the lack of dynamic cell libraries.

1.2.2 Computer-aided design for low-power devices

As manufacturing feature sizes have decreased, transistors have become increasingly leaky and power budgets have become increasingly difficult to meet. This has motivated a range of improvements to the low-power ASIC flow. In particular, low-leakage power-efficient cell design, multiple supply voltages on a single die, gated power supplies, and more advanced clock-gating techniques are continually being developed and incorporated in ASIC flows. In addition, a few full-custom techniques for low power have also been explored and are just emerging in ASIC tools; these techniques include low-voltage swings for long-range wires. However, these low-swing circuits have reduced noise margins, which necessitates careful wire planning, shielding, and some analog verification. Moreover, it is well known that low-power latches can be used instead of flip-flops to reduce clock-tree capacitance.

1.3 Asynchronous design basics

As a result of the increasing limitations and growing complexity of semi-custom synchronous design, asynchronous circuits are gaining in interest. In the absence of a global clock that controls register and state updating, asynchronous designs rely on handshaking to transfer data between functional blocks. One common way

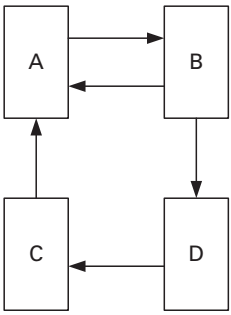

Figure 1.2. Asynchronous blocks communicating using channels.

to design asynchronous circuits is to organize the data transfer in *channels* that group a bundle of data wires using handshaking signals. The channels are unidirectional and typically point-to-point; they are represented as arrows in Figure 1.2. Notice that the bi-directional communication of data between blocks A and B requires two channels in opposite directions.

Many different design styles have been proposed for asynchronous circuits. They differ in the method in which data is encoded in channels, the handshaking protocol, and the number and type of timing assumptions required for the designs to work properly. These different design tradeoffs make it difficult to make general statements regarding the relative merits of the various benefits typically associated with asynchronous design. In particular, some design styles yield low power but are best suited for low-performance applications. Others yield high performance at the expense of more power and additional timing assumptions. All, however, provide a rigorous framework for exploring alternatives to synchronous design.

1.4 Asynchronous design flows

There are many different possible design flows for asynchronous circuits. Here we describe three, to demonstrate the diversity of flows being explored.

The first of these design flows is called *refinement* and involves the decomposition of asynchronous blocks into a hierarchical network of *leaf cells*, where a leaf cell is the smallest block that communicates with its neighbors via channels. Each leaf cell is typically implemented with a small set of transistors, typically between 10 and 100. Early attempts to automate this process are encouraging, but as of today industry relies on significant manual effort and a large re-usable library of macro cells (functional blocks, registers, and crossbars) and leaf cells. This technique was pioneered by Alain Martin at Caltech and has been applied to several asynchronous microprocessors [9] and digital-signal processing chips [25]. It has been commercialized by Fulcrum Microsystems and has led

to circuit designs whose performance exceeds that available through semi-custom standard-cell design flows (see e.g. [9]).

The second design flow involves re-using synchronous synthesis tools and translating a synchronous gate-level netlist (which conveys connectivity information) into an equivalent asynchronous netlist. This involves removing the global clock and replacing it with local handshaking circuitry. This flow has the benefits of reducing the barrier of adoption for asynchronous design and employing the power of synchronous synthesis tools. This approach was initially proposed by a start-up company called Theseus Logic [28] and subsequent results are encouraging [26]–[32]. Nevertheless, more work is necessary for it to produce circuits that compete with manual decomposition.

The third design flow is based on syntax-directed translation from a high-level language that includes handshaking primitives such as *sends* and *receives* [15]–[17]. Syntax-directed translation makes the high-level estimation of power and performance characteristics computationally efficient and enables designers to control the resulting circuits more directly by altering the high-level language specification. The results of the translation are typically far from optimal, and forms of peep-hole optimization at gate level are needed to improve efficiency. This technique was pioneered by several researchers at Phillips Research and is now being commercialized by Handshake Solutions. It has produced very-low-power designs from a digital compact cassette (DCC) error detector [15], an 80C51 micro-controller [14], and an Advanced RISC Machines (ARM) micro-processor [44] that compare quite favorably with their synchronous counterparts.

There are also significant differences in back-end flows for these design flows. Some rely on using synchronous standard-cell libraries while others rely on the advantage of using non-standard gates such as C-elements and domino logic. In both cases, however, commercial place and route flows are being explored to automate the physical design.

In addition, commercial static timing and power analysis tools are used to verify timing assumptions pre- and post-layout, and synchronous test tools are being adopted for both automated test generation and test coverage analysis. In all these cases, the presence of combinational cycles in asynchronous circuits is a distinguishing feature that often stretches the capabilities of these tools and requires novel approaches.

1.5 Potential advantages of asynchronous design

Asynchronous circuits have demonstrated potential benefits in many aspects of system design (e.g. [9][18]–[23]). Their advantages include improvements in high-performance, low-power, ease of use and reduced electromagnetic interference (EMI) but these are not universally applicable. Some advantages may be application specific and dependent on the particular asynchronous circuit design style. Others depend on whether the comparison is made with semi-custom or full-custom

synchronous design or, more generally, the level of effort put into the comparable synchronous design.

1.5.1 High performance

Performance can be measured in terms of system latency or throughput or a combination of the two. The potential for high performance in asynchronous design stems from a number of factors. Several of these are inherent in any design lacking a global clock and are independent of the asynchronous design style. They include the following:

- *Absence of clock skew* Clock skew is defined as the arrival time difference of the clock signal to different parts of the circuit. In traditional standard-cell design, the clock period may need to be increased to ensure correct operation in the presence of clock skew, yielding slower circuits. In recent design flows, however, a portion of this skew is regarded as *useful skew* and is accounted for during logic synthesis, thus mitigating the impact on the feasible clock frequency. Moreover, in full-custom design, more sophisticated clock-tree analysis and design reduce this effect further. Nevertheless, as process variations increase, the clock-skew impact on clock frequencies is likely to grow.
- *Average-case performance* Synchronous circuit designers have to consider the worst-case scenario when setting the clock speed to ensure that all the data has stabilized before being sampled. However, many asynchronous designs, including those that use static logic, can have average-case delay due to a data-dependent data flow and/or functional units that exhibit data-dependent delay [37]. In both cases, the average-case delay may be less than the synchronous worst-case delay.

Other performance advantages are specific to different sub-classes of asynchronous designs and include the following:

- *Application of domino logic* As mentioned earlier, domino logic is often used in high-performance full-custom synchronous designs because its logical effort is lower than that required for static logic [6]. Domino logic is limited to full-custom design flows because of its reduced noise margin and because its timing assumptions are not currently supported by semi-custom design tools. Some asynchronous design flows embed domino logic within a *pipeline template* [22] [25]. Instead of a clock that controls the precharge and evaluate transistors, distinct asynchronous control signals are used. Recent research advances are aimed at determining whether these templates can be designed within a standard-cell flow for asynchronous design [33]–[36]. Compared with current ASIC synchronous flows, they have the potential performance advantages of dynamic logic as well as removal of the latency overhead and setup margins associated with explicit flip-flops and latches.
- *Automatic adaptation to physical properties* The delay on a path may change owing to variations in the fabrication process, temperature, or power supply

voltage. Synchronous system designers must consider the worst case and set the clock period accordingly. Many asynchronous circuits can adapt robustly to changing conditions, yielding improved performance [22][24]. This is far more difficult to achieve in a synchronous design, as the variations can be local whereas the impact on the clock is far more global.

- *At-speed testing* Asynchronous design styles that embed data and validity into the same wires provide a potential additional performance advantage. In principle, it is possible to add logic to verify whether the asynchronous data arrives before a fixed clock, both during performance testing and/or on-line. This enables mixed asynchronous–synchronous designs which avoid the large margins associated with synchronous application-specific ICs, for which performance testing would not be affordable.

1.5.2 Low power

The constant activity of a global clock causes synchronous systems to consume power even though some parts of the circuit may not be processing any data. Even though clock gating can avoid the sending of the clock signal to the un-active blocks, the clock driver still has to constantly provide a powerful clock that reaches all parts of the circuit. The removal of the global clock in favor of power-efficient control circuits can sometimes lead to significant power savings. Moreover, asynchronous architectures can reduce power consumption by minimizing data movement to only where and when it is needed.

The event-driven nature of asynchronous design leads to circuits with low standby power, which provides a significant advantage in mobile applications that must react to external stimuli (e.g. smart cards [9] and pagers [12]). The alternative in synchronous design would be a standby-power-inefficient circuit that continually polls external signals.

A third power advantage of some asynchronous design techniques is the application of level-sensitive latches. Latches have less input capacitance and consume less switching power than comparable flip-flops, and their use can lead to substantial savings in power [13].

However, these power advantages are not universal in asynchronous design styles. In particular, some asynchronous circuits designed for high performance have more average transitions per data bit than comparable synchronous designs, owing to the dual-rail or other multi-rail data encoding and/or completion-detection logic. While the performance advantage associated with some techniques may be turned into lower power through voltage scaling, the overall power saving is not as clear and is likely to be application dependent.

1.5.3 Modularity and ease of design

Another advantage of asynchronous design is the modularity that comes from the send and receive channel-based discipline. Blocks that communicate using the

same handshaking discipline can very easily be connected, offering a plug-and-play approach to design. Moreover, the handshaking discipline offers an immediate notion of flow control within the design. If some stage is not ready to receive new tokens then the sender will block until the receiver is ready.

In general, well-designed asynchronous circuits are one form of latency-insensitive design, and they make changing the level of pipelining late in the design cycle substantially less disruptive. This is particularly advantageous in enabling long wires to be pipelined late in the design cycle, as we will explore in Chapter 4.

More generally, asynchronous circuits provide an effective component in designs that are globally asynchronous and locally synchronous. They can offer the high-throughput low-latency power-efficient interconnect technology that is essential to creating the backbone of networks on chips. This has been the focus of Silistix, an asynchronous start-up company out of the University of Manchester [45]. We will explore this aspect of asynchronous design in Chapter 14.

1.5.4 Reduced electromagnetic interference

In a synchronous design, all activity is locked into a very precise frequency. The result is that nearly all the energy is concentrated in very narrow spectral bands around the clock frequency and its harmonics. Therefore, there is substantial electromagnetic noise at these frequencies, which can adversely affect neighboring analog circuits. Activity in an asynchronous circuit is uncorrelated, resulting in a more distributed noise spectrum and lower peak noise. A good example of this is the Amulet 2e asynchronous micro-processor, which displayed a lower overall emission level and much less severe harmonic peaks than similar clocked circuits [20].

1.6 Challenges in asynchronous design

Despite these advantages, which have been evident for some time, asynchronous circuits are only now gaining acceptance in industry. Two main reasons have to do with the challenges they present in testing and debugging and the general lack of CAD tools that support asynchronous design.

1.6.1 Testing and debugging

Testing for synchronous ASICs is made very efficient by the use of specialized scannable flip-flops, advanced tools for automated test-pattern generation, and IEEE test circuit standards such as the Joint Test Action Group (JTAG). The basic challenge associated with many asynchronous design styles is the presence of loops in the circuit that are not cut by specific latches or flip-flops. Many of these loops must be cut with additional circuitry in order to achieve sufficient observability and controllability in the circuit for test purposes and, perhaps more

importantly, for automated test-pattern-generation techniques to be applicable. Research in this area is extensive (e.g. [39][40]) but the commercialization of these techniques is only just beginning.

In addition, in comparison with synchronous design, asynchronous design can be difficult to debug. In synchronous design, when the circuit fails one can lower the clock frequency and investigate the failure. In asynchronous designs, however, there is no clock and the circuit operates at the maximum possible speed. The lack of this natural control can make debugging more challenging.

1.6.2 Industry-supported CAD tools

Only one start-up company, Handshake Solutions, has developed a commercially supported design flow for asynchronous circuits [46]. It is based on their proprietary language Haste and a syntax-directed design flow. It is geared toward low-power circuits and uses commercially available static CMOS libraries. All other asynchronous start-up companies use internal flows to design chips or intellectual property cores (e.g. [47]).

Although research shows that it is possible to design asynchronous circuits with common industry-supported CAD tools that are geared for synchronous design (e.g. [28]), this process in general requires modifications to fool these tools into thinking that the design is synchronous, making logic, timing, and performance verification more challenging. Whether traditional electronic design automation (EDA) companies or another start-up will put efforts into completing and optimizing these flows will depend on the industrial demand for asynchronous circuits. However, this demand depends on the ease of designing high-quality asynchronous circuits and integrating them into otherwise synchronous systems, which in turn depends on the availability of CAD tools optimized for asynchronous design. Solving this "catch-22" situation depends on many factors, including the growing limitations of synchronous design and the collaboration between academia and industry.

1.7 Organization of the book

In summary, asynchronous design embodies a wide range of circuit families and handshaking disciplines that hold out a promise of low power, high performance, low electromagnetic emissions, and ease of use. This book provides an introduction to this diverse area of VLSI from a designer's point of view. Our goal is to enable designers to appreciate the many asynchronous design choices that may be readily available in the near future.

To do this in an organized fashion, we have divided the remainder of this book into three parts. The first focuses on general asynchronous architectures without going into implementation details. In particular, in Chapter 2 we discuss the variety of handshaking disciplines available and how handshaking is used to

synchronize the behavior among asynchronous blocks forming sequential, parallel, and pipelined implementations. In Chapter 3 we explain how these handshaking forms can be modeled at an abstract level and, in particular, focus on a means of modeling them in the well-known Verilog hardware description language. In Chapters 4 and 5 we consider the unique performance analysis and optimization techniques specific to asynchronous design, with an emphasis on asynchronous pipelines. Chapter 6 completes this part of the book with an explanation of the deadlock issues associated with asynchronous architectures.

The second part of the book focuses on the variety of implementation strategies that are available for asynchronous architectures. Chapter 7 provides a detailed overview of the different design options. Chapter 8 focuses on two common methods of automatically synthesizing asynchronous controllers, one using state machines and the other using event-based specifications. Chapter 9 covers the famous *micropipelines* approach to asynchronous design, which uses static logic, delay lines, and small asynchronous controllers [41]. In Chapter 10 we then cover the syntax-directed design approach being commercialized by Handshake Solutions [46]. In Chapters 11 through 13 we consider a variety of asynchronous templates geared towards high-speed fine-grain asynchronous pipelines.

The third part of the book comprises specific design examples. Chapter 14 focuses on an asynchronous crossbar-based solution to globally asynchronous locally synchronous design. Finally, Chapter 15 contains an asynchronous implementation of the Fano algorithm that exemplifies the benefits of asynchronous design described throughout the book.

References

[1] "International technology roadmap for semiconductors – 2005 edition," http://www.itrs.net/Common/2005ITRS/Home2005.htm.

[2] S. H. Unger, *Asynchronous Sequential Switching Circuits*, Wiley Interscience, John Wiley & Sons, 1969.

[3] A. Chandrakasan, W. J. Bowhill, and F. Fox, *Design of High Performance Microprocessor Circuits*, IEEE Press, 2001.

[4] D. Chinnery and K. Keutzer, *Closing the Gap Between ASIC & Custom: Tools and Techniques for High-Performance ASIC Design*, Kluwer Academic, 2002.

[5] S. Schuster, W. Reohr, P. Cook, D. Heidel, M. Immediato, and K. Jenkins, "Asynchronous interlocked pipelined CMOS circuits operating at 3.3–4.5 GHz," in *IEEE ISSCC Conf. Digest of Technical Papers*, February 2000, pp. 292–293.

[6] D. Harris, *Skew-Tolerant Circuit Design*, Morgan Kaufmann, 2000.

[7] K. Bernstein, K. M. Carrig, C. M. Durham, *et al.*, *High Speed CMOS Design Styles*, Kluwer Academic, 1998.

[8] H. P. Hofstee, S. H. Dhong, D. Meltzer, *et al.*, "Designing for a gigahertz guTS integer processor," *IEEE Micro*, vol. 18, no. 3, pp. 66–74, May 1998.

[9] A. J. Martin, M. Nyström, and C. G. Wong, "Three generations of asynchronous microprocessors," *IEEE Design & Test of Computers, special issue on clockless VLSI design*, November/December 2003.

[10] U. Cummings, "Terabit clockless crossbar switch in 130nm," in *Proc. 15th Hot Chips Conf.*, August 2003.
[11] J. Kessels, T. Kramer, G. den Besten, A. Peeters, and V. Timm, "Applying asynchronous circuits in contactless smart cards," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, April 2000, pp. 36–44.
[12] J. Kessels and P. Marston, "Designing asynchronous standby circuits for a low-power pager," *Proc. IEEE*, vol. 87, no. 2, pp. 257–267, February 1999.
[13] K. van Berkel, R. Burgess, J. Kessels, *et al.* "A single-rail re-implementation of a DCC error detector using a generic standard-cell library," in *Proc. Conf. on Asynchronous Design Methodologies*, May 1995, pp. 72–79.
[14] H. van Gageldonk, K. van Berkel, A. Peeters, D. Baumann, D. Gloor, and G. Stegmann, "An asynchronous low-power 80C51 microcontroller," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, April 1998, pp. 96–107.
[15] A. Peeters, "Single-rail handshake circuits," Doctoral dissertation, Technische Universiteit Eindhoven, 1996.
[16] J. Kessels and A. Peeters, "The Tangram framework: asynchronous circuits for low power," in *Proc. Asia and South Pacific Design Automation Conf.*, February 2001, pp. 255–260.
[17] A. Bardsley and D. A. Edwards, "The Balsa asynchronous circuit synthesis system," in *Proc. Forum on Design Languages*, September 2000.
[18] S. B. Furber, D. A. Edwards, and J. D. Garside, "AMULET3: a 100 MIPS asynchronous embedded processor," in *Proc. Int. Conf. on Computer Design (ICCD)*, September 2000, pp. 329–334.
[19] M. Benes, S. M. Nowick, and A. Wolfe, "A fast asynchronous Huffman decoder for compressed-code embedded processors," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, 1998, pp. 43–56.
[20] S. B. Furber, J. D. Garside, P. Riocreux, S. Temple, P. Day, J. Liu, and N. C. Paver, "AMULET2e: an asynchronous embedded controller," *Proc. IEEE*, vol. 87, no. 2, pp. 243–256, February 1999.
[21] S. Rotem, K. Stevens, R. Ginosar, *et al.*, "RAPPID: an asynchronous instruction length decoder," *IEEE J. Solid-State Circuits*, vol. 36, no. 2, pp. 217–228, 2001.
[22] A. J. Martin, A. Lines, R. Manohar, M. Nystroem, P. Penzes, R. Southworth, and U. Cummings, "The design of an asynchronous MIPS R3000 microprocessor," in *Adv. Res. VLSI*, pp. 164–181, September 1997.
[23] H. Terada, S. Miyata, and M. Iwata, "DDMPs: self-timed super-pipelined data-driven multimedia processors," *Proc. IEEE*, vol. 87, no. 2, pp. 282–296, February 1999.
[24] K. Y. Yun, P. A. Beerel, V. Vakilotojar, A. E. Dooply, and J. Arceo, "The design and verification of a high-performance low-control-overhead asynchronous differential equation solver," *IEEE Trans. VLSI Systems*, vol. 6, no. 4, pp. 643–655, Dec. 1998.
[25] U. Cummings, A. Lines, and A. Martin, "An asynchronous pipelined lattice structure filter," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, November 1994, pp. 126–133.
[26] A. Davare, K. Lwin, A. Kondratyev, and A. Sangiovanni-Vincentelli, "The best of both worlds: the efficient asynchronous implementation of synchronous specifications," in *Proc. ACM/IEEE Design Automation Conf.*, June 2004.

[27] A. Kondratyev and K. Lwin, "Design of asynchronous circuits using synchronous CAD tools," *IEEE Design & Test of Computers,* vol. 19, no. 4, pp. 107–117, 2002.

[28] M. Ligthart, K. Fant, R. Smith, A. Taubin, and A. Kondratyev, "Asynchronous design using commercial HDL synthesis tools," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, April 2000, pp. 114–125.

[29] A. Smirnov, A. Taubin, M. Karpovsky, and L. Rozenblyum, "Gate transfer level synthesis as an automated approach to fine-grain pipelining," in *Proc. Workshop on Token Based Computing (ToBaCo)*, Bologna, June 2004.

[30] A. Smirnov, A. Taubin, and M. Karpovsky, "Automated pipelining in ASIC synthesis methodology: gate transfer level," in *Proc. ACM/IEEE Design Automation Conf.*, June 2004.

[31] D. H. Linder and J. C. Harden, "Phased logic: supporting the synchronous design paradigm with delay-insensitive circuitry," *IEEE Trans. Computers*, vol. 45, no. 9, pp. 1031–1044, September 1996.

[32] R. Reese, M. A. Thornton, and C. Traver, "A coarse-grain phased logic CPU," *IEEE Trans. Computers*, vol. 54, no. 7, pp. 788–799, July 2005.

[33] M. Renaudin, P. Vivet, and F. Robin, "ASPRO-216: a standard-cell QDI 16-bit RISC asynchronous microprocessor," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, 1998, pp. 22–31.

[34] M. Ferretti and P. A. Beerel, "High performance asynchronous design using single-track full-buffer standard cells," *IEEE J. Solid-State Circuits*, vol. 41, no. 6, pp. 1444–1454, June 2006.

[35] R. Ozdag and P. Beerel, "A channel based asynchronous low power high performance standard-cell based sequential decoder implemented with QDI templates," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, April 2004, pp. 187–197.

[36] P. Golani and P. A. Beerel, "Back-annotation in high-speed asynchronous design," *J. Low Power Electronics*, vol. 2, pp. 37–44, 2006.

[37] S. M. Nowick, "Design of a low-latency asynchronous adder using speculative completion," *IEE Proc. Computers and Digital Techniques*, vol. 143, no. 5, pp. 301–307, September 1996.

[38] S. Kim, Personal communication, 2006.

[39] F. Te Beest, A. Peeters, K. Van Berkel, and H. Kerkhoff, "Synchronous full-scan for asynchronous handshake circuits," *J. Electron. Test.*, vol. 19, no. 4, pp. 397–406, August, 2003.

[40] A. Kondratyev, L. Sorensen, and A. Streich, "Testing of asynchronous designs by 'inappropriate' means. Synchronous approach," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, April, 2002 pp. 171–180.

[41] I. E. Sutherland, "Micropipelines," *Commun. ACM*, vol. 32, no. 6, pp. 720–738, June 1989.

[42] J. Teifel and R. Manohar, "An asynchronous dataflow FPGA architecture," *IEEE Trans. Computers*, vol. 53, no. 11, pp. 1376–1392, November 2004.

[43] D. Fang, J. Teifel, and R. Manohar, "A high-performance asynchronous FPGA: test results," in *Proc. 13th Annual IEEE Symp. on Field-Programmable Custom Computing Machines*, April 2005, pp. 271–272.

[44] A. Bink, R. York, and M. de Clercq, "ARM996HS: the first licensable, clockless 32-bit processor core," in *Proc. Symp. on High-Performance Chips (Hot Chips 18)*, August 2006.
[45] Silistix, www.silistix.com.
[46] Handshake Solutions, www.handshakesolutions.com.
[47] Fulcrum Microsystems Inc., www.fulcrummicro.com.
[48] Achronix Semiconductor Co., www.achronix.com.

2 Channel-based asynchronous design

This chapter is an introduction to asynchronous handshaking and typical asynchronous modules, with an emphasis on asynchronous designs that follow a channel-based discipline. The detailed implementation of these modules will be described in later chapters.

2.1 Asynchronous channels

As mentioned earlier, asynchronous designs are often composed of a hierarchical network of *blocks*, which contain ports interconnected via *asynchronous channels*. These channels are simply a bundle of wires and a protocol for synchronizing computation and communicating data between blocks. The smallest block that communicates with its neighbors using asynchronous channels is called a *leaf cell*. Larger blocks that communicate via channels may be called *modules*.

Numerous forms of channels have been developed that trade robustness to timing variations with improved power and performance, and this section reviews some of the most popular forms.

2.1.1 Bundled-data channels

Bundled-data channels consist of a single request line bundled with a uni-directional single-rail data bus that is coupled with an acknowledgement wire. In the typical bundled-data *push* channel, illustrated in Figure 2.1, the sender initiates the communication and tells the receiver when new valid data is available.

Bundled-data channels can be implemented with two-phase handshaking, in which there is no distinction in meaning between the rising and falling transitions of the request and acknowledge handshaking wires. More specifically, both the rising and falling transitions of the request wire indicate the validity of new data at the output of the sender, and both the rising and falling transitions of the acknowledge signal indicate that the data has been *consumed* by the receiver, as illustrated in Figure 2.2. Note that after the data is consumed the sender is free to change the data. Consequently, it is the voltage transitions of these wires that carry meaning rather than their absolute voltage levels. Thus, this type of two-phase protocol is often called *transition signaling* [9].

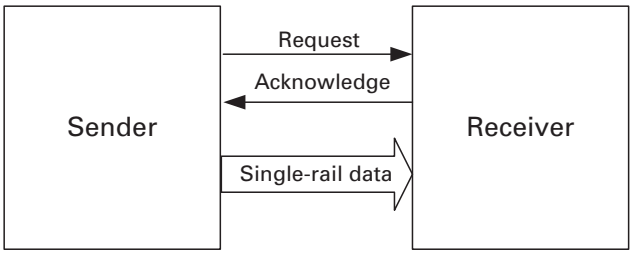

Figure 2.1. The elements of a bundled-data channel.

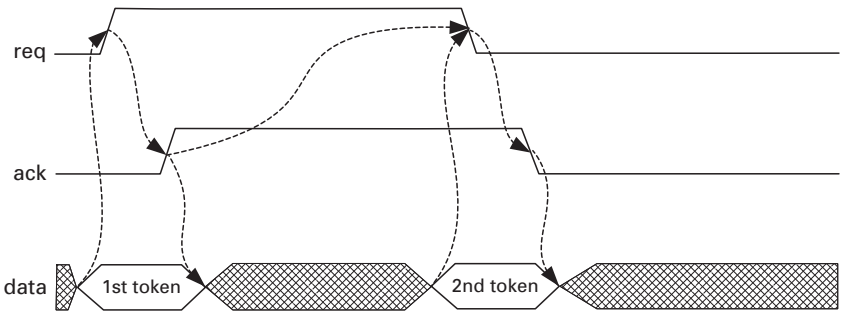

Figure 2.2. Transition signaling in a bundled-data push channel.

In contrast, in four-phase bundled-data protocols, only the request line going high signifies the validity of data and a single transition of the acknowledge signal identifies that the data has been consumed by the receiver and can be safely altered by the sender. In the *narrow* protocol [13], also known as the *early* protocol [6], the rising transition of the acknowledge signal signifies that its data has been consumed and the single-rail data can be changed, as illustrated in Figure 2.3(a). After the acknowledge signal has risen the sender resets the request line and the receiver then resets the acknowledgement line, bringing both wires back to the initial state in preparation for the next token transfer. By contrast, in the *late* [6] protocol the data is valid from when the request falls to when the acknowledge signal falls, and the first two phases can be considered as a warm-up in preparation for the latter two active transitions; this is illustrated in Figure 2.3(b). In the *broad* protocol [13], the signals follow the same four transitions but the falling transition of the acknowledge signal signifies that the data can be reset, as illustrated in Figure 2.3(c). As the name implies, the time for which the data must be stable is much larger in the broad protocol than in the narrow protocol. Also, notice that in all these protocols the only purpose of two of the four phases of the communication is to reset the handshaking wires in preparation of the next data transfer.

Bundled-data channels are area efficient because the request and acknowledge line overhead is distributed over the width of the entire data bus which encodes data with one wire per bit, as in a typical synchronous circuit. They are also power efficient because the activity on the data bus may be low and the power consumption of the handshaking wires is relatively small if the data path is wide

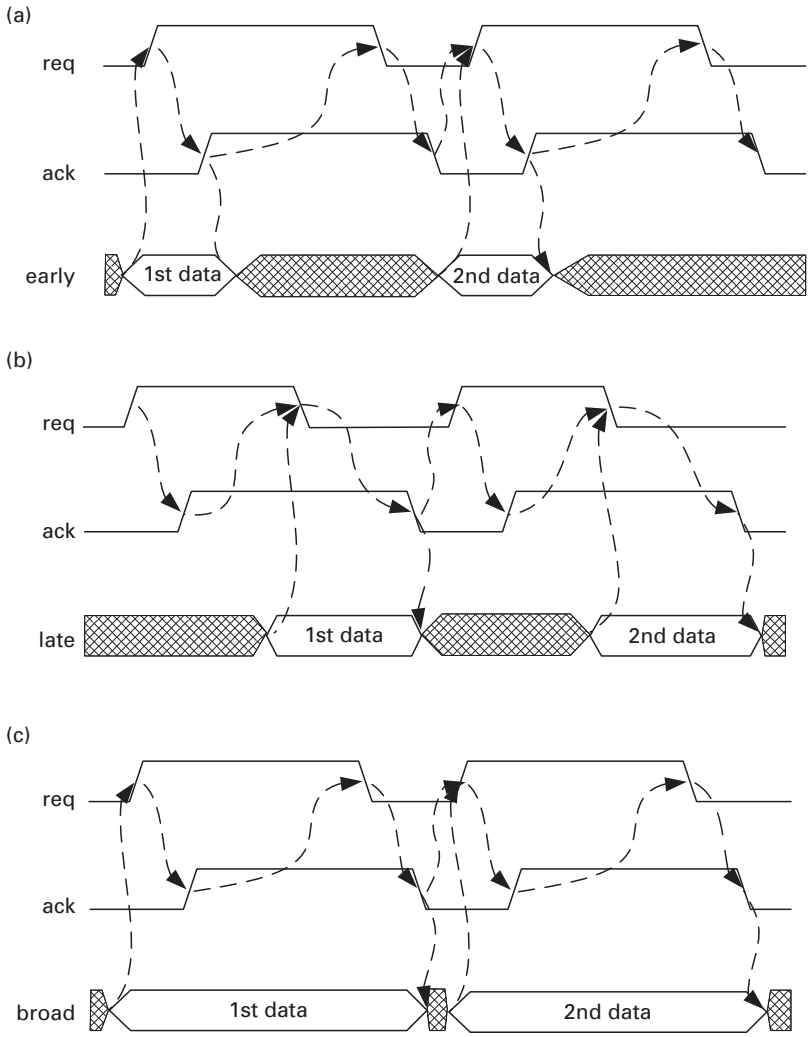

Figure 2.3. Four-phase bundled-data push protocols: (a) early, (b) late, (c) broad.

(e.g. 64 bits). A disadvantage of bundled-data channels, however, is that they involve a timing assumption, i.e. that the data at the receiver is valid upon the associated transition of the request line. Moreover, unlike in a synchronous circuit there is no associated clock that can be slowed down to ensure that this timing assumption is satisfied. Consequently, significant margins are typically necessary to ensure proper operation. These margins are on the forward latency path of the circuit, which is often critical to system performance.

2.1.2 One-of-*N* channel

A 1-of-N channel uses N data wires to send $\log_2 N$ bits of data, as illustrated in Figure 2.4, and is typically designed with four phases. After a single data wire has

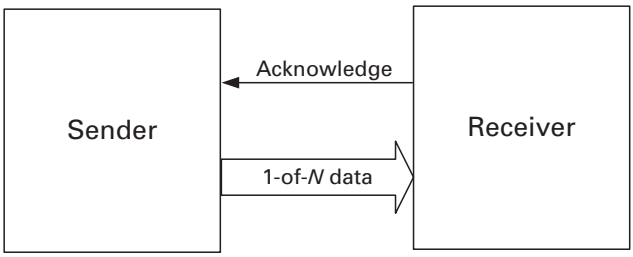

Figure 2.4. The 1-of-N channel.

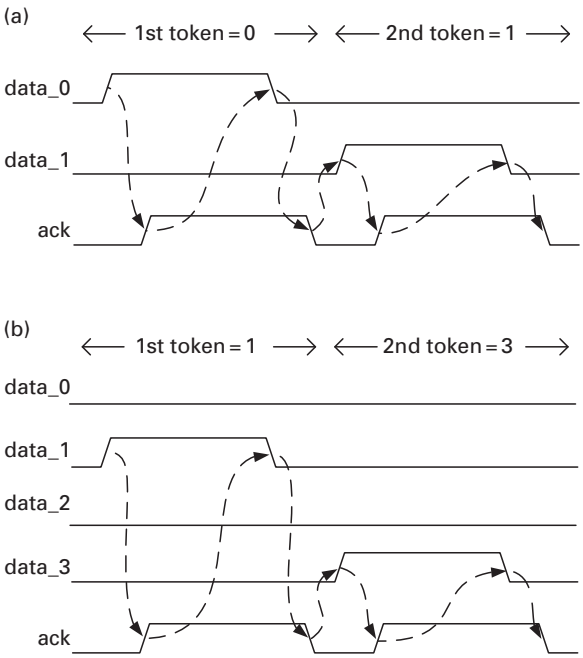

Figure 2.5. (a) The 1-of-2 (dual-rail) channel; (b) the 1-of-4 channel.

risen, the acknowledge wire also rises, indicating that the data has been consumed by the receiver and that the sender can safely reset the risen data wire. Once the sender resets the data signal, the receiver can reset the acknowledge signal, completing the four phases. This protocol facilitates *delay-insensitive communication* between blocks in that the data and its validity are encoded on the same N wires rather on separate request and data wires. Consequently, no timing assumption is needed. This increases robustness to variations, reducing the amount of timing verification required.

The most well-known form of this channel is dual-rail, also known as 1-of-2, and uses two data wires or *rails* per bit of data, as illustrated in Figure 2.5(a). One

wire is referred to as the data_1 or true wire and the other as the data_0 or false wire. Handshaking is initiated when a data wire rises; this represents both the value of the data wire and the validity of the data. The receiver can detect the validity of the input data simply by ORing the data_0 and data_1 wires. In many templates the data triggers a domino dual-rail logic datapath concurrently.

Other forms of 1-of-*N* datapath are also possible. A 1-of-4 channel systems communicates two bits of data by changing only one data wire, as illustrated in Figure 2.5(b), yielding a lower power consumption than dual-rail channels yet with no additional data wires per bit (i.e. two data wires are still used per bit communicated). Higher radix channels are also possible and have additional power advantages but require more wires per data bit. For example, a 1-of-8 channels system requires 8/3 data wires per bit rather than the two data wires per bit required by dual-rail and 1-of-4 channels. The most trivial form is a 1-of-1 channel, which has the same form as a bundled-data channel with no data and is used to synchronize operations between sender and receiver.

It is also possible to implement 1-of-*N* channels with the acknowledge signal inverted, in which case it is often referred to as an *enable* because a high value indicates that the channel is ready to accept a new token. In addition, as in bundled-data transition signaling, it is also possible to use transition signaling, in which each transition on the 1-of-*N* data wires represents a new token. However, creating senders and receivers that react to both phases of the data and control wires often leads to less efficient control circuits, owing to the substantially lower mobility in P-transistors compared with that in N-transistors.

Various block implementations for 1-of-*N* channels will be discussed in Chapters 11 and 12. In addition, block implementations that use a variant known as a 1-of-$N+1$ channel will be discussed in Chapter 11. The 1-of-$N+1$ channel has an additional request signal between sender and receiver to enable higher performance.

2.1.3 Single-track 1-of-*N* channel

A single-track 1-of-*N* channel have 1-of-*N* data with no acknowledgement, as illustrated in Figure 2.6.

The 1-of-*N* single-track channel is implemented with two phases, as illustrated in Figure 2.7. The sender drives one of the *N* wires high, thereby sending a token, and after receiving this token the receiver drives the same wire low. After driving the wire to its desired state, the sender and receiver(s) must tri-state the wire (i.e. hold it in a high-impedance state) to ensure that they do not try to drive it in opposite directions at the same time. If the drivers have a drive gap, additional staticizers are necessary to hold the wires at their driven state to combat leakage current and noise.

The two-phase single-track protocol avoids the overhead of the reset phases without requiring the sender and receiver to react to different transitions, yielding substantially higher performance than four-phase protocols. Moreover, compared

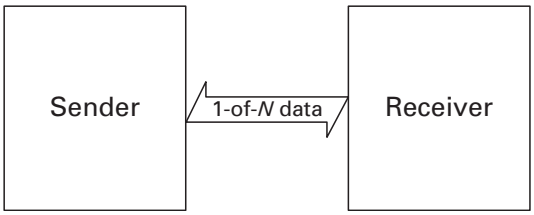

Figure 2.6. Single-track 1-of-N channel.

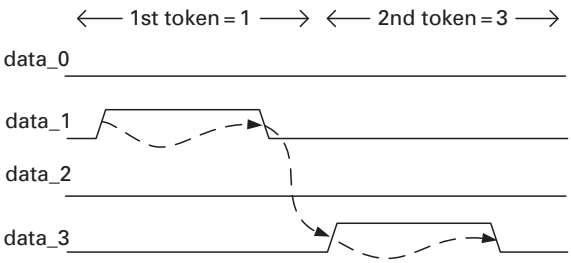

Figure 2.7. Handshaking on a 1-of-4 single-track channel.

with bundled-data protocols there is no timing assumption that requires margins on the forward latency; this yields additional performance improvement. In comparison with four-phase 1-of-*N* protocols there are fewer transitions per bit, resulting in substantially lower power. In comparison with bundled-data channels, however, the number of transitions per bit is often larger, yielding a higher power consumption per bit transmitted.

Single-track handshaking has been proposed as a replacement for the handshaking signals in ultra-high-performance bundled-data channels [8][10]. (Note that in the GasP implementation [10] the default value of the request and acknowledge wire is high rather than low.) Single-track handshaking was later extended to 1-of-*N* data for ultra-high-performance design [11]. Single-track senders and receivers will be described in detail in Chapter 13.

2.1.4 Shared channels

Asynchronous channels are typically point-to-point channels, but it is also possible to design shared asynchronous channels that connect more than two blocks. In this case, multiple blocks can be designated as senders and multiple blocks as receivers. The key to designing shared channels is to guarantee that only a single sender and single receiver are active at a given time, i.e. all senders are, in a mutually exclusive way, trying to drive the channel. The receivers can be designed to receive the data on the channel mutually exclusively or they can all be required to acknowledge receipt of the data.

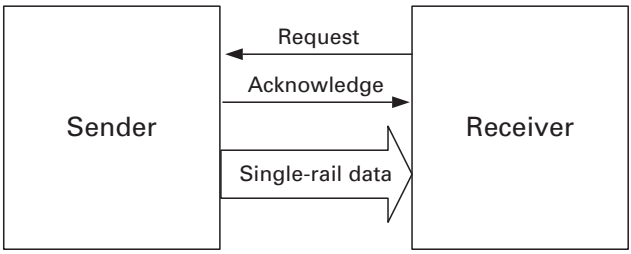

Figure 2.8. Pull channels in which the request and acknowledge wires are reversed.

2.1.5 Pull channels

Each channel connects an active and a passive port, where the active port initiates a communication and the passive port responds. In push channels the sender is the active port and starts the communication by either raising the request or raising one of the *N* data wires. This represents a *data-driven* channel. Alternatively, if the data transfer is initiated by the receiver then the receiver has the active port and the channel is said to be a *pull* channel and represents a *demand-driven* channel.

In pull channels, the directions of the request and acknowledge signals are reversed compared with those in push channels, as illustrated in Figure 2.8. Communication is initiated when the receiver raises the request signal, requesting data from the sender. As in push channels there are early, late, and broad data-validity protocols that dictate when data is stable, as illustrated in Figure 2.9. In the early scheme, once the sender has stable data at its output, it raises the acknowledge wire. The receiver then acknowledges receipt of the data by lowering its request signal. This allows the sender to change data and reset the acknowledge wire, preparing for another communication. The data is stable from acknowledge rising to request falling, which is why this protocol is sometimes also referred to as a *middle* data-validity protocol [6]. In the late and broad protocols, however, note that the data must be stable in between successive handshakes.

2.1.6 Abstract channel diagrams

It is often useful to depict asynchronous channels independently of the underlying handshaking style. This allows the diagram to focus on an architectural view of the design. In an abstract channel diagram, the direction of the channel arrow is in the direction of the data transfer and the active port is designated by a dark circle, as illustrated in Figure 2.10(a), (b). Channels without data are sometimes referred to as 1-of-1 channels, as previously mentioned, when they are used to carry request tokens for shared resources [17]. They are called synchronization or nonput channels when they do not send tokens that are consumed by a receiver but, rather, are used to manage synchronization among modules [6], as will be explained in the next section.

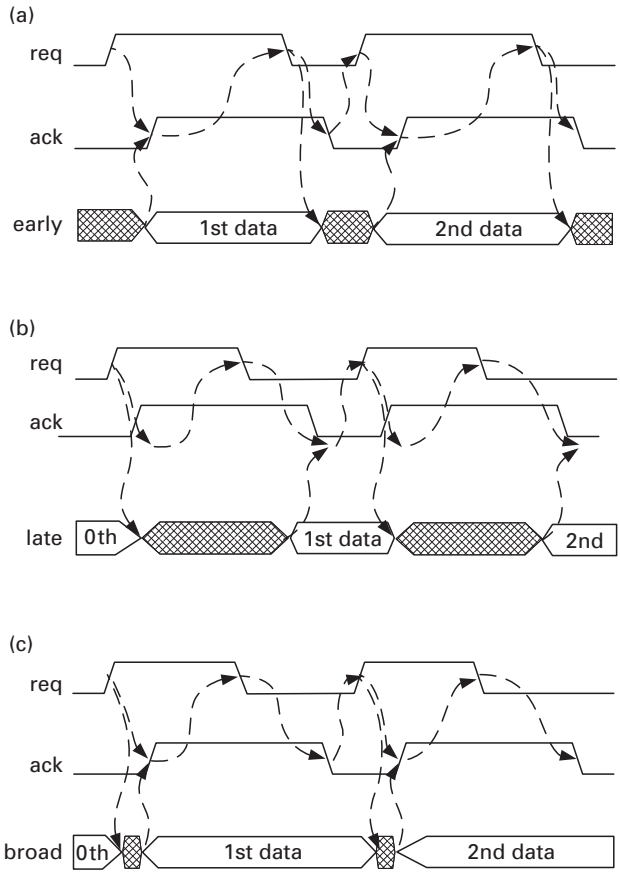

Figure 2.9. Data-validity schemes in four-phase pull channels: (a) early, (b) late, (c) broad.

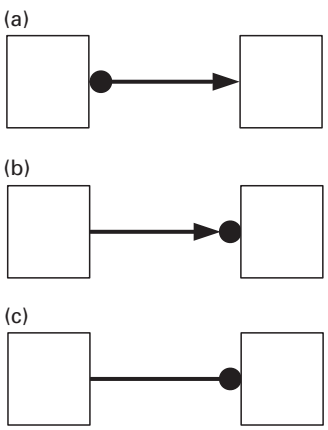

Figure 2.10. Abstract channels: (a) push channel with data, (b) pull channel with data, (c) synchronization or nonput channel with active side on the right.

2.2 Sequencing and concurrency

In addition to communicating data between blocks, channels are used for sequencing operations performed by the modules they connect. Clearly, a module must receive data on its input channels before it can operate on the data and send its result(s) on its output channel(s). However, it is often useful for the handshaking on the input and output channels to be overlapped. The specific interleaving chosen between the transitions on the input and output channels changes the degree of concurrency among the associated modules, which ranges from sequential to parallel to pipelined behavior.

2.2.1 Enclosed handshaking

A form of interleaving commonly used for sequencing is the *enclosed handshake*, in which the handshaking on one channel is enclosed within the handshaking of another channel. To illustrate this behavior we will use a two-phase protocol, as in Figure 2.11, even though efficient implementations typically use a four-phase protocol. Four-phase alternatives with detailed implementations will be discussed in more detail in Chapter 10.

Notice that communications on the R synchronization channel (R1 and R2, see Figure 2.11) are enclosed within the handshaking on the L channel (L1 and L2). The application of this form of handshaking depends on the semantics of the enclosed handshake. Typically, the enclosed handshake represents the completion of some function and thus the enclosing handshake L represents the calling of a function represented by the communication on R.

Two common applications of enclosed handshaking are described below, implementing sequential and parallel behavior, respectively.

SEQ module

The handshaking on multiple active channels can be ordered, thereby ensuring that the associated tasks are sequenced. As depicted in Figure 2.12, upon receipt of

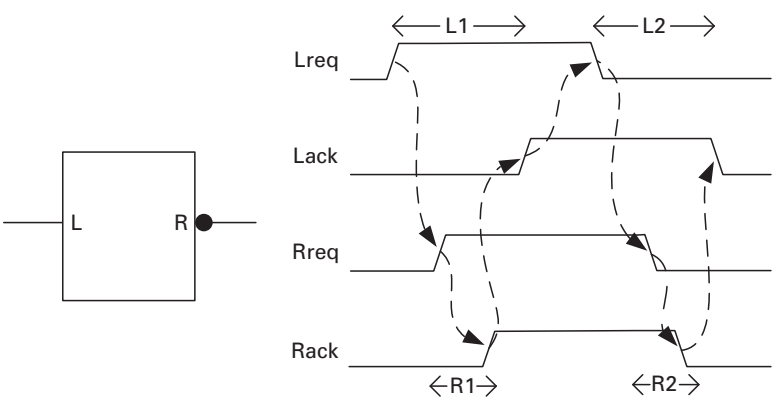

Figure 2.11. Enclosed handshaking for a two-phase protocol.

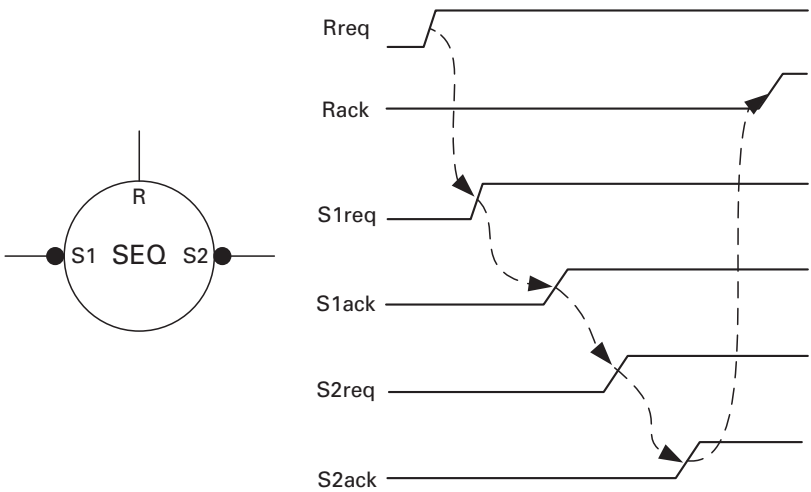

Figure 2.12. SEQ module using a two-phase protocol.

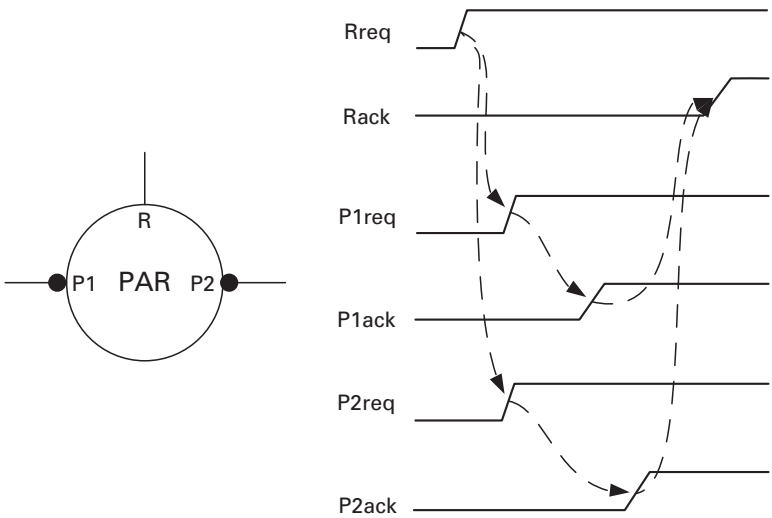

Figure 2.13. PAR module using a two-phase protocol.

a request on passive port R the sequence (SEQ) module completes a full handshake on active port S1 followed by a full handshake on active port S2 before acknowledging R. The SEQ module is used to sequence operations associated with handshakes on S1 and S2. Moreover, because the resulting sequence is enclosed within the handshake on R, the acknowledgement on R indicates that the sequence is complete.

PAR module

The handshaking on multiple active channels can occur simultaneously, thereby ensuring that the associated tasks operate in parallel. In particular, as depicted in Figure 2.13, upon receipt of a request on port R the parallel (PAR) module

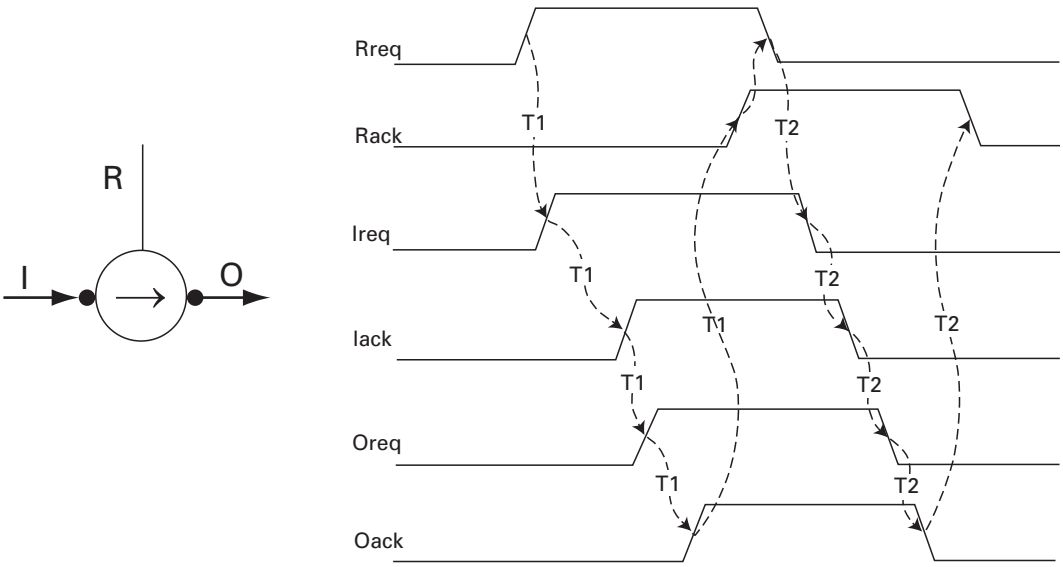

Figure 2.14. A transferer: (a) its symbol and (b) its timing diagram, assuming two-phase handshaking over two requests on R.

completes a full handshake on port P1 in parallel (i.e. concurrently) with a full handshake on port P2 before acknowledging R. As in the SEQ module, because this sequence is enclosed within the handshake on R the acknowledgement on R indicates that the pair of operations is complete. This enables this pair to be properly sequenced with subsequent operations.

The handshaking procedure using the two-phase protocol is straightforward and there are no variants to consider. In four-phase protocols, however, several variants exist depending on which phases represent active handshaking rather than being needed solely to reset the wires. In particular, an overlap of the reset phases of one handshake with the active phases of the next handshake can reduce control overhead and improve performance [15].

Transferer

A *transferer*, denoted by →, waits for a request on its passive nonput port and then initiates a handshake on its pull input port, as illustrated in Figure 2.14. The handshake on the pull input channel is then relayed to the push output channel in order to pull data from its input channel and push it (i.e. *transfer* it) to its output channel. Finally, the transferer completes the handshaking on its passive nonput channel.

2.2.2 Pipelined handshaking

Pipelining can improve system performance by enabling multiple instances of an algorithm to operate concurrently. This is achieved by decomposing the algorithm

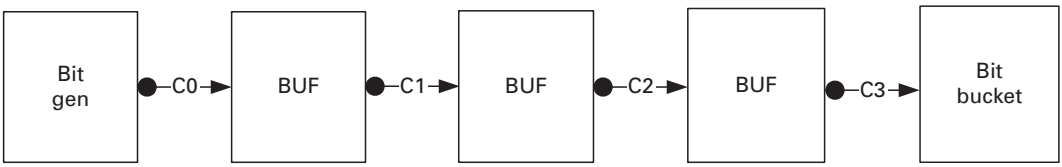

Figure 2.15. Three-stage FIFO surrounded by bit-generator and bit-bucket modules.

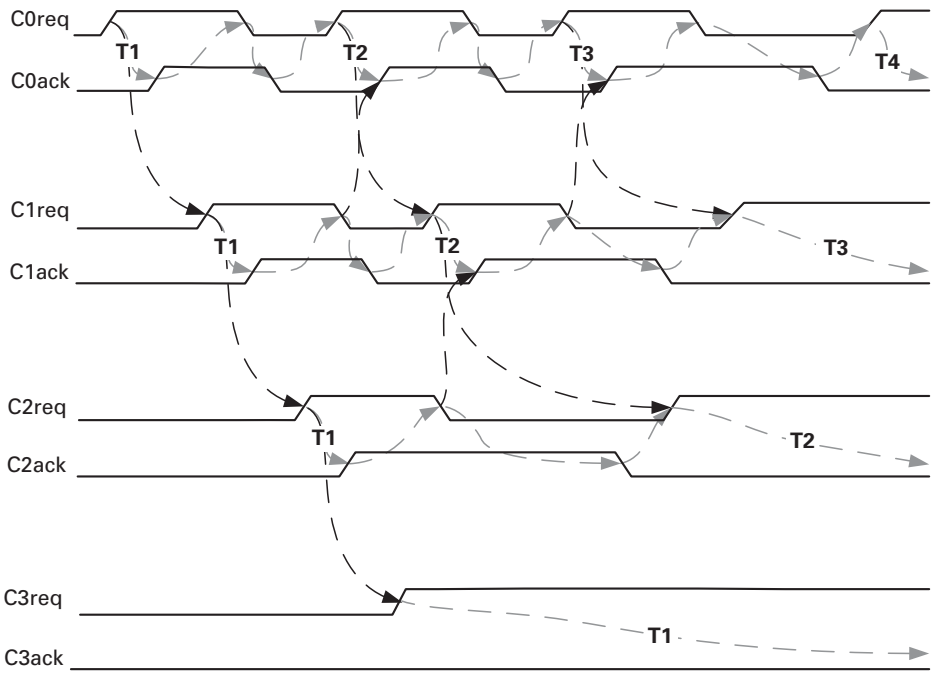

Figure 2.16. Four-phase full-buffer handshaking for a 1-of-1 pipeline.

into steps, associating steps with pipeline stages, and enabling earlier pipeline stages to accept new instances of the algorithm while later pipeline stages are concurrently operating on earlier problem instances. Pipelining can be implemented explicitly using parallel PAR operators or implicitly through pipelined handshaking between blocks.

To illustrate the latter, consider a three-stage pipeline of simple buffers, each denoted BUF, surrounded by an environment modeled by a bit generator on the left that randomly generates tokens and a bit bucket on the right that simply consumes tokens, as illustrated in Figure 2.15.

After consuming the data, each BUF generates its output in parallel with acknowledging the input; this enables the previous stage to reset and generate new tokens. One specific sequence of events is shown in Figure 2.16 for a 1-of-1 pipeline that follows the *fully decoupled handshake protocol* [12]. Note that the gray arrows represent the four-phase handshaking protocol within a channel and the

black arrows identify the interleavings of events across handshaking protocols. While the first token, T1, is passing through C2 a second token, T2, is able to enter the C0 channel. Also notice that in this figure we have assumed that the bit bucket is slow and therefore the signal C3ack does not rise. Consequently, tokens begin to pile up one behind the other and towards the end of the simulation four distinct tokens, T1, T2, T3, and T4, exist simultaneously in the pipeline. The handshaking guarantees that each channel holds at most one token and that no token is lost, despite the relative speed of the bit bucket. Thus the pipeline inherently provides *flow control* of the tokens.

An asynchronous pipeline can have a simple linear configuration, such as the first-in–first-out (FIFO) arrangement in Figure 2.15, or it can have non-linear elements such as forks, joins, cycles, or rings. Despite the configuration, the common theme of pipelining is that many data tokens that represent intermediate results of different instances of the same algorithm can exist simultaneously in the pipeline. We may refer to the class of leaf cells that form a pipeline stage as *buffers* because they inherently store tokens on their input and output channels. The functionality and type of buffer should be clear from the context.

Full buffers versus half buffers

While it is assumed in pipelined handshaking that a channel cannot hold more than one token, not all types of asynchronous leaf cell support this one-token maximum. A leaf cell is a *full buffer* [5], also known as a *high-capacity* cell [14], when it can support distinct tokens on its input and output channels. *Half-buffer* leaf cells, however, cannot support distinct tokens on their input and output channels. More specifically, an N-stage FIFO made up of half buffers can support a maximum of $N/2$ tokens. The maximum number of tokens that a pipeline can support is called its *capacity* or *static slack* [5].

The difference between half and full buffers can be understood better by analyzing the difference in the handshake interleavings on their input and output channels. To appreciate this difference, consider again the operation of the three-stage pipeline shown in Figure 2.15 (with 1-of-1 channels and a slow bit bucket) when each buffer is changed to a half buffer that follows the *simple four-phase protocol* [12]. As illustrated in Figure 2.17, C0ack− depends on C1req−, which depends on C1ack+, which in turn depends on C2req+. These dependencies imply that the first token must pass into channel C2 before the handshake on channel C0 is complete and the latter can accept a new token. In fact, because of the additional causality of this protocol, a token must always pass through two buffers before the first of the two buffers completes its handshake and can accept a second token. In other words, only every other stage can hold a distinct token. Consequently, at the end of the simulation waveforms illustrated in Figure 2.17, because the bit bucket is very slow tokens pile up and are present in channels C1 and C3 while channels C0 and C2 are empty.

There are full and half-buffer design templates that use bundled-data, 1-of-N, on single-track encodings; they vary in terms of power, performance, and

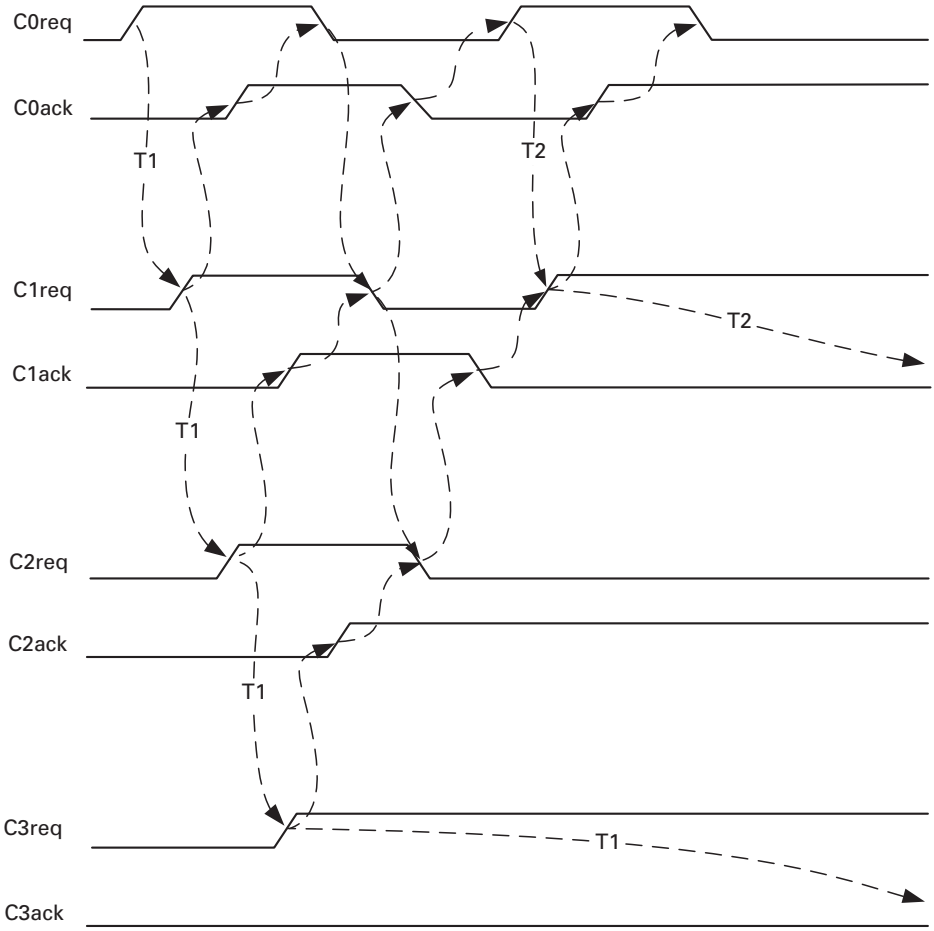

Figure 2.17. Four-phase half-buffer handshaking for a 1-of-1 pipeline.

robustness. In bundled-data pipelines, the handshake circuit is independent of the datapath but drives a local clock or enable signal that controls latches or flip-flops. It will be studied in more detail in Chapters 9 and 10. In 1-of-*N* and single-track templates, the storage transistors are often integrated within the cell itself. They will be studied in Chapters 11–13. In all cases, notice that the acknowledge signal typically means that the associated data has been consumed rather than some operation has been completed, in contrast with the situation for an enclosed handshake.

Non-linear pipelines

Most high-performance architectures are not linear pipelines and thus buffers with multiple input and output channels are often needed.

A *fork* is a buffer with one input channel and multiple output channels. The simplest type of fork is a two-output COPY, which simply copies the token on the

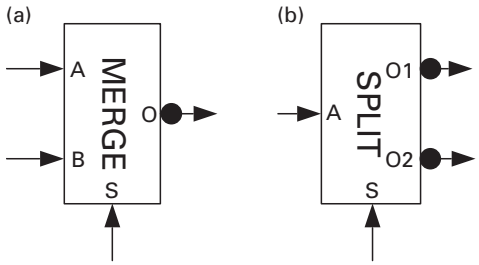

Figure 2.18. (a) Merge and (b) split non-linear conditional pipeline buffers.

input channel to both output channels. A *join* is a buffer with multiple input channels. Buffers with multiple inputs and outputs are both forks and joins. Buffers are *unconditional* if they wait for input tokens on all input channels and generate tokens on all output channels. They are *conditional* if the input channels that are read or output channels that are written depend on the value of a subset of the tokens read. A typical conditional cell is a merge, illustrated in Figure 2.18(a), which, on the basis of the value of a select token on channel S, waits for an input token on one of the other two input channels and then routes it to the output channel. The merge cell is essentially a synchronous multiplexor with built-in flow control. A typical conditional output cell is a split, illustrated in Figure 2.18(b), which, on the basis of the value of a select token on S, routes an input token to one of the two output channels.

As with linear buffers, non-linear buffers have inherent flow control. No token is lost, because no buffer is allowed to overwrite a token. Moreover, as was the case for BUF elements, both full- and half-buffer implementations of merge and split elements exist. As an example, the timing diagram of a merge with two-phase handshaking on A, B, and dual-rail S channels is shown in Figure 2.19. Notice that this implementation is an example of full-buffer handshaking because tokens can exist on both input and output channels at the same time. In particular, the second token on A (identified by Areq−) arrives while the first output token on O is still being acknowledged (i.e. before Oack+).

2.3 Asynchronous memories and holding state

Notice that the SEQ and PAR modules operate on only request and acknowledge signals or 1-of-1 channels and are thus distinct from the datapath. Despite having an internal state variable, which is necessary to ensure the proper handshaking sequence, these modules do not include multi-bit latches or flip-flops in which data is stored. In pipeline buffer modules, the value of the data is stored on the channels. Neither type of module represents a means of storing data or building a memory.

One type of asynchronous memory is a cell that stores a single variable, denoted as VAR. It is depicted in Figure 2.20(a). It has a passive write port W attached to a

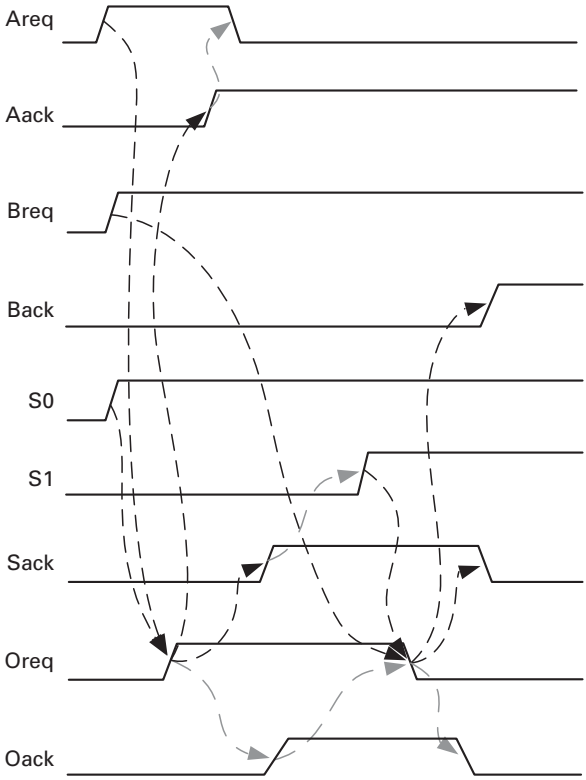

Figure 2.19. Timing diagram of a merge cell using full-buffer two-phase handshaking and a dual-rail select channel.

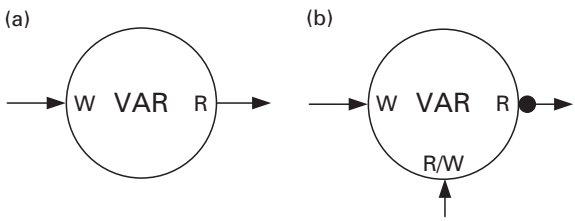

Figure 2.20. Illustration of VAR modules for data storage (a) with mutually exclusive read and write channels, (b) with a one-bit read or write control channel.

push channel and a passive read port R attached to a pull channel. Other modules actively send data to this module to be stored and actively request to read the data stored. To avoid malfunction, the write and read requests are assumed to be mutually exclusive. Alternatively, one can design a VAR module with a one-bit control port R/W, illustrated in Figure 2.20(b), which indicates whether the module should wait for a token on its passive write port, which it will then store, or generate a previously stored value on its active read port. Notice that with a

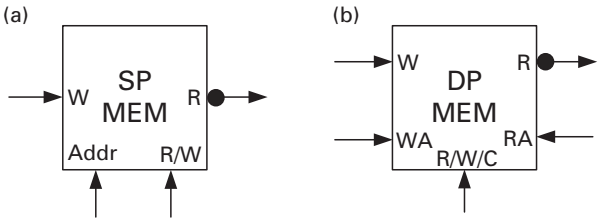

Figure 2.21. Multi-variable (a) single-ported (SP) and (b) dual-ported (DP) memories.

one-bit control the environment need not ensure mutual exclusivity on the read and write channels.

Another way to implement the memory of a single variable is by not generating an acknowledge signal on the channel in which the associated token is sent. This prevents reset of the data, which can then be re-used in multiple computations. Although this technique can be used for specific problems such as *loop control* [5], it is very limited.

Other memory modules are designed to hold more than one variable [18]. Such modules use additional address channels to identify which location to read to or write from the data. A one-ported memory has a single address channel and can either write to or read from distinct input and output channels, as illustrated in Figure 2.21(a). Note that it is also possible to combine read and write channels into a common bi-directional channel if access to the channel can be guaranteed to be mutually exclusive. Dual-ported memories have two address channels and a three-way R/W/C control channel that indicates whether the memory should read, write, or simultaneously read and write to different addresses, as illustrated in Figure 2.21(b). In both cases, the only active channel is associated with the read port, which generates the resulting memory token.

Another form of memory in an asynchronous system is a finite-state machine (FSM). An FSM is a state-holding circuit, which only changes state when the expected inputs for that state become available. To build an asynchronous FSM, we create distinct blocks for *output* and *next-state logic* and feed the outputs of the next-state logic back to the inputs using pipeline buffers to hold the data [5]. This technique is similar to synchronous FSM design, but the flip-flops are replaced with pipeline buffers, as illustrated in Figure 2.22.

In this figure, each channel either is an input or output or holds a state. The blocks labeled C are COPY buffers. The next-state and output logic blocks may involve complex logic. For efficient implementation, these blocks may be decomposed into multiple leaf cells interconnected with additional channels. In addition, the input and next-state bits may be decomposed into multiple channels. For example, one strategy is to design the FSM using *one-hot-state encoding*, in which each different state is represented by a different 1-of-1 next-state channel. In this case, the next-state logic will conditionally write to the specific output channel corresponding to the desired next state. Moreover, next-state logic cells may

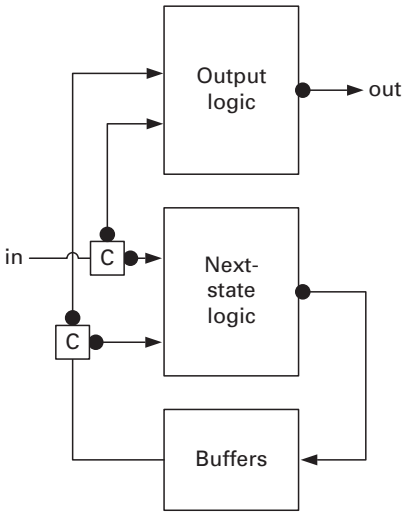

Figure 2.22. An abstract asynchronous FSM.

conditionally read primary input tokens on the basis of the state inputs. Thus, these cells may exhibit the conditional reading of input channels and writing of output channels, similarly to the non-linear pipeline templates described earlier.

The simplicity of this method for designing FSMs enables synchronous design techniques for generating Boolean next-state and output expressions to be used for asynchronous design. In particular, synchronous output and next-state logic can be readily implemented with an acyclic network of pipelined leaf cells.

Notice that at least one buffer in the feedback path must be a *token buffer*, which generates a token upon reset and acts as a buffer in all other respects. The initial token identifies the initial state of the controller. Token buffers will be described in more detail in subsequent chapters.

2.4 Arbiters

Arbiters are often used to provide mutually exclusive access to a resource shared by two or more requesters. In asynchronous systems, the access to shared resources is typically given to the earliest request; however, implementing a notion of priority among requesters is also possible. In this section we discuss first non-pipelined arbiters, which can handle only one arbitration event at a time, and then pipelined arbiters, which can simultaneously support multiple arbitration events.

2.4.1 Non-pipelined arbiters

Typically, two types of two-way arbiters are used in non-pipelined designs, as illustrated in Figure 2.23. The simplest form has two synchronization channels,

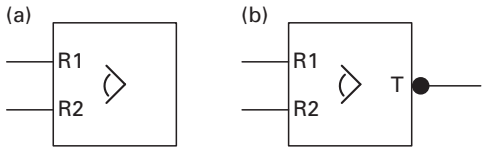

Figure 2.23. Non-pipelined arbiters: (a) two-way, (b) multi-way tree.

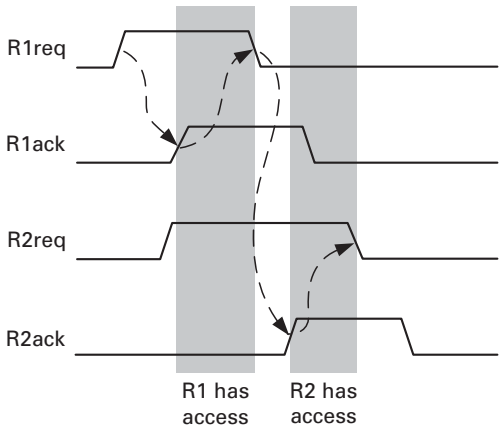

Figure 2.24. Timing diagram of the two-way arbiter in Figure 2.23(a).

connected to passive ports R1 and R2 as illustrated in Figure 2.23(a), and is typically implemented using a four-phase protocol. If the request on R1, say, is received first, it wins the arbitration and is acknowledged. Any request on the non-winner remains pending until the winning channel is reset. In this way, the winner can be guaranteed access to a shared resource until it resets its request, as illustrated in Figure 2.24.

An important facet of all two-way arbiters is that if the requests on R1 and R2 arrive sufficiently close together then the winner is randomly determined. Moreover, in this situation, the time to resolve arbitration may be arbitrarily long and its delay has an exponential distribution.

Arbiters that are *N*-way can be generated using the two-way tree arbiters shown in Figure 2.23(b), in which the 1-of-1 output T is the input of subsequent two-way arbiters. In this arbiter the winner is not acknowledged until the output T is generated and acknowledged, which occurs only when it wins the *N*-way arbitration. Once the output T is acknowledged, the tree arbiter waits until after the winning request is reset before resetting T, guaranteeing mutually exclusive access to the resource. A balanced tree yields fair arbitration but unbalanced trees are needed if certain request inputs should have higher priority.

A four-way arbiter is shown in Figure 2.25. Note that if the output T is generated at the same time that the winner of the arbitration is decided, the four-way arbiter will typically choose the request that arrives first. If T is

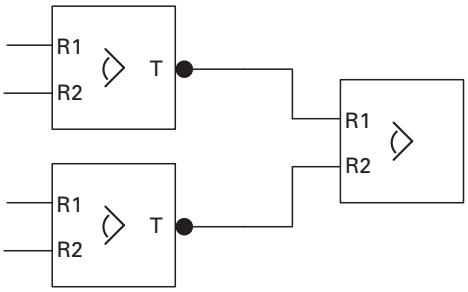

Figure 2.25. Non-pipelined four-way tree arbiter.

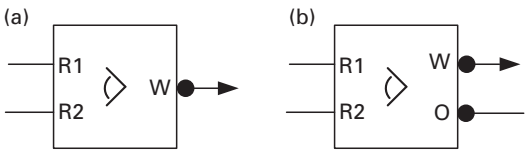

Figure 2.26. Two-way arbiters used in (a) pipelined designs and (b) pipelined multi-way tree arbiters.

generated only after a winner is decided, however, the four-way arbiter may not always choose the first-arriving request. In particular, if two requests arrive at one arbiter and one request arrives slightly later at the other arbiter, this later request may win because it can generate a request on T while the first arbiter decides between the two other requests. This tree arbiter is non-pipelined or *slackless*, because it can operate only on one arbitration event at a time.

2.4.2 Pipelined arbiters

In pipelined systems, the two-way arbiter often has an active one-bit output port, W, that identifies who has won the arbitration, as illustrated in Figure 2.26(a). In this case, the input environment can reset the winning request concurrently with the sending of the output token on W, in contrast with the arbiter in Figure 2.23(a), which delays the resetting of input channels until the shared resource is released. This is possible because the winner token captures the information about who won the arbitration event and will control access to the shared resource. This will be illustrated in a simple crossbar design in Section 2.5.

Figure 2.26(b) shows an extension to the basic pipelined two-way arbiter that includes a one-bit nonput channel. This is useful for pipelined tree arbiters, such as that illustrated in Figure 2.27. Here, the nonput ports O of the two first-level arbiters are connected to a second-level arbiter. To identify which of the four requests wins the arbitration, the first-level one-bit winner tokens are re-coded into two bits by the addition of an appropriate most significant bit (MSB). The output of the second-level arbiter controls a final MERGE, which selects between

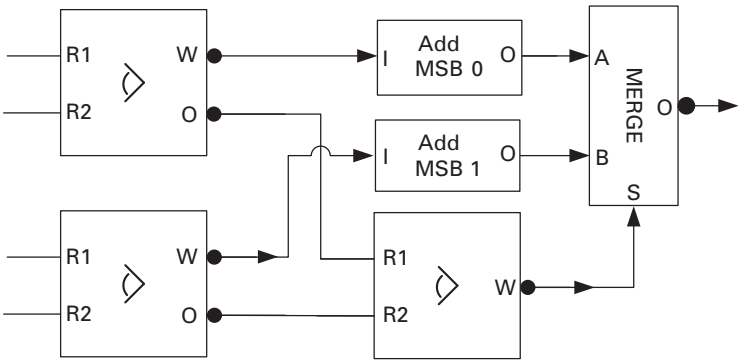

Figure 2.27. Four-way pipelined tree arbiter.

the two-bit tokens and identifies the overall winner. The key feature of this tree arbiter is that it can operate on more than one arbitration request at a time. In particular, assuming that each handshake is implemented with full buffers, after the first arbitration event is processed by the first-level arbiters and propagated to the second level of the tree a new set of input requests can arrive and be arbitrated.

2.5 Design examples

This section uses the building blocks described above in two slightly more complex design examples.

2.5.1 Two-place FIFO

As mentioned earlier, pipelining can be implemented explicitly via parallel operators. As an example, consider the simplest form of a pipeline, a two-place FIFO, as illustrated in Figure 2.28.

The two-place FIFO has capacity 2 because it can hold a maximum of two tokens. The bottom row of handshake components includes two variables, x and y, and three transferer elements. The control is implemented with a parallel operator that triggers the operation of the two halves (i.e. places) of the two-place FIFO. In particular, each half of the pipeline is controlled by a repeater (denoted by an asterisk) and a sequencer. The two halves are synchronized using the join element, denoted by the large solid circle. The repeaters are indicated by question marks.

To understand the operation of this FIFO better, we first describe the handshaking behavior of the repeater and join elements in more detail. Upon receiving a request on its request channel R, the repeater element indefinitely repeats handshaking with its active nonput channel O. The request on R is thus never acknowledged. This is illustrated in Figure 2.29(a), which shows four complete two-phase handshakes on O, labeled T1–T4 following the initial request on R.

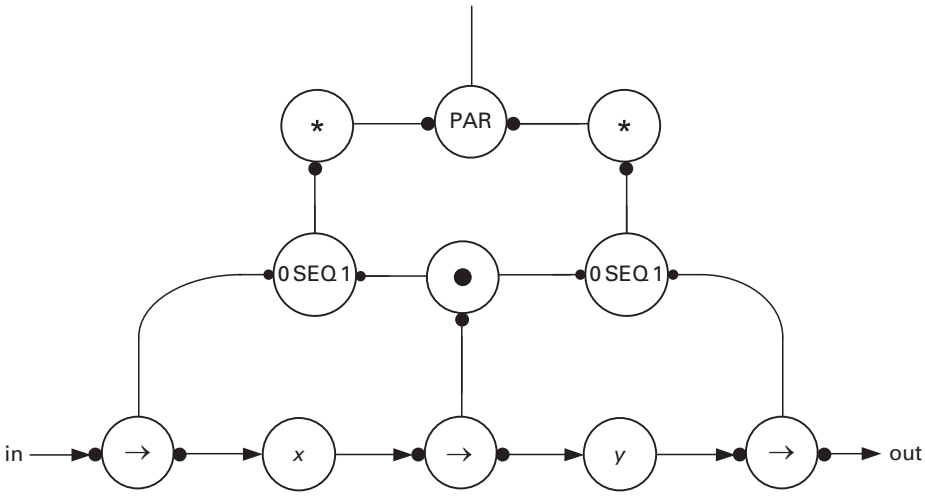

Figure 2.28. Two-place FIFO using explicit parallelism [6].

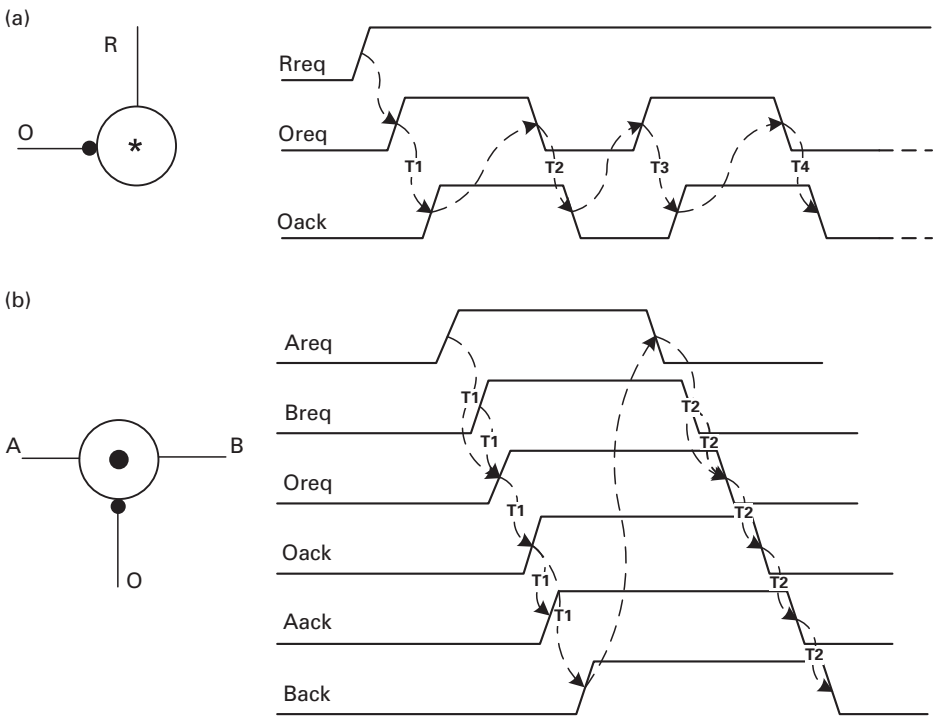

Figure 2.29. (a) A repeater element and its two-phase timing diagram. (b) A join element and its two-phase timing diagram.

The join element synchronizes communication among two input passive ports by enclosing a handshake on the active output port O after receiving requests from both A and B and before acknowledging A and B. This is illustrated in Figure 2.29(b), which shows two complete two-phase handshaking cycles.

In the case of the two-place FIFO, the left-hand repeater repeatedly handshakes its active nonput channel, which is tied to the left-hand sequencer. This sequencer is responsible for first transferring the input to the variable x and then to the variable y. The right-hand repeater repeatedly handshakes its active nonput channel, which is tied to the right-hand sequencer. This sequencer is responsible for transferring the variable x first to y and then to the output. The join element is responsible for synchronizing the two sides in such a way that the transfer from x to y occurs only when data is stored in x and when the variable y is empty. In particular, notice that the left-hand sequencer sends a request to the join only after it knows that the data is stored in x. Moreover, the right-hand sequencer sends a request to the join either the first time the FIFO is triggered (when y is known to be empty) or after the old value of y is transferred out.

Notice that the transfer of data is strictly control-driven and the transfer elements have active (pull) inputs. This is in contrast with the pipelined handshaking we described in subsection 2.2.2, which is data-driven, consisting of pipeline buffers that have passive (push) inputs.

The 2 × 2 asynchronous crossbar

The top-level view of a simple 2 × 2 asynchronous crossbar with representative sending and receiving environments is illustrated in Figure 2.30. The general idea is that the two senders will intermittently send data to either Receiver 0 or Receiver 1, as specified by an associated one-bit address channel. The receivers will intermittently consume the data sent to them but are not required to do this immediately upon its arrival. That is, senders may be idle and receivers may stall the communication. Notice that the senders will always send out a data token with an address token.

The key feature of the crossbar is that parallel communication is allowed when it is possible. For example, Sender 0 can communicate with Receiver 0 at the same time as Sender 1 communicates with Receiver 1. Similarly, Sender 0 can communicate with Receiver 1 at the same time as Sender 1 communicates with Receiver 0. Another important feature of this system is that it never drops any data packet. In particular, any sender must be stalled to prevent it from overwriting its data when the addressed receiver does not consume the data. Finally, this system should never deadlock. In other words, communication can always progress.

A straightforward implementation of a crossbar based on a split and merge pipelined handshaking architecture is illustrated in Figure 2.31. There is one merge element per output and one split element per input. The splits send data to the appropriate merge on the basis of a copy of the associated address token. Each "special split" accepts one-bit address tokens and converts them to requests on its

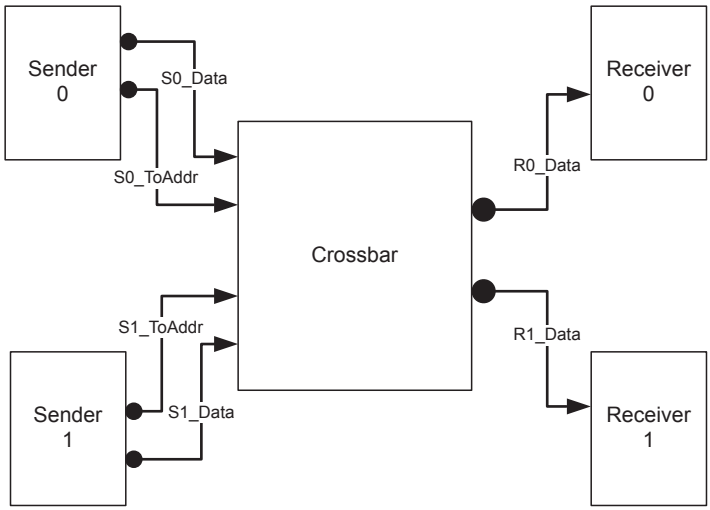

Figure 2.30. Top-level diagram of a crossbar and its environment.

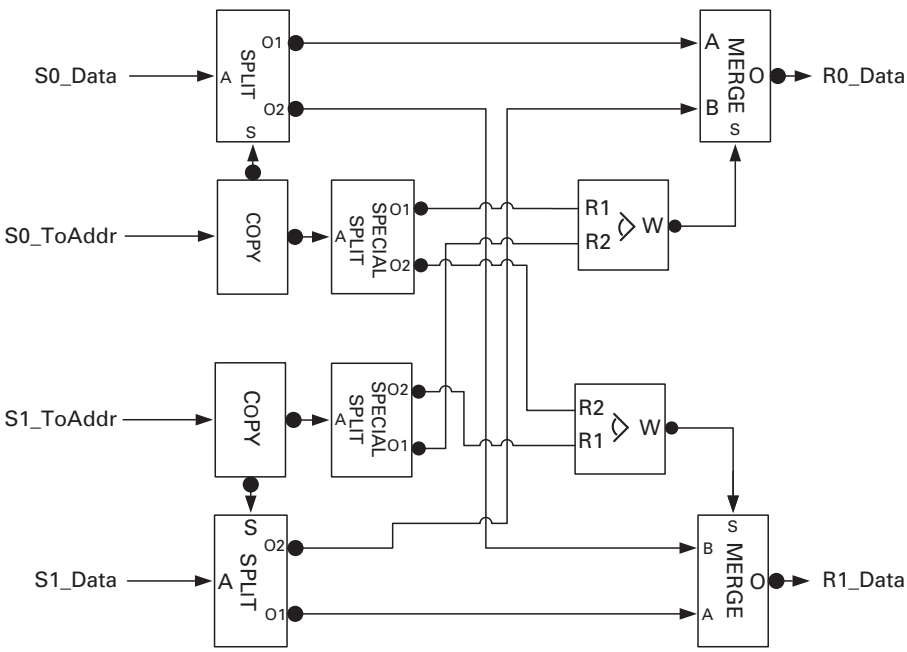

Figure 2.31. Straightforward implementation of a 2 × 2 asynchronous crossbar.

1-of-1 output channel connected to the corresponding arbiter (see Figure 2.26). The arbiter resolves any simultaneous receiver requests and sends the winner-information to the output merge. The merge uses this control data to determine which input token to wait for and to route to the output.

Because all elements in the design use pipelined handshaking, the design is pipelined. While tokens are flowing through the design, new tokens can be processed at the inputs. Nevertheless, the design does not drop any token because it consists of leaf cells that by definition cannot overwrite tokens on their output channels. If a leaf cell output is for some reason stalled and a generated output token is not consumed (i.e. acknowledged), the leaf cell will not consume new input tokens. In particular, assuming that each element uses full-buffer interleaving, one token can exist between each pair of split and merge elements while another token waits to be consumed by the output environment. Thus, the design has capacity or static slack 2. This asynchronous pipelined behavior yields a built-in notion of flow control, which makes system design using a crossbar for module-to-module communication somewhat easier than using synchronous alternatives.

With some effort it can also be shown that this implementation is deadlock-free. If the channels can be arbitrarily pipelined to optimize performance while preserving correctness then the design is called *slack elastic* [16]. Unfortunately, the simple design described above is not slack elastic. We will discuss enhancements that address this issue in Chapter 6.

2.6 Exercises

2.1. Draw the timing diagram of a merge element with 1-of-1 inputs A and B and output O. Assume that S is attached to a dual-rail channel. Assume full-buffer interleaving between input and output handshakes. Assume that there is a two-phase protocol on all channels, that tokens on A and B arrive nearly simultaneously, and that two tokens arrive on S in succession with alternate bit values. Repeat for the case when S is implemented with the two-phase bundled-data protocol.

2.2. Draw the timing diagram of the VAR module in Figure 2.20(a). Assume a two-phase one-bit bundled-data protocol and that the environment performs a write followed by two reads.

2.3. Draw the timing diagram of the four-way slackless arbiter shown in Figure 2.25 when simultaneous requests from the two R1 inputs arrive.

2.4. Design a four-way arbitrated merge with N-bit inputs A, B, C, and D and output O. As the basis of your design, use two-way merge elements, the two-way arbiters shown in Figure 2.26(a), and special copy elements that produce 1-of-1 tokens on one output rather than a duplicate N-bit data.

2.5. Using four four-way arbitrated merge blocks as a basis, design a 4×4 crossbar. Assume each of the four senders has a two-bit address and an N-bit data output channel.

2.6. Repeat Exercise 2.5 but instead use the pipelined tree arbiters shown in Figure 2.26(b).

2.7. Label all internal channels in Figure 2.28. Draw a timing diagram showing the two-place FIFO in Figure 2.28 consuming two tokens. Assume a two-phase handshaking protocol. Lastly, assuming that each action takes two time units, what is the cycle time of the design (i.e. the time between the consumption of each token)?

References

[1] A. J. Martin, "Synthesis of asynchronous VLSI circuits," Caltech-CS-TR-93–28, California Institute of Technology.

[2] A. J. Martin, "The probe: an addition to communication primitives," *Information Proc. Lett.*, vol. 20, no. 3, pp. 125–130, 1985.

[3] P. Endecott and S. B. Furber, "Modelling and simulation of asynchronous systems using the LARD hardware description language," in *Proc. 12th European Simulation Multiconf.*, Society for Computer Simulation International, June 1998, pp. 39–43.

[4] D. Nellans, V. K. Kadaru, and E. Brunvand, "ASIM – an asynchronous architectural level simulator," in *Proc. Great Lake Symp. on VLSI (GLSVLSI)*, April 2004.

[5] A. M. Lines, "Pipelined asynchronous circuits," M.S. thesis, California Institute of Technology, 1995.

[6] A. Peeters, "Single-rail handshake circuits", *doctoral dissertation*, Technische Universiteit Eindhoven, 1996.

[7] P. A. Beerel, J. Cortadella, and A Kondratyev, "Bridging the gap between asynchronous design and designers," tutorial presentation at *17th Int. Conf. on VLSI Design (VLSID'04)*, January 2004.

[8] K. van Berkel and A. Bink, "Single-track handshaking signaling with application to micropipelines and handshake circuits," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, March 1996, pp. 122–133.

[9] I. E. Sutherland, "Micropipelines," *Commun. ACM*, vol. 32, no. 6, pp. 720–738, June 1989.

[10] I. Sutherland and S. Fairbanks, "GasP: a minimal FIFO control," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, March 2001, pp. 46–53.

[11] M. Ferretti and P. A. Beerel, "Single-track asynchronous pipeline templates using 1-of-*N* encoding," in *Proc. Conf. on Design, Automation and Test in Europe (DATE)*, March 2002, pp. 1008–1015.

[12] S. B. Furber and P. Day, "Four-phase micropipeline latch control circuits," *IEEE Trans. VLSI Systems*, vol. 4, no. 2, pp. 247–253, June 1996.

[13] E. Brunvand and R. F. Sproull. "Translating concurrent programs into delay-insensitive circuits," in *Proc. Int. Conf. on Computer-Aided Design (ICCAD)*, November 1989, IEEE Computer Society Press, pp. 262–265.

[14] M. Singh and S. M. Nowick. "High-throughput asynchronous pipelines for fine-grain dynamic datapaths," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, April 2000, IEEE Computer Society Press, pp. 198–209.

[15] L. A. Plana and S. M. Nowick. "Concurrency-oriented optimization for low-power asynchronous systems," in *Proc. Int. Symp. on Low Power Electronics and Design (ISLPED)*, August 1996, pp. 151–156.
[16] R. Manohar and A. J. Martin. "Slack elasticity in concurrent computing," in *Proc. 4th Int. Conf. on the Mathematics of Program Construction*, Lecture Notes in Computer Science vol. 1422, pp. 272–285, Springer-Verlag, 1998.
[17] A. Lines, Private communication, 2002.
[18] J. Dama and A. Lines. "GHz Asynchronous SRAM in 65 nm," in *Proc. Int. Sump. on Advanced Research in Asynchronous Circuits and Systems (Async)*, May 2009, pp. 85–94.

3 Modeling channel-based designs

Digital system designers usually use hardware description languages (HDLs) to design and model their circuits at several levels of abstraction; Verilog and VHDL have been the most popular. Asynchronous circuit designers, however, often use some form of communicating sequential process (CSP) [1] to model the intended architectural behavior because it has two essential features: channel-based communication and fine-grained concurrency. The former makes data exchange between modules abstract actions. The latter allows one to define nested sequential and concurrent threads in a model. Thus, a practical HDL for high-level asynchronous design should implement the above two constructs. Furthermore, as found in many standard HDLs, the following features are highly desired:

- *Support for various levels of abstraction* There should be constructs that describe the module at both high and low levels of abstraction (e.g. at module level and at transistor level). This feature enables the modeling of designs at mixed levels of abstraction, which provides incremental verification as units are decomposed into lower levels and also enables arrayed units (e.g. memory banks) to be modeled at high levels of abstraction in order to decrease simulation run-time. Also, this enables the *mitered co-simulation* of two levels of abstraction, in which the lower-level implementation can be verified against the higher-level, golden, specification with a common input stream [12].
- *Support for synchronous circuits* A VLSI chip might consist of both synchronous and asynchronous circuits [10]. The design flow is considerably less complex if a single language can describe both, so that the entire design can be simulated using a single tool. Consequently, the modeling of clocked units should be straightforward.
- *Support for timing* Modeling timing and delays is important at both the module level and low levels of the design. Early performance verification and analysis of the high-level architecture using estimated delays is critical to avoid costly redesign later in the design cycle. Later, it is useful to be able to verify the performance of the more detailed model of the implementation that is now available, using accurate back-annotated delays.
- *Use of supporting CAD tools* In addition to powerful simulation engines, hooks to debugging platforms (e.g. graphical-user-interface-based waveform viewers), synthesis tools, and timing analyzers should also be available. Many powerful

CAD tools are available in these areas, but in most cases they only support standard hardware design languages such as VHDL and Verilog.

- *A standard syntax* The circuit description should be easily exchangable among a comprehensive set of CAD tools. Using a non-standard syntax causes simulation of the circuit to become tool dependent.

We will begin this chapter by outlining CSP. We then review several languages that have been used for designing asynchronous circuits and cover in more detail one specific application of Verilog for modeling CSP and channel-based communication that we have adopted for this book.

3.1 Communicating sequential processes

A process is a sequence of atomic or composite actions. In CSP, a process P that is composed of a sequence of atomic actions $s_1, s_2, \ldots, s_n$ repeated forever is shown as follows:

```
P = *[s1; s2; . . . ; Sn].
```

Usually, processes do not share variables but communicate via *ports* connected by *channels*. Each port is either an input or an output port. A communication action consists of either sending a variable to a port or receiving a variable from a port.

Suppose that we have a process S that has an output port "out" and a process R that has an input port "in", and suppose that S.out is connected to R.in via channel C. The "send" action is defined to be an event in S that outputs a variable to the out port and suspends S until R executes a "receive" action. Likewise, a receive action in R is defined to be an event that suspends or blocks R until a new value is put on channel C. At this point, R resumes activity and reads the value. The completion of the send action in S is said to *coincide* with the completion of the receive action in R. In CSP notation, sending the value of the variable v on the port "out" is denoted as

```
(out!v).
```

Receiving a value v from the port "in" is denoted as

```
(in?v).
```

Notice that the S and R processes are thus synchronized at their respective points of send or receive communication. In CSP there is no notion of time and communication is performed instantaneously. Synchronization between processes can also occur with no data transfer by having paired communication actions with no data, i.e. `out!` and `in?`.

Another abstract construct, called a *probe,* is also defined. In this construct a process p_1 can determine whether another process p_2 is suspended on the shared channel C waiting for a communication action to happen in p_1 [2]. Using a probe a process can avoid deadlock by not waiting to receive from a channel on

which no other process has a send pending. A probe also enables the modeling of arbitration [2]. The probe construct is an additional primitive to CSP and the resulting CSP variant is sometimes called communicating hardware processes (CHP).

For two processes P and Q, the notation $P||Q$ is used to denote that processes P and Q operate concurrently. The notation $P;Q$ denotes that Q is executed after P. We can also use a combination of these operators; for example

```
*[(p1 || (p2; p3) || p4); p5].
```

Here the process p_1 will be executed in parallel with p_4. At the same time $p_2;p_3$ will be executed. Finally, once all the processes p_1, p_2, p_3, and p_4 finish, p_5 will be executed. These nested serial and concurrent processes enable the modeling of fine-grained concurrency. Selection in CSP is implemented with a guarded command

```
[g1 -> p1 □ g2 -> p2],
```

where "□" is the alternate operator. This command process executes p_1 if the Boolean guard g_1 is true and p_2 if the Boolean guard g_2 is true. If both are true then the process *non-deterministically* chooses to execute p_1 or p_2.

An illustrative application of CSP is its model of the dining philosophers' problem, originally proposed by E. W. Dijkstra. The general idea is that a number of philosophers surround a dining table on which is a large plate of food, to be eaten with chopsticks as illustrated in Figure 3.1. Adjacent philosophers share one chopstick. They spend time either thinking or trying to eat. A philosopher must have both the chopstick on the left and the chopstick on the right to eat. Consequently, adjacent philosophers cannot eat at the same time. The dining philosophers' problem consists of finding an algorithm for sharing chopsticks that prevents deadlock and starvation.

The CSP formulation shown in Figure 3.2 is a simpler version than that described in [1] and consists of three different process types: the philosopher, the chopstick, and the table. The table process is a combined process which includes four philosophers and four chopsticks and connects the ports of these processes through channels. The philosophers request and release access to a chopstick

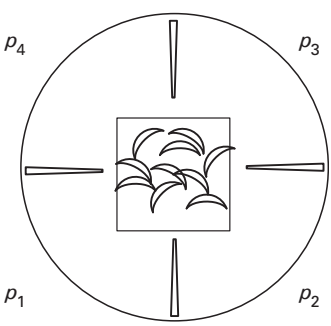

Figure 3.1. Illustration of four dining philosophers, four chopsticks, and a plate of Chinese potstickers.

```
 PHILOSOPHER = process(leftChopStickPort: syncport,
                       rightChopStickPort: syncport)
 *[
   THINK;                  // THINK command
   leftChopPort!;          // send sync on left ChopStick port
   rightChopPort!;         // send sync on right ChopStick port
   EAT;                    // EAT command
   leftChopPort!;          // send sync on left ChopStick port
   rightChopPort!;         // send sync on right ChopStick port
 ]
 end

 CHOPSTICK = process(left PhilospherPort: syncport,
                     rightPhilospherPort: syncport)
*[
  [
  PROBE(leftPhilPort) ->  // if left philosopher sent sync
  leftPhilPort?;          // then, receive sync from left
                          // philosopher for pick up
  leftPhilPort?;          // and one more timefor putdown

  []                      // alternate operator

  PROBE(rightPhilPort) -> // If right philosopher sent sync
  rightPhilPort?;         // then, receive sync from left
                          // philosopher for pick up
  rightPhilPort?;         // and one more time for putdown
  ]
]
end

TABLE = process
[
  p(i:0..3)::PHILOSOPHER  // p(i) is a process of type PHILOSOPHER

  ||                      // parallel operator

  c(i:0..3)::CHOPSTICK    // c(i) is a process of type CHOPSTICK

  chan<i:0..3:(p(i).leftChopStickPort, c(i).rightPhilospherPort>
  chan<i:0..3:(p(i).rightChopStickPort, c((i+1)%5).leftPhilospherPort>
]
end
```

Figure 3.2. Communicating sequential process (CSP) for the dining philosophers.

through successive synchronization actions with the chopstick. Chopsticks communicate only to synchronize data, not to send or receive data. If two philosophers want to access a chopstick at the same time then the chopsticks non-deterministically arbitrate among requesting philosophers, using the alternate operator □.

3.2 Using asynchronous-specific languages

Several new languages with syntaxes similar to CSP have been created, including LARD [2], Balsa [11], and Tangram [13]. The simulation of these languages has been dependent on academic simulation environments, which generally have

limited capabilities and support. Also, these languages do not support circuit modeling at lower levels of abstraction such as the transistor and logic levels. Tangram has evolved into the commercially supported language HASTE [17], which can be translated into Verilog for simulation and integration purposes.

In addition, a commercial variant of CSP called CAST has the captured semantics of CSP and has developed into a front-end design environment [20]. CAST has evolved to include a wide range of object-oriented refinement and inheritance features that yield compact specifications. It features built-in environments at all levels of abstraction to support verification and includes user-specified integrated-design directives to support a semi-automated design flow that guides simulation, transistor sizing, and physical design. Like HASTE, CAST can be translated to Verilog for simulation and integration purposes.

Although both CAST and HASTE are efficient new languages that work quite effectively in the context of their respective start-up companies, convincing developers that they should adopt a new language remains a stumbling block to widespread adoption.

3.3 Using software programming languages

Java has been enhanced with a new library, JCSP [5][10], in order to support CSP constructs in Java. However, this approach does not support timing and mixed level simulation. Furthermore, integration with commercially available CAD tools is challenging.

3.4 Using existing hardware design languages

Because of the popularity of VHDL (Very High Speed IC hardware description language) and Verilog among hardware designers, as well as the wide availability of commercial CAD tool support, several approaches have been used to enhance these languages to model channel-based asynchronous circuits. Works by Frankild and Sparsϕ [6], Renaudin *et al*. [7], and Myers [8] have employed VHDL in the design of asynchronous circuits. In VHDL, however, implementing fine-grained concurrency is cumbersome because modeling the synchronization between VHDL processes requires extra signals. Moreover, in some cases the native design language must be translated into VHDL [6]. This makes the debugging phase more cumbersome, because the code that is debugged in the debugger is different from the original code. Signals may have been added, and port names may have been changed, forcing the designer to be familiar with the details of the conversion.

Bjerregaard *et al*. [15] proposed using SystemC to model asynchronous circuits and have created a library to support CSP channels. As in VHDL, implementing fine-grained concurrency in SystemC is cumbersome. Also, the modeling of timing is not addressed in their approach.

In [8], Verilog together with its programming language interface (PLI) was proposed. Using Verilog, modeling fine-grain statement-level concurrency can be done via the built-in fork and join constructs. The PLI is used to interface Verilog and precompiled C-routines at simulation. Using the PLI, however, has two disadvantages: first, the PLI interface significantly slows down the simulation speed and, second, the C-code must be recompiled for all system environments, making compatibility across different system environments a challenge. Additionally, in the Verilog–PLI approach, handshaking variables are shared among all channels of a module. Unfortunately, this scheme breaks down for systems such as non-linear pipelined circuits in which multiple channels of a module are simultaneously active.

More recently, Saifhashemi and Beerel proposed the development of Verilog macros to model send and receive primitives [18]. This set of macros, called VerilogCSP, provides the benefits of a channel-based model in Verilog without the computational costs of the PLI interface. These macros can also be extended to capture both pipelined and enclosed handshaking, which enables designers to model not only specifications based on CSP but also architectural decisions involving sequencing and pipelining decisions between modules.

The use of Verilog automatically includes support for adding timing to the design, enabling performance modeling and efficient debugging based on existing waveform viewers. In addition, support for the back-annotation of delays after placement and routing is also provided. Furthermore, using Verilog also enables one to migrate to SystemVerilog [15], a superset of Verilog that commercial CAD tools are beginning to adopt.

3.5 Modeling channel communication in Verilog

In this book we will adopt VerilogCSP macros for the high-level modeling of channel-based designs and Verilog for the behavioral description of leaf cells.

3.5.1 Using send and receive macros

In particular, the VerilogCSP macros for send and receive are SEND(_port_, _value_) and RECEIVE(_port_, _var_). The INPORT and OUTPORT macros declare the input and output channel ports and CHANNEL declares the channel. The macros require an accompanying USES_CHANNEL macro, which declares a signal used internally to the macros.

The application of these macros to model a linear pipeline is illustrated in Figure 3.3. This model does not yet include any delays. Consequently, all events happen in zero time and time does not progress. In simulation, this will appear as an infinite loop. However, delays can be easily added to the module using the standard Verilog #delay construct [14]. In Chapter 4, we will use VerilogCSP and delays to model the performance of asynchronous pipelines.

```
module BitGen(R);
 `OUTPORT(R,width);
 `USES_CHANNEL;
  parameter width = 8;
  reg [width-1:0] d;
  always
  begin
      d = {$random}%256;
     `SEND(R,d);
  end
endmodule

module BUF(L, R);
 `INPORT(L,width);
 `OUTPORT(R,width);
 `USES_CHANNEL;
  parameter width = 8;
  reg [width-1:0] d;
  always
  begin
      `RECEIVE(L,d);
      `SEND(R,d);
  end
endmodule

module BitBucket(L);
 `INPORT(L,width);
 `USES_CHANNEL;
 parameter width = 8;
 reg [width-1:0] d;
 always
 begin
     `RECEIVE(L,d);
 end
endmodule

module top;
  `CHANNEL(C0,8);
  `CHANNEL(C1,8);
  `CHANNEL(C2,8);
  `CHANNEL(C3,8);
  BitGen #8 BG0(C0);
  BUF I1(C0,C1);
  BUF I2(C1,C2);
  BUF I3(C2,C3);
  BitBucket #8 BB1(C3);
endmodule
```

Figure 3.3. Example of linear pipeline in VerilogCSP.

Notice that in Figure 3.3 each buffer is a full buffer. Modeling half buffers requires more intricate connections between input and output handshaking; this, while possible, is beyond the scope of this text.

3.5.2 Using synchronization channels and probes

Supporting synchronization channels requires macros different from SEND and RECEIVE because there is no value to be sent or received. The modified macros are SEND_SYNC and RECEIVE_SYNC. In addition, the two macros PROBE_IN and PROBE_OUT provide the ability to identify pending communications on INPORTS and OUTPORTS.

These macros enable one to model the pipelined arbiter with a winner output, as described in Chapter 2 and illustrated in Figure 3.4. The arbiter first probes its two input ports, `AccessReqPort1` and `AccessReqPort2`, to see whether any other process is trying to gain access by sending a sync message on these ports. If only one process is sending a sync message then the arbiter performs a RECEIVE_SYNC with that process and then sends the winner process number on `WinnerNumPort`. If both processes request access at the same time, i.e. the probes on both `AccessReqPort1` and `AccessReqPort2` are evaluated as true, then the arbiter randomly chooses one process and declares it as the winner.

Notice that the port definitions for synchronization channels are simply zero-width INPORT and OUTPORT channels. Moreover, at this level of modeling the active or passive nature of the ports is abstracted; in other words, ports are not specified as active or passive, this decision is being left as a lower-level implementation detail.

3.5.3 Using enclosed handshaking macros

The SEND/SEND_SYNC, RECEIVE/RECEIVE_SYNC, and PROBE macros are sufficient to model most CSP processes in Verilog and thus are effective for the specification of asynchronous designs. However, as described in Chapter 2, channel-based architectures have concurrency and sequencing properties beyond these applications of atomic communication actions.

In particular, in this subsection we discuss extensions to VerilogCSP that model enclosed handshaking. In enclosed handshaking, one communication action is split into two parts and other communication actions occur in between these two parts. In order to support this type of communication, split communication action macros are defined in VerilogCSP; they include SEND_P1, SEND_P2, RECEIVE_P1, and RECEIVE_P2. In addition, similar macros are defined that split SEND_SYNC and RECEIVE_SYNC communication actions.

As an example, an application of a two-part RECEIVE_SYNC macro is illustrated in a sequencer and parallel module in Figure 3.5. In the SEQ module,

```
module PipelineArbiter (AccessReqPort1, AccessReqPort2,
                        WinnerNumPort);
    `USES_CHANNEL;
    `INPORT(AccessReqPort1,0);
    `INPORT(AccessReqPort2,0);
    `OUTPORT(WinnerNumPort,1);
    always
    begin
         wait(`PROBE_IN(AccessReqPort1)||`PROBE_IN(AccessReqPort2));
         if (`PROBE_IN(AccessReqPort1) && `PROBE_IN(AccessReqPort2))
         begin
              //Both ports have tokens: select one randomly
              if ({$random}%2==0)
              begin
                  `RECEIVE_SYNC(AccessReqPort1);
                  `SEND(WinnerNumPort,0);
              end else
              begin
                  `RECEIVE_SYNC(AccessReqPort2);
                  `SEND(WinnerNumPort,1);
              end
         end else if (`PROBE_IN(AccessReqPort1))
              begin // Only AccessReqPort1 has a token
                  `RECEIVE_SYNC(AccessReqPort1);
                  `SEND(WinnerNumPort,0);
              end
         else
              begin // Only AccessReqPort2 has a token
                  `RECEIVE_SYNC(AccessReqPort2);
                  `SEND(WinnerNumPort,1);
              end
         end
    endmodule
```

Figure 3.4. Arbiter with one-bit output modeled in VerilogCSP.

after receiving a request on input port R the SEQ module performs a complete handshake first on output port S1 and then on the second output port S2, before completing the handshake on R. In the PAR module, the Verilog fork and join construct is used to complete the handshakes on ports P1 and P2 concurrently within the full handshake on input port R.

```
module SEQ (R, S1, S2);
     `USES_CHANNEL;
     `INPORT(R,0);
     `OUTPORT(S1,0);
     `OUTPORT(S2,0);
     begin
        `RECEIVE_SYNC_P1(R);
        `SEND_SYNC(S1);
        'SEND_SYNC(S2);
        `RECEIVE_SYNC_P2(R);
     end
endmodule

module PAR (R, P1, P2);
     `USES_CHANNEL;
     `INPORT(R,0);
     `OUTPORT(P1,0);
     `OUTPORT(P2,0);
     begin
        `RECEIVE_SYNC_P1(R);
        fork
                `SEND_SYNC(P1);
                `SEND_SYNC(P2);
        join
        `RECEIVE_SYNC_P2
     end
endmodule
```

Figure 3.5. Application of an enclosed handshake in SEQ and PAR modules.

3.5.4 Modeling the dining-philosophers problem in VerilogCSP

Our VerilogCSP model is slightly different from the CSP version. As in CSP, the chopstick is a resource that determines which philosopher gains its access. It has one channel per right-hand and left-hand philosopher, with which a philosopher requests access. However, as illustrated in Figure 3.6, the philosophers request and release access via split synchronization communication rather than via two successive synchronization communications. This allows the chopstick to be implemented with the pipelined arbiter module described in Chapter 2. In particular, the philosopher is modeled by first a wait for a random length of time (while the philosopher is thinking about some deep concept), then a request for access to the left-hand and right-hand channels (to obtain the chopsticks), then a wait for another random

```
module ChopStick(L,R);
 `INPORT(R,0);
 `INPORT(L,0);
 `USES_CHANNEL;
 `CHANNEL (W,1); // Winner
  // Implemented with arbiter
  // and bit bucket
  PipelineArbiter ARB1(L,R,W);
  // Winner token not needed
  BitBucket #1 BB1(W);
endmodule

module Philosopher(L,R);
 `OUTPORT(L,0);
 `OUTPORT(R,0);
 `USES_CHANNEL;
  always
  begin
    #({$random}%10);    // think
    // get left chopstick
    `SEND_SYNC_P1(L);
    // get right chopstick
    `SEND_SYNC_P1(R);
    #({$random}%5);     // eat
    `SEND_SYNC_P2(L);   // release
    `SEND_SYNC_P2(R);
  end
endmodule

module top;
  `CHANNEL(P1_L,0); `CHANNEL(P1_R,0);
  `CHANNEL(P2_L,0); `CHANNEL(P2_R,0);
  `CHANNEL(P3_L,0); `CHANNEL(P3_R,0);
  `CHANNEL(P4_L,0); `CHANNEL(P4_R,0);
  ChopStick CS0 (P3_R, P0_L); Philosopher P0 (P0_L, P0_R);
  ChopStick CS1 (P0_R, P1_L); Philosopher P1 (P1_L, P1_R);
  ChopStick CS2 (P1_R, P2_L); Philosopher P2 (P2_L, P2_R);
  ChopStick CS3 (P2_R, P3_L); Philosopher P3 (P3_L, P3_R);
endmodule
```

Figure 3.6. VerilogCSP model of the dining-philosophers problem.

length of time (while the philosopher is eating), and finally the release of both channels. The entire process then repeats. Notice that the {$random}%$N$ [14] line is used to create a random length of time equal to between 0 and $N - 1$ time units.

The model correctly illustrates the dining philosophers problem – it will deadlock if all philosophers simultaneously access their left-hand chopstick as then they will all be waiting for their right-hand chopstick and no philosopher will make progress

```
module Sender(Addr,Data);
   parameter width = 8;
   `OUTPORT(Addr,1);
   `OUTPORT(Data,width);
   `USES_CHANNEL;
   always
   begin
     'SEND(Addr,{$random}%2);
     #({$random}%100);
   end
   always
   begin
     `SEND(Data,{$random}%256);
     #({$random}%100);
   end
endmodule

module Testbench;
   `USES_CHANNEL;
   parameter width = 8;
   `CHANNEL(A0,1);
   `CHANNEL(A1,1);
   `CHANNEL(D0, width);
   `CHANNEL(D1, width);
   `CHANNEL(R0, width);
   `CHANNEL(R1, width);
   Xbar XbarImpl #width (A0, D0,
            A1, D1, R0, R1);
   Sender S0 #width (A0,D0);
   Sender S1 #width (A1, D1);
   Bit Bucket BB0 #width (R0);
   Bit Bucket BB1 #width (R1);
endmodule
```

Figure 3.7. A simple X-bar testbench.

in eating. If and when the Verilog simulator reaches this condition, it will terminate the simulation as there are no more events to process. In addition, this model can lead to the starvation of slow philosophers as there is no guarantee that they will ever gain access to the chopsticks. Thus, solutions to the dining-philosphers problem must avoid deadlock [19]. (See Exercise 3.12 at the end of this chapter.)

3.5.5 Modeling a 2 × 2 asynchronous crossbar in VerilogCSP

Modeling the crossbar implementation presented in Chapter 2 is a straightforward instantiation of all leaf cell components along with channels that properly connect

them. Modeling an environment to test the implementation is more challenging. It is recommended that each port be modeled as a concurrent process so that unintended synchronization between independent ports can be avoided. A simple testbench is shown in Figure 3.7. The testbench uses a process called Sender that randomly sends data to a random destination port via separate "always" blocks, allowing addresses and data tokens to flow independently. The crossbar output ports are connected to simple bit buckets but more complicated alternatives, which can verify whether the data received on each output port is expected, are also possible.

3.6 Implementing VerilogCSP macros

The challenge in implementing CSP macros is to overcome the limited syntax and semantics of Verilog macros. Verilog macros only support textual substitution with parameters; they do not support the creation of new variables via parameter concatenation, as is available in software languages like C.

Our approach is to make the macros implement an underlying four-phase handshaking protocol in zero time. Because there are many possible handshaking protocols, there are a variety of different possible macro implementations. However, it is important to emphasize that the purpose of the macros is to abstract the underlying handshaking, so that the architecture can be modeled without regard to the specific handshaking protocol. Thus a particular underlying handshaking protocol should not be interpreted as a requirement or specification for the more detailed circuit implementation.

In this section we will describe a set of macros that use an active–active protocol. This protocol is not appropriate for most circuit implementations but does enable efficient debugging by providing more visibility of the state of the various channels.

3.6.1 Send and receive macros

The send and receive macros SEND(_port_, _value_) and RECEIVE (_port_, _var_) are shown in Figure 3.8. Note that the backslashes "\" are required to concatenate lines, as the macros by default end at the next new line.

The two least significant bits of _port_ hold the concrete handshaking signals and the remaining MSBs hold the transmitted data. The RECEIVE macro uses a two-bit "dummy" signal as a convenience for extracting the transmitted data bits of _port_ into the _var_ variable using the line {_var_,dummy} = _port_. Note that users should not use variables named _port_, _value_, _var, or dummy when using these macros.

The keyword "force" in the macros is needed to set the value of the dummy handshaking signals. This is necessary because Verilog does not allow inputs to be assigned [14]. The alternative of making these signals "inout" does not

```
`define SEND(_port_, _value_)\
   begin \
   force _port_ = \
        {_value_,_port_[1],1'b1}; \
   #0; \
   wait (_port_[1] == 1'b1); \
   force _port_ = \
       {_value_,_port_[1],1'b0}; \
   wait(_port_[1] == 1'b0); \
   end
```

```
`define RECEIVE(_port_, _var_) \
   begin \
   force _port_[1] = 1'b1; \
   wait (_port_[0] == 1'b1); \
   {_var_,dummy} = _port_; \
   wait(_port_[0] == 1'b0); \
   force _port_[1] = 1'b0; \
   #0; \
   end
```

Figure 3.8. SEND and RECEIVE macros.

```
`define USES_CHANNEL          reg [1:0] dummy
`define OUTPORT(port,width)   output[width+1:0] port
`define INPORT(port,width)    input[width+1:0]  port
`define CHANNEL(c,width)      wire[width+1:0]   c
```

Figure 3.9. Dummy signal definition and port declarations.

work either, because writing to an inout port is also illegal in a Verilog "always" block [14]. The #0 delays are necessary in some simulators to ensure that the potential zero-delay pulse in the handshaking signals releases any corresponding pending waits.

As mentioned earlier, we hide the dummy signal definition and input or output port declarations, as shown in Figure 3.9.

The designer should use the USES_CHANNEL macro in modules that incorporate any communication macro because it declares the dummy signal. The INPORT, OUTPORT, and CHANNEL macros declare the ports with two more bits added for handshaking. Notice that an OUTPORT port is declared as an output and supports the SEND macro, while the INPORT port is declared as an input and supports the RECEIVE macro. This is necessary because the port variable in the RECEIVE macro is used on the left-hand side of a Verilog assignment statement.

3.6.2 Synchronization macros

As mentioned earlier, the support of synchronization channels requires macros different from SEND or RECEIVE because there is no value to be sent or received. The modified macros, SEND_SYNC and RECEIVE_SYNC, are illustrated in Figure 3.10.

```
`define SEND_SYNC(_port_) \
   begin \
   force _port_[0]=1'b1; \
   #0; \
   wait (_port_[1]==1'b1); \
   force _port_[0]=1'b0; \
   wait (_port_[1]==1'b0); \
   end
```

```
`define RECEIVE_SYNC(_port_) \
   begin \
   force _port_[1]=1'b1; \
   wait (_port_[0]==1'b1); \
   wait (_port_[0]==1'b0); \
   force _port_[1]=1'b0; \
   #0; \
   end
```

Figure 3.10. The SEND_SYNC and RECEIVE_SYNC macros.

```
`define PROBE_IN(_port_)      (_port_[0]===1'b1)
`define PROBE_OUT(_port_)     (_port_[1]===1'b1)
```

Figure 3.11. The PROBE_IN and PROBE_OUT macros.

```
`define RECEIVE_SYNC_P1(_port_)
   begin \
   force _port_[1]=1'b1; \
   wait (_port_[0]==1'b1); \
   end
`define RECEIVE_SYNC_P2(_port_)
   begin \
   wait (_port_[0]==1'b0); \
   force _port_[1]=1'b0; \
   #0; \
   End
```

```
`define SEND_SYNC_P1(_port_)
   begin \
   force _port_[0]=1'b1; \
   #0; \
   wait (_port_[1]==1'b1); \
   end
`define SEND_SYNC_P2(_port_)
    begin\
    force _port_[0]=1'b0;\
    wait (_port_[1]==1'b0);\
    end
```

Figure 3.12. Two-part RECEIVE_SYNC and SEND_SYNC macros.

3.6.3 Probe macros

Because send and receive macros both use an active protocol, the implementation of these probe macros is straightforward. In particular, PROBE_IN tests _port_[0] === 1 and PROBE_OUT tests _port_[1] === 1, as illustrated in Figure 3.11. PROBE_IN returns "true" when a pending SEND/SEND_SYNC exists on an INPORT, and PROBE_OUT returns "true" when a pending RECEIVE/RECEIVE_SYNC exists on an OUTPORT.

3.6.4 Enclosed handshaking macros

It is straightforward to split the RECEIVE/RECEIVE_SYNC macros into two parts to model enclosed handshakes, as illustrated in Figure 3.12. Again it is

important to emphasize that the specific partition of the four-phase handshake into two parts is arbitrary and should not be considered as a specification of the circuit implementation.

3.7 Debugging in VerilogCSP

In comparison with the debugging of traditional software programs, such as C, debugging VerilogCSP and CSP can be significantly more challenging. The main reason is that there are usually multiple processes in a CSP program that run in parallel, which makes the system more complicated than a traditional single-threaded software program. Often, the designers need to keep track of the state of multiple processes and understand how the processes interact to understand the overall system behavior and find and resolve bugs.

One of the most common bugs in VerilogCSP, as well as in other parallel programming languages, is deadlock. Formally proving that a program is deadlock-free is usually very difficult if not computationally impossible. Therefore, usually designers must rely on multiple (random) simulations to gain sufficient confidence that the design is deadlock-free under all possible environments.

In this section we introduce a few debugging tools designed to make the debugging task simpler and more efficient.

3.7.1 When does deadlock happen?

Deadlock happens when a set of processes in the system cannot progress, no matter how much time passes. Consider the example in Figure 3.13. Modules M1 and M2 are not able to progress, because they are both blocked on the respective SEND actions. Thus M1 is waiting for M2, and M2 is waiting for M1. This is illustrated in the *wait graph* of Figure 3.14. In a wait graph each node represents a module, and there is an edge between two nodes if one is waiting for the other.

One source of deadlock occurs when a cycle is generated in a system's wait graph. To fix such a deadlock, a designer must first determine where and under what conditions the cycle appears. To do so, he or she must understand the state of the processes involved in the cycle, determine the necessary conditions for the cycle to form, and find a solution that prevents the cycle from forming. Keeping track of the state of each process, however, is cumbersome. Alternatively, we propose keeping track of the state of the process ports. This is helpful because processes synchronize with each other by communicating on their ports. If a designer knows the status of the communication action on the ports, he or she can find the source of the deadlock. In the next subsection we show how to monitor the state of the process ports.

3.7.2 Monitoring the state of ports and channels

As explained in the previous section, VerilogCSP uses an active–active protocol. This allows us to monitor the state of each port and channel on the basis of the

```
module M1(inP1, outP2);
  `INPORT(inP1,1);
  `OUTPORT(outP2,1);
  `USES_CHANNEL;
  reg a,b;
  always
  begin
        #10;
        `SEND(outP2,a);
        `RECEIVE(inP1,b);
  end
endmodule

module M1(outP1,inP2);
  `INPORT(inP2,1);
  `OUTPORT(outP1,1);
  `USES_CHANNEL;
  reg a, b;
  always
  begin
        #20;
        `SEND(outP1,a);
        `RECEIVE(inP2,b);
  end
endmodule

module top;
  `CHANNEL(C0,1);
  `CHANNEL(C1,1);
  M1 mod1(.inP1(C0), .outP2(C1));
  M2 mod2(.outP1(C0), .inP2(C1));
endmodule
```

Figure 3.13. An example of deadlock in VerilogCSP.

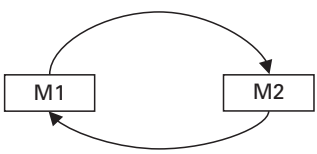

Figure 3.14. Wait graph for the example in Figure 3.13.

values of the handshaking signals. Table 3.1 shows these states. Notice that the SYNC_PENDING case exists for zero time during SEND and RECEIVE synchronization. However, it may last longer during split communication actions, when both processes have finished the first part of the communication action and before the second half of the split communication is complete.

Table 3.1 VerilogCSP port states

State	Bit 1	Bit 0	Description
IDLE	0	0	No pending communication
SEND_PENDING	0	1	Blocked on SEND
RECEIVE_PENDING	1	0	Blocked on RECEIVE
SYNC_PENDING	1	1	Blocked on SEND and RECEIVE

```
virtual type {IDLE SEND_PENDING RECEIVE_PENDING SYNC_PENDING}
       VerilogCSPType
virtual function {(VerilogCSPType)  (2*(int)(/top/C0[1]===1'b1)+
        (int)(/top/C0[0]===1'b1) )} TOP_C0_STATUS
virtual function {(VerilogCSPType)  (2*(int)(/top/C1[1]===1'b1)+
        (int)(/top/C1[0]===1'b1) )} TOP_C1_STATUS
```

Figure 3.15. Defining virtual functions for channel status.

Commercial Verilog simulators allow the user to assign mnemonic values to certain values on signals. For example, one can assign the mnemonic value IDLE to signals whose two least significant bits are zero. In this section we show how to define these mnemonics in ModelSim [21]. This simulator lets the user assign mnemonic values to signals by using virtual types and functions. Similar methods are possible using other simulators.

First we define a new virtual type, which consists of four different values: IDLE, SEND_PENDING, RECEIVE_PENDING, SYNC_PENDING. To do so, we use the following Modelsim command:

```
virtual type {IDLE SEND_PENDING RECEIVE_PENDING
SYNC_PENDING} VerilogCSPType.
```

Then, for each channel or port of interest, we define a new virtual function that creates a new signal whose value is a function of other signals. For example, the following command creates a new virtual signal of Boolean type that shows whether bit 0 of channel C0 in module top is 1:

```
virtual function {(/top/C0[0]===1'b1)} C0_IS_ONE.
```

Similarly, the command that defines a new virtual function for the status of channel C0 is

```
virtual function {(2*(int)(/top/C0[1]===1'b1) +
(int)(/top/C0[0]===1'b1))} TOP_C0_STATUS.
```

In particular, the above command creates a new signal TOP_C0_STATUS, which has two bits. Bit 0 is 1 if bit 0 of C0 is 1, otherwise it is 0. Bit 1 is 1 if bit 1 of C0 is 1, otherwise it is 0. What we really want, however, is to type cast these values to the virtual type VerilogCSPType. Figure 3.15 illustrates the complete simulator

Table 3.2. Summary of the actions of the VerilogCSP macros

Macro	Description
USES_CHANNEL	Must be used in variable declaration of any module that uses VerilogCSP macros.
INPORT(p,l)	Defines an input port p of size l + 2. The extra two least significant bits are used by the macros for handshaking.
OUTPORT(p,l)	Defines an output port p of size l + 2. The extra two least significant bits are used by the macros for handshaking.
CHANNEL(c,l)	Defines a channel c of size l + 2. The extra two least significant bits are used by the macros for handshaking. Channels should be used to connect two modules that perform communication actions.
SEND(p,v)	Sends value v on port p
RECEIVE(p,v)	Receives the value of port p and copies it into variable v
SEND_SYNC(p)	Sends a sync message on port p
RECEIVE_SYNC(p)	Receives a sync message from port p
SEND_P1(p,v)	Splits SEND, part 1
SEND_P2(p,v)	Split SEND, part 2
RECEIVE_P1(p,v)	Split RECEIVE, part 1
RECEIVE_P2(p,v)	Split RECEIVE, part 2
SEND_SYNC_P1(p,v)	Split SEND_SYNC, part 1
SEND_SYNC_P2(p,v)	Split SEND_SYNC, part 2
RECEIVE_SYNC_P1 (p,v)	Split RECEIVE_SYNC, part 1
RECEIVE_SYNC_P2 (p,v)	Split RECEIVE_SYNC, part 2
PROBE_IN(p)	Returns true if the process connected to input port p is trying to SEND.
PROBE_OUT(p)	Returns true if the process connected to output port p is trying to RECEIVE.

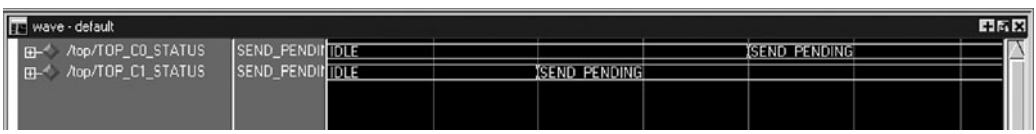

Figure 3.16. Waveforms illustrating the channel status.

commands to achieve this goal, yielding new virtual signals that show the status on channels C0 and C1. The resulting waveforms are shown in Figure 3.16.

3.8 Summary of VerilogCSP macros

Table 3.2 gives a summary of the action of all the VerilogCSP macros introduced in this chapter and can be used as a reference.

3.9 Exercises

3.1. Implement a Boolean one-bit register in VerilogCSP with the CHP description given in Figure 3.17. Assume that the environment always communicates on only one of the register's ports, D, Q, or Q_bar. Write a testbench to test your design.

3.2. Implement an unconditional merge module. This module has two input ports, called inPort1 and inPort2, and one output port, called outPort. Input data comes on either inPort1 or inPort2 (not both). The module receives the input data and sends it to outPort. Write a testbench to test your design.

3.3. Implement a merge module that is similar to the unconditional merge but has an extra input port, called controlPort. The module first receives a value from controlPort. If the value is 1, the module receives a value from inPort1. If the control value is 2, it receives a value from inPort2. Finally, it sends the received value to outPort. Write a testbench to test your design.

3.4. Implement a split module. This module has two input ports, inPort and controlPort. The module receives a value from controlPort and inPort in parallel. If the value on controlPort is 1, the module sends the input value to outPort1. If the value on controlPort is 2, it sends the input value to outPort2. Write a testbench to test your design.

3.5. A *lazy stack* of depth $n > 0$ is a linear chain of n modules called stackElement defined as follows [3] (a lazy stack of depth 3 is illustrated in Figure 3.18).

The environment communicates with the stack using the "in" and "out" ports of the first stack element (on the left). The stack element module has a state variable called "state." If "state" is empty and there is an input request on the port "in" then the data is stored; otherwise it is passed to the next element using the port "put." If there is an output request on the port "out" then this is sent to the next module using the "get" port. Observe that there is no attempt to reshuffle the full stack of elements. Hence it could arise that data is scattered in different stack elements with empty stack elements in between (that is why we call it a lazy stack). Write the VerilogCSP description of a stack element, and then connect eight of these modules to form a stack of depth 8. Assume that stack overrun or underrun never happens. Write a testbench to test your design.

3.6. Design a channel combiner module which has four eight-bit input ports and one 32-bit output port. The function of the module is to receive data from all input channels in parallel, make a 32-bit value, and then send it to the output. Write a testbench to test your design.

3.7. Design a one-bit arithmetic logic unit (ALU) with input ports operand1, operand2, cin, and opcode and output ports result and cout. The ALU can perform four different operations on the basis of the value it receives from the opcode port: AND, OR, XOR, ADD. The opcode for the AND operation is 0, for OR is 1, for XOR is 2, and for ADD is 3. The ALU first receives a value

```
*[[
      D̄ → D?x
      []
      Q̄ → Q!x
      []
      Q̄_bar → Q_bar!(~x)
]]
```

Figure 3.17. CHP of a one-bit register.

```
stackElement ≡
*[[
      (state = empty) →
            (in̄ → in?x; state = full  []  out‾ → get?x; out!x)
      []
      (state = full) →
            (in̄ → put!x; in?x  []  out‾ → out!x; state = empty)
]]
```

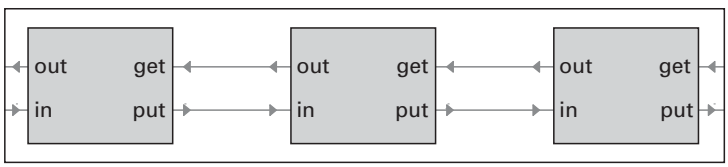

Figure 3.18. A linear chain of three stack elements.

from the opcode port, and then receives the values from operand1 and operand2. If the operation is ADD, it also expects to receive a value from the port cin. After calculating the result, the ALU sends the value to the result port. If the operation is ADD, it also sends the value of the carry-out to the cout port. Write the VerilogCSP code of the ALU and a testbench to test your design.

3.8. Use the one-bit ALU of the previous problem to design an eight-bit ALU. The top module has similar ports, except that operand1, operand2, and result are each eight-bit ports. *Hint* First design the following modules:
- a module to split the input eight bits of each operand to eight one-bit ALU modules;
- a module to split the opcode to all one-bit ALU modules;
- a module to combine the results from eight one-bit ALUs.

Draw a schematic of your design and then implement it in VerilogCSP. Write a testbench to test your design.

3.9. Implement a four-stage two-bit linear pipeline driven by a bit bucket and bit generator. Add a random delay in the range 1 to 10 between the receive and

send commands in the buffers. Add a random delay in the range 1 to 20 between successive sends in the generator and receives in the bucket. Verify via simulation that the linear pipeline propagates data as expected and does not deadlock.

3.10. Repeat Exercise 3.9 but add statistics in the bit-bucket module to determine the long-term average rate at which tokens are received. Repeat with the delay in generator and bit bucket between 1 and 5. Repeat with a 10-stage pipeline. Explain how and why the numbers change.

3.11. Implement a dining-philosophers simulation with four philosophers, using VerilogCSP. Change the random delay setting in such a way that deadlock occurs within a reasonable amount of simulation time.

3.12. Implement a solution to the dining-philosphers problem in VerilogCSP by introducing a waiter at the table. Philosophers must ask the waiter permission to access a chopstick, and the waiter tells a philosopher to wait when deadlock might otherwise be likely. Show that even with random delays your simulation does not deadlock over a reasonably long period of time.

3.13. Implement a simple 2×2 crossbar having eight-bit datapaths and its environment using VerilogCSP. Verify via simulation that data streams through the crossbar are as expected and that deadlock does not occur.

3.14. Request Exercise 3.13 but add cells to the crossbar to create two one-bit "from" channel outputs indicating from which sender the data comes.

3.15. Repeat Exercise 3.14 with a more complex sender and receiver such that the receiver verifies that all the data received was expected. *Hint* This can be done by creating shared circular queues for each input–output pair in the testbench.

3.16. Repeat Exercise 3.13 but modify your implementation to add some statistics logic that counts the number of packets sent to each output channel. Design some means for the environment to read and reset these counters.

References

[1] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[2] A. J. Martin, "The probe: an addition to communication primitives," *Information Processing Lett.*, vol. 20, no. 3, pp. 125–130, 1985.

[3] A. J. Martin, *"Synthesis of asynchronous VLSI circuits,"* Caltech-CS-TR-93–28, California Institute of Technology, 1993.

[4] P. Endecott and S. B. Furber, " Modelling and simulation of asynchronous systems using the LARD hardware description language", in *Proc. 12th European Simulation Multiconf., Society for Computer Simulation International*, June 1998, pp. 39–43.

[5] D. Nellans, V. Krishna Kadaru, and E. Brunvand, "ASIM – an asynchronous architectural level simulator," in *Proc. Great Lake Symp. on VLSI (GLSVLSI)*, April 2004.

[6] S. Frankild and J. Sparsø, "Channel abstraction and statement level concurrency in VHDL++", in *Proc. 4th Asynchronous Circuit Design Workshop (ACiD)*, National Polytechnic Institute of Grenoble, 2000.

[7] M. Renaudin, P. Vivet, and F. Robin, " A design framework for asynchronous/synchronous circuits based on CHP to HDL translation," in *Proc. 5th Int. Symp. on Advanced Research in Asynchronous Circuits and Systems (ASYNC)* April 1999, pp. 135.
[8] C. Myers, *Asynchronous Circuit Design*, John Wiley & Sons, 2001.
[9] A. Saifhashemi and H. Pedram, "Verilog HDL, powered by PLI: a suitable framework for describing and modeling asynchronous circuits at all levels of abstraction," in *Proc. Design Automation Conf. (DAC)* June 2003, pp. 330.
[10] P. H. Welch, J. R. Aldous, and J. Foster, "CSP networking for Java (JCSP.net)," in *Proc. Int. Conf. on Computational Science–Part II* April 2002, pp. 695–708.
[11] A. Bardsley,*"Balsa: an asynchronous circuit synthesis system,"* Ph.D. thesis, University of Manchester, 1998.
[12] P. A. Beerel, J. Cortadella, and A. Kondratyev, "Bridging the gap between asynchronous design and designers," *tutorial presentation at 17th Int. Conf. on VLSI Design (VLSID'04)*, January 2004.
[13] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij, "The VLSI-programming language Tangram and its translation into handshake circuits," in *Proc. European Conf. on Design Automation (EDAC)*, 1991, pp. 384–389.
[14] "IEEE std 1364–2001," IEEE Standard for Verilog Hardware Description Language, 2001.
[15] T. Bjerregaard, S. Mahadevan and J. Sparsø, "A channel library for asynchronous circuit design supporting mixed-mode modeling", in *Proc 14th Int. Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, September 2004, Springer pp. 301–310.
[16] System Verilog website, http://www.systemverilog.org.
[17] Handshake Solutions website, www.handshakesolutions.com.
[18] A. Saifhashemi and P. A. Beerel, "High level modeling of channel-based asynchronous circuits using Verilog," in *Proc. Conf. on Communicating Process Architectures 2005 (WoTUG-28)*, vol. 63 of *Concurrent Syst. Eng.*, 2005.
[19] K. M. Chandy and J. Misra, "The drinking philosophers problem", *ACM Trans Programming Lang. Syst.*, vol. 6, no. 4, pp. 632–646, 1984.
[20] A. Lines, Private communication, 2006.
[21] ModelSim – Advanced Simulation and Debugging, Mentor Graphics, www.model.com

4 Pipeline performance

In the synchronous world, the performance of a pipelined design is straightforward and is characterized by *throughput* and *latency*. Throughput is measured in terms of results per second. It is the inverse of the clock frequency for a synchronous system that generates a new result every clock cycle. It can be a multiple of this value if the synchronous system generates multiple results every clock cycle, and it is a fraction of this value if it requires multiple clock cycles between the generation of results. Latency is measured as the number of clock cycles required to generate a result after the associated primary inputs are made available. In pipelined synchronous systems, the latency is a measure of the pipeline depth of the system.

The cycle time in an asynchronous system is the period between two results. Unlike in a synchronous circuit it is not dictated by a clock frequency but rather by the time between successive output tokens. Because this time can vary between tokens, it is often taken to be the average time between output tokens. Since asynchronous systems often have a warm-up period during which there is irregular or no generation of output tokens, the average measured cycle time is typically a *long-term average* for which the behavior during the warm-up period is insignificant. An asynchronous system's throughput is simply the inverse of its cycle time.

The latency of an asynchronous system is the time between input tokens being consumed at the primary inputs and output tokens being generated. Latency is measured by the presentation of one set of primary input tokens in isolation, to avoid the possibility of congestion caused by previous tokens impacting the measurement.

Both the latency and cycle time of an asynchronous system can be data dependent, for multiple reasons. The delay of specific units may be data dependent in, for example, an asynchronous adder that completes faster when one operand is zero. Moreover, the data flow within an asynchronous system may also be data dependent because the system includes asynchronous blocks that exhibit choice behavior such as arbitration, splits, and merges. Such a system is called *non-deterministic*. Even in a deterministic system, which has no data-dependent token flow, determining the cycle time can be challenging for two reasons. First, the handshaking interaction between neighboring components is complex and has many local cycles that dictate throughput. Second, asynchronous architectures exhibit performance bottlenecks when blocks are starved by the lack of input tokens or stalled due to the backing up of tokens in their output channel(s).

We will mathematically formalize these metrics, along with techniques to model, analyze, and optimize cycle time, in Chapter 5. In this chapter, however, we will provide the intuition behind these metrics, using simple design examples to illustrate the issues.

4.1 Block metrics

The performance of an asynchronous system is depends upon the performance of the communication blocks that make up the system and the performance of the protocols that guide the communication.

4.1.1 Forward latency

The forward latency of a block is the time difference between tokens enter the block and when they are subsequently generated at the block outputs. Like the latency of the system, the input tokens must be presented in isolation to ensure that congestion due to previous tokens does not influence this measurement. Moreover, it is assumed that the output channels are free to accept new tokens, ensuring that the measurement is independent of the output environment. For blocks with multiple input and output channels, the forward latency from different input channels to different output channels may vary. Even for an input–output channel pair, the delay may be dependent on the state of the block or the specific data values of the input and output tokens.

The latency of a system is the sum of the forward latencies along the critical path through the blocks where tokens flow. As for critical path analysis in synchronous combinational circuits, the causality among the tokens in each block should be taken into account. In particular, for blocks with multiple input channels the earliest arriving token may cause the generation of the output token, as in an arbiter. Alternatively, it may be the latest arriving token that causes the generation of an output token, as is the case for a generic join cell such as an asynchronous two-input XOR. Consequently, a simple worst-case path analysis that sums forward latencies and does not consider causality can lead to a conservative estimate of the forward latency.

In fact, this issue arises even when one is modeling small asynchronous blocks. Consider the VerilogCSP model of an OR gate shown in Figure 4.1. The Verilog fork–join construct [1] enables the module to receive on both L1 and L2 input channels concurrently. The Verilog construct "#FL" [1] after the fork–join construct makes the module wait *FL* time units after input tokens arrive at **both** input channels before sending the output result to output channel R. This model is not always accurate because some designs do not wait for a second input token to arrive if the result is known, e.g. the other input channel received a 1. For such implementations, this model sends some tokens later than in reality and in this way is conservative. More accurate models are possible but are more complex.

```
module OR(L1, L2, R);
 `INPORT(L1,1);
 `INPORT(L2,1);
 `OUTPORT(R,1);
 `USES_CHANNEL
  parameter FL = 2;
  reg d1,d2;
  always
  begin
       fork
      `RECEIVE(L1,d1)
      `RECEIVE(L2,d2)
       join
       #FL;
      `SEND(R,d1 | d2)
  end
endmodule
```

Figure 4.1. Example of VerilogCSP OR with forward latency.

4.1.2 Local cycle time

The handshaking system between neighboring cells limits the rate at which the cells process tokens. The shortest length of time needed by neighboring cells to complete a handshake on a channel c, i.e. the time between the generation (or consumption) of two successive tokens on c, is referred to as the channel's *local cycle time*. When back-to-back communication on a channel is required, as is typical in deterministic systems, the local cycle time is a lower bound for the system cycle time. For example, the cycle time of a linear pipeline is no smaller than the worst-case local cycle time of all channels in the pipeline.

The local cycle time may be a function of the sender and receiver of a single channel or the senders and receivers of neighboring channels. In the latter case this arises because it is sometimes necessary for a block to receive the acknowledgement of an output channel before resetting the acknowledgement of a corresponding input channel. This possible interleaving between input and output channel handshaking, which is typical in many four-phase half-buffer blocks, causes the local cycle time to be a function of three blocks in sequence.

To generalize this point, we say that each block contributes specific delays to a complex function that represents all the various cyclic dependencies associated with the handshaking protocol and their interleaving. Techniques to model the performance of this interleaving behavior using graph structures will be described in Chapter 5.

4.1.3 Backward latency

The time difference between the local cycle time and the forward latency of a channel is often referred to as the channel's *backward latency*. For example, in a

```
module BUF(L, R);
  parameter width = 8; parameter FL = 2; parameter BL = 4;
 `INPORT(L,width);
 `OUTPORT(R,width);
 `USES_CHANNEL
  reg [width -1:0] d;
  always
  begin
      `RECEIVE(L,d)
       #FL;
      `SEND(R,d)
       #BL;
  end
endmodule
```

Figure 4.2. Example of VerilogCSP buffer with forward and backward latencies.

typical four-phase protocol it is the time difference from the output request to a subsequent input request. The notion of backward latency captures the time elapsed after the receipt of a token on a channel while the receiver resets and becomes able to accept new tokens on that channel. Thus, if the input tokens are sufficiently spread apart in time, the backward latency does not create a system bottleneck. If the input tokens are presented quickly, however, it is the backward latency that prevents new input tokens from being processed, stalling the system.

In fact, owing to the possible interleaving nature of the input and output channel handshaking, the calculation of the backward latency can be very complex. Nevertheless, approximate models of performance that reduce this complex interaction are often very useful in estimating and optimizing system performance. For example, an abstract VerilogCSP model of a buffer including forward latency *FL* and backward latency *BL* is shown in Figure 4.2 [2]. As with the VerilogCSP model of the OR gate in Figure 4.1, this model uses the Verilog parameters to set default values for *FL* and *BL* which can be overridden when instantiating this module. Because *FL* and *BL* are constants, this abstract model does not capture the data-dependent characteristics of delays. Moreover the local cycle time, which is $FL + BL$, implies an upper bound on how fast this buffer can send or receive tokens. Consequently, the time between successive sends or receives along any channel is implicitly bounded by the maximum of the local cycle times of the associated senders and receivers.

4.2 Linear pipelines

As mentioned earlier, one of the simplest configurations of asynchronous blocks that make up a system is a linear pipeline. We first analyze the performance of a

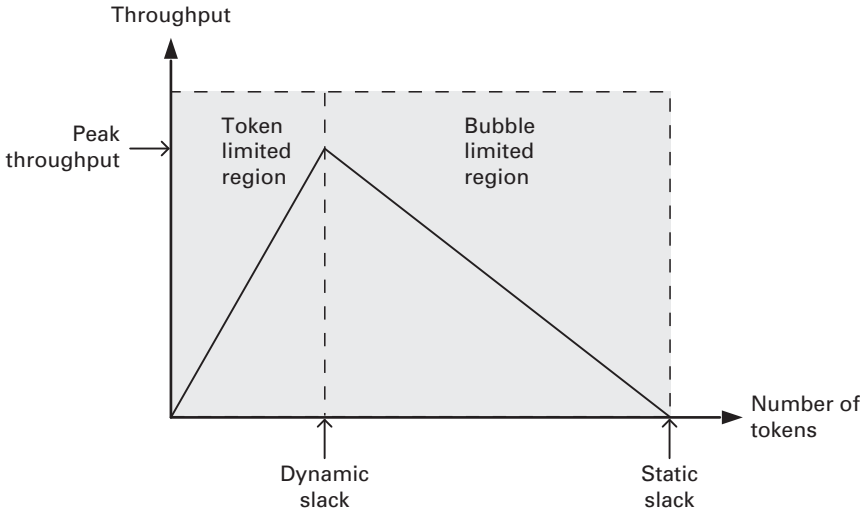

Figure 4.3. Graph of throughput versus number of tokens for a linear pipeline.

homogeneous pipeline in which each pipeline stage has the same forward and backward latency. We then generalize it to non-homogeneous pipelines before discussing techniques to improve performance.

4.2.1 Homogeneous linear pipelines

To describe the throughput of a homogeneous linear pipeline, a *throughput versus token graph* is often used [3], as illustrated in Figure 4.3. This graph shows the throughput of a linear pipeline as a function of the number of tokens in the pipeline. For both throughput and number of tokens, long-term averages are used to account for the fact that both these quantities may vary as the pipeline operates, even assuming fixed delays among all components. The left-hand side of the graph shows the characteristic of an asynchronous pipeline operating in a *data-limited region*. In this region, as the data tokens are inserted more frequently the pipeline operates at a higher throughput. The speed of the pipeline is limited by how fast data can be inserted into the pipeline, i.e. the speed of the input environment. The right-hand side of the graph shows the characteristic of an asynchronous pipeline operating in a *bubble-limited region* [6]. In this region the output environment cannot consume the data provided by the asynchronous pipeline sufficiently fast and the data tokens start to accumulate in the pipeline. To help understand this behavior, we often view the places into which tokens can move as *bubbles*. A bubble is inserted into the pipeline when the output environment consumes a token and moves backwards through the pipeline as tokens move forward.

Notice that the two extreme points on the horizontal axis yield zero throughput. When no tokens are in the pipeline then by definition the pipeline is yielding 0

tokens per second and thus has throughput 0. In addition, when the pipeline is full of tokens the throughput is 0. This is counterintuitive to most students of the subject and deserves some explanation. When a token leaves the pipeline, it takes a non-zero time for a new token to move into the bubble created. Thus, there will be some time during which the number of tokens in the pipeline is less than the maximum number of tokens, the *static slack*. In other words, the only way for the long-term average number of tokens in the pipeline to be equal to the static slack is for the pipeline to be full and for no tokens to leave the pipeline. This can happen if the bit bucket is not operating during the time period of interest. Notice this is true for both half buffers and full buffers. The only difference is that for N full buffers the static slack is N whereas for N half buffers the static slack is $N/2$ (assuming that N is even).

A related counterintuitive feature of asynchronous pipelines is that the throughput is not correlated with the number of tokens in the pipeline. Whereas readers can often appreciate the argument that if the pipeline is full the throughput must be zero, it still seems intuitive that, up to this full point, the more tokens in the pipeline the higher throughput the pipeline will have. This counterintuitive result again stems from the non-zero backward latency of asynchronous buffers.

For example, for a linear pipeline of full buffers, as the number of tokens that reside in the pipeline increases, the number of consecutive buffers that are blocked on send commands increases. As the last stage of this chain of full buffers completes its send command it waits a time BL before receiving its next token and unblocking the next to last stage of the chain. This stage then waits an additional time BL before the next token arrives, and the receipt of this token unblocks the second to last stage. This continues until the end of the chain, where a stage not blocked on send is reached. Thus, in the extreme case where there is only one bubble in the pipeline, the time between new tokens is the sum of the backward latencies of all pipeline stages, i.e. $N \times BL$, where N is the number of pipeline stages. A similar behavior occurs with a linear pipeline of half-buffers; however, the extreme case typically has a smaller total backward latency because of the alternating empty buffers.

The pipeline's throughput is thus maximized when the number of tokens is somewhere between 0 and the static slack of the pipeline. This number is referred to as the *dynamic slack* [3]. The throughput corresponding to the dynamic slack is the inverse of the maximum local cycle time of any buffer stage. For the VerilogCSP full buffer shown in Figure 4.2, this is $1/(FL + BL)$. The value of the dynamic slack can be computed as the forward latency of the entire pipeline divided by the maximum local cycle time. The dynamic slack is

$$\frac{N \times FL}{FL + BL} = \frac{N}{1 + BL/FL}.$$

Thus if $BL = FL$ then the dynamic slack equals $N/2$. Typically BL is larger than FL, as asynchronous blocks are typically optimized for forward latency. The larger the ratio BL/FL, however, the smaller the dynamic slack, meaning that peak

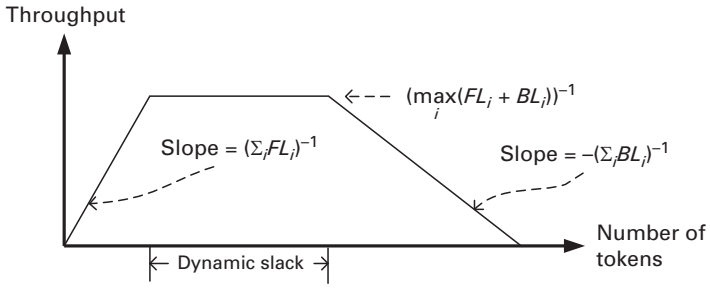

Figure 4.4. Impact of a slow stage in a linear pipeline of full buffers.

throughput occurs with fewer tokens in the pipeline. For half buffers the same underlying logic holds. In particular, the dynamic slack is the total forward latency divided by the local cycle time. As an example, the precharged half-buffer template to be described in Chapter 11 has $FL = 2$ and cycle time 18 when approximated to unit delays. Consequently, for an N-stage pipeline constructed from these half buffers the dynamic slack is $N/9$.

4.2.2 Series composition of linear pipelines

The throughput of a series composition of two linear pipelines with different performance characteristics is limited by the slower pipeline and can thus be no larger than the minimum of their respective peak throughputs. Assume that, for a given throughput τ, the series composition of two pipeline can support a range of tokens $[d_{\min}(\tau), d_{\max}(\tau)]$. Then $d_{\min}(\tau)$ and $d_{\max}(\tau)$ are respectively the sums of the smallest and largest numbers of tokens capable of achieving a throughput τ in the two constituent pipelines. In other words, $d_{\min}(\tau)$ is obtained by summing the number of tokens in each constituent pipeline for the data-limited region and $d_{\max}(\tau)$ is obtained by summing the number of tokens in each pipeline for the bubble-limited region.

More generally, for a non-homogeneous linear pipeline in which each stage i is a full buffer with different FL_i and BL_i values, the pipeline throughput is limited by the maximum local cycle time of all stages and the throughput versus token graph is a trapezoid, as illustrated in Figure 4.4. The slopes of the lines in the data-limited (bubble limited) regions generalize to the inverse of the sum of the forward (backward) latencies of all stages. The dynamic slack is the range of tokens $(d_{\min}, d_{\max})$ that can reside in the pipeline when maximal throughput is achieved. More specifically,

$$d_{\min} = \frac{\sum_i FL_i}{\max_i(FL_i + BL_i)},$$

$$d_{\max} = s - \frac{\sum_i BL_i}{\max_i(FL_i + BL_i)},$$

where s is the static slack of the pipeline.

For half buffers the slope in the data-limited region is the same as in the full-buffer case but the slope of the line in the bubble-limited region is higher than in the full-buffer case owing to the reduced slack. We refer the reader to [3] for more details.

4.2.3 Improving throughput

There are several ways to improve the throughput of a linear pipeline. If the throughput is limited by either the left-hand or right-hand environment then no change in the pipeline itself can improve performance. When the limiting factor is the local cycle time of a particular stage, however, the throughput can often be improved by splitting the bottleneck stage into smaller pipeline stages, which may have smaller forward and backward latency components.

Notice that this optimization increases the slack of the pipeline. A natural question to ask is "Does adding slack affect the behavior of the system?" In fact, in most carefully designed asynchronous systems, slack can be added to any channel without affecting correctness. The class of systems for which this is true has been studied and coined *slack-elastic* by Manohar and Martin [4]. In particular, they showed that all asynchronous systems that do not exhibit arbitration behavior are slack-elastic. In systems that are not slack-elastic, however, care must be taken as adding slack can cause deadlock. This will be explored further in Chapter 6.

One practical use of adding slack to a channel is to combat the impact of long channel wires in the layout. Adding slack essentially pipelines long wires without needing to change the environment, whereas in synchronous circuits it would effectively change the clock edge at which the receiver would expect the data. If the design were not latency-insensitive, such a change would require a redesign of its FSM control and this would make the pipelining of long wires late in a chip's design-cycle prohibitively time consuming. Herein lies a significant advantage of latency-insensitive design and in particular slack-elastic asynchronous design.

In addition, because the backward latency is a function of the design of neighboring pipeline stages, adding a fast buffer in between two relatively slow pipeline stages can also improve throughput. This happens because the long local cycle time associated with the handshaking between the two slow stages is effectively replaced with two shorter local cycle times, each involving handshaking between one slow stage and the fast buffer.

4.3 Pipeline loops

Pipeline loops, also called rings [5][7], comprise a set of pipeline stages arranged in a loop. In most cases, the ring implements an iterative algorithm and each token in the ring operates independently of other tokens, representing an independent thread of execution.

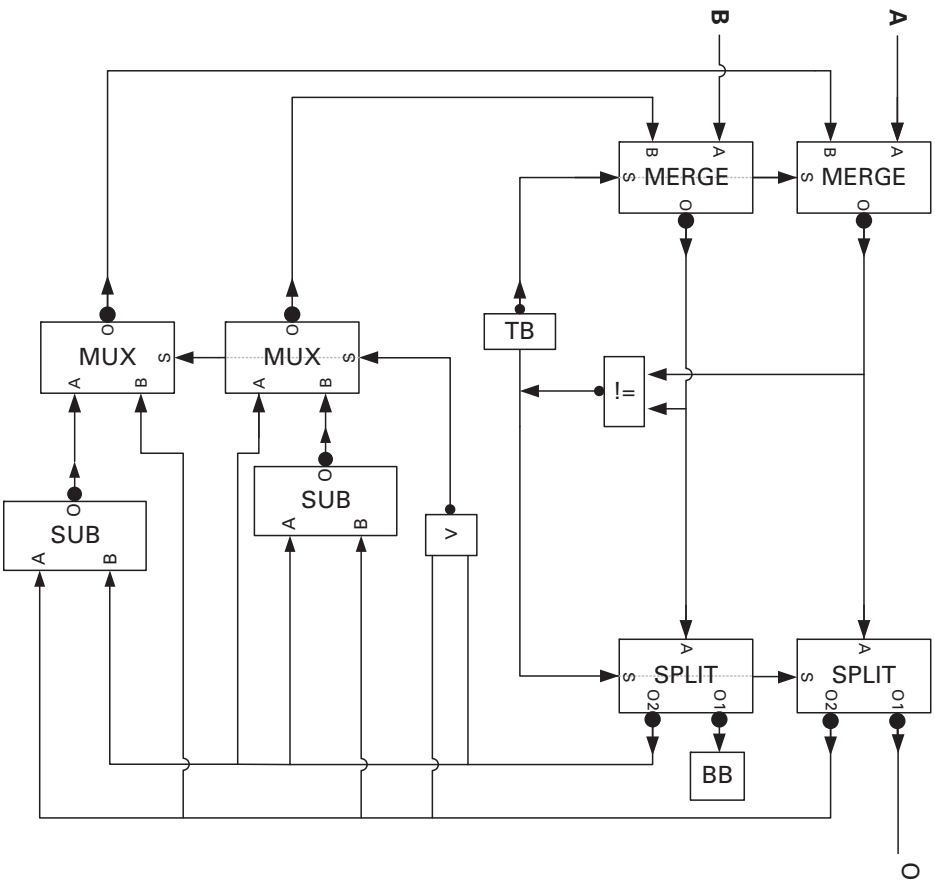

Figure 4.5. Pipeline ring-based implementation of Euclid's algorithm.

4.3.1 Design example: implementation of Euclid's algorithm

As an example, consider the implementation of Euclid's algorithm to find the greatest common divisor of two numbers. The algorithm can be stated as follows:

```
function gcd(A, B)
        while A ≠ B
              if A > B
                    A: = A − B
              else
                    B: = B − A
        return O = A
```

One possible implementation is shown in Figure 4.5; this performs both subtractions in parallel, using two sub elements, and conditionally updates both variables using two mux elements that are controlled by the result of a single ">" element. Note that the mux elements each consume both data input tokens independently

of the control value whereas in the merge element only one data input token is consumed. In this way, unwanted tokens do not pile up at the inputs of the mux elements. Note also that, to simplify the figure, we have omitted the necessary copy elements (they are implied by the forking of channels).

All loops in the design are restricted to having only one token, i.e. operating on only one problem instance at a time, via the single *token buffer*, denoted TB. Upon initialization TB generates a token and acts as a regular buffer for the rest of the time. This enables a single token from each of the external inputs A and B to enter the pipeline upon initialization. Once the end of the algorithm is reached, the corresponding tokens represent the results of the algorithm and leave the loop through the split elements and a new set of inputs is allowed to enter the loop. In particular, notice how this behavior is controlled by the value of the one-bit output token of the "! = " (equality operator) module. When the value is "false," the split elements send the result tokens out to the environment and the merge elements concurrently accept new inputs token from the environment. When the value is "true," however, the action of the merge and split elements recirculates the tokens around the loop in order to perform another iteration on the old data.

Interestingly, this design can be made to multi-thread many problem instances simply by the addition of token buffers after the equality operator. The inherent token flow ensures that the different problem instances remain independent as they are iteratively processed along the ring and outputted to the environment.

A VerilogCSP model of a token buffer is shown in Figure 4.6. Notice how the Verilog "init" block [1] is used to send out the initial token upon reset.

```
module TOK_BUF(L, R);
  parameter width = 8; parameter init = 8'b0;
 `INPORT(L,width);
 `OUTPORT(R,width);
 `USES_CHANNEL
  parameter FL = 2; parameter BL = 4;
  reg[width -1:0] d;
  initial
  begin
      `SEND(R,init)
  end
  always
  begin
      `RECEIVE(L,d)
       #FL;
      `SEND(R,d)
       #BL;
  end
endmodule
```

Figure 4.6. VerilogCSP model of a TOK_BUF.

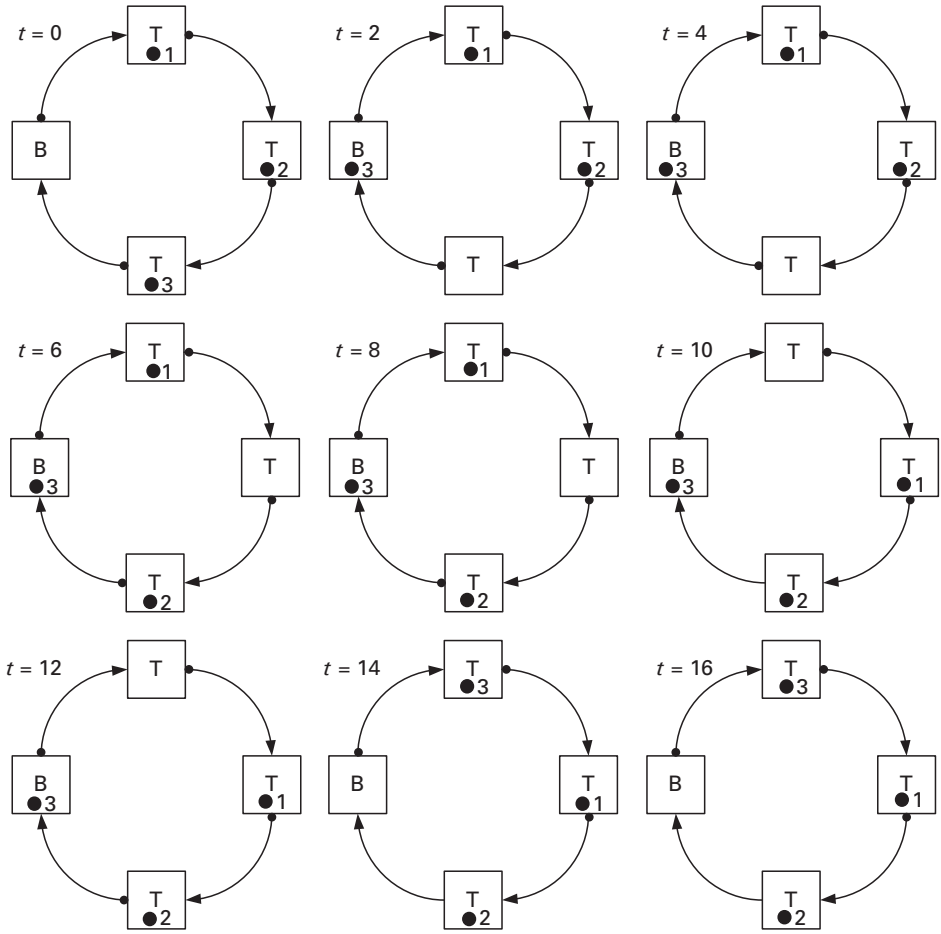

Figure 4.7. Bubble-limited performance bottleneck in a ring.

In implementations of a TOK_BUF this initialization is often triggered by a global reset signal. However, the use of the initial block allows us to model this reset behavior abstractly in our architectural model.

4.3.2 Performance analysis of rings

For the purpose of understanding the underlying performance bottlenecks in such architectures, simple rings of buffers, including some token buffers, are an effective abstraction that we can more easily analyze. For example, consider the behavior of a four-stage ring with three token buffers, illustrated in Figure 4.7, in which the token buffers are labeled T and the buffer is labeled B.

One can think of a ring as a linear pipeline where the input and output channels are tied together. As in a linear pipeline, the performance of a ring is optimal if no

buffer in the ring starves owing to a lack of tokens on its input channel or stalls because the output channels are busy. If there are too few data tokens in a ring (the data-limited case), the cycle time is determined by the sum of the forward latencies around the ring divided by the number of data tokens in the ring. If there are too many data tokens in the ring (the bubble-limited case), the cycle time is determined by the sum of the backward latencies around the ring divided by the number of bubbles in the pipeline. Notice, however, that after initialization the number of tokens in a ring is constant and equals the number of token buffers in the ring, in this case 3.

In Figure 4.7, all buffers are assumed to have FL and BL equal to 2 and 4 respectively, yielding a local cycle time equal to 6. Because the pipeline is bubble-limited, however, the cycle time is much larger. In particular, after token 3 is sent to on received by a new buffer at time 2, a new token cannot be sent to on received by the vacated buffer until BL time units later, i.e. at time 6. A similar stall occurs after token 2 is sent and received and subsequently after token 1 is sent and received. This leads to a cycle time equal to 16, as can be seen from the figure.

For an N-stage ring of homogeneous full buffers, the optimal number of tokens or dynamic slack is determined by finding when these two conditions yield the same value. This occurs when the number of tokens in the pipeline equals

$$\frac{N}{1 + BL/FL}.$$

As an example, if $BL = FL$ then the dynamic slack is $N/2$. A similar analysis works for half buffers, remembering that the number of bubbles is $N/2$ minus the number of tokens, assuming an even number of stages. With this assumption, the optimal performance occurs at

$$\frac{N/2}{1 + BL/FL}.$$

We leave the analysis and simulation of the more general case of a non-homogeneous ring as an exercise for the reader.

This performance analysis can be illustrated in a throughput versus token number graph, as for linear pipelines. However, unlike in the linear-pipeline case, a non-integral dynamic slack is not achievable. Thus on the throughput versus token number graph we use dots to highlight the feasible points, as shown for a four-stage ring in Figure 4.8. In particular, the dynamic slack equals $N/(BL/FL + 1) = 4/(2 + 1) = 4/3$. Because this is non-integer, the optimal throughput $(FL + BL)^{-1} = 1/6$ associated with the inverse of the local cycle time cannot be achieved and the best throughput achievable is actually $1/8$, with either one or three tokens in the ring.

4.3.3 Improving ring throughput

If a ring operates in the bubble-limited region, its performance can be improved by adding more pipeline buffers to the ring. This result may be counterintuitive to

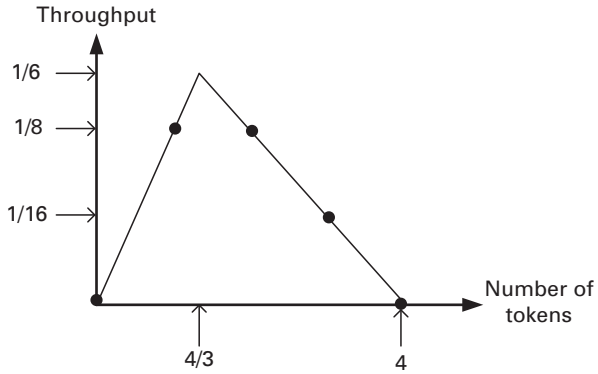

Figure 4.8. Throughput versus token number graph for a four-stage ring with $FL=2$ and $BL=4$.

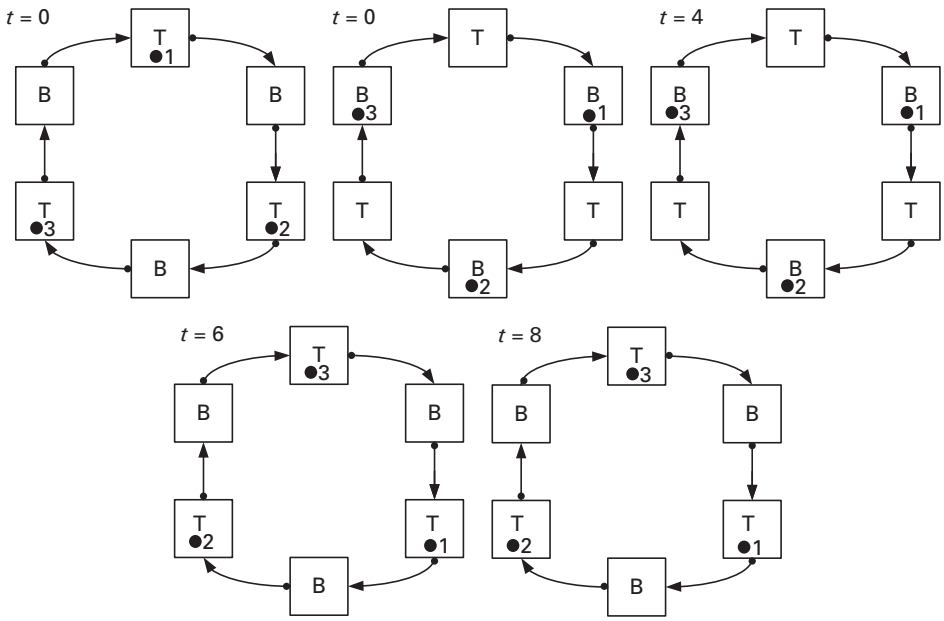

Figure 4.9. Adding buffers to a ring can improve the cycle time.

some readers because it seems to suggest that by adding delay to the pipeline performance can be improved. However, the addition of a pipeline stage also adds slack to the ring, which provides more bubbles and so shortens the backward bubble propagation that can otherwise stall the pipeline. This is illustrated in Figure 4.9. Here two buffers have been added to the ring shown in Figure 4.7, and this improves the cycle time from 16 to 8.

If the ring operates in the data-limited region then to improve throughput either more tokens must be inserted into the ring or the latency around the ring

must be shortened. Shortening the latency around the ring can be done with logic optimization and transistor sizing but is inherently limited by the complexity of the operations in the ring. Adding tokens to the ring implies an architectural change that involves increasing the number of parallel executing threads within the ring, because each ring token represents a distinct problem instance. In practice this optimization must be done in the context of designing the entire system to handle the added complexity associated with these additional threads of execution. This is not always trivial or possible and, for this reason, analyzing the delay of algorithmic loops and determining the number of tokens in the associated rings should be part of the early architectural analysis for any pipelined asynchronous design.

4.4 Forks and joins

A reconvergent fanout path occurs when a pipeline forks, has multiple independent paths, and then rejoins. It is a common pipeline structure and can be part of a loop. As an example, in the implementation of the Euclid algorithm in Figure 4.5, both the variables A and B fork to the comparator ">" and the two subtractors SUB before rejoining at merge elements.

We illustrate the performance issues in fork–joins using the more abstract and straightforward fork–join pipeline shown in Figure 4.10. The fork S0 outputs tokens to both the top and bottom pipeline branches. The join S8 waits for a token to arrive at each of its inputs before consuming both and generating an output. As with the linear pipeline and pipeline rings, multiple tokens representing different problem instances can reside in each branch of the fork, but in this case a problem instance is represented by a pair of tokens, one in each branch. In Figure 4.10 the fork–join pipeline is processing three pairs of tokens.

From a performance perspective, the key feature of this fork–join pipeline is that the number of tokens in each branch is identical. In particular, one way to view the throughput versus token number graph for this structure is first to analyze the performance of each branch independently as a linear pipeline and then to add the constraint that the number of tokens in the branches must be the same. In particular, at each value for the number of tokens in the pipeline, the throughput is limited by the lowest throughput of both branches. Consequently, the throughput versus token number graph for a fork–join structure can be obtained by taking for each value of the number of tokens, on the horizontal axis, the minimum of the throughputs associated with each branch.

Figure 4.11 illustrates some important points. First, notice that the static slack of the fork–join pipeline is equal to the lower static-slack value of its two branches. Second, notice that the peak throughput of the fork–join structure can be lower than the peak throughput of either branch taken separately. This happens because individually the branches can have different dynamic slack values and forcing them to work in parallel can force the shorter fork to operate with more tokens

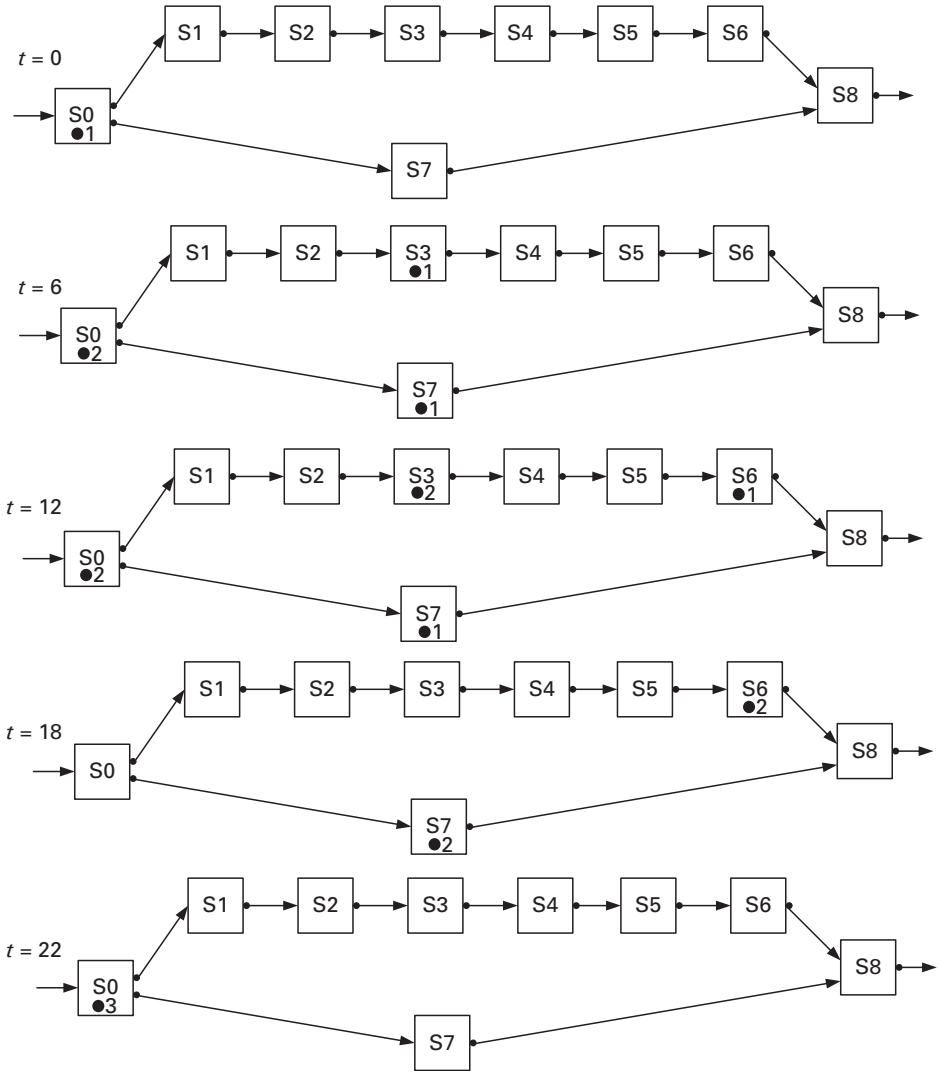

Figure 4.10. A fork–join pipeline at five different times.

than specified by its dynamic slack. In this example, the *FL* and *BL* values of of all pipeline stages are 2 and 4, respectively. Ideally, then, we would be able to insert a new token into the pipeline every 6 time units, i.e. at the rate of the local cycle time. However, because of the differences in slack, the bottom (shorter) branch is forced to operate in its bubble-limited region and the overall throughput becomes 1/11. This can also be seen from Figure 4.10, as it takes 22 time units to receive two new tokens into the fork–join pipeline, corresponding to an (average) cycle time equal to 11.

A simplifying design constraint aimed at avoiding this bottleneck is to require that the slacks along both branches are identical. This heuristic is reasonable

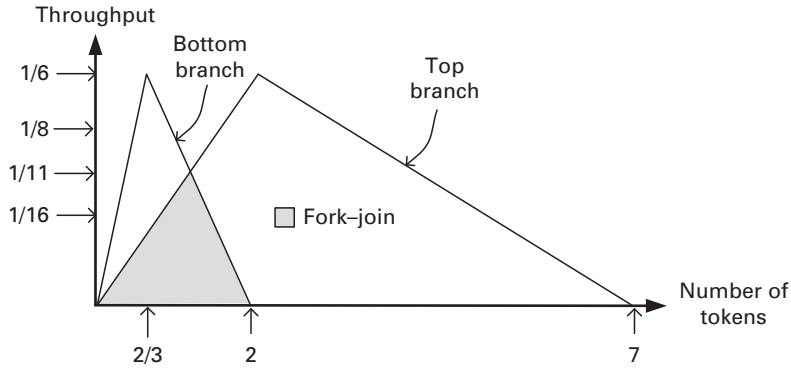

Figure 4.11. Throughput versus token number graph for a fork–join pipeline.

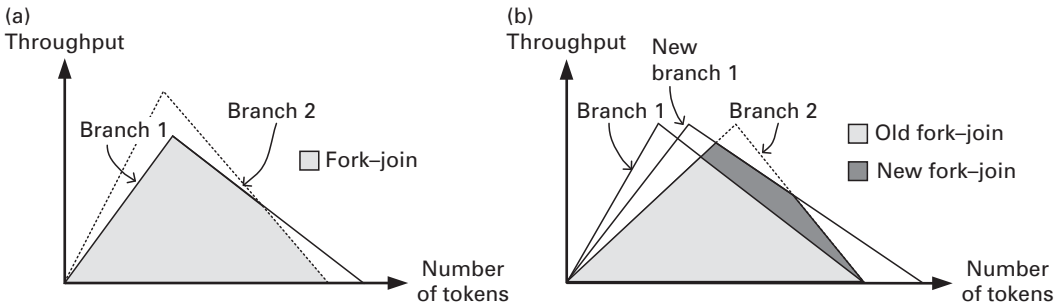

Figure 4.12. Fork–join performance characteristics.

because disparate slacks will cause the branch with the larger slack to operate with far fewer tokens than the optimal value identified by its dynamic slack. Nevertheless, the heuristic is not ideal when the peak throughputs or dynamic slacks of each branch are different. For example, a shorter branch with less slack but higher peak throughput may not be the cause of the bottleneck when paired with a slower longer branch, as illustrated in Figure 4.12(a). In addition, adding pipeline buffers to a branch may increase the fork–join throughput despite causing a slack imbalance, as illustrated in Figure 4.12(b).

4.5 More complex pipelines

We hope from these examples that the reader can appreciate that the analysis and optimization of asynchronous pipelines has several clear differences from the synchronous pipeline case. In general, it is important to identify bottlenecks and to consider adding slack (so-called *slack matching*) or redesigning existing stages to remove them. However, when one is dealing with more complex fork–join structures, conditional communication, and the avoidance of long wire delays in an

automated place-and-route flow this can be challenging to achieve manually [8]. Thus more mathematical approaches that can lead to automated techniques for performance optimization are needed. We will study these in Chapter 5.

4.6 Exercises

4.1. Generalize the analysis presented in this chapter to find the dynamic slack of a non-homogeneous ring of N full-buffer stages.

4.2. Design in VerilogCSP a 20-stage homogeneous full-buffer linear pipeline with $FL = 4, BL = 2$, and configurable local cycle times on the bit generator and bit bucket. Add statistics logic to either your bit generator or bit bucket that will measure and display the long-term average throughput of the pipeline. Now add Verilog code to your bit generator and bit bucket and to the environment that includes both of these, in order to measure and display the long-term average number of tokens in the pipeline.

Run simulations with a variety of bit-bucket and bit-generator delay values in the data- and bubble-limited regions. Use the points to plot the throughput versus token number graph. Explain any deviations from the theory. What is the dynamic slack? For which values of the local cycle times of the bit generator and bit bucket does the pipeline run at maximum throughput?

4.3. Repeat Exercise 4.2 with a 30-stage ring of full buffers instead of a linear pipeline. In this case modify the token-buffer logic to measure the throughput statistics. Will it matter from which token-buffer instance you take your measurements?

4.4. Draw the throughput versus token number graph for the six-stage ring shown in Figure 4.9. What is the dynamic slack and the optimal throughput? Verify that the theory yields a cycle time of 8 when three tokens are in the ring.

4.5. Verify your analysis in Exercise 4.4 by simulating the ring with different numbers of token buffers using VerilogCSP.

4.6. In Figure 4.9, the two additional buffers were placed between the token buffers. Would the result change if instead the three buffers were in a group in between two token buffers? Verify your analysis by simulating both cases in VerilogCSP.

4.7. Consider the fork–join pipeline in Figure 4.10. Identify the times at which the first 10 pairs of tokens enter the join stage S8. Analyze the resulting throughput and confirm that the (long-term average) cycle time is 11, as suggested by the corresponding throughput versus token number graph in Figure 4.11.

4.8. Consider a fork-join pipeline with seven buffers in the top branch and four buffers in the bottom branch. Let each fork or join and buffer have $FL = 2$ and $BL = 8$. Draw the throughput versus token number graph for this fork–join pipeline. Consider adding special pipeline buffers with $FL = 2$ and $BL = 4$

to the short fork. How many are needed to achieve the optimal throughput, 1/10? How many are needed to achieve a throughput of 1/12?

4.9. Design a VerilogCSP model of Exercise 4.8 with fast bit generators and bit buckets connected to the input and output of the fork–join structure. Print out waveforms and identify the critical cycle of operations that leads to the original cycle time. Repeat for the optimized design, which achieves a throughput of 1/12.

4.10. Design a VerilogCSP model of the implementation of Euclid's algorithm in Figure 4.5. Assume that each block has $FL = 2$ and $BL = 4$. Slack-match the design to achieve optimal throughput.

4.11. Repeat Exercise 4.10 but with a second token buffer added to the design that enables the pipeline to operate on two independent greatest common divisor problem instances simultaneously.

References

[1] "IEEE Std 1364–2001," *IEEE standard for Verilog hardware description language*, 2001.

[2] A. Saifhashemi and P. A. Beerel, "High level modeling of channel-based asynchronous circuits using Verilog," in *Proc. Conf. on Communicating Process Architectures 2005 (WoTUG-28)*, vol. 63 of *Concurrent Systems Engineering*, 2005.

[3] A. M. Lines, Pipelined asynchronous circuits, Master's thesis, California Institute of Technology, June 1998.

[4] R. Manohar and A. J. Martin, "Slack elasticity in concurrent computing," in *Proc. 4th Int. Conf. on the Mathematics of Program Construction*, Lecture Notes in Computer Science, vol. 1422, Springer-Verlag, 1998, pp. 272–285.

[5] J. Sparsø and J. Staunstrup, "Delay-insensitive multi-ring structures," *Integration, VLSI J.*, vol. 15, no. 3, pp. 313–340, October 1993.

[6] M. Greenstreet and K. Steiglitz, "Bubbles can make self-timed pipelines fast," *J. VLSI and Signal Processing*, vol. 2, no. 3, pp. 139–148, November 1990.

[7] T. E. Williams, "Self-timed rings and their application to division," Ph.D. thesis, Stanford University, June 1991.

[8] P. Golani, G. D. Dimou, M. Prakash, and P. A. Beerel, "Design of a high-speed asynchronous turbo decoder," in *Proc. Async'07*, 2007.

5 Performance analysis and optimization

Many different mathematical tools have been used for modeling concurrent systems. One of the most common is Petri nets. This chapter focuses on their use in architectural performance analysis and the optimization of asynchronous pipelines. In later chapters we will discuss the more specific forms of Petri nets used in specifying asynchronous controllers for automated synthesis. In addition, we will also use Petri nets to describe the abstract behavior of various implementation templates. More extensive analyses of Petri nets can be found in references [1]–[3].

5.1 Petri nets

A Petri net is a four-tuple $N = (P, T, F, m_0)$, where P is a finite set of places p_i and T is a finite set of transitions t_i (see Figure 5.1); $F \subseteq (P \times T) \cup (T \times P)$ is a flow relation and $m_0 \in N^{|P|}$ is the initial marking (see below), where N is the set of natural numbers. As can be seen in the figure, a Petri net is usually represented as a bipartite graph in which the p_i and t_i are the nodes. For any two nodes x and y, if $(x, y) \in F$ then there is a directed *arc* from x to y. An arc runs between a place and a transition or between a transition and a place, but arcs do not run between two places or two transitions.

In the example Petri net in Figure 5.1, $P = \{p_1, p_2, p_3, p_4, p_5, p_6\}$, $T = \{t_1, t_2, t_3, t_4\}$, $F = \{(t_1, p_1), (t_1, p_2), (p_1, t_3), (p_2, t_4), (t_2, p_3), (t_2, p_4), (p_3, t_3), (p_4, t_4), (t_3, p_5), (p_5, t_1), (t_4, p_6), (p_6, t_2)\}$, and $m_0 = [0\ 0\ 0\ 0\ 1\ 1]$.

A *marking* is an assignment of tokens to places and represents the state of the system. Formally, a marking is a $|P|$-vector m, where the number of tokens in place p under marking m, denoted by $m(p)$, is a natural number. We say for a place or transition $x \in P \cup T$ that $\bullet x$ is the *preset* of x, defined as $\bullet x = \{y \in P \cup T \mid (y, x) \in F\}$ and that $x\bullet$ is the *postset* of x, defined as $x\bullet = \{y \in P \cup T \mid (x, y) \in F\}$. A transition t is enabled at marking m if each place p from which an arc comes into t (these places constitute its preset, $\bullet t$) is marked with at least one token. When the transition is enabled, it can fire by removing one token from each place in its preset, $\bullet t$, and adding one token to each place in its postset, $t\bullet$. For example, in Figure 5.1 the transition t_1 can fire, and in doing so removes the token from p_5 and adds it to p_1 or p_2.

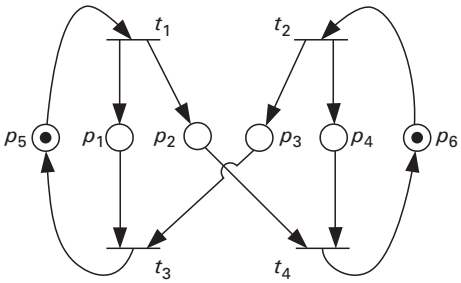

Figure 5.1. An example of a Petri net: the open circles indicate places and the short lines indicate transitions.

5.1.1 Petri net types

A state machine (SM) is a type of Petri net where every transition has at most one input and one output place, which is represented as $| \bullet t | \leq 1 \wedge | t\bullet | \leq 1, \forall t \in T$. A marked graph (MG) is a type of Petri net in which every place has at most one input and output transition, which is represented as $| \bullet p | \leq 1 \wedge | p\bullet | \leq 1, \forall p \in P$. A state machine can model *choice* but not *concurrency*, whereas a marked graph can model *concurrency* but not *choice*. Notice that the Petri net in Figure 5.1 is not a state machine because, for instance, $t_1\bullet = \{p_1, p_2\}$ and thus has size 2. It is, however, a marked graph, because all the places p_i have presets and postsets of size 1.

When a place p has more than one postset transition, that is $| p\bullet | > 1$, we say that p is a *choice place*. There are a several types of choice place. A place p is an extended-free-choice place if all transitions in $p\bullet$ can be enabled regardless of external influence, in other words, if the selection of the transition to fire is non-deterministic. A special case of extended free choice is a free-choice place p for which all output transitions reached from p have only one input place, as illustrated in Figure 5.2(a). More mathematically, extended-free-choice places satisfy the constraint that all such places $p \in P$ have either the same postset or non-intersecting postsets. As an example, the places p_2 and p_3 in Figure 5.2(b) are both extended-free-choice places because they have exactly the same postset transitions and no other place has an intersecting postset transition. A place p is a unique-choice place if no two output transitions of p can be enabled at the same time. In this case, the selection of the transition to be fired is deterministic. This is illustrated in Figure 5.2(c), in which, despite the fact that place p_1 is a choice place with two postset transitions t_1 and t_2, only one of these two transitions can be enabled to fire at a time. Any choice place that is neither extended-free-choice nor unique-choice is classified as arbitration-choice, as illustrated by place p_5 in Figure 5.2(d). It has two postset transitions, t_2 and t_4, which are simultaneously enabled if tokens exist both at places p_1 and p_4. This implies, however, that the choice also may depend on the relative arrival time of tokens at places p_1 and p_4. Namely, if a token exists only in one of these places, say p_1, then only one transition, in this case t_2, is enabled to fire. Moreover, if t_2 does not fire

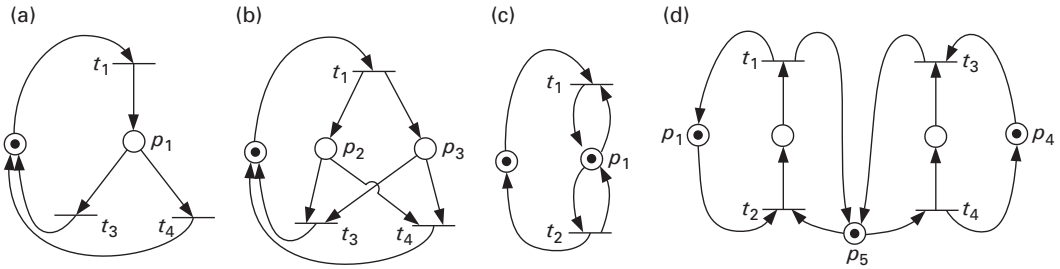

Figure 5.2. Petri nets with choice.

immediately and in the meantime a token arrives at p_4 then the choice of which transition will be fired at that point becomes non-deterministic.

5.1.2 Reachability graph

A marking m' is reachable from m if there is a sequence of firings $t_1\ t_2\ \ldots\ t_n$ that transforms m into m'; this sequence is denoted $m[t_1t_2\ \ldots\ t_n> m'$. The set of reachable markings from m_0 is denoted by $[m_0>$. By considering the set of reachable markings as the set of states of the system and the transitions between these markings as the transitions between the states, a *reachability graph* can be obtained representing the underlying behavior of the Petri net. A Petri net is *k-bounded* if no marking in $[m_0>$ assigns more than k tokens to any place of the net. It is *safe* if it is 1-bounded. In contrast, an unbounded net allows an unbounded number of tokens to accumulate at some place. It is *live* if and only if (iff) all transitions can eventually be enabled from every $m \in [m_0>$.

As an example, the reachability graph for the Petri net in Figure 5.1 is shown in Figure 5.3. Each node in the graph is a 6-ary (six-dimensional) vector of natural numbers $[p_1, p_2, p_3, p_4, p_5, p_6]$ identifying the number of tokens in each place. Each edge represents a state transition and is labeled with the transition that fired. Notice that this Petri net is not safe because some reachable markings have more than one token in places p_2 and p_3. It is however 2-bounded and live.

For the purpose of modeling asynchronous designs we typically restrict ourselves to k-bounded live Petri nets because the designs have limited memory but never-ending behavior in which all transitions can repeatedly fire.

5.1.3 Modeling delays in Petri nets

In some variants of Petri nets, each transition or place can be annotated with delay information, such as (i) a single value representing a fixed delay, (ii) a pair of values denoting the lower and upper bounds of the delay, or (iii) a more general stochastic delay distributions [2][7][23]. These Petri nets with timing constraints are often called *timed Petri nets* and are further classified as timed-place Petri nets (TPPNs) or timed-transition Petri nets (TTPNs), depending on whether the timing bounds annotate places or transitions.

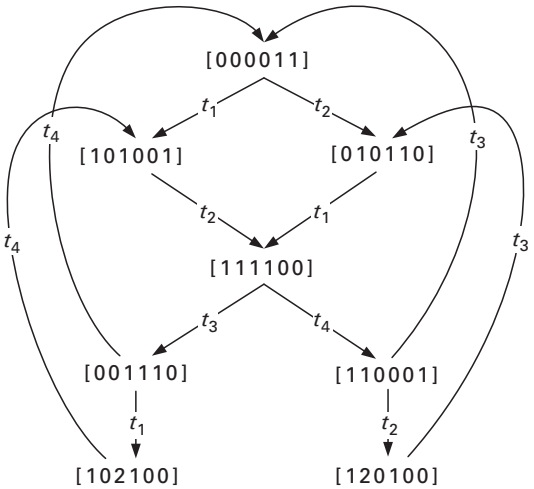

Figure 5.3. Reachability graph for the Petri net shown in Figure 5.1.

When, on the one hand, delays are associated with places, a token flowing into a place p experiences a delay $d(p)$ before it can be consumed by an output transition of p. The delays experienced in different places are independent. The actual firing of a transition is assumed to be *instantaneous*. When, on the other hand, delays are associated with transitions, the transition takes a delay $d(t)$ to fire.

We will focus on modeling and performance analysis using timed marked graphs with fixed delays, for which there is a rich set of analysis and optimization techniques. While such timed marked graphs are sufficient to analyze deterministic pipelined systems, they cannot model pipelines that have choice-behavior. Consequently, our analysis and optimization techniques will often be limited to analyzing one specific or typical mode of operation (which can be modeled with marked graphs) at a time. More advanced techniques that handle more general timed Petri nets will be briefly discussed in Section 5.5.

As with all Petri nets, the delay information in a *timed marked graph* can be associated with transitions or places; however, associating delays with places is less constrained than associating them with transitions. The reason is that every MG with delays associated with transitions can be translated into one in which these delays are instead associated with all places in its preset. Because there is no choice in marked graphs, there is no significant difference between the two types of marked graph model [7]. The semantics of a timed MG state that a transition t fires after the tokens in places $p \in \bullet t$ have resided in p for at least $d(p)$ time units.

5.1.4 Cycle time

A cycle c is a sequence of places $p_1\ p_2\ \ldots\ p_1$ connected by arcs and transitions, whose first and last places are the same. The cycle time $\mathrm{CT}(c)$ of a cycle c is the sum $d(c)$ of the delays of all associated places (or transitions) along the cycle c divided

by the number $m_0(c)$ of tokens that reside in the cycle, i.e. $CT(c) = d(c)/m_0(c)$. The *cycle time* of an MG is defined as the largest cycle time of all cycles in the MG, i.e. $max\ \forall c \in Y\ \Omega\ CT(c)$, where Y is the set of all cycles in the timed MG [19]. The intuition behind this well-known result is that the performance of any computation modeled with a timed MG is dictated by the cycle time of the timed MG and thus the largest cycle time for a cycle in the MG.

5.2 Modeling pipelines using channel nets

5.2.1 Full-buffer channel nets

We will model a network of leaf cells with a novel timed marked graph called a full-buffer channel net (FBCN), a specific form of pipelined Petri net (see e.g. [14]). It is defined as $N = (P \cup \overline{P}, T, F, m_0)$ and satisfies place symmetry, i.e. $|P| = |\overline{P}|$. It also satisfies channel connectivity, i.e. F satisfies the property that for every $p \in P$ there exist (t_i, p) and $(p, t_j) \in F$ and a $\overline{p} \in \overline{P}$ such that $(t_j, \overline{p})$ and $(\overline{p}, t_i) \in F$. Finally the channels are single-token i.e., for every $p \in P, m_0(p) + m_0(\overline{p}) = 1$. The idea behind a channel net is that each leaf cell is modeled as a transition and each channel between the ports of cells becomes two places. Four arcs in the flow relation connect the places to the relevant leaf cell transitions.

The performance of the network is represented by modeling the performance of the channels according to how they are connected to the leaf cells. In our model, the local cycle time is attributed to the output channels to which the leaf cell is connected. The delay $d(p)$ represents the forward latency of a channel while the corresponding $d(\overline{p})$ value represents its backward latency. Intuitively, the forward latency represents the delay through an empty channel (and associated leaf cell) and the backward latency represents the time it takes for the handshaking circuitry within the neighboring leaf cells to reset, enabling a second token to flow. The cycle time $c(p \circ \overline{p})$ associated with the cycle defined by p and $\overline{p}$ represents the local cycle time of the channel and equals the sum of the two place delays, i.e.

$$\begin{aligned} c(p \circ \overline{p}) &= d(p) + d(\overline{p}) / [m(p) + m(\overline{p})] \\ &= d(p) + d(\overline{p}). \end{aligned}$$

As an example, we illustrate two channel nets in Figure 5.4. The open circles represent forward places $p \in P$ and the open boxes represent backward places $\overline{p} \in \overline{P}$. In this example, both channels have forward latency 2 and backward latency 8 and thus local cycle times equal to $2 + 8 = 10$. We have choosen the forward latency to be significantly smaller than the backward latency since this is the case for many asynchronous templates, as will be described in the second part of the book. In Figure 5.4(a) (Figure 5.4(b)), the forward (backward) channel is marked, indicating that immediately after reset this channel is full (empty). For illustrative purposes only, the forward places are represented by circles and the backward places by squares. Note that this model implies that every channel can

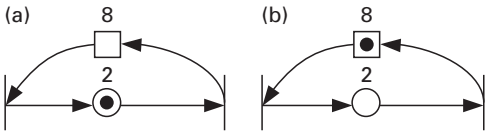

Figure 5.4. Full-buffer channel nets.

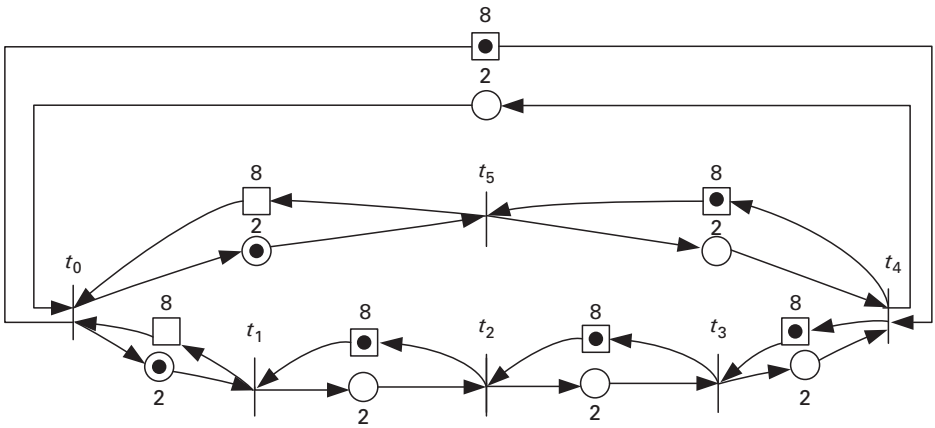

Figure 5.5. Full-buffer channel-net model of a homogeneous unbalanced fork–join pipeline.

hold at most one token and thus that the leaf cells are *full buffers* [11]. An extension to model *half buffers* [11] more accurately is an area of research and development.

Notice that in any marking (see Figure 5.1), if the forward place is marked then it represents a state in which there exists a token in the channel. Otherwise, the backward place is marked. This configuration guarantees that only one data or control token can reside in the channel and that the marked graph is safe.

5.2.2 Cycle time and throughput

The cycle time of an asynchronous pipeline is captured by the cycle time of its FBCN model (see the previous subsection). The throughput of the circuit is the reciprocal of this value. Very often additional buffers, also known as slack, must be added to the model to balance the pipelines and thereby improve the cycle time. We model the addition of slack between leaf cells by creating new transitions, places, and arcs that represent buffer leaf cells and their corresponding channels.

As an example, consider a homogeneous non-linear pipeline fork–join channel structure in which there are three buffers (represented by vertical bars) in one path and one buffer in the other path. The FBCN model of this structure is illustrated in Figure 5.5. Notice also that the forking transition t_0, which represents the leaf-cell

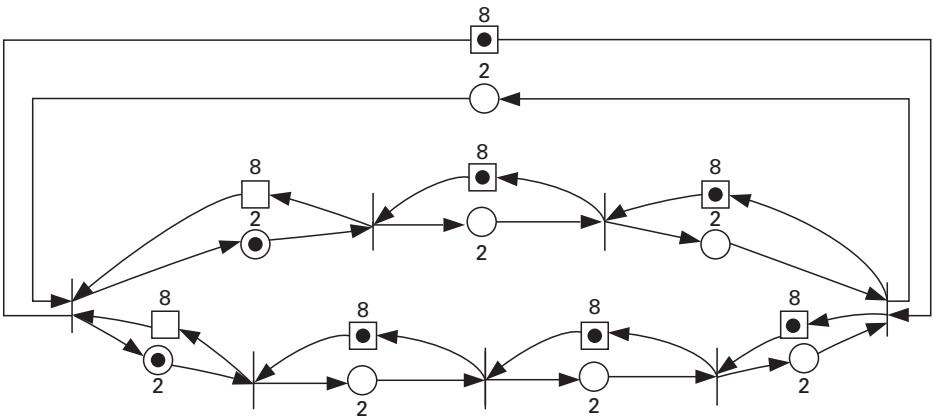

Figure 5.6. Adding a pipeline buffer can improve the cycle time of an FBCN.

fork has both output channels initially full, whereas all other channels are initially empty, indicating that this leaf cell is the only token buffer in the system. The cycle yielding the cycle time consists of the forward latency path through the long fork and the backward latency path of the short fork. Its cycle time is $(2 + 2 + 2 + 2 + 8 + 8)/2 = 24/2 = 12$.

Note that if a second buffer is inserted in the short forked path at t_5, as illustrated in Figure 5.6, the cycle time reduces to $(2 + 2 + 2 + 2 + 8 + 8 + 8)/3 = 32/3$.

5.3 Performance analysis

There are a variety of ways to determine the cycle metric of timed marked graphs. The most obvious approach is to enumerate all cycles, evaluate their individual cycle times, and identify the maximum value. While sometimes efficient, the worst-case complexity is poor because the number of cycles in a marked graph may be exponentially larger than its size. Consequently, this has been a research topic for which several more efficient approaches have been identified. We first review two direct approaches and then an approach based on solutions to a related problem, called the *maximum cycle mean problem*, for more general directed graphs.

5.3.1 Shortest-path-based algorithm

Ramamoorthy and Ho were among the first workers to propose an efficient algorithm for the performance analysis of timed marked graphs [6]. In particular, their algorithm verifies whether a timed marked graph has a cycle time no larger

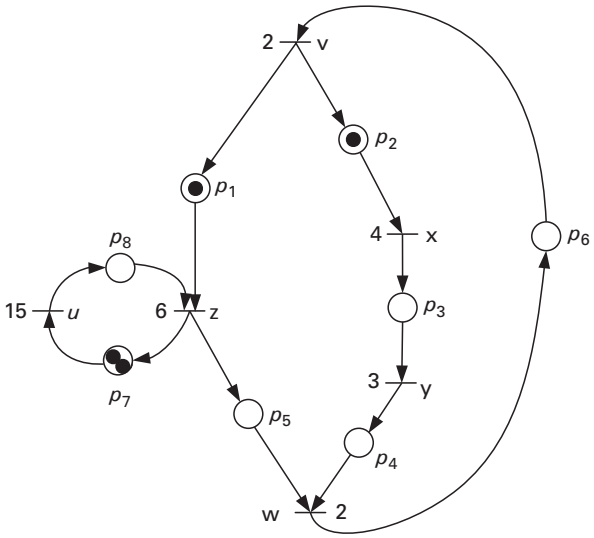

Figure 5.7. Example of a timed marked graph.

than a given value C. Their approach applied to timed marked graphs having delays associated with transitions and involved creating two $n \times n$ matrices P and Q, where n is the number of places in the Petri net. Each entry (i, j) in P has the value $m_0(i)$ if there is a transition between i and j. All other entries of P have a zero. Each entry (i, j) in Q has the value $d(i)$ if there is a transition between places i and j. Otherwise the entry is $-\infty$.

Using the matrix $CP - Q$ as a distance matrix between place pairs, a matrix S is derived that represents the shortest path (in terms of cumulative delays) between any two places. If all diagonal entries in S are positive then the system's cycle time is less than C. If some diagonal entries are zero but none are negative, the cycle time is C. Otherwise, the cycle time is greater than C.

As an example, the matrix $CP - Q$ for the timed marked graph in Figure 5.7 for $C = 11$ is as follows:

$$CP - Q = \begin{bmatrix} \infty & \infty & \infty & \infty & 5 & \infty & 5 & \infty \\ \infty & \infty & 7 & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & -3 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & -2 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & -2 & \infty & \infty \\ -2 & -2 & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 7 \\ \infty & \infty & \infty & \infty & -6 & \infty & -6 & \infty \end{bmatrix}.$$

In particular, the (2, 3) entry equals 7 because it corresponds to p_2 and p_3, which are separated by the transition x with $d(x) = 4$ and $11 - 4 = 7$. Notice also that for any pair of places (i, j) that are not directly connected via a transition the

distance matrix has an entry ∞. The corresponding all-pairs shortest path matrix is as follows:

$$S = \begin{bmatrix} 1 & 1 & 8 & 5 & 5 & 3 & 5 & 12 \\ 0 & 0 & 7 & 4 & 5 & 2 & 5 & 12 \\ -7 & -7 & 0 & -3 & -2 & -5 & -2 & 5 \\ -4 & -4 & 3 & 0 & 1 & -2 & 1 & 8 \\ -4 & -4 & 3 & 0 & 1 & -2 & 1 & 8 \\ -2 & -2 & 5 & 2 & 3 & 0 & 3 & 10 \\ -3 & -3 & 4 & 1 & 1 & -1 & 1 & 7 \\ -10 & -10 & -3 & -6 & -6 & -8 & -6 & 1 \end{bmatrix}.$$

As an example, $S(1, 3) = 8$ because the shortest path between p_1 and p_3 goes through p_5, p_6, and p_2, yielding the cumulative delay $5 + (-2) + (-2) + 7 = 8$. Note that because no diagonal sums in S are negative and some are zero, the cycle time of the marked graph is exactly 11.

The distance matrix can be computed with any all-pairs shortest-path algorithm, including the well-known Floyd–Warshall algorithm, which takes $O(n^3)$ time. This is substantially lower than the worst-case exponential time complexity for computing the cycle times of all cycles individually. Readers interested in the details of all-pairs shortest-path algorithms can refer to [15] for a more in-depth analysis of various algorithms and their complexities.

5.3.2 Linear-programming-based approaches

Magott [5] proposed determining the cycle time of a timed marked graph using linear programming. He created arrival-time variables a_i for every transition i and also the following set of constraints for every pair of transitions (i, j) connected by a place p:

$$a_j \geq a_i + d(i) - \tau m_0(p).$$

Maggot proved that when this set of constraints is satisfied, the cycle time of the marked graph is no greater than τ. With this result, he proposed a linear programming (LP) problem based on these constraints and an objective function that minimizes τ. This is a useful problem formulation for several reasons. First, there are many freely available LP solvers (e.g. [18]), including those built into MATLAB. Second, many of these solvers are very computationally efficient. The most common are based on the simplex algorithm developed by Dantzig [16] and in practice often take a linear time to solve the problem but have worst-case exponential complexity. Other algorithms have been developed that have polynomial worst-case complexity but often take longer in practice. Interested readers are referred to [17] for a more thorough review of various algorithms.

As an example of the LP approach, the formulation for the example in Figure 5.7 is as follows:

$$
\begin{aligned}
&\text{minimize } \tau \text{ subject to}\\
&a_x \geq a_v + 2 - \tau,\\
&a_z \geq a_v + 2 - \tau,\\
&a_z \geq a_u + 15,\\
&a_u \geq a_z + 6 - 2\tau,\\
&a_y \geq a_x + 4,\\
&a_w \geq a_y + 3,\\
&a_v \geq a_w + 2,\\
&a_w \geq a_z + 6.
\end{aligned}
$$

Running this algorithm through an LP may provide many different results for the arrival times but always generates a minimum τ equal to 11, corresponding to the one-token cycle $< v, x, y, w >$.

Extending this result to marked graphs with delays on places is also possible. The only change is that the arrival-time constraint for a pair of transitions (i, j) connected by place p uses the delay on the place rather than on the transition i, as follows:

$$a_j \geq a_i + d(p) - \tau m_0(p).$$

The proof of this result can be trivially obtained from [25], which proves a similar result using equality constraints that include a non-negative slack variable.

Burns and Martin proved similar results, using a slightly different formalism called *event-rule systems* rather than timed marked graphs [12][13].

With this modified formulation we can now use linear programming to find the cycle time of full-buffer channel nets. As an example, the linear program for finding the cycle time of the FBCN in Figure 5.5 is:

$$
\begin{aligned}
&\text{Minimize } \tau \text{ subject to}\\
&a_1 \geq a_0 + 2 - \tau,\\
&a_0 \geq a_1 + 8,\\
&a_2 \geq a_1 + 2,\\
&a_1 \geq a_2 + 8 - \tau,\\
&a_3 \geq a_2 + 2,\\
&a_2 \geq a_3 + 8 - \tau,\\
&a_4 \geq a_3 + 2,\\
&a_3 \geq a_4 + 8 - \tau,\\
&a_0 \geq a_4 + 2,\\
&a_4 \geq a_0 + 8 - \tau,\\
&a_0 \geq a_5 + 8,\\
&a_5 \geq a_0 + 2 - \tau,\\
&a_4 \geq a_5 + 2,\\
&a_5 \geq a_4 + 8 - \tau.
\end{aligned}
$$

The minimum τ is 12, matching the value described in Subsection 5.2.2. One solution to the arrival times is $[a_0\ a_1\ a_2\ a_3\ a_4] = [10\ 0\ 2\ 4\ 6]$. The specific values for the arrival times are, however, not unique. For example, adding a constant k to all arrival times also leads to a valid solution.

5.3.3 Relation to maximum-cycle-mean problem

Let $G = (N, E, w)$ be a weighted directed graph with n nodes N and m edges $E \subseteq N \times N$. A cycle in G is a sequence of nodes in N in which the first and last nodes are identical. Let C represent the set of all cycles of nodes in G. The weights are then described by the function w: $E \cup C \rightarrow \mathrm{R}$, applied to both edges and cycles. In particular, the weight $w(c)$ of the cycle c is equal to the sum of the weights of the edges in c. The (cycle) *mean* of c is defined as $w(c)/|c|$. This is the average weight of the edges in c. The *maximum cycle mean* is simply the largest cycle mean.

We can use this maximum-cycle-mean problem to solve for the cycle time of marked graphs with a maximum of one token per place, as follows. Create a vertex for each marked place in the marked graph. Create an edge between every pair of vertices for which there exists at least one path between their corresponding places that does not go through other marked places. Now associate with each such path the sum of the delays on its places and transitions. Then let the weight of the edge be the largest such sum for the various possible paths. For timed marked graphs with delays associated with places, this sum should include the delay of the source place but exclude the delay of the sink place. The maximum cycle mean of this graph equals the cycle time of the marked graph.

Marked graphs with more than one token per place in the initial marking can also be supported by expanding the multi-marked place into a sequence of dummy places and zero-delay transitions with one token per place. The cycles in this new marked graph are in one-to-one correspondence with the original marked graph and have the same cycle time. As an example, the marked graph in Figure 5.7 has two tokens in place p_7. Expanding the graph with a dummy transition and creating a second place p'_7 and distributing these two tokens between p_7 and p'_7 yields the marked graph shown in Figure 5.8(a).

The associated weighted graph is shown in Figure 5.8(b). The maximum cycle mean of this weighted graph corresponds to the self loop at place p_2 with mean weight 11. As a second example, the directed graph that corresponds to the FBCN model shown in Figure 5.5 is shown in Figure 5.9. Notice that the maximum cycle mean, 12, arises from the two-edge cycle $< p_{01}, p_{45}, p_{01} >$.

5.3.4 Karp's algorithm

In this subsection we briefly review Karp's well-known algorithm for automatically finding the maximum cycle mean [4]. We refer the interested reader to [9][10] for in-depth reviews and analyses of Karp's algorithm among others.

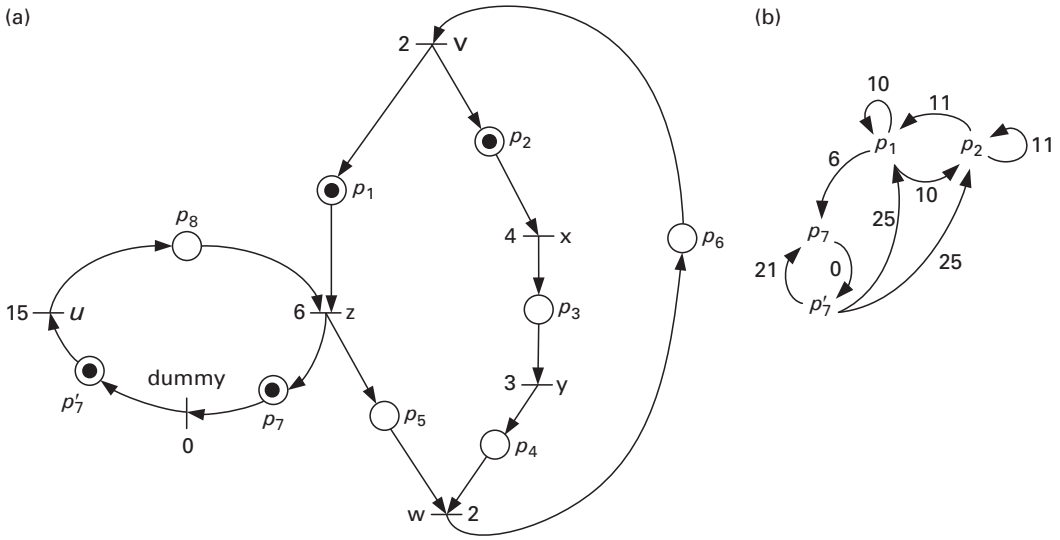

Figure 5.8. (a) A marked graph and (b) its corresponding weighted directed graph.

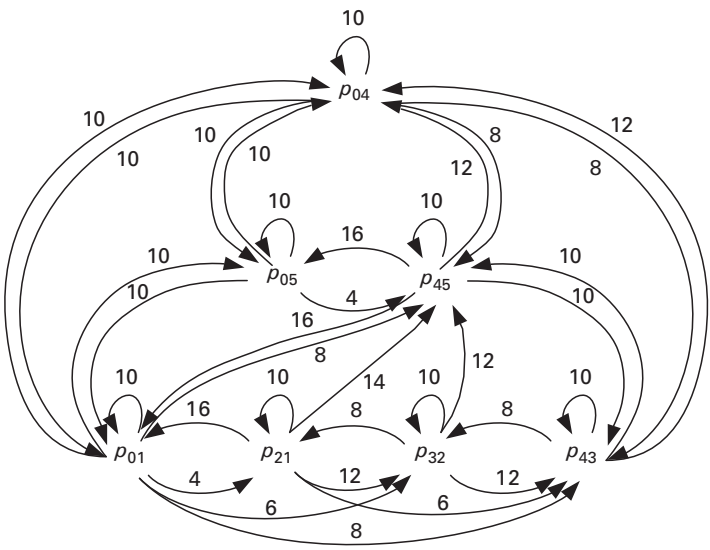

Figure 5.9. Weighted directed graph corresponding to the FBCN in Figure 5.5.

Let s be an *arbitrary* node, called the *source*, in a weighted directed graph G. For every v in N and every non-negative integer k, define $D_k(v)$ as the maximum weight of a path of length k from s to v; if no such path exists then $D_k(v) = -\infty$. Then, the maximum cycle mean λ^* of G is given by the following formula, as proven in [4],

$$\lambda^* = \max_{v \in V} \min_{0 \le k \le n-1} \frac{D_n(v) - D_k(v)}{n - k}.$$

Table 5.1. Calculation of the maximum cycle mean for the graph in Figure 5.8(b); $D_k(v)$ is the maximum weight of a path of length k from the source node to v

k	$D_k(p_1)$	$D_k(p_2)$	$D_k(p_7)$	$D_k(p_7')$
0	0	$-\infty$	$-\infty$	$-\infty$
1	10	10	6	$-\infty$
2	21	21	16	6
3	32	32	27	16
4	43	43	38	27
$\min_{0\le k\le 4-1} \frac{D_4(v)-D_k(v)}{4-k}$	$10\frac{3}{4}$	11	$10\frac{2}{3}$	$10\frac{1}{2}$

If G is not strongly connected then we can find the maximum cycle mean λ^* by finding the strongly connected components of G (in linear time), determining the maximum cycle mean for each component, and then taking the largest of these as the maximum cycle mean. Unless stated otherwise, we will consider only strongly connected graphs, so that to $n \le m$. The weights $D_k(v)$ can be determined by the following recurrence relation [4]:

$$D_k(v) = \max_{(u,v)\in E}[D_{k-1}(u) + w(u,v)], \qquad k = 1, 2, \ldots, n,$$

with the initial conditions $D_0(s) = 0$ and $D_0(v) = -\infty$ for all other vertices v.

As an example, consider the application of Karp's algorithm to the directed weighted graph shown in Figure 5.8(b) (which has one strongly connected component) and let the source node s be p_1. The calculation of the weights $D_k(v)$ is shown in Table 5.1. As can be seen, the maximum cycle mean is $\lambda^* = 11$. As expected, this equals the cycle time for the corresponding marked graph, as determined in Section 5.2.1.

5.4 Performance optimization

5.4.1 Slack matching: an intuitive analysis

An intuitive understanding of the slack-matching problem can be obtained by analyzing further the unbalanced fork–join pipeline in Figure 5.5. In particular, a key observation is that if tokens arrive at a join stage at different times then the early token will stall and the stall will propagate backwards and slow the entire system down. We illustrate this intuition by annotating the FBCN in Figure 5.5 with token arrival times, as illustrated in Figure 5.10. The unbalanced nature of the pipeline causes the first token processed by the top "short" fork, stage t_5, to stall for 4 time units while waiting for the lower "longer" fork to propagate its first token. Because t_5's backward latency is 8, the output channel of t_5 can accept a second token no sooner than $t = 16$, i.e. t_5 fires for the second time at $t = 16$. This means that the second token that arrives at the input channel of t_5 is stalled for 4 time units during this reset period. This stall delays the time at which t_0 generates

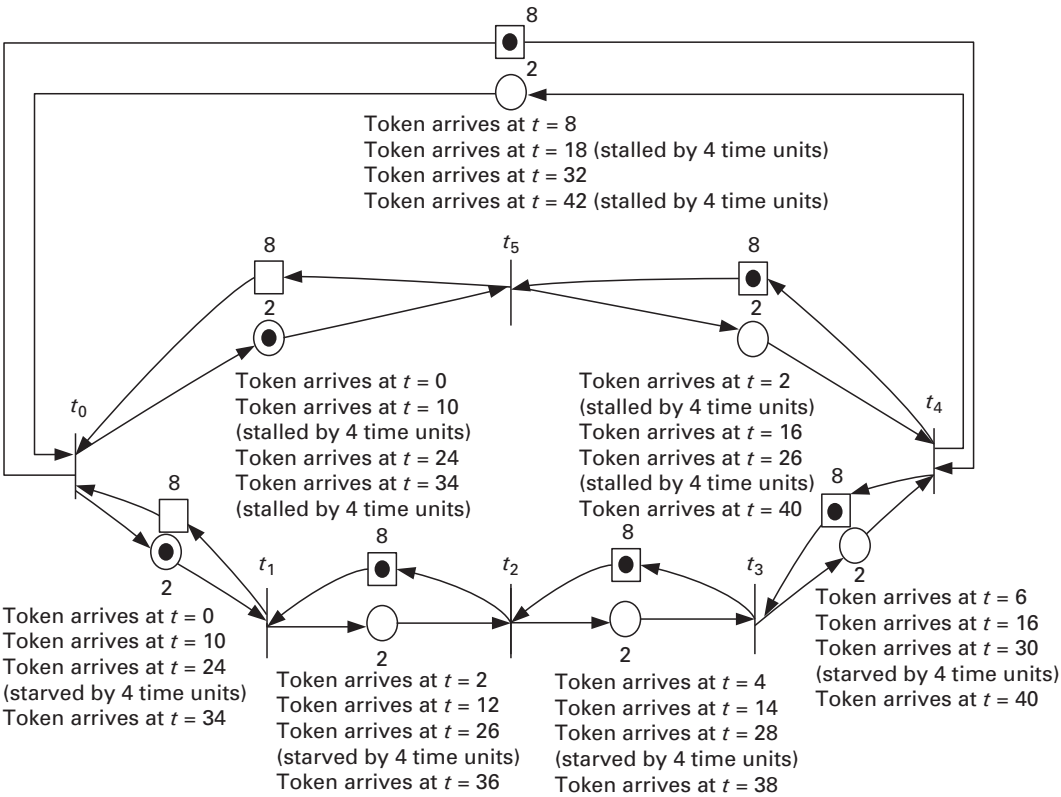

Figure 5.10. Annotation of Figure 5.5 showing how an unbalanced fork–join pipeline causes pipeline stalls.

the third token by 4 time units, starving t_1 by 4 time units. This propagation of starving on stalling continues and every other token processed by every channel is delayed by 4 time units. Consequently, instead operating at a peak local cycle time of 10 time units, each channel operates at an average cycle time of 12 time units. The intuition gained from this example is as follows: for the global cycle time to equal the local cycle time, all tokens must arrive at all join stages at the same time.

The above intuition is a necessary condition for the global cycle time to be equal to the local cycle of the channels in a homogeneous pipeline in which all channels have the same cycle time. However, it is not sufficient. To see this, consider another important case, that of a short loop of leaf cells, when a token can propagate around the loop faster than the local cycle time. In this case a token will be stalled while the local channel resets.

Like the above case, this stall will propagate backward and increase the global cycle time. In particular, it is the backward propagation of bubbles into which a token can leads to a move that throughput bottleneck. This is illustrated in Figure 5.11.

Alternatively, if the latency around a one-token loop is larger than τ, the cycle time will be greater than τ. Consequently, another necessary condition is that for

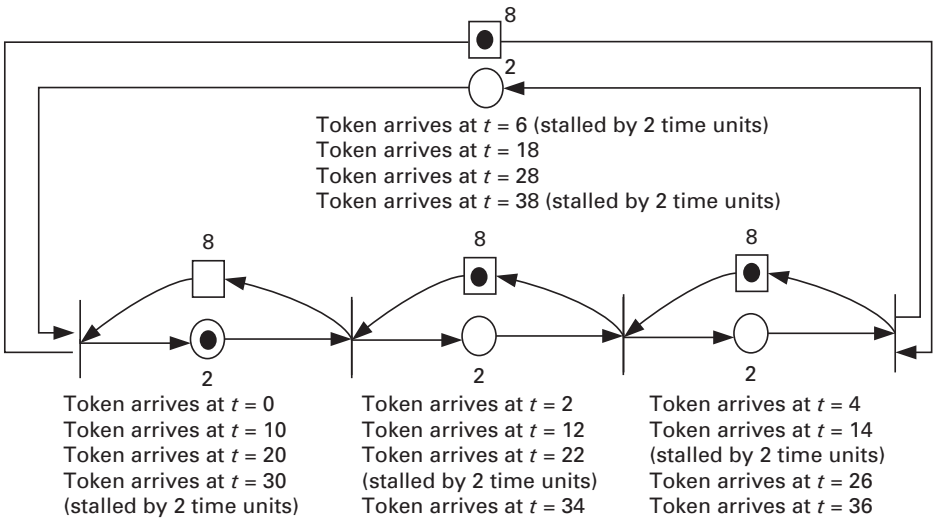

Figure 5.11. Illustration of how a short loop can cause pipeline stalls.

one-token loops the propagation delay along every cycle is equal to the local cycle time. Multi-token loops, with multiple token buffers, yield a generalization of this condition. Specifically, for the global cycle time to equal the local cycle time the latency along any m-token loop must be $m\tau$.

This intuition must be modified when considering the case in which one channel resets faster than others, i.e. it has a smaller backward latency. If a token is stalled in this channel by a small amount, the channel may still be able to reset in time to accept the next token within the desired global cycle time. If the forward latency plus stall time plus backward latency equals the desired global cycle time then this stall will not bottleneck the design. We call the difference between desired global cycle time and the local cycle time the *free slack*.

For example, if the channel driven by the top buffer in Figure 5.4 had backward latency of 4 instead of 8, it would have a local cycle time of 6 and a free slack equal to $10-6=4$. This free slack would be able to absorb the stall, and the worst-case cycle metric of the pipeline would be 10, matching the worst local cycle time. Similarly, if the added pipeline buffer in Figure 5.6 has a free slack of 2 rather than 0, the worst-case cycle metric would be 10.

Note that this intuition also applies to cases when the desired global cycle time is larger than the homogeneous local cycle time, in that then all channels have free slack. Similarly, to meet the desired global cycle time, stalls caused by the backward propagation of bubbles must be less than the sum of the free slack around the cycle.

5.4.2 Slack matching: an MILP optimization framework

One possible goal of slack matching is to maximize throughput. A more general goal is to minimize the number or cost of adding slack buffers while achieving a target throughput. In [25], a mixed-integer linear program (MILP) for this problem was developed based on the FBCN model of the circuit:

$$
\begin{aligned}
&\text{Minimize the sum } s_c \text{ subject to}\\
&a_j = a_i - m + f_c + l_c + l_s s_c \qquad \text{(channel constraints)},\\
&f_c \le \tau - \tau_c + s_c(\tau - \tau_s) \qquad \text{(free slack constraints)},\\
&a_i \ge 0 \text{ and } f_c \ge 0 \qquad \text{(variable bounds)},\\
&s_c \in \mathrm{N} \qquad \text{(integral constraints)},
\end{aligned}
$$

for all transitions t_i and channels c between leaf cells i and j, where

- l_c, τ_c, and f_c are the latency, local cycle time, and free slack of channel c between leaf cells i and j,
- the a_i are free variables representing the arrival time of tokens at leaf cells, where there are as many variables a_i as leaf cells,
- the s_c are independent variables that identify the amount of slack added to channel c between leaf cells i and j,
- l_s and τ_s are the latency and local cycle time of a pipelined buffer,
- $m = 1$ if this channel upon reset has a token and $m = 0$ otherwise.

The channel constraints guarantee that the density of data tokens along a loop of leaf cells is sufficiently low that no token is stalled waiting for a channel to reset and that the latency along the loop is not too high, so that leaf cells are never starved. In addition, the constraints guarantee that the sum of the free slack along the short path of a fork–join path is sufficient to balance the two paths. Notice that the free slack of a channel, as constrained by the free-slack constraints, is upper bounded by the sum of two components. The first of these is the free slack associated with the difference between the cycle time of the channel and the target cycle time, $\tau - \tau_c$. The second component of the free slack is that obtained by adding s_c pipeline buffers, each contributing slack $\tau - \tau_s$. Note that we restrict s_c to be an integer because a fraction of a pipeline buffer cannot be added to a channel.

A simple extension that accounts for the differing costs of channels – these may depend on the physical size of the pipeline buffer and/or the width (i.e. number of rails) of a channel – is described in [25].

As an example, the MILP for the non-homogeneous fork–join pipeline described in Figure 5.5 (with local cycle time of the channel driven by the top buffer of 6 rather than 10) is as follows:

$$
\begin{aligned}
&\text{Minimize } s_{0a} + s_{0b} + s_1 + s_2 + s_3 + s_4 + s_5 \text{ subject to}\\
&\quad a_1 = a_0 - 8 + 2s_{0a},\\
&\quad a_5 = a_0 - 8 + 2s_{0b},\\
&\quad a_2 = a_1 + 2 + 2s_1,\\
&\quad a_3 = a_2 + 2 + 2s_2,\\
&\quad a_4 = a_3 + 2 + 2s_3,\\
&\quad a_4 = a_5 + 2 + f_5 + 2s_5,\\
&\quad a_0 = a_4 + 2 + 2s_4,\\
&\quad 4 \ge f_5 \ge 0.
\end{aligned}
$$

An optimal solution to this MILP is $a = [8\ 0\ 2\ 4\ 6\ 0]$, $f_5 = 4$, and $s = [0\ 0\ 0\ 0\ 0\ 0\ 0]$ with minimum objective function 0. That is, no slack is needed in this example, which is consistent with our intuition in subsection 5.4.1.

A fundamental problem with an MILP formulation is its computational complexity, which can be exponential in the size of the formulation [17]. While in some special cases the problem can be reduced to a linear program (LP) without losing exactness [25], an exact solution to most large-sized problems is often too computationally challenging to obtain. A practical approach that addresses this issue is to simply relax the integral restriction on the variables s_c, use an LP solver and round up the slack results to the nearest integral number of buffers. Then, if necessary, the buffer latencies are tuned further by transistor sizing or by constraining the physical design placement and/or routing to address any remaining performance bottlenecks. This approach may not lead to the minimum number of buffers or the optimal performance but experience has shown that it performs reasonably well in an industrial setting [30].

Other techniques for slack matching have also been proposed in the literature. Kim and Beerel analyzed the complexity of more general pipeline optimization problems and proposed a branch and bound algorithm that also was able to optimize the degree of pipelining of datapath units [24]. Prakash and Martin proposed a more complex MILP formulation based on a more exact model of the handshaking protocol [26]. Venkataramani and Goldstein proposed a simulation-based approach which, while not optimal, appears to work well in practice and can be applied to circuits with choice [27].

5.5 Advanced topic: stochastic performance analysis

More sophisticated performance-analysis techniques use random variables to model the delays. For a place p, we write $X(p)$ as the random variable denoting the delay associated with it and $F_{X(p)}$: $R\mu \rightarrow [0\mu\ 1]$ as the distribution function of $X(p)$, i.e., $F_{X(p)}(x) = \text{Prob}(X(p) \leq x)$.

For Petri nets with free or unique choices, these assumptions imply that there is no race condition among transitions in structural conflict, i.e. those in the postset of a choice place. For each free-choice place p, we assume there is a probability mass function (p.m.f) $\mu(p, \cdot)$ to resolve the choice. That is, for $t \in p\bullet$, $\mu\ (p,\ t)$ is the probability that t consumes the token each time p is marked. Of course, $\Sigma_{t \in p\bullet}\mu(p,\ t) = 1$.

We call such Petri nets stochastic timed Petri nets (STPNs) [88] and STPNs in which only fixed delays are allowed are called probabilistic timed Petri nets (PTPNs) [43]. Figure 5.12 shows an example of a PTPN in which p_8 is a free-choice place with a given probability mass function.

In a run of an STPN or PTPN, called a *timed execution*, choices are resolved and places are assigned delay values. In particular, we call the firing of a transition an *event*. A timed execution can be described as a sequence of events and their

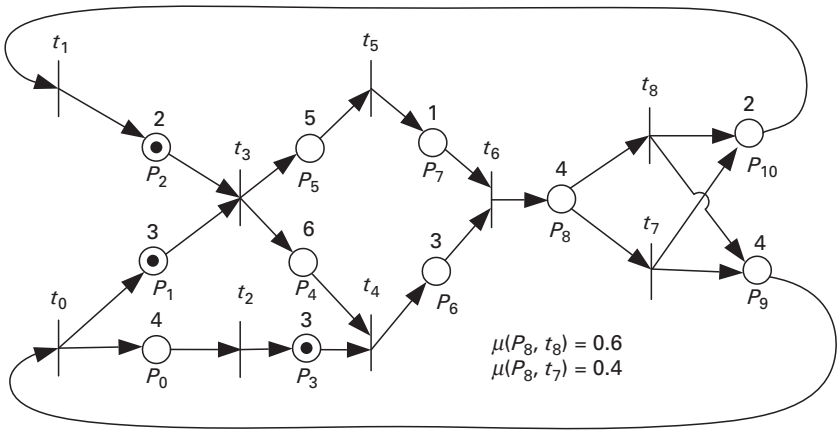

Figure 5.12. Example of a probabilistic timed Petri net.

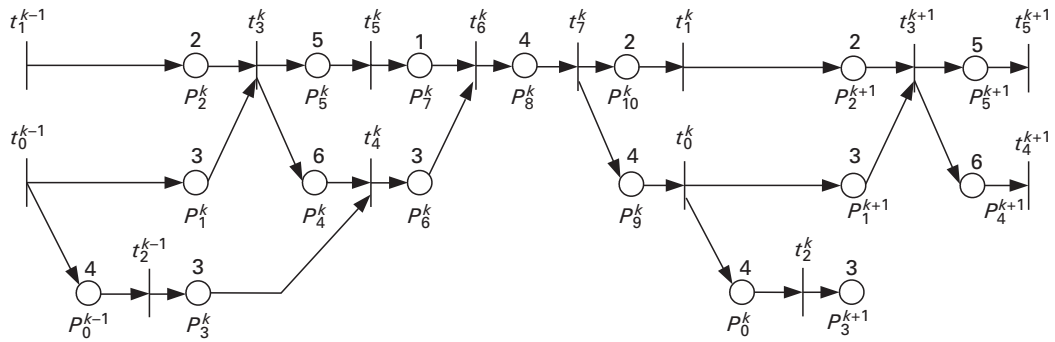

Figure 5.13. Example of a timed event graph illustrating a timed execution of the PTPN shown in Figure 5.12.

occurrence times. Alternatively, it can be depicted as an acyclic *timed event graph* in which the choice decisions are resolved, the causality among events is shown, and the place delays are shown.

As an example, Figure 5.13 shows the timed event graph of a timed execution of the PTPN in Figure 5.12. In particular, the numbers above the places denote the delay values. For convenience, we write t_k and p_j to denote the kth event due to the firing of t and the jth place, respectively.

Given a timed execution, the time separation of an event pair (TSE) is the time between the two events $s^{(k)}$, $t^{(k+\varepsilon)}$. The average TSE due to the separation triple $\gamma^{(k)}(s, t, \varepsilon)$ is the average over k of the corresponding TSEs of an infinite timed execution of the STPN. Note that there are cases where an average TSE does not exist. The difference between the numbers of occurrences of two transitions could tend to infinity as time progresses. However, for a wide class of event pairs, their average TSEs do exist [23].

Many system performance metrics can be expressed directly as the average TSEs of some indicator event pairs. For example, the latency of a linear pipeline can be measured as the average difference between the time t_r^i of the ith generation of an output token at the output of the pipeline and the time t_l^i at which the corresponding input token is consumed at the input. Similarly, the cycle time of the pipeline can be measured as the long-term average difference between t_r^i and t_r^{i+1}.

Consequently, the TSE framework is a very general framework for analyzing performance and many techniques have been developed in connection with it. However, modeling the delays as random variables and modeling choice increases the complexity of the approaches designed to solve for the TSEs.

One approach is based on discretizing time and then producing a reachability graph in which every state transition represents the advance of one time unit. In the presence of choices, the graph can be modeled as a Markov chain and Markovian analysis can be used to find the TSEs. The difficulty is that the reachability graph can have millions of states and the Markovian analysis quickly becomes computationally intractable. Techniques using symbolic methods to address the computational challenges of this approach were explored in [21]. For marked graphs, however, McGee, Nowick, and Coffman suggest using their periodic nature to simplify the required Markovian analysis [29].

Another approach uses Monte Carlo simulation to find the TSEs; more specifically, this technique analyzes the longest path of independently generated segments of timed executions to find bounds on average TSEs. This approach has been shown to be successful for very large STPNs but is limited to free-choice nets [22][23]. Extensions to nets that model arbitration is an interesting area of future work.

Lastly, Chakraborty and Angrish simplified the problem by analyzing only moments of distributions rather than detailed distributions [28].

5.6 Exercises

5.1. Draw the reachability graph for the marked graph in Figure 5.7 (ignoring delays). Is the marked graph safe? Is it live? Is it 2-bounded?

5.2. Illustrate Karp's algorithm for the weighted directed graph shown in Figure 5.9.

5.3. The cycle time of a marked graph will be identical for any initial marking $m_0 = m$, where m is any reachable marking, i.e. $m \in [m_0 >$. Consider the timed marked graph in Figure 5.7 with an alternative initial marking $m_0 = [1\ 0\ 1\ 0\ 0\ 0\ 2\ 0]$, i.e. where place p_3 is marked instead of p_2. Show the corresponding weighted directed graph and illustrate Karp's algorithm by finding the new cycle time. Verify that it is the same as the original cycle time.

5.4. A possible half-buffer model is shown in Figure 5.14. Notice how the addition of *BL2* placed between alternating transitions constrains the neighboring channels to not hold distinct tokens.

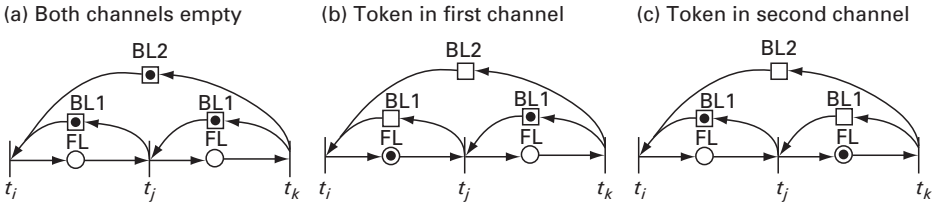

Figure 5.14. Possible half-buffer performance model.

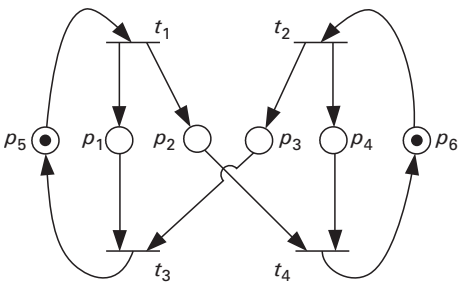

Figure 5.15. Example timed marked graph for Exercise 5.6.

Consider the fork–join loop structure shown in Figure 5.6 but with each full buffer replaced by a half buffer (include fork and join modules). Model the performance of this modified circuit using the above half-buffer model. Assume that $FL = 2$, $BL1 = 8$, and $BL2 = 6$. What is the cycle time of this timed marked graph model? Is it the same as that with full buffers?

5.5. Write the LP formulation for the half-buffer model in the above problem. Solve with any available LP solver.

5.6. Write an LP formulation to find the cycle time of the timed marked graph in Figure 5.15. Notice that the delays are associated with the places. Show a solution to your LP that yields the cycle time. Explain your answer.

References

[1] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, 1981.

[2] J. Wang, *Timed Petri Nets*, Kluwer Academic, 1998.

[3] T. Murata, "Petri nets: properties, analysis and application," *Proc. IEEE*, vol. 77, no. 4, pp. 541–579, 1989.

[4] R. M. Karp, "A characterization of the minimum cycle mean in a digraph," *Discrete Math.*, vol. 23, pp. 309–311, 1978.

[5] J. Magott, "Performance evaluation of concurrent systems using Petri nets," *Information Proc. Lett.*, vol. 18, pp. 7–13, January 1984.

[6] C. V. Ramamoorthy and G. S. Ho, "Performance evaluation of asynchronous concurrent systems using Petri nets," *IEEE Trans. Software Eng.*, vol. SE-6, no. 5, pp. 440–449, 1980.

[7] J. Campos, G. Chiola, J. M. Colom, M. Silva, "Properties and performance bounds for timed marked graphs," *IEEE Trans. Circuits and Systems I: Fundamental Theory and Applications*, vol. 39, no. 5, pp. 386–401, May 1992.
[8] M. Silva and J. Campos, "Structural performance analysis of stochastic Petri nets," in *Proc. IEEE Int. Computer Performance and Dependability Symp.*, 1995, pp. 61–70.
[9] A. Dasdan, "Experimental analysis of the fastest optimum cycle ratio and mean algorithms," *ACM Trans. Design Automation of Electronic Systems*, vol. 9, no. 4, pp. 385–418, October 2004.
[10] A. Dasdan and R. K. Gupta, "Faster maximum and minimum mean cycle algorithms for system-performance analysis," *IEEE Trans. CAD Integrated Circuits and Systems*, vol. 17, no. 10, pp. 889–899, 1998.
[11] A. M. Lines, "Pipelined asynchronous circuits," Master's thesis, California Institute of Technology, June 1998.
[12] S. M. Burns, "Performance analysis and optimization of asynchronous circuits," Ph.D. thesis, California Institute of Technology, 1991.
[13] S. M. Burns and A. J. Martin, "Performance analysis and optimization of asynchronous circuits," in *Adv. Res. VLSI*, MIT Press, 1991, pp. 71–86.
[14] V. I. Varshavsky, M. A. Kishinevsky, V. B. Marakhovsky, *et al.*, *"Self-timed Control of Concurrent Processes*, Kluwer Academic, 1990.
[15] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.
[16] G. B. Dantzig, *Linear Programming and Extensions*. Princeton University Press, 1963.
[17] C. H. Papadimitriou and K. Steiglitz, *Combinational Optimization: Algorithms and Complexity*, Dover, 1998.
[18] GLPK: GNU linear programming kit package, http://www.gnu.org/software/glpk/glpk.html.
[19] R. M. Karp, "A characterization of the minimum cycle mean in a diagraph," *Discrete Math.*, vol. 23, pp. 309–311, 1978.
[20] R. Manohar and A. J. Martin, "Slack elasticity in concurrent computing," in *Proc. 4th Int. Conf. on the Mathematics of Program Construction,* Lecture Notes in Computer Science vol. 1422, Springer-Verlag, 1998, pp. 272–285.
[21] P. A. Beerel and A. Xie, "Performance analysis of asynchronous circuits using Markov chains," in *Proc. Conf. on Concurrency and Hardware Design,* 2002, pp. 313–344.
[22] A. Xie and P. A. Beerel, "Performance analysis of asynchronous circuits and systems using stochastic timed Petri nets," in *Proc. 2nd Workshop on Hardware Design and Petri Net (HWPN'99) of the 20th Int. Conf. on Application and Theory of Petri Nets (PN'99),* 1999, pp. 35–62.
[23] A. Xie, S. Kim, and P. A. Beerel, "Bounding average time separations of events in stochastic timed petri nets with choice," in *Proc. 5th IEEE Int. Symp. on Asynchronous Circuits and System (Async)* 1999, p. 94.
[24] S. Kim and P. A. Beerel, "Pipeline optimization for asynchronous circuits: complexity analysis and an efficient optimal algorithm," *IEEE Trans. CAD Integrated Circuits and Systems*, vol. 25, no. 3, pp. 389–402, 2006.
[25] P. A. Beerel, A. Lines, M. Davies, and N. Kim, "Slack matching asynchronous designs," in *Proc. 12th IEEE Int. Symp. on Asynchronous Circuits and System (Async)*, March 2006, pp. 184–194.

[26] P. Prakash and A. J. Martin, "Slack matching quasi delay-insensitive circuits," in *Proc. 12th IEEE Int. Symp. on Asynchronous Circuits and System (Async)*, March 2006, pp. 195–204.
[27] G. Venkataramani and S. C. Goldstein, "Leveraging protocol knowledge in slack matching," in *Proc. ICCAD 2006*, pp. 160–171.
[28] S. Chakraborty and R. Angrish, "Probabilistic timing analysis of asynchronous systems with moments of delays," in *Proc. 8th Int. Symp. on Asynchronous Circuits and Systems*, April 2002, pp. 99–108.
[29] P. B. McGee, S. M. Nowick, and E. G. Coffman, "Efficient performance analysis of asynchronous systems based on periodicity," in *Proc. 3rd IEEE/ACM/IFIP Int. Conf. on Hardware/Software Codesign and System Synthesis*, September 2005, pp. 225–230.
[30] A. Lines. Private communication, 2006.

6 Deadlock

With new technology come new challenges. Deadlock is an important issue that can manifest itself far more commonly in asynchronous circuits than in their synchronous counterparts.

Deadlock is the failure or inability of a system to proceed and arises when two or more processes expect a response from each other before completing an operation and thus mutually block each other from progressing. A more formal definition is as follows: a set P of processes is said to be deadlocked if

- each process p_i in P is waiting for an event $e \in E$,
- each event $e \in E$ is generated only by a process $p_j \in P$.

The dining-philosophers problem, covered in Chapter 3, is a classic problem illustrative of deadlock. The general idea is that a number of philosophers surround a dining table with a big plate of food (in this case Chinese pot stickers) in the middle, as illustrated in Figure 6.1 for four philosophers. They spend time either thinking or eating, but because adjacent philosophers share one chopstick, they cannot eat simultaneously. A philosopher must have both the chopsticks to his or her left and right in order to eat. The philosophers never speak to each other, an aspect that introduces the possibility of a deadlock in which every philosopher holds, say, the chopstick on the left and is waiting for the chopstick on the right. This system has reached deadlock because there is a cycle of ungranted requests. In this case philosopher p_1 is waiting for the chopstick held by philosopher p_2, who is waiting for the chopstick of philosopher p_3, and so forth, making a circular chain.

The lack of available chopsticks is analogous to the locking of shared resources in computer programming. Locking a resource is a common technique to ensure that the resource is accessed by only one process at a time. When a resource in which a process is interested is already locked by another process, the first process waits until it is unlocked.

For the purposes of considering deadlock in asynchronous circuits we will cover a number of examples from four categories:

1. deadlock caused by incorrect implementation of the handshake protocol;
2. deadlock caused by architectural token mismatch;

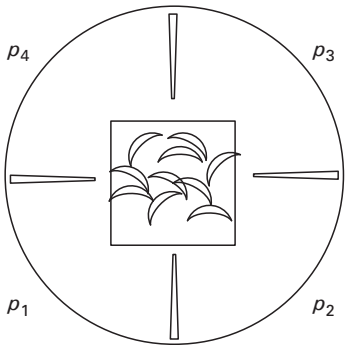

Figure 6.1. A set of processes, consisting of four dining philosophers, four chopsticks, and a plate of Chinese potstickers, which is liable to deadlock.

3. deadlock caused by arbitration;
4. deadlock caused by delay: depending on the delay value, deadlock may or may not occur.

In the following subsections we will discuss each deadlock category in turn.

6.1 Deadlock caused by incorrect circuit design

A poorly designed pipeline unit, which does not implement the intended communication protocol on its channels, can easily cause deadlock. Figure 6.2 illustrates such a case. It shows the waveform for a four-phase handshaking protocol that has been implemented incorrectly by a stage in the pipeline illustrated in Figure 6.3. Because the ack signal generated by stage S3, which acknowledges that the inputs are consumed, is not set back to 0 (see Figure 6.2), the sender, stage S2, cannot send a new data token. Thus the sender has data to send but is waiting for the ack signal to go low and the receiver, S3, is expecting new data because it thinks it has implemented the protocol correctly. Therefore this pipeline will deadlock and no data transmission will occur. All the tokens before S3 will sit there forever or until the system is reset. If the system is restarted and another token arrives, deadlock will occur again. While the cause of the deadlock in this particular example may be obvious and easy to spot, as a cell becomes more complicated, with conditional multi-inputs and multi-outputs and with internal states, verification to ensure that the four-phase protocol is implemented correctly on all channels can be a daunting task, which can easily be let slip. The detection of deadlock may require either exhaustive simulation, if a simulation-based verification method is used, or a cleverly implemented formal verification tool to ensure deadlock-free operation.

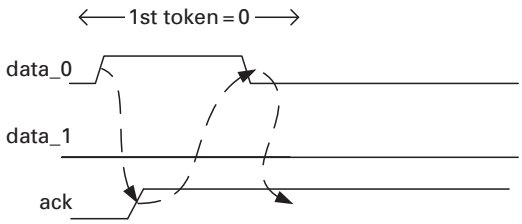

Figure 6.2. An incorrect four-phase handshake.

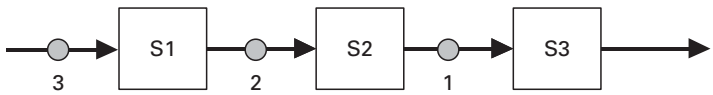

Figure 6.3. Stage S3 deadlocks the pipeline.

The deadlock introduced because the pipeline stage has been implemented incorrectly will be observed independently of the relative delays or data values chosen in the environment or arbitration within the circuit. This type of deadlock can also happen at the behavioral level, in for example VerilogCSP, if the underlying handshaking macros are incorrect.

If the incorrect handshaking was implemented as part of a conditional communication then the deadlock may appear to be data dependent. In other words, the block in which this pipeline stage is instantiated may deadlock only when the faulty conditional channel is used.

6.2 Deadlock caused by architectural token mismatch

The modules typically found in large designs expect to receive or generate a particular sequence of tokens. In general these tokens are used to initialize storage elements such as random access memory (RAM) or configuration registers, as part of a boot process. When the modules involved in this initial configuration process disagree on the number of tokens that must be exchanged, this often leads to deadlock. An insufficiency of data tokens in the system can cause deadlock via a starvation of data tokens to be processed, and having more tokens than the system can handle can cause deadlock via a starvation of empty space, i.e. bubbles enabling the system to move tokens around. Both situations can occur, depending on the data values used in the system. We will cover the two cases with examples.

6.2.1 Data token starvation

Consider the system illustrated in Figure 6.4. In this system the left-hand environment initializes the first 100 entries of SRAM_A with consecutive values

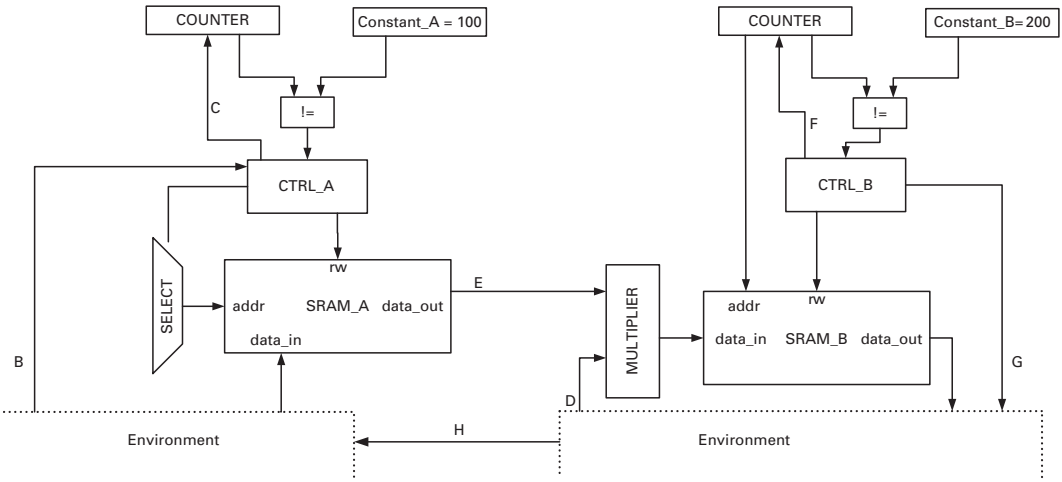

Figure 6.4. Deadlock caused by an incorrect value in a register.

increasing from 0 to 99. Then the right-hand environment reads these 100 values, multiplies them by a certain number, and stores them in SRAM_B. A deadlock may occur if the registers in the left- and right-hand circuits, Constant_A and Constant_B respectively, are initialized with different numbers while the system is expecting the same number of tokens to be exchanged.

The left-hand environment initializes SRAM_A with a set of numbers. It first sets the CTRL_A module, via control channel B to write to SRAM_A. The CTRL_A module drives the rw input of SRAM_A, thus setting SRAM_A to be read from or written to. The left-hand environment writes to the addresses 0–99. The register Constant_A is compared with the counter value and set to 100. Once the left-hand environment has completed the initialization of SRAM_A, it sets the block CTRL_A to read from the SRAM. The CTRL_A module increments the counter via control channel C. The counter value is compared with the value 100 stored in Constant_A. If they are not equal then the counter output is provided as the address input to SRAM_A. In this way the SRAM_A contents are read from addresses 0 to 99. Once the counter reaches 100, the comparison results in equality and thus the operation stops. The output of SRAM_A is multiplied by the data provided by the right-hand environment via the data channel E. Then this multiplication result is stored in SRAM_B. Again the results are stored from addresses 0 to 99. However, the register on the right-hand side, Constant_B, is initialized to 200 rather than 100. Therefore this part of the design expects 200 data tokens to be multiplied and then stored from 0 to 199. The CTRL_B module generates a data token to the right-hand environment via control channel G to indicate that it has stored all the received and multiplied data. In response to this, the right-hand environment should send a token to the left-hand environment to indicate that the initialization process is complete. However, because the design on the right-hand

side expects more tokens than are being provided by the design on the left, the system will deadlock. The deadlock could have been avoided by setting the registers to have the same value and therefore is data dependent. This type of deadlock is known as an architectural deadlock.

6.2.2 Bubble starvation

The previous example illustrated the case where there are no more data tokens in the system and therefore no more processing is done. This token starvation corresponds to the leftmost point in the throughput versus token number graph presented in Chapter 4. At this point the performance of the system is 0. There is another point on such a graph for which the performance is 0, the rightmost point of operation where there are too many tokens and not enough space or bubbles for the tokens to move.

Consider the design in Figure 6.5, which consists of four stages connected as a loop. The environment will insert a number of tokens into the system using a merge unit. The environment provides the ctrl channel to the merge. Once the environment has finished inserting tokens, it will set the merge unit to accept tokens from stage S3 and send them to S1. However, if the environment inserts more tokens than the system can handle and, therefore, there are no places to which the tokens can move then the system will deadlock and will be incapable of processing any data.

Let us reconsider the example in Figure 6.4. This time we will assume that the register on the left, Constant_A, is set to 200 and the register on the right, Constant_B, is set to 100. In this case, the left-hand part of the design will try to send more tokens than the right-hand part expects. The tokens will not be able to pass the multiplier. They will back up all the way to CTRL_A and the counter, causing the system to deadlock.

It is also possible to design systems that do not depend on any programmable data yet can result in such a deadlock. These cases can still be considered as examples of deadlock caused by incorrect design, however. Such a case is illustrated in Figure 6.6. Here there are four stages in the loop and all are capable of inserting a token immediately after reset. The loop is illustrated before and after reset.

6.3 Deadlock caused by arbitration

Sometimes the use of arbiters is unavoidable in a design. While they give the benefit of reducing control circuitry and increasing parallelism, they typically introduce non-determinism into the system that can cause deadlock. We will explore how deadlock due to arbitration can occur, using two examples.

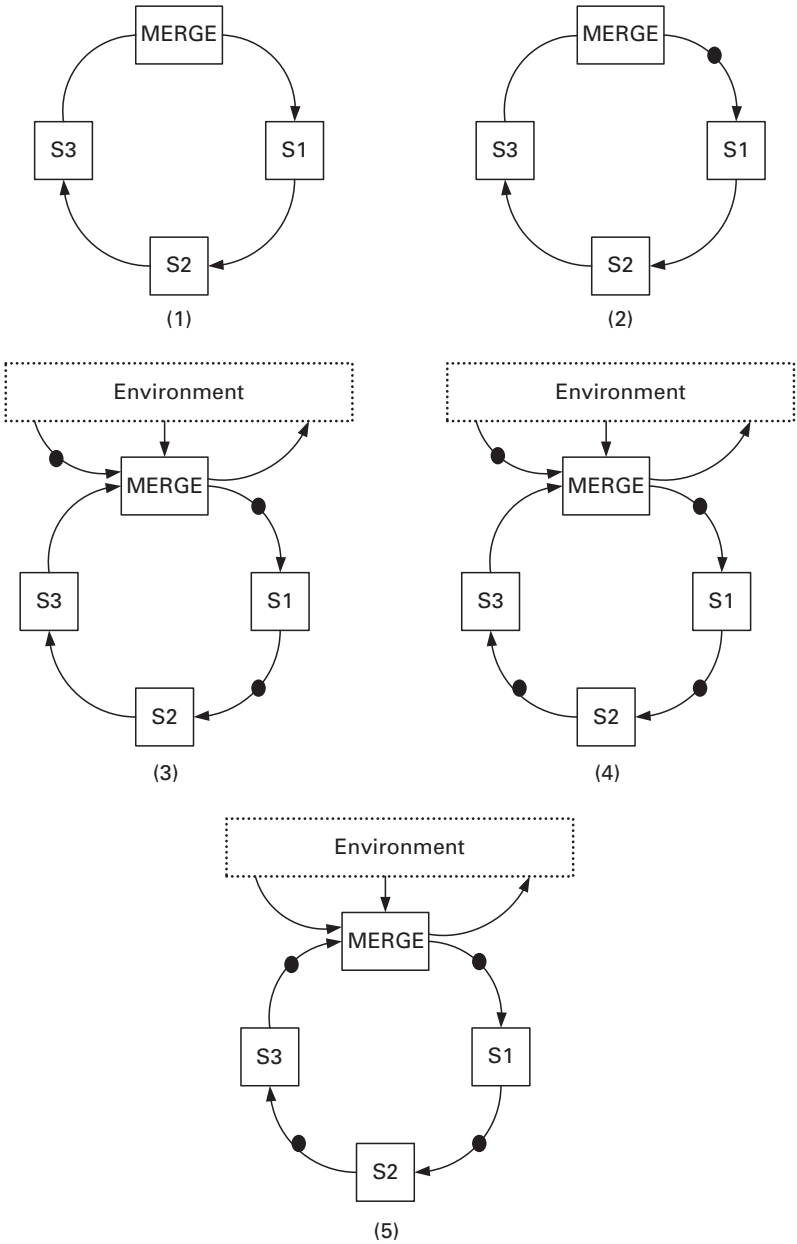

Figure 6.5. Deadlock due to bubble starvation.

6.3.1 Arbiter deadlock Example 1

Consider the basic 2 × 2 crossbar presented in Chapter 2. Assume that in this arbiter the *_Data and *_ToAddr channels, where the asterisks represent S0 or S1, are independent of each other and that the tokens illustrated in Figure 6.7 arrive at

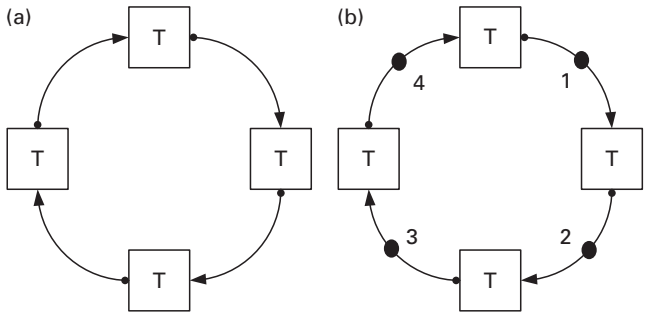

Figure 6.6. Static deadlock of four token buffers T due to bubble starvation, (a) before and (b) after reset.

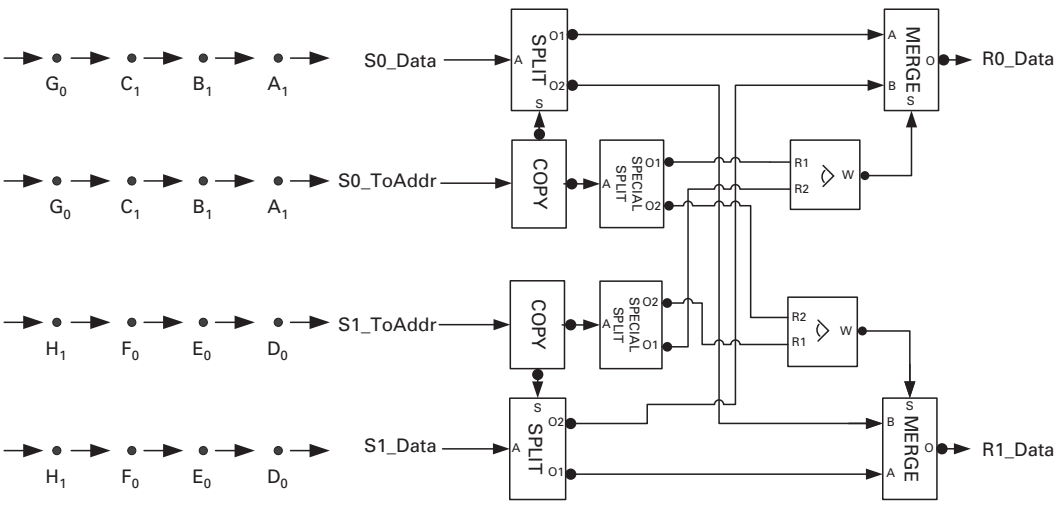

Figure 6.7. Tokens (solid circles) arriving at the inputs of a crossbar.

the inputs of the crossbar. In this case, the tokens A_1, B_1, C_1 should be sent from S0 to R1, the tokens G_0 from S0 to R0, the tokens D_0, E_0, F_0 from S1 to R0, and the tokens H_1 from S1 to R1. The subscripts 0 and 1 indicate the destination ports. We assume that all tokens are sent back to back and as fast as possible.

Because of the unpredictable delays of the internal wires of the crossbar, the arbiter of output R1 may see token H_1 arriving before token A_1 and so grant token H_1 first. Similarly, the arbiter of output R0 may see token G_0 before D_0 and grant token G_0 first. These events may happen even though A_1 was sent before H_1 and D_0 was sent before G_0.

The merge block of output R1 is waiting for data token H_1 and the merge block of output R0 is waiting for data token G_0. Until these data tokens are consumed, all the other tokens, waiting on the other input of this block, will stall. However, because S0 sent all its data tokens (A_1, B_1, and C_1) even before they were granted access to output R1, now S0 cannot consume G_0. The output of the split block is

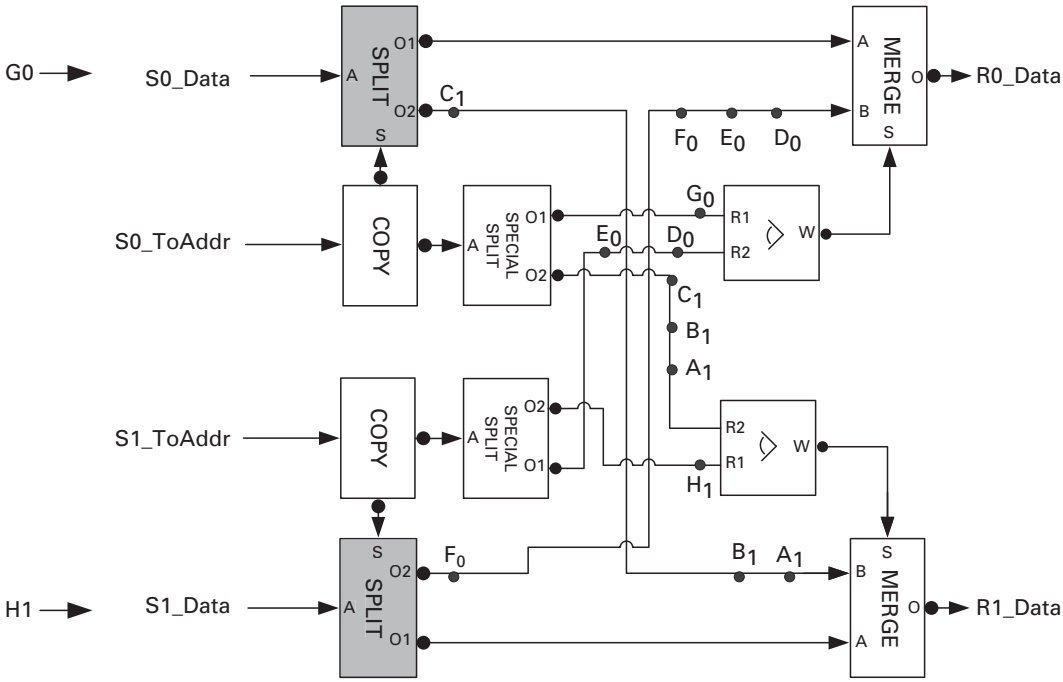

Figure 6.8. Crossbar deadlocked due to lack of slack.

stalled because of the data token C_1 sitting on the 02 output channel. Similarly, the split block of input S1 cannot consume the data token on its inputs, H_1, because the pipeline between its output and the input of the R0 merge block is full of data tokens. As illustrated in Figure 6.8, the data tokens that need to be consumed by the output merge blocks cannot be consumed because there is not enough slack in the pipeline and therefore the system deadlocks. Note that the buffers on the channels to provide slack to store multiple tokens are not shown explicitly.

6.3.2 Arbiter deadlock Example 2

Assume that, for the same crossbar as above, the *_Data and *_ToAddr channels are independent of each other and that there are two requests arriving back to back on the S0 input channel, one requesting output R0 and the other requesting output R1, and only one request arriving on the S1 input channel, requesting output R0 (see Figure 6.9). Notice that while the data for input S1 is arriving at the inputs, none of the data tokens for input S0 is present. This can happen if the latency on this channel is longer than on others, or it may be a design requirement.

Assuming that there is enough slack in the design, it is possible that input S0 can win both arbitrations, since it can issue them one after another even if its data has not been sent to the first winning output or has not arrived. In this case input S0 will monopolize both inputs and hold access to them, even though its data is absent. This is illustrated in Figure 6.10.

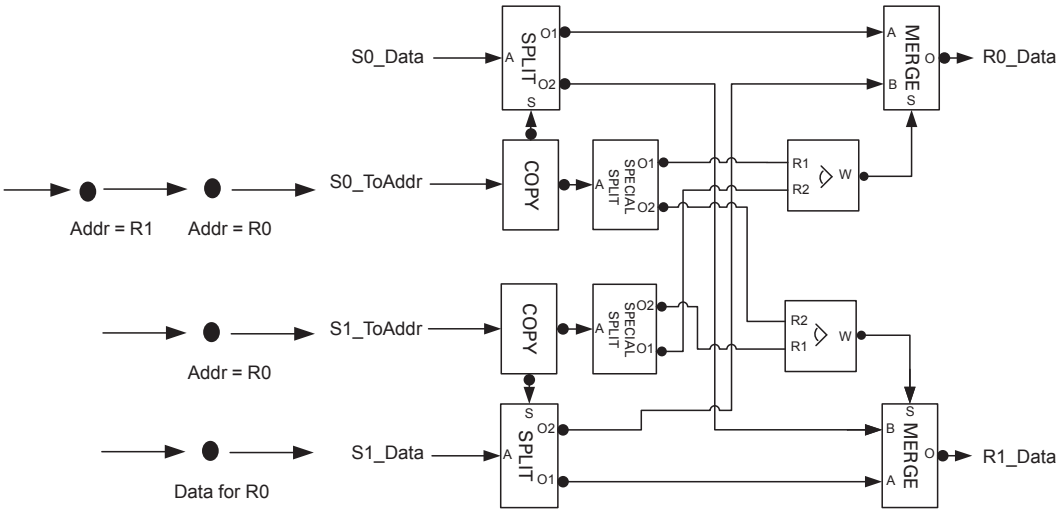

Figure 6.9. A 2 × 2 arbiter with requests on both inputs.

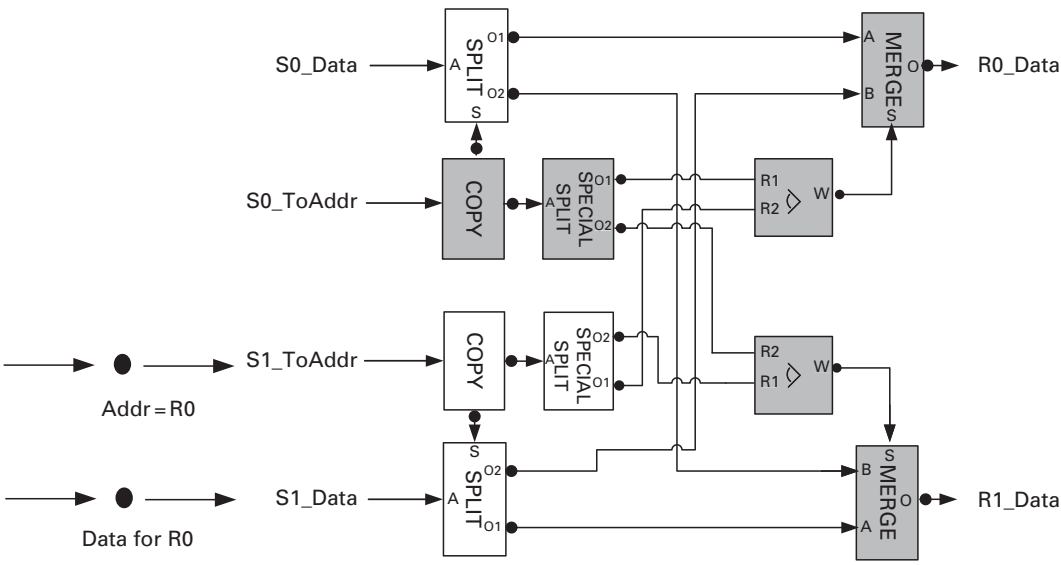

Figure 6.10. Input S0 wins both arbitrations.

Even though S1 has both address and data tokens on its inputs it cannot send to either output since they are both allocated for the use of input S0. Ideally the data tokens (two needed for each output) will arrive some time later, allowing S0 to release the outputs, and then S1 can win the arbitration and send its data to output R0. However, depending on the system in which this crossbar is used, it is possible that the arrival of the data tokens at input S0 actually depends on the data input at

S1 being sent to output R0 first. In this case the output will be used by the system that instantiates this crossbar to generate the data for S0, which corresponds to the address tokens that have already arrived. This system will deadlock because S0 has pre-emptively or prematurely requested resources, even though it was not ready to use them, and this request was granted by the arbiter. However, now it cannot release them because its data token will never arrive since it depends on S1 sending its data. This is a typical case of the incorrect use of this particular arbiter, which is too flexible for this system. It is possible to make either the crossbar stricter or the system that uses it cleverer in order to avoid this deadlock; for example, one can ensure that the data is not requested until both the address and the data are ready.

Reference

[1] C. L. Seitz, "System timing", in *Introduction to VLSI Systems*, C. A. Mead and L. A. Conway, eds., Addison-Wesley, 1980.

7 A taxonomy of design styles

In addition to the various types of handshaking protocol discussed in Chapter 2, there is a variety of other means to characterize asynchronous design styles, which provide insight into the relative advantages and challenges associated with each design style. In this chapter we discuss several of these forms of taxonomy.

7.1 Delay models

The delay model dictates the assumptions made about delays in the gates and wires during the design process. Generally speaking the less restrictive the delay assumptions, the more robust is the design to delay variations caused by a variety of factors such as manufacturing process variations, unpredictable wire lengths, and crosstalk noise. This robustness, however, often comes at a cost such as larger area, lower performance, and/or higher power.

7.1.1 Delay-insensitive design

Delay-insensitive (DI) design is the most robust of all asynchronous circuit delay models [44]. It makes no assumptions on the delay of wires or gates, i.e. they can have zero to infinity delay and can be time varying. Delay-insensitive circuits are thus very robust to all forms of variations.

Although this model is very robust, Martin showed that no practical single-output gate-level delay-insensitive circuits are possible [2]. This result has profound significance. First, to build delay-insensitive circuits the smallest building block must be larger than a single-output gate. As one example, a network of leaf cells that communicate with delay-insensitive channels (e.g. 1-of-N channels) is a delay-insensitive circuit if we consider each leaf cell to be an atomic multi-output gate. This model is realistic if each leaf cell is a library cell that is physically designed as a single unit, because the delays within the cell are then small or at least well verified to be hazard-free under all environmental conditions including slewed input rates, output loading, and cross-talk due to over-the-cell routing. Second, Martin's result motivates the need for less restrictive delay models appropriate for gate-level design, such as those described below.

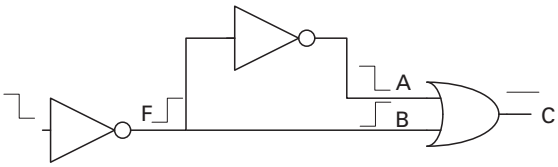

Figure 7.1. Illustration of the isochronic-fork assumption.

7.1.2 Quasi-delay-insensitive design

Martin proposed a compromise to delay insensitivity for gate-level design called quasi-delay-insensitive (QDI) design [3]. In a QDI circuit, as in a DI circuit, all gates and wires can have arbitrary delays, except for a set of designated wire forks labelled as *isochronic*. Isochronic forks, as the name suggests, have the additional constraint that the delay to the different ends of the fork must be the same.

This strict and unrealistic definition is often interpreted to mean that the difference in the times at which the signal arrives at the ends of an isochronic fork must be less than the minimum gate delay. In fact, the underlying timing assumption related to an isochronic fork is often much less restrictive. The purpose of the isochronic-fork assumption is to ensure that the ordering of transitions at the various inputs of a gate, necessary to preserve hazard freedom, can be guaranteed. Consider the reconvergent circuit depicted in Figure 7.1. If the fork F is isochronic, it is guaranteed that the rising transition at B arrives before the falling transition at A. This means that there is no glitch at C. If the fork were not isochronic then a glitch at C could occur and the circuit would be hazardous. In other words, the isochronic-fork assumption can be relaxed to mean that the delay from one end of the fork to its terminal gate G must be less than the delay of any reconvergent fanout path through any other end of the fork that also terminates at G.

The isochronic-fork concept has also been extended in a different way to consider isochronic timing through a number of logic gates [4]. As in the isochronic-fork case, the practical timing assumption often relates to the relative delays of reconvergent paths but, unlike the isochronic-fork case, the shorter path may now include gates.

Another interesting feature of a QDI design is that, practically speaking, primary inputs to the design should be unordered. The reason is that, even if the specification indicates they are ordered, because of the unbounded wire delays to the gates that they drive the ordering is not guaranteed at these gates.

7.1.3 Speed-independent design

In the speed-independent (SI) delay model, all gate delays can be arbitrarily large but all wire delays are taken to be negligible [13]. This means that all forks are isochronic and, unlike a QDI circuit, the transitions of primary inputs to the circuit

can be ordered. Consequently, the specifications for speed-independent circuits may include an ordering of inputs that makes little sense for a QDI circuit.

In today's processes, wire delays can represent a large fraction of the total delay; thus this delay model is very difficult to guarantee for large circuits. Like a QDI circuit, however, the underlying delay assumptions in an SI circuit necessary for correct operation often reduce to the relative delays of reconvergent fanouts, which are much more likely to be satisfied. Thus, the SI model may be a practical model from which to do synthesis as long as a more detailed analysis of the underlying timing assumptions is used to verify correctness after physical design. In particular, methods to synthesis SI control circuits are well covered in the text book by Myers [1] and will be briefly reviewed in Chapter 8.

7.1.4 Scalable delay-insensitive design

In the scalable delay-insensitive (SDI) model, a large chip is divided into many small blocks that communicate using the delay-insensitive (DI) model. Similarly to the QDI model, the SDI model is an unbounded delay model, i.e. no upper bound is assumed on the gate and wire delays. However, unlike the DI or QDI models, in the SDI model it is assumed that the ratio of the delays between any two components (gates and wires) is specifically bounded [5][6].

7.1.5 Bounded-delay design

In the bounded-delay model, values are assumed for the minimum and maximum bounded delays of all gates in the circuit. The circuit is guaranteed to work correctly if these bounds are satisfied. Such timed circuits can have the advantages of being faster, smaller, and lower in power than their more robust counterparts, but their design and synthesis procedures are generally more complex. In addition, extensive post-physical-design analysis is needed to ensure that all bounds are met. In contrast with QDI and SI, the underlying timing assumptions are explicit in the form of delay bounds rather than implicit in the form of the relative delays of reconvergent fanout paths.

7.2 Timing constraints

As we have already discussed, the delay model used during designs has a direct influence in the timing assumptions necessary to ensure correctness. The more relaxed these timing constraints are, the more likely it is that the design will work or can be modified to work. We will describe three categories of timing constraint. The first is the *one-sided* timing constraint, which states that the delay of a gate, wire, or path must be less than or greater than some fixed value. The second is the *two-sided* timing constraint, which states that the delay of a specified gate, wire, or path must be smaller than some value Δ and larger than another value δ.

Two-sided timing constraints are in general less attractive than one-sided timing constraints because it is possible that the constraint will not be satisfied in the case where δ is larger than Δ. In contrast, one-sided timing constraints can usually be satisfied by adding delays to the design, typically at the cost of some area, power, and/or performance.

The third type of common timing constraint is referred to as *relative timing* and was introduced by Stevens *et al*. [8]. It constrains the relative delay of two paths in a circuit, such as the reconvergent paths shown in Figure 7.1. The essential aspect of this constraint is that the bounds on the paths are not fixed numbers and consequently correlation between path delays is an integral feature of these constraints. This form of constraint is also well suited for highlighting the underlying constraints associated with a design that, typically, relates to a necessary ordering among the inputs of a gate. Relative timing constraints can be used to guide synthesis [9], can be automatically generated from circuits [7], and may be a good basis for post-layout verification.

7.3 Input–output mode versus fundamental mode

Another fundamental difference between forms of asynchronous circuits is in the assumptions placed on the environment. In one form of design, the environment is assumed to satisfy the *fundamental-mode assumption*, which states that the circuit is allowed to stabilize before a new set of inputs is applied to the circuit. This assumption simplifies design and is typically easy to satisfy, particularly if the environment is slow-changing. However, the assumption must be verified through post-layout analysis and adjusted as necessary. This assumption is integral to the burst-mode control-synthesis algorithms to be discussed in Chapter 8, where its significance will be assessed in more detail.

Alternatively, asynchronous circuits may operate in *input–output mode*. In this case, an input transition is causally associated with other transitions on inputs and outputs. In other words, an input transition can occur as soon as other specified input and/or output transitions occur. As we described in the context of speed-independent design, input transitions may be ordered among themselves and can also be concurrent with expected output transitions. This is a more explicit means of defining when input and/or output transitions can occur and leads to very different design and synthesis approaches. In particular, speed-independent circuits are typically designed assuming this input–output mode, as will be described in Chapter 8.

7.4 Logic styles

Another significant dimension to asynchronous design is in the style of logic used. As mentioned in Chapter 1, the two most common logic styles are static and

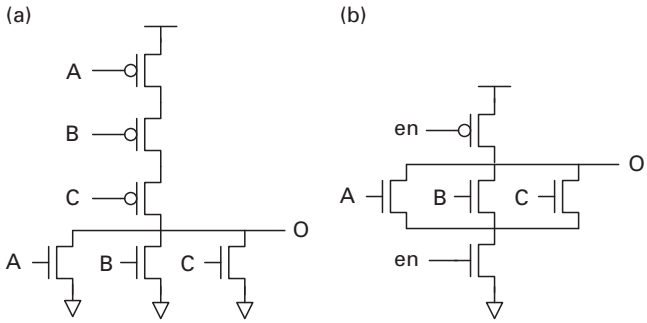

Figure 7.2. Three-input NOR gate implementations: (a) static logic, (b) dynamic logic.

dynamic. We provide a brief introduction to these styles in this section and refer the reader to classic VLSI textbooks for a more complete review [10][11].

7.4.1 Static logic

Static logic units comprise complementary networks of N- and P-transistors. The N-network ties the output to ground and the P-network ties the output to the drain supply V_{dd}. The networks are designed so that for every input combination there is either an electrical path from the output node to ground or to V_{dd}. Moreover, it guarantees that that no input combination yields an electrical path between ground and V_{dd}, which would cause static short-circuit current. Consider, for example, the static three-input NOR gate shown in Figure 7.2(a). The output O is connected to V_{dd} when all inputs are 0 and to ground otherwise.

Static logic is robust to noise, because the switching threshold for the gates is typically around $V_{dd}/2$, and they can recover from transient glitches on the input or output nodes. Moreover, while the sizes of the individual transistors may impact performance, they will not impact correct functionality.

7.4.2 Dynamic logic

Dynamic logic can have non-complementary N and P networks and may rely on the output capacitance to hold the output value temporarily. Consider the dynamic logic implementation of the three-input NOR gate in Figure 7.2(b). In this implementation the N and P transistors connected to the enable (en) signal control the precharge and evaluate phases of the gate. When en is high and all inputs are low, there is no path from the output to V_{dd} or ground and the output value is held solely by the output capacitance to ground. The intention is that after the output is precharged high by en going low, en will go high and the output will only discharge if at least one of the three inputs is high, thereby implementing the three-input NOR function.

Most forms of dynamic logic rely on the output being precharged high and then conditionally discharged. For example, in the dynamic NOR gate it is expected that the evaluation cycle, i.e. the period for which en is high, is sufficiently small that the output capacitance on O is large enough to prevent the output high signal going low via a leakage current through the N-stack. Notice also that a dynamic gate is more sensitive to noise than its static logic counterpart because glitches on the input or output of the dynamic gate may accidentally discharge the output, which will not recover its charge. As we will see later, dynamic logic can be made *pseudo-static* by adding a small *staticizer* to the output, which is designed to prevent leakage current and hold the state.

The principal advantage of dynamic logic is performance. If the precharge cycle can be hidden from the critical path, the discharge path uses only N-transistors. Because N-transistors have a significantly higher mobility factor than P-transistors they can drive more current for the same transistor width. Consequently, by removing the P-transistors from the critical path the latter has less gate capacitance, enabling a 30%–50% increase in performance [10]. Moreover, removing P-transistors can sometimes lead to area savings.

7.4.3 Muller C-element implementations

To illustrate further the differences between static and dynamic logic, we will use different implementations of the Muller C-element, a common memory element found in many asynchronous control circuits [13]. It has the next-state logic function

$$C' = AB + AC + BC$$

where A and B are inputs and C is the output. Thus, when both inputs A and B go high, the output C goes high. It then stays high until A and B both go low, at which time it goes low and the process repeats. Illustrations of the abstract C-element and a pseudo-static logic implementation are shown in Figures 7.3(a),(b). Two static logic implementations, one in the form of a complex gate and one using gates typically found in standard cell libraries, are shown in Figures 7.3(c),(d).

Notice how, in the pseudo-static implementation of the C-element, when A and B do not have the same value there is no direct path from the output to ground or to V_{dd} and the output state is held via the staticizer in the form of a weak feedback inverter. The latter is typically designed to be between one-fifth and one-tenth the strength of the driving logic (the main N- and P-stacks) because it needs to be sufficiently weak to be easily overcome by the driving logic when the output C switches values. If the size of the staticizer is too small then the value on the internal node may leak away and the state of the C-element may be lost. If the size of the staticizer is too large then the C-element may not switch values when required. Consequently, unlike in static logic, the relative size of transistors is important in ensuring correctness. This also means that the functional correctness

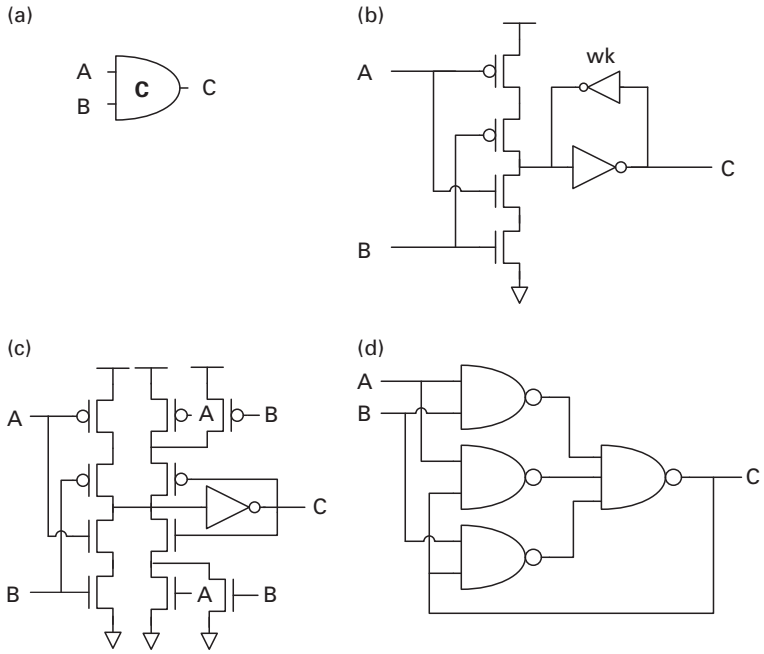

Figure 7.3. An abstract C-element and its various descriptions: (a) abstract symbol, (b) pseudo-static implementation, (c) static complex-gate implementation, (d) static standard-cell implementation.

of dynamic implementations is sensitive to the process variations that often impact transistor strengths.

Another concern with the dynamic implementation is charge sharing. Charge sharing occurs when transistors turn on in the N- or P-network but it is not intended that the output should switch values. In particular, the charge on the output node must equalize with all nodes electrically connected to it and thus the output node may change from its nominal V_{dd} or ground value. If the change is too great, the output node may inadvertently switch. Again, careful transistor sizing and post-layout simulation and verification is necessary to ensure that the degree of charge sharing is manageable. During this analysis it is important to account for or leave margins for other noise events that may simultaneously make the output node switch [10].

The complex-gate static implementation shown in Figure 7.3(c) avoids the charge-sharing problems arising with dynamic logic and is somewhat compact. The standard-cell alternative is substantially larger but has the advantage of using readily available library cells. The standard-cell implementation also has an implicit fundamental-mode timing assumption: the feedback path must stabilize before subsequent input transitions can occur. This timing assumption is generally easily satisfied, but care must be taken if it cannot be ensured that the feedback delay will be relatively small.

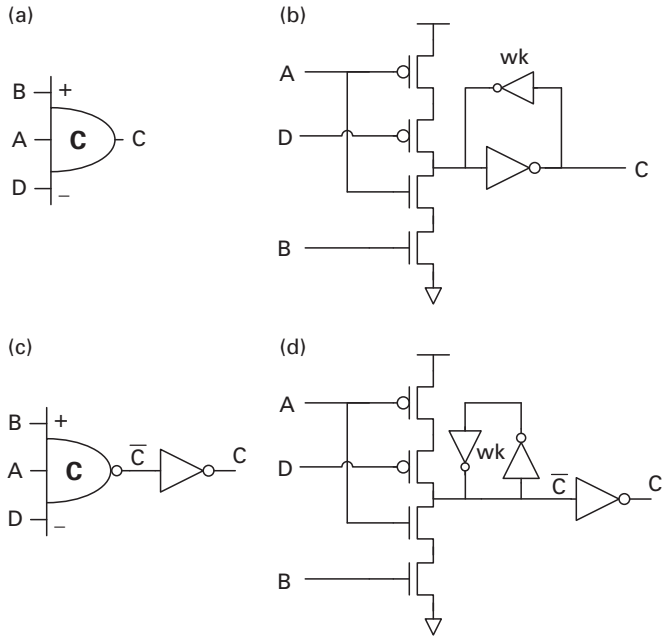

Figure 7.4. Asymmetric C-element gate notation and pseudo-static implementations.

7.4.4 Asymmetric C-elements

Another common form of an asynchronous latch is the asymmetric C-element, in which some inputs are involved only in the rising or falling transition, but not both. In this book we indicate asymmetric C-elements as in Figure 7.4, in which signals required for a positive transition of the output are labeled with a plus and signals required for a negative transition are labeled with a minus (this is similar to what was proposed in [17]). In addition, inverted inputs and outputs are denoted as usual with a small empty circle at the associated ports. In particular, in a pseudo-static implementation of an inverted C-element, the output can be taken directly after the dynamic logic and the output inverter becomes part of the staticizer, as illustrated in Figure 7.4(d). Extending the asymmetric input convention to inverted C-elements, as illustrated in Figure 7.4(c), a plus (minus) on an input port means that an input is required for the inverted output to fall (rise).

7.5 Datapath design

Datapath design in asynchronous circuits is different from that in synchronous circuits, in that often it must be accompanied with circuitry that indicates when the computation is complete and the output is valid. The methods to accomplish this vary and provide tradeoffs in robustness, area, power, and performance.

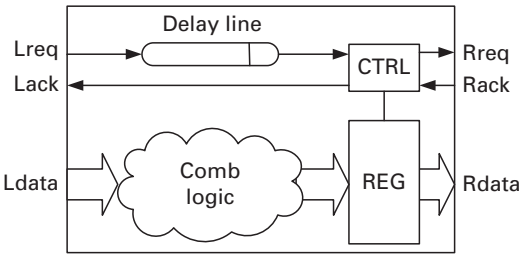

Figure 7.5. A bundled-data pipeline stage.

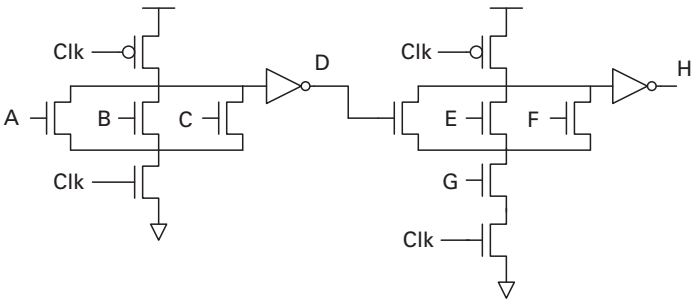

Figure 7.6. Example of two stages of domino logic.

7.5.1 Bundled data

The simplest form of datapath design uses standard static logic as in synchronous ASIC design. This has the advantage that standard libraries and synthesis tools that can automatically generate the datapath logic can be employed. The logic is accompanied by a delay line triggered when the input data is valid, the output of which signals that the computation is complete and that the output of the logic is valid and should be stored, as illustrated in Figure 7.5. Various forms of delay line and storage element have been developed and these will be explored in Chapter 9.

In some high-performance full-custom synchronous designs, the datapath is designed using domino logic. Domino logic comprises alternating stages of dynamic and static logic gates. It has the benefit that data input to the dynamic gates has 0-to-1 transitions during the evaluation mode. As early stages of the logic evaluate, this triggers the evaluation of later stages. In the example illustrated in Figure 7.6, when Clk goes high, the domino logic enters the evaluation mode. If A is 1 then the first dynamic stage will discharge, causing D to rise. This in turn will cause the second stage to discharge, if G is also 1, thereby causing H to rise. This successive discharging can be seen as dominoes falling in a cascading fashion. Notice that in Figure 7.6 the static gate is simply an inverter; however, more complex static gates are also possible as long as they are still inverting. This is essential to ensure that the inputs to the domino logic satisfy the *monotonicity*

requirement, according to which inputs can only make transitions from 0 to 1 during the evaluation period. Violations of this requirement, transitions from 1 to 0 on inputs, would cause an accidental discharge of the domino logic [10].

When domino logic is applied to asynchronous design, the Clk signal is replaced with a local asynchronous control signal and staticizers are sometimes added to internal nodes to avoid the need for explicit latches or flip-flops. Specific templates that use domino logic will be discussed in Chapters 11 through 13.

7.5.2 Quasi-delay-insensitive design

The most robust form of datapath design uses delay-insensitive or quasi-delay-insensitive techniques. This involves designing the datapath using dual-rail or 1-of-*N* encoding and building completion sensing to establish when the outputs are valid. The 1-of-*N* logic can be implemented statically or dynamically using domino or other dynamic logic families. The key difference from synchronous design is the requirement that the inputs and outputs are in the form of 1-of-*N* channels and that *completion-sensing circuitry* is used to determine the validity of the channels. A 1-of-*N* channel can be completed by a simple ORing of the channels. The OR gate goes high when one channel goes high, indicating validity, and stays high until that channel is lowered, indicating neutrality. Determining when a bundle of channels is valid or neutral can be done by combining the OR gate output's with a tree of C-elements. Figure 7.7 illustrates a simple completion-sensing circuit for three channels: A and B are both dual-rail 1-of-2 channels and C is a 1-of-3 channel. Of course, bubble shuffling can be used to map the design to typical library cells, take inverted channel inputs, or simply optimize the design.

A typical problem with completion-sensing logic is that it adds significant amounts of delay to the design that can overshadow the performance advantages of the 1-of-*N* logic. For example, for a 32-bit datapath consisting of 32 dual-rail channels, the C-element tree would be responsible for merging 32 two-input OR gate outputs, which requires five levels of two-input C-elements. Fortunately, the completion-sensing delay can be largely hidden from the forward latency of a series of computations if the next computations are performed in parallel with the completion sensing. In particular, if the control logic ensures that the next block in the series of computations is in evaluation mode, so that it can evaluate as soon as it receives valid data from the previous block output, the only overhead in latency is the extra side load associated with the completion-sensing logic. In fact, this is a common theme of many QDI and timed pipeline templates, which will be discussed in later chapters, and consequently many templates achieve close to pure domino logic latency.

That said, a large completion-sensing delay can limit throughput. One means of mitigating this throughput limitation is to build blocks that operate on only a small number of bits by decomposing wide datapaths into communicating parallel blocks. This means that, in addition to pipelining the datapath via its function (i.e. vertically), the datapath is also pipelined via its bitwidth (i.e. horizontally),

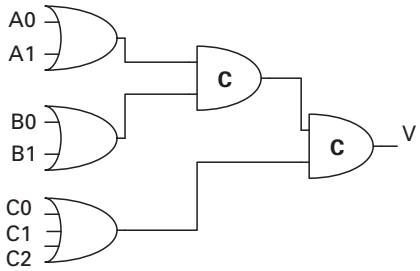

Figure 7.7. A simple completion sensing circuit for three channels.

forming a *two-dimensional pipeline* of asynchronous blocks. This technique is also common in many of the design templates that we will see in later chapters.

7.5.3 Hybrid techniques

It is also possible to have a hybrid approach that combines the completion-sensing and timing assumptions associated with the delay lines or, more generally, the control delay. A good example of this is the adders presented by Yun *et al.* [17]. The carry logic is implemented in dual-rail domino logic and has completion-sensing circuitry that identifies whether the carry signals are valid. The sum logic, however, is implemented by normal domino logic, and there is a timing assumption that ensures that the sum logic is valid before the completion-sensing circuitry triggers a "done" signal. Yun *et al.* also proposed to hide the completion-sensing delay in the delay due to static routing of the output signals to the next functional unit [17]. These techniques require a post-layout verification that valid data arrives at its destination only after the associated control signal has indicated its validity.

In addition, Yun *et al.* proposed to precharge the completion-sensing unit. In this way the unit checks validity, but checking neutrality is not explicitly performed and reliance is placed on a safe timing assumption. With removal of P-transistors, the completion-sensing logic turns into a precharged OR–AND circuit. Consequently the falling transition of the valid signal, V, can be relatively quick, reducing cycle times and thereby improving performance. An optimized transistor-level design of a precharged completion-sensing circuit for the carry outputs of a 32-bit adder from [17] is illustrated in Figure 7.8. This type of timing assumption has also been called *implied neutrality* [45].

7.5.4 Indictability

Another means of describing datapath logic is based on the concept of *indictability* – the relationship between valid outputs and valid inputs. There are three possibilities. First, if when the outputs go valid (neutral) this indicates that all inputs have become valid (neutral) then the function block is said to be

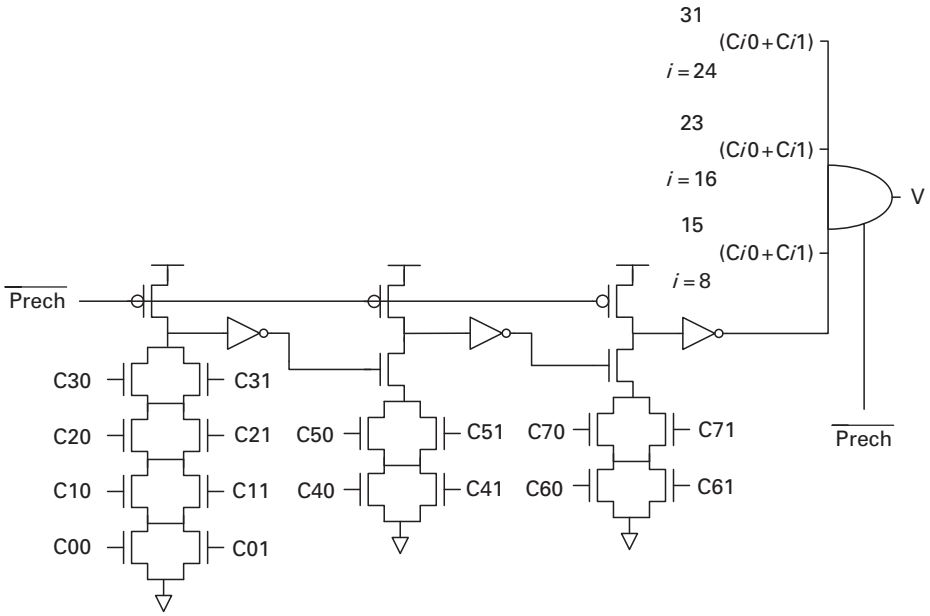

Figure 7.8. Precharged completion sensing unit.

weak-conditioned or strongly indicating [12][14][16]. Second, if when the outputs go valid this indicates that all inputs are valid but that the outputs can go neutral before all the inputs go neutral then the function block is said to be semi-weak-conditioned [15]. Third, if when the outputs go valid (neutral) this indicates that only some, but not necessarily all, inputs have become valid (neutral) then the function block is said to be weakly indicating [12][14] or not weak-conditioned [16].

As an example, consider three transistor-level implementations of a two-input dual-rail OR gate, as shown in Figure 7.9. The implementation in Figure 7.9(a) is weak-conditioned because (i) the output cannot go valid until both dual-rail input channels have become valid and (ii) the output cannot go neutral until both dual-rail inputs channels have gone neutral. Condition (i) is guaranteed because every path to ground through the N-network goes through an N-transistor connected to either the 0 or the 1 rail of both input channels. Condition (ii) is guaranteed by the stack of four P-transistors. The domino-logic implementation in Figure 7.9(b) is not weak-conditioned, because the P-network has been replaced with a single precharge transistor. Consequently, if the "enable" input is lowered before all other inputs become neutral, the output will immediately become neutral. In fact this implementation is semi-weak-conditioned, because the N-network has not changed and consequently the output cannot become valid until all inputs (including the enable) have become valid. The last implementation, in Figure 7.9(c), is more efficient in terms of the transistor count but is only weakly indicating,

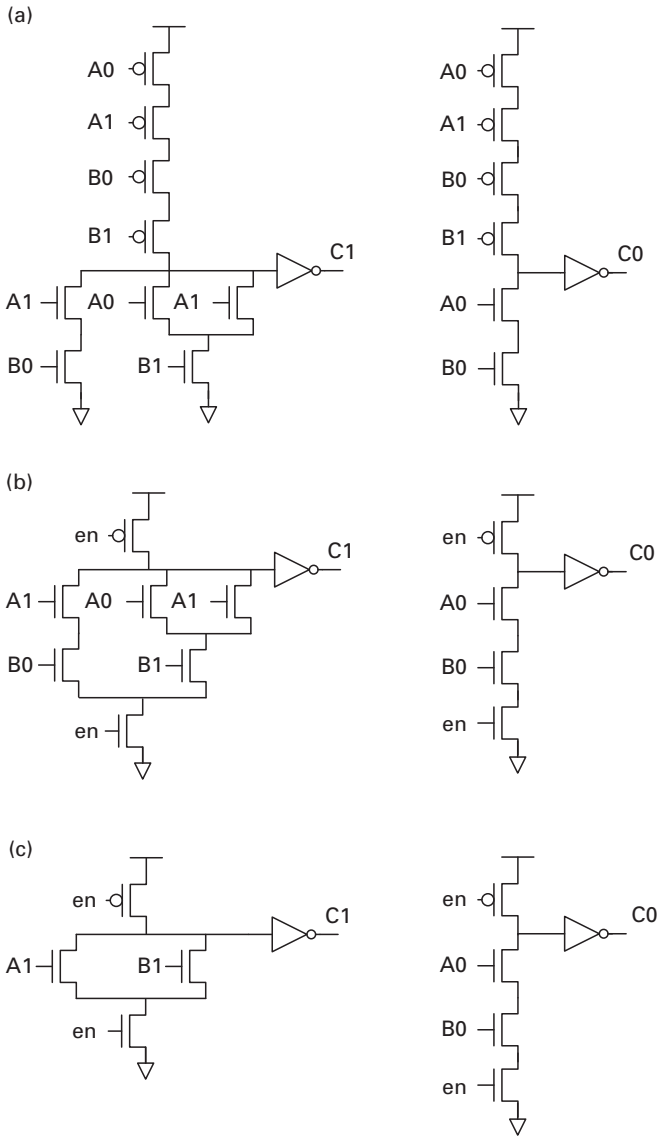

Figure 7.9. Three implementations of a two-input dual-rail OR gate: (a) weak-conditioned, (b) semi-weak-conditioned, (c) not weak-conditioned. The enable inputs are indicated by en.

because if one input channel becomes valid and the enable signal is high then the domino logic will evaluate without waiting for the second input channel to become valid.

While we have illustrated indictability for a single-level dual-rail function block, the concept extends naturally to multi-level function blocks with no cycles and no dangling internal nets. Moreover, it has been shown that if every gate is strongly (weakly) indicating then the multi-level function network is also strongly (weakly)

indicating [12]. We believe that the result extends to semi-weak-conditioned logic but this has not been formally proven yet.

The indictability of the logic has an important consequence in the design of the control circuits surrounding the function blocks that generate acknowledgement signals. In general, it is necessary that an input channel is not acknowledged until valid data is received on it. Consequently, in semi- or weakly indicating function blocks, additional completion sensing on the *input* channels (in addition to the output channels) is sometimes needed to ensure that the input channels are valid before sending their associated acknowledgement. The role of the input completion-sensing block will be discussed in more detail in the QDI templates described in Chapter 11.

7.6 Design flows: an overview of approaches

A variety of different design flows have been proposed for asynchronous circuits, and this section describes the more popular approaches.

7.6.1 Communicating sequential process language refinement

Martin and coworkers pioneered a correct-by-construction approach for the design of quasi-delay-insensitive circuits. The process begins with a sequential specification in communicating hardware processes [3], a variant of the CSP [19] that we described in Chapter 3.

This sequential specification is decomposed into concurrent processes communicating via asynchronous channels. The decomposition introduces both parallelism and pipelining. Because the target is a high-speed QDI design, the decomposition often introduces two-dimensional pipelining, i.e. pipelining is added both vertically and horizontally.

Once a leaf-cell communicating sequential process (CSP) is achieved, the next step is to expand the handshaking on the channels and synthesize the control circuits into gates and transistors. Many different templates of leaf cells and handshaking expansions have been used in Caltech designs; Martin and Burns developed a synthesis procedure to generate QDI designs from any possible handshake interleaving [24]. Lines analyzed these expansions and concluded that only a few templates are practically useful [16]; all these studies focused on 1-of-*N* delay-insensitive channels. We will discuss template designs in Chapter 11. Beerel and coworkers adopted this decomposition procedure for other types of channel and timed template, including single-track templates, which will be discussed in Chapter 13.

Efforts to automate CSP decomposition have been made by Wong and Martin [30][31] and are called *data-driven decomposition*. In this approach, a sequential CSP process is decomposed into a set of smaller concurrent CSP processes on the basis of the data dependencies and a projection technique proposed in [31].

Similarly, Teifel and Manohar demonstrated data-driven decomposition that efficiently maps CSP processes into predefined pipelined circuit blocks implemented in a field-programmable gate array (FPGA) [29].

7.6.2 Syntax-driven translation

In a similar way, Philips developed a synthesis tool using the CSP-like programming language Tangram [21][20]. A behavioral description expressed by Tangram is translated using syntax directed translation into an intermediate form based on *handshake circuits*, which represent abstract asynchronous blocks. The key difference from the data-driven decomposition in Martin's decomposition-based approach is that the ";" in CSP is translated explicitly as sequencing even if the implicit data dependencies allow more parallelism. This provides a level of transparency between the specification and final implementation but makes achieving high-performance designs more challenging.

Handshake circuits can map to multiple design back-end implementations, i.e. QDI and/or bundled-data [22][23]. Additionally, peephole optimizations have been proposed to further improve the area and speed of a design. Bardsley and Edwards adopted the syntax-directed translation approach of Tangram and built a new synthesis tool called Balsa [33][34][35], which is freely available for downloading. Additionally, Nowick and coworkers developed controller optimizations targeting a burst-mode oriented back-end implementation for Balsa [37][38].

7.6.3 Gate-level netlist translation

Linder and Harden developed a synthesis approach that uses commercial synthesis tools to generate a netlist and then translates this netlist to the delay-insensitive asynchronous implementation called *phased logic* [39]. Each gate in the netlist is replaced with a dual-rail gate using a special encoding called level-encoded two-phase dual-rail (LEDR) [40]. This encoding scheme is beneficial for reducing power consumption, since fewer transitions are required per data token propagating through a stage. However, it requires additional feedback connections to be inserted in the original netlist to ensure safeness [39].

Theseus Logic introduced a design flow that uses a new logic family called null convention logic (NCL) [27][41]. In this design flow, large designs are synthesized using commercial synthesis tools. Then synchronous registers in the datapath netlist are replaced by asynchronous registers that communicate using a delay-insensitive handshaking protocol. However, this design flow produces pipelined circuits that are architecturally equivalent to the synchronous register transfer-level (RTL) implementation and typically coarse-grained. Thus, large completion-detection circuits slow down the speed of the design. Another key drawback is that the behavioral specification is extended to include the notion of channels to implement handshake mechanisms between datapath and control. This requires

designers to specify manually some handshake signals in the specification, which makes the existing RTL hard to reuse [41].

Another gate-level-translation approach that automates the design of asynchronous circuits from synchronous netlists is *de-synchronization*, proposed in [25][26]. This approach translates synchronous netlists synthesized by commercial synthesis tools to asynchronous implementation by replacing the global clock network with a set of asynchronous controllers. Several new and existing four-phase handshaking protocols for latch controllers are proposed. The benefits of this de-synchronization approach are that it provides a fully automated synthesis flow, does not require any knowledge of asynchronous design by the designer, and does not change the structure of synchronous datapath and controller implementation since it only affects the synchronization network. However, the target architecture of this method is restricted to micropipeline design.

None of the above approaches can support the synthesis of automatic pipelining and yield a performance on the level of the original specification. Smirnov and coworkers proposed a design flow that translates the behavioral specification to QDI pipelined asynchronous circuits by synthesizing a synchronous implementation using commercial tools and translating ("weaving") it into an asynchronous pipelined implementation [28]. The advantage of this approach is that it demonstrates a general framework for the automated synthesis of pipelined asynchronous circuits. However, fine-grained pipelining can eliminate opportunities for resource sharing in low-performance applications and also incurs a high area penalty.

Furthermore, none of the approaches so far discussed includes slack elasticity, implying that further performance optimization may be needed. Moreover, these approaches do not fully address the development of conditional communication and/or gated clocking. Consequently, unnecessary tokens may be generated and transmitted between blocks, which causes unnecessary power consumption and also may limit system performance. Thus, adding conditional communication in this approach is an interesting area of active research.

7.6.4 High-level synthesis-based approaches

Many synthesis procedures utilize commercial synchronous high-level synthesis engines, which translate behavioral hardware description languages to gate-level netlists as the front end of their tools. The back-end tools map and optimize such netlists to asynchronous designs. The key drawback of this approach is that the netlists generated may not be optimally synthesized for the required targets, since current commercial synchronous synthesis tools do not support several key parameters that can profoundly impact such targets. In particular, the tools typically do not understand the notions of global and local cycle time, thus disabling an opportunity to share resources. We briefly describe these approaches. Jacobson and coworkers proposed a synthesis framework called ACK [36]. This approach utilizes existing synchronous synthesis flow and tools for datapath

synthesis. Control circuits are modeled and synthesized using distributed burst-mode specifications [42]. Yoneda and coworkers synthesized a high-level C-like language using SpecC to generate asynchronous timed gate-level circuits, where the control is modeled and synthesized using a signal transition graph (STG) -based approach allowing global timing optimization [43]. Theoretical work was performed by Tugsinavisut *et al*. [46] to explore algorithms for resource sharing in multi-threaded applications.

7.7 Exercises

7.1. Design a two-input AND gate in (a) weak-conditioned, (b) semi-weak-conditioned and (c) non-weak-conditioned logic.
7.2. Is a dual-rail domino block by definition semi-weak-conditioned? Why or why not?
7.3. Design a three-input XOR gate in domino logic and in dual-rail domino logic, using two-input domino and dual-rail domino XOR gates. Compare the numbers of transistors needed in your two implementations.
7.4. Design a three-input AND gate in domino logic and in dual-rail domino logic out of two-input domino and dual-rail domino AND gates. Compare the numbers of transistors needed in your two implementations.
7.5. A function f is *positive unate* if, for any two n-bit inputs $a \leq b, f(a) \leq f(b)$. Here $a \leq b$ means $a_i \leq b_i$ for all n bits. Consider two functions f_1 and f_2, one of which is positive unate and another which is not. What can you say about the transistor count ratios of dual-rail domino and domino implementation of these two functions?

References

[1] C. Myers, *Asynchronous Circuit Design*, John Wiley & Sons, 2001.
[2] A. J. Martin, "The limitations to delay-insensitivity in asynchronous circuits," in *Proc. 6th MIT Conf. on Advanced Research in VLSI* W. J. Dally, ed., MIT Press, 1990, pp. 263–278.
[3] A. J. Martin, "Compiling communicating processes into delay-insensitive VLSI circuits," *Distributed Computing*, vol. 1, no. 4, pp. 226–234, 1986.
[4] K. van Berkel, F. Huberts, and A. Peeters, "Stretching quasi delay insensitivity by means of extended isochronic forks," in Proc. 2nd *Working Conf. on Asynchronous Design Methodologies*, May 1995, pp. 99–106.
[5] T. Nanya, A. Takamura, M. Kuwako, *et al*., "Scalable-delay-insensitive design: a high-performance approach to dependable asynchronous systems," in *Proc. Int. Symp. on the Future of Intellectual Integrated Electronics*, March 1999, pp. 531–540.
[6] T. Nanya, A. Takamura, M. Kuwako, *et al*., "TITAC-2: a 32-bit scalable-delay-insensitive microprocessor," in *Symp. Record of HOT Chips IX*, August 1997, pp. 19–32.

[7] H. Kim, P. A. Beerel, and K. Stevens, "Relative timing based verification of timed circuits and systems," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, April 2002, pp. 115–124.
[8] K. Stevens, S. Rotem, and R. Ginosar, "Relative timing," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, April 1999, pp. 208–218.
[9] K. Stevens, S. Rotem, S. Burns, *et al.*, "CAD directions for high performance asynchronous circuits," in *Proc. ACM/IEEE Design Automation Conf.*, 1999, pp. 116–121.
[10] N. Weste and D. Harris, *CMOS VLSI Design, A Circuits and Systems Perspective (3rd Edn)*, Addison Wesley, 2005.
[11] J. M. Rabaey. *Digital Integrated Circuits – A Design Perspective*, Prentice-Hall, 1996.
[12] C. L. Seitz, "System timing," in *Introduction to VLSI Systems*, C. A. Mead and L. A. Conway, eds, Addison-Wesley, 1980.
[13] R. E. Miller, *Switching Theory, Volume II: Sequential Circuits and Machines*, Wiley, 1965.
[14] J. Sparsø and S. Furber, eds., *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer Academic Publishers, 2001.
[15] M. Ferretti and P. A. Beerel, "Single-track asynchronous pipeline templates using 1-of-*N* encoding," in *Proc. Conf. on Design, Automation and Test in Europe (DATE)*, pp. 1008–1015, March 2002.
[16] A. Lines, *"Pipelined asynchronous circuits,"* Master's thesis, California Institute of Technology, 1995.
[17] K. Y. Yun, P. A. Beerel, V. Vakilotojar, A. E. Dooply, and J. Arceo, "The design and verification of a high-performance low-control-overhead asynchronous differential equation solver," *IEEE Trans. VLSI Systems*, vol. 6, no. 4, pp. 643–655, December 1998.
[18] S. B. Furber and P. Day, "Four-phase micropipeline latch control circuits," *IEEE Trans. on VLSI Systems*, vol. 4, no. 2, pp. 247–253, June 1996.
[19] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.
[20] J. Kessels and A. Peeters, "The Tangram framework: asynchronous circuits for low power," in *Proc. Asia and South Pacific Design Automation Conf.*, February 2001, pp. 255–260.
[21] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij, "The VLSI-programming language Tangram and its translation into handshake circuits," in *Proc. European Conf. on Design Automation (EDAC)*, 1991, pp. 384–389.
[22] K. van Berkel, *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, International Series on Parallel Computation, vol. 5, Cambridge University Press, 1993.
[23] A. M. G. Peeters, "Single-rail handshake circuits," Ph.D. thesis, Eindhoven University of Technology, 1996.
[24] S. M. Burns, "Automated compilation of concurrent programs into self-timed circuits." Master's thesis, California Institute of Technology, 1988.
[25] I. Blunno, J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou,"Handshake protocols for de-synchronization," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, April 2004, pp. 149–158.
[26] J. Cortadella, A. Kondratyev, L. Lavagno and C. Sotiriou, "A concurrent model for de-synchronization," in *Handouts of Int. Workshop on Logic Synthesis*, pp. 294–301, 2003.

[27] M. Ligthart, K. Fant, R. Smith, A. Taubin, and A. Kondratyev, "Asynchronous design using commercial HDL synthesis tools," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, April 2000, pp. 114–125.
[28] A. Smirnov, A. Taubin, and M. Karpovsky, "Automated pipelining in ASIC synthesis methodology: gate transfer level," in *Proc. ACM/IEEE Design Automation Conf.*, June 2004.
[29] J. Teifel and R. Manohar, "Static tokens: using dataflow to automate concurrent pipeline synthesis," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, April 2004, pp. 17–27.
[30] C. G. Wong and A. J. Martin, "Data-driven process decomposition for the synthesis of asynchronous circuits," in *Proc. IEEE Int. Conf. on Electronics, Circuits and Systems*, 2001.
[31] C. G. Wong and A. J. Martin, "High-level synthesis of asynchronous systems by data-driven decomposition," in *Proc. ACM/IEEE Design Automation Conf.*, June 2003, pp. 508–513.
[32] R. M. Badia and J. Cortadella, "High-level synthesis of asynchronous systems: scheduling and process synchronization," in *Proc. European Conf. on Design Automation (EDAC)*, February 1993, pp. 70–74.
[33] A. Bardsley and D. A. Edwards, "The Balsa asynchronous circuit synthesis system," in *Proc. Forum on Design Languages*, September 2000.
[34] A. Bardsley and D. A. Edwards, "Synthesising an asynchronous DMA controller with Balsa," *J. Systems Architecture,* vol. 46, pp. 1309–1319, 2000.
[35] A. Bardsley, "Implementing Balsa handshake circuits." Ph.D. thesis, Department of Computer Science, University of Manchester, 2000.
[36] H. Jacobson, E. Brunvand, G. Gopalakrishnan, and P. Kudva, "High-level asynchronous system design using the ACK framework," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, April 2000, pp. 93–103.
[37] T. Chelcea, A. Bardsley, D. Edwards, and S. M. Nowick, "A burst-mode oriented back-end for the Balsa synthesis system," in *Proc. Conf. on Design, Automation and Test in Europe (DATE)*, March 2002, pp. 330–337.
[38] T. Chelcea and S. M. Nowick, "Balsa-cube: an optimising back-end for the Balsa synthesis system," in *Proc. 14th UK Asyncronisation Forum*, 2003.
[39] D. H. Linder and J. C. Harden, "Phased logic: supporting the synchronous design paradigm with delay-insensitive circuitry," *IEEE Trans. Computers*, vol. 45, no. 9, pp. 1031–1044, September 1996.
[40] M. Dean, T. Williams, and D. Dill, "Efficient self-timing with level-encoded 2-phase dual-rail LEDR," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, 1991, pp. 55–70.
[41] A. Kondratyev and K. Lwin, "Design of asynchronous circuits by synchronous CAD tools," *IEEE Design and Test of Computers*, vol. 19, no. 4, pp. 107–117, July/August 2002.
[42] P. Kudva, G. Gopalakrishnan, and H. Jacobson, "A technique for synthesizing distributed burst-mode circuits," in *Proc. ACM/IEEE Design Automation Conf.*, 1996.
[43] T. Yoneda, A. Matsumoto, M. Kato, and C. Myers, "High level synthesis of timed asynchronous circuits," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, IEEE Computer Society Press, March 2005.

[44] J. T. Udding, "A formal model for defining and classifying delay-insensitive circuits," *Distributed Computing*, vol. 1, no. 4, pp. 197–204, 1986.
[45] U. Cummings, Private communication, 2004.
[46] S. Tugsinavisut, R. Su, and P. A. Beerel, "High-level synthesis for highly concurrent hardware systems," in *Proc. Int. Conf. on Applications of Concurrency to System Design*, 2006, pp. 79–90.

8 Synthesis-based controller design

This chapter focuses on synthesis-based asynchronous controller design. Such designs use a bounded delay model, similarly to synchronous circuits. In this particular design methodology the datapath is decoupled from the control logic. Often the very same synchronous datapath can be used directly, with minor modifications.

Figure 8.1 below illustrates a synthesis-based controller in a synchronous pipeline, providing the clock (Clk) to the flip-flops. A separate synthesis-based controller can be designed uniquely for each stage of the pipeline, or a more general circuit can be specified and synthesized to fit all pipeline stages. Figure 8.1(a) illustrates a standard synchronous pipeline. In Figure 8.1(b) the Clk signal has been stripped out of the pipeline, and the controllers are placed so as to provide the latching signal for the flip-flops.

In this chapter we will first present some background on burst-mode circuits and on ways to build these controllers. We will then review approaches for building input–output circuits from signal transition graph specifications.

8.1 Fundamental-mode Huffman circuits

In the fundamental-mode style [1], circuits consist of a network of combinational gates that take inputs, move the circuit from one state to another, generate outputs, and also generate the next-state logic, similarly to the way in which standard synchronous circuits operate. The next-state outputs feed back through the delay elements and arrive at the inputs of the controller as the current state. Sometimes delays on feedback paths are necessary, however, they are not shown explicitly in Figure 8.2.

The circuit is traditionally expressed as a *flow table*, which has a row for each internal state and a column for each combination of inputs, as illustrated on the left in Figure 8.3. The entries indicate the next state entered and the output generated when the column's input combination is seen while the circuit is in the row's state. States where the next state is identical to the current state are called *stable states*. It is assumed that each unstable state leads directly to a stable state, with at most one transition occurring on each output variable. As in finite-state-machine synthesis in synchronous systems, state reduction and state encoding are performed on the flow table and Karnaugh maps are generated for each resulting signal.

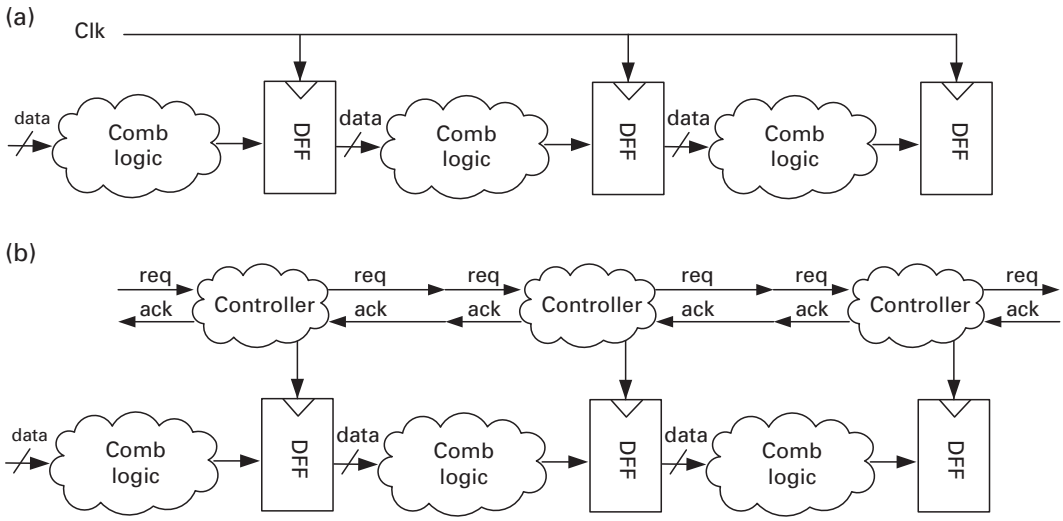

Figure 8.1. (a) Synchronous and (b) synthesis-based controller-based asynchronous pipeline.

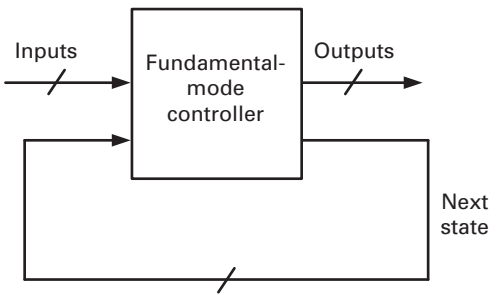

Figure 8.2. Fundamental-mode Huffman circuit diagram.

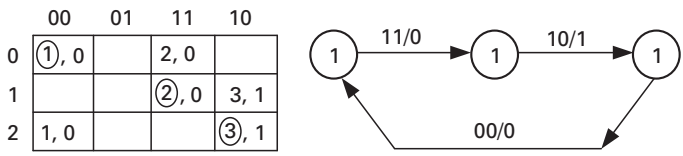

Figure 8.3. Fundamental-mode Huffman circuit flow table.

Several points need to be considered for this design method. The system responds to input changes rather than clock ticks. A new set of inputs is allowed to change only after the circuit has responded to the previously applied input and settled down in a stable state. Circuits where only one input is allowed to change are known as single-input-change (SIC) circuits; the more flexible versions, where multiple inputs are allowed to change, are known as multiple-input change (MIC) circuits.

Since hazards, static or dynamic, can cause the circuit to enter an unstable state, they must be eliminated. One way to achieve this is for the designer to add a sum-of-products circuit that has a functionally redundant product. Hazards will be covered further in subsection 8.2.2.

8.1.1 Burst-mode design

Advanced synthesis tools to optimize burst-mode designs have been pioneered by Steven Nowick and his group at Columbia University. This includes various logic synthesis and state assignment techniques that tradeoff algorithmic complexity, area, and latency [2][3][4]. They have included many of these techniques in a public-domain downloadable package called MINIMALIST [5]. Professor Nowick has also developed various interesting designs using burst-mode techniques, including low-latency "relaystations" for network-on-chip systems [6] and applications to peep-hole optimization of syntax-directed-translation-based circuits [7][8].

8.1.2 Burst-mode circuits

The *burst-mode* design style is a generalization of the traditional multiple-input change (MIC) mode, developed in [9][10]. This design style is based on the earlier work at the Hewlett–Packard laboratories by [11] and is an attempt to move even closer to synchronous design than the Huffman method. In burst-mode circuits, a number of inputs can change simultaneously; hence, in a burst fashion, and in response, a number of output signals can also change. In this method circuits are specified via a standard state machine, in which each arc is labeled by a non-empty set of inputs (an *input burst*) and a set of outputs (an *output burst*). The assumption is that, in a given state, only the inputs specified on one of the input bursts leaving that state can occur. The inputs are allowed to occur in any order. The state reacts to the inputs only when all the expected inputs have occurred. The state machine then fires the specified output bursts and enters the specified next state. New inputs are only allowed to occur after the system has reacted completely to the previous input burst. The fundamental-mode assumption must still hold between transitions occurring in different input and state bursts. Figure 8.4(a) shows a block diagram for a burst-mode machine. Figure 8.4(b) illustrates the burst-mode state diagram specification.

8.1.3 Burst-mode specification

To design or synthesize burst-mode circuits successfully, the user-defined burst-mode specification must satisfy a number of important restrictions. These are understood more easily from the examples given in Figures 8.5–8.7 and discussed below.

Starting in a given state, the machine remains stable in that state until a complete input burst arrives. Individual inputs within that burst may arrive in any order and at any time. Once the last such input arrives, the burst is complete.

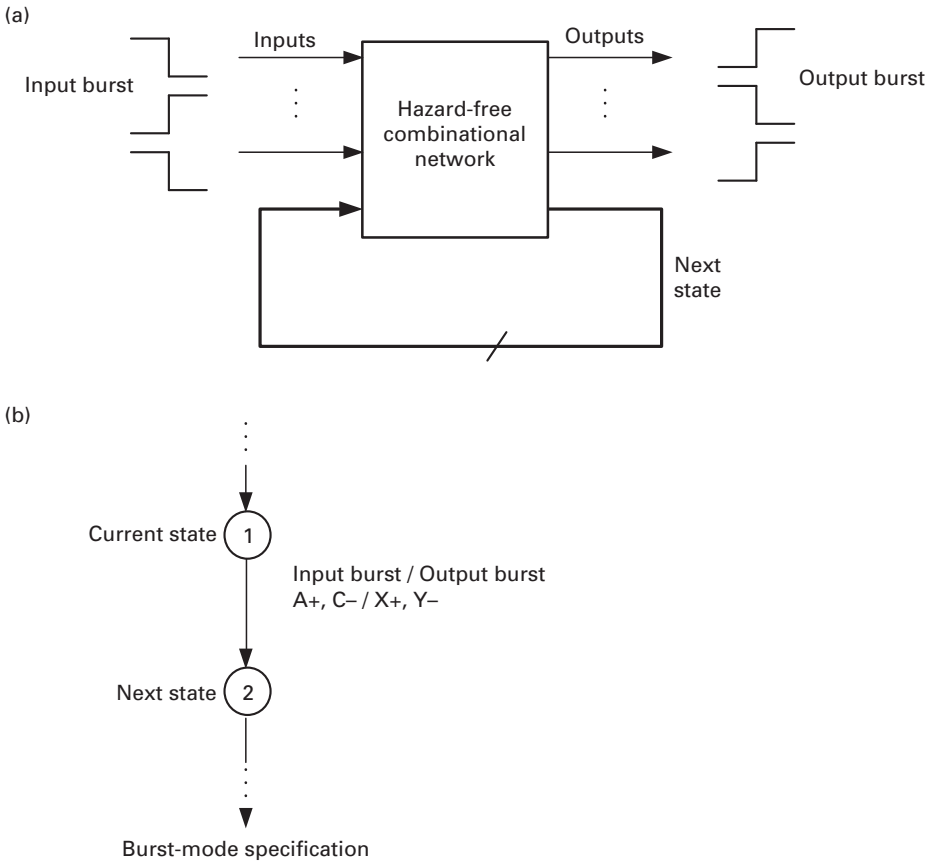

Figure 8.4. (a) Burst-mode circuit diagram and (b) state diagram.

The machine then generates the corresponding output burst, if any, and moves to the specified next state. The environment allows the machine to settle, and the next cycle begins.

1. *The input bursts cannot be empty but the output bursts can be empty* In the absence of input changes, the machine remains stable in its current state. This is illustrated in Figure 8.5. The transition from state 5 to state 0 requires the input to go low; however, none of the outputs have changed and therefore the output burst is empty.
2. *Maximal set property* No arc leaving a given state may possess an input burst that is a subset of any other arc leaving that state. This property ensures that, at all times, the state machine can decide unambiguously whether to follow a transition or remain stable. This is illustrated in Figure 8.6. If this property is violated then the problem is that there will be ambiguity when only A+ arrives. What happens in this case? Does the state machine wait for C+ and then output Z− and go to state 1 or does it immediately output Z−, Y+ and go to state 2? There is an ambiguity here that has to resolved.

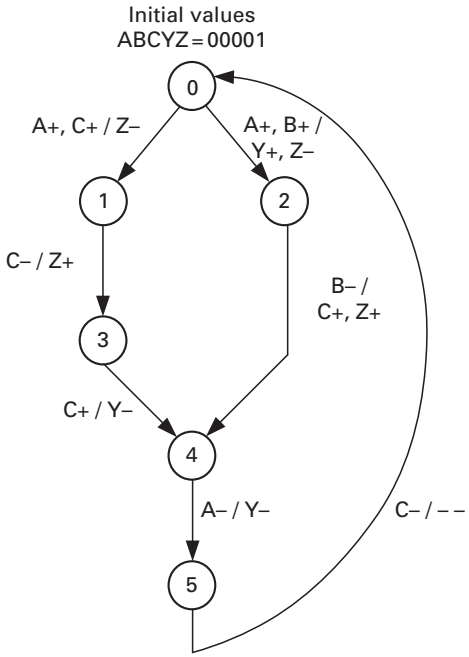

Figure 8.5. Burst-mode specification (see Figure 8.4) with empty output burst.

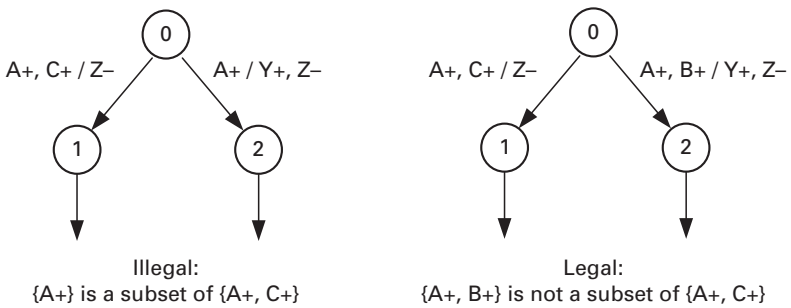

Figure 8.6. Burst-mode specification: maximal-set property.

3. *Unique entry point* Each specification state must be entered by a unique set of input values. In the example illustrated in Figure 8.7, there are two paths from the initial state 0 to state 3. Assume that at state 0 the initial values of the input signals are given by ABCD = 0000. Then A must go high to go to state 1 and from there C must go high to go to, or in other words enter, state 3. Now consider the other path. Starting at the initial state 0, the input signals again being set to all zeros, B must go high to enter state 2 and from there D must go high to enter state 3. While no matter which path we take we end up with the same output signals high or low, this is not the case for the input signals. This constitutes a violation of the burst-mode specification: there must be a unique set of inputs to reach or enter a given state. One solution is illustrated below, in

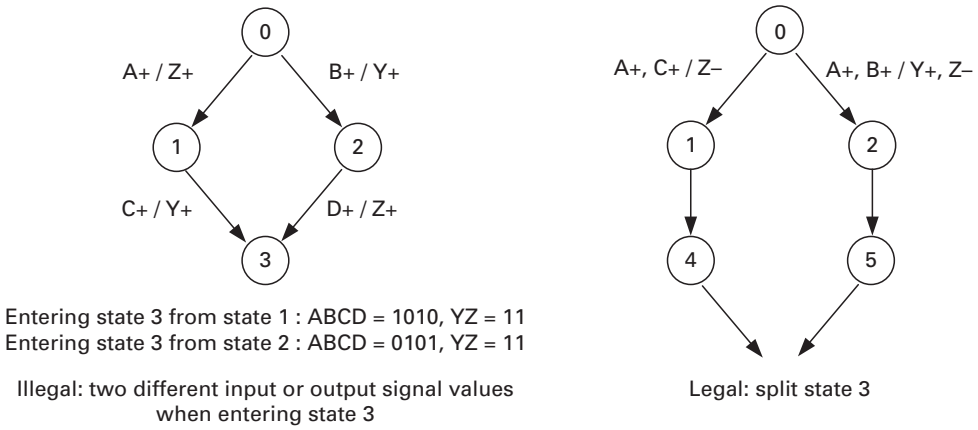

Figure 8.7. Burst-mode specification: unique entry point.

which more states are inserted on the path by splitting state 3 to states 4 and 5. Now we have at least one more state on each path to allow us to unify the entry point.

4. The burst-mode specification of the state machine must capture all expected input events. For example, in the specification illustrated in Figure 8.5 B+, C+ is not specified and therefore is an unexpected event. If this input burst should happen, it would be a violation of what the state machine expects and so there is no way to predict how the state machine would behave.

8.1.4 Hazards

One of the biggest differences between asynchronous and synchronous circuits is the importance of hazards and the way these circuits deal with them. In a synchronous system, the signals are free to transition as much as the logic in between the flip-flops permits, as long as they become stable before the rising edge of the clock or setup time is reached. In other words the only requirement is that the signals be stable around the point at which they are sampled.

A synchronous pipeline is illustrated in Figure 8.8. The shaded area indicates that signal b is allowed to change in response to a change in signal a. At the rising edge of the clock, signal a starts feeding the combinational logic. Signal b has almost the whole clock period to stabilize before it is sampled.

However, in an asynchronous system, since there is no clock, signals are not always sampled at discrete intervals but, rather, their transition from 0 to 1 or from 1 to 0 is used to communicate events. Therefore a glitch in the control logic of asynchronous circuits may be interpreted as a request to transition state, making the state machine go to an unexpected state and causing the system to malfunction. Therefore, it is important to eliminate hazards in asynchronous control circuits.

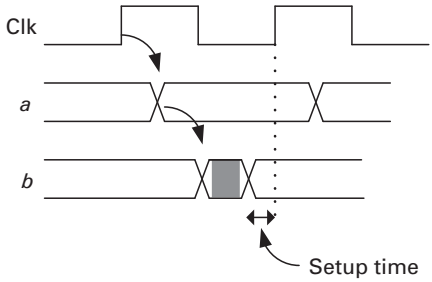

Figure 8.8. Synchronous pipeline.

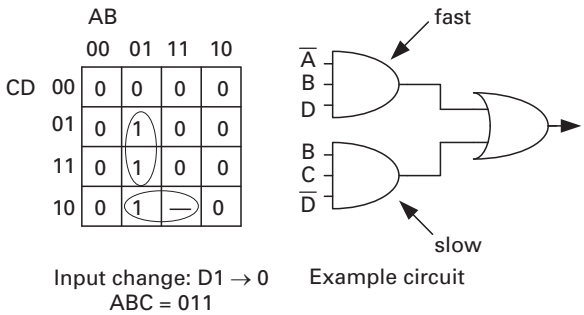

Figure 8.9. Static hazard example.

Static hazards

A static hazard occurs when a single-variable change at the input causes a momentary change on the output(s). Consider the example in Figure 8.9.

If the signal D goes from 1 to 0, the output of the top fast AND gate will go to 0 faster than the output of the bottom slow AND gate goes to 1 (see Figure 8.10). For this reason, for a short time the output of the OR gate will go low and then it will go high again. If both AND gates operated at the same speed then we would not observe this static hazard.

One way to eliminate such a hazard is to change the sum-of-products to the alternative illustrated in the Karnaugh map in Figure 8.11. In this case the signal D is no longer an input to the slow AND gate and therefore the resulting circuit is hazard-free.

Dynamic hazards

A dynamic hazard occurs when an output changes more than once, unlike a static hazard, which occurs when an output changes only once. Dynamic hazards exist when there are multiple paths with multiple delays from the changing input to the changing output. Figure 8.12 illustrates a Karnaugh map and the resulting circuit, which has a dynamic hazard.

In this example the initial value of ABCD is 0101. As signals A and C transition from 00 to 11, they propagate through the wires and gates at different rates

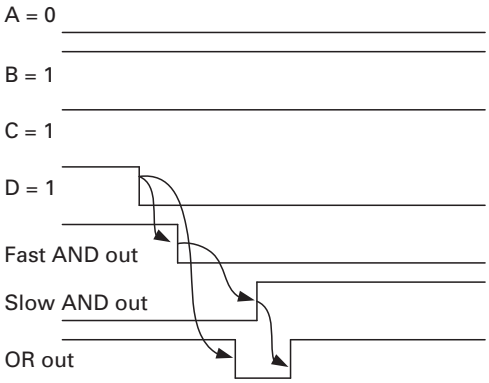

Figure 8.10. Static hazard timing diagram.

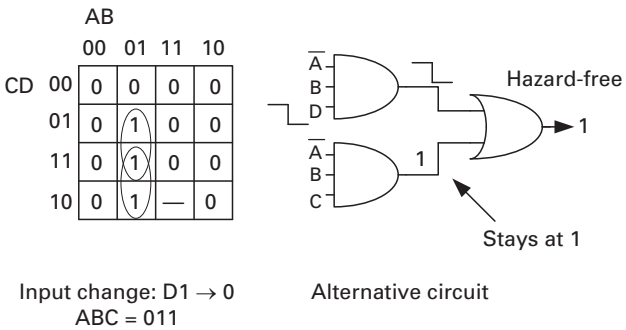

Figure 8.11. Static hazard removal.

because of different delays. As A transitions to 1, the outputs of AND_1 and AND_2 go low. However, if AND_3 is slow then its output will remain at 0 and in this case the output of the OR gate will go low. As the C input goes high the output of AND_3 will go high, pulling the output of the OR gate high. After a while, finally the $\overline{A}$ input to AND_3 will go low, transitioning the output of AND_3 low, and the output of the OR gate will go low again. Therefore even though the output of the OR gate is finally 0, it has experienced a dynamic glitch by going first low, then high, and finally low again. The timing diagram is illustrated in Figure 8.13.

This dynamic hazard can be removed by rearranging the sum-of-products as indicated in Figure 8.14.

8.1.5 Burst-mode design example

In this subsection a four-phase controller for an asynchronous linear pipeline that satisfies the fundamental-mode requirement will be implemented as a design example. A three-stage synchronous pipeline is illustrated in Figure 8.15(a). The pipeline consists of three basic parts, BitGen, Incrementer, and eight-bit registers. BitGen generates tokens of value 0, 10, 20, and so on, one time unit after the clock

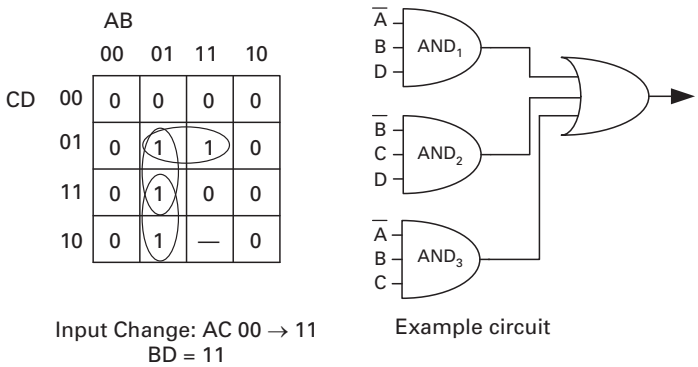

Figure 8.12. Dynamic hazard example.

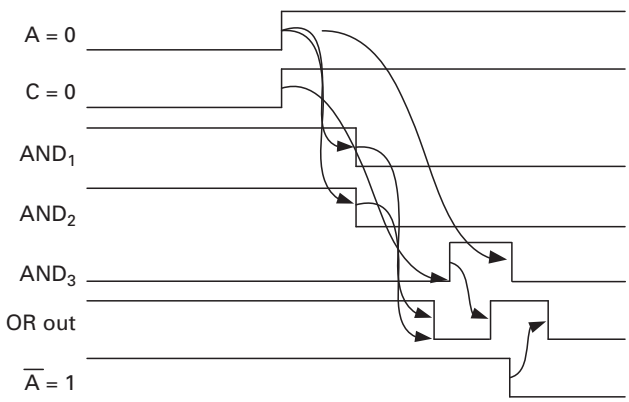

Figure 8.13. Dynamic hazard example timing diagram.

goes high. The delay through the register is also one time unit. In this asynchronous design the clock signal Clk is replaced with controllers created using MINIMALIST, a tool that can synthesize burst-mode circuits. These asynchronous controllers are placed as illustrated in Figure 8.15(b).

The first step in building the main pipeline controller is to design a state machine. First we will consider a basic three-state bust mode specification, as illustrated below in Figure 8.16. In the three-state solution the machine waits at state S0 until there is a request from the left-hand channel on Lreq. When such a request is present Lreq is pulled high by the preceding stage. The state machine moves to state S1 and forwards the request to the following stage by pulling the right-hand request signal, Rreq high. At the same time it pulls the left-hand acknowledge signal Lack high to indicate to the preceding state that it has received the request. It also latches the D flip-flop (DFF) to store the data from the incrementer or the previous stage by pulling the en signal high.

The potential problem with this solution is that after Lack− fires and before Rack− occurs, Lreq+ may occur. Also, even if Rack− occurs first, Lreq+ may

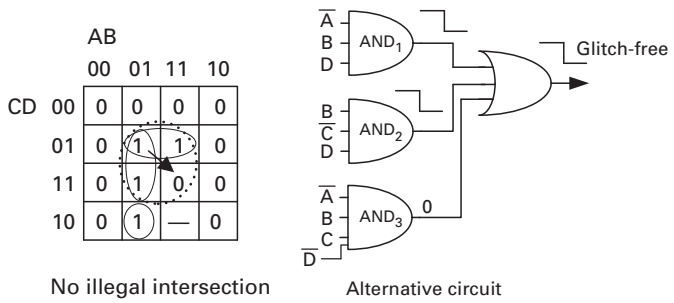

Figure 8.14. Dynamic hazard removal using a new sum-of-products.

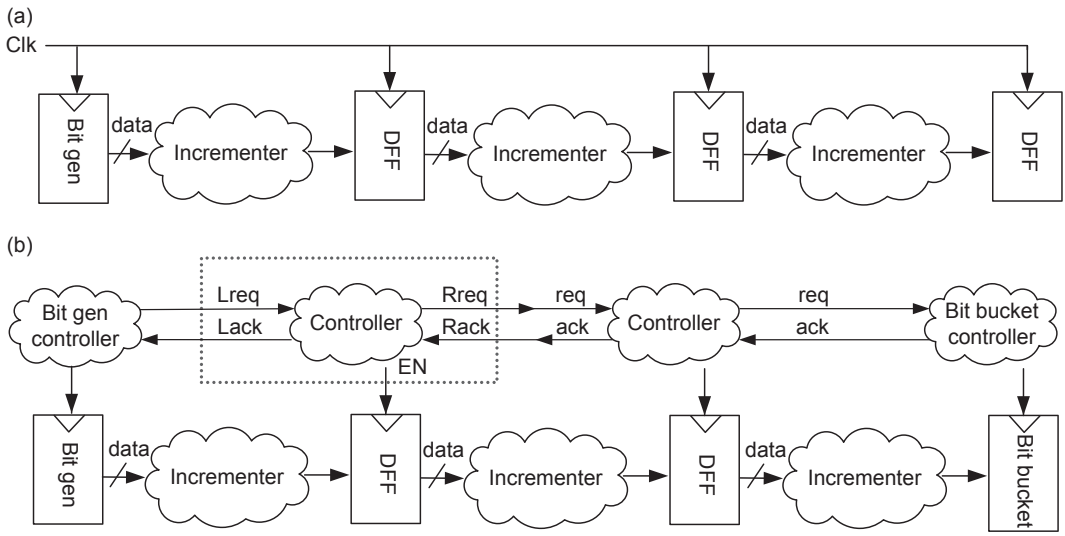

Figure 8.15. Three-stage (a) synchronous and (b) asynchronous pipelines.

occur before the circuit settles down again. The former is a violation of the specification and the latter is a violation of the fundamental-mode assumption. It is possible to argue, however, that a sufficiently large asymmetric delay line after Rreq guarantees that Rack− will come well before the next Lreq+ signal. Therefore, using such a delay and making sure that the implementation satisfies this assumption, this machine should work.

Figure 8.17 illustrates the same controller specified using a two-state burst-mode machine and Figure 8.18 illustrates the circuit synthesized from the state machine.

The primary inputs to the controller are Lreq and Rack. The Rack, en, and Rreq outputs are all generated from the same logic; note how the feedforward logic creates the primary outputs and the next-state data. The above implementation lacks reset logic, however. Upon reset the circuit must be forced to go to a deterministic initial state, upon which it can begin operation. Ideally, reset is added to the feedback path rather than the forward path, so that the forward path's delay,

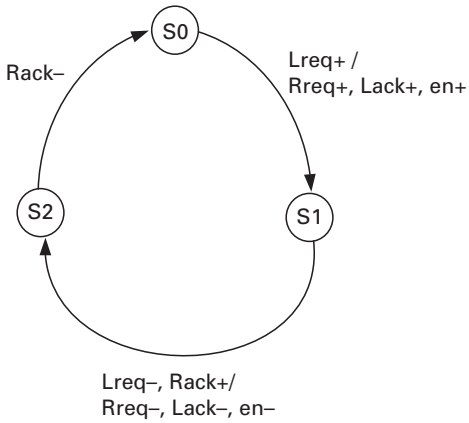

Figure 8.16. A three-state burst-mode state machine.

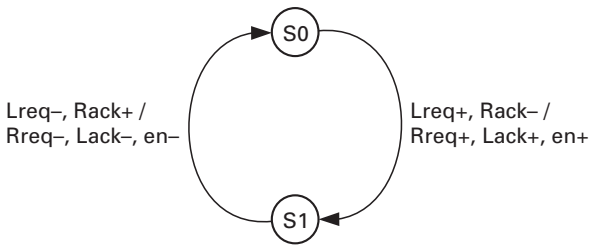

Figure 8.17. A two-state burst-mode state machine.

which is typically critical, is not increased. However, applying this approach to the case above will have the result that the path from the top-most AND gate, AND (Lreq, Rack), to the primary outputs (Rack, en, and Rreq) cannot reset properly. In particular, consider the case where the bit-generator controller is not reset. Then, there is a cycle, from the buffer's Lack output back to the buffer's Lreq through the bit-generator controller, that is not broken by reset. In fact, at initialization and even during reset the logic values of these gates are unknown. In simulation the signals along this cycle and all following nets will be an "X" throughout the waveform. Simply put, after reset the circuit will be in an unknown state.

To avoid this, we can add reset to the bit-generator controller. In particular, at reset the first Lreq coming from bit generator as Rreq should be 0. This will make all the outputs of all buffers go to 0. Reset should be asserted long enough for this 0 to trickle to the end of the pipeline.

8.2 STG-based design

An alternative means of specifying asynchronous control circuits is using signal transition graphs, first introduced by Chu 1985. STGs are interpreted Petri-nets, in which each Petri-net transition represents a rising or falling transitions of an

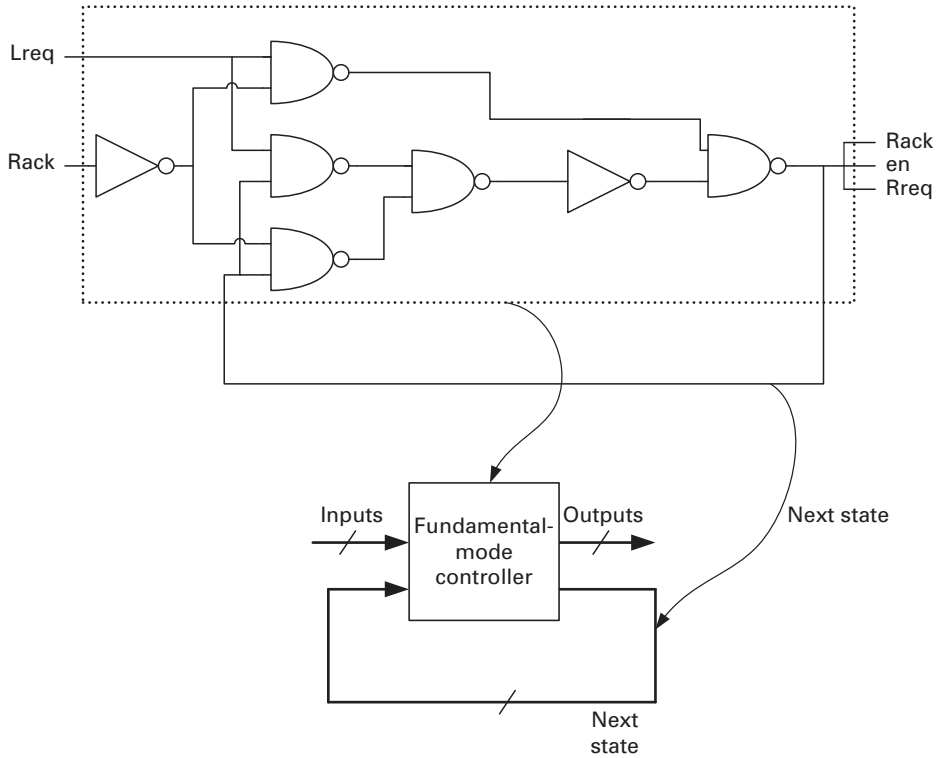

Figure 8.18. A burst-mode circuit synthesized from the two-state specification.

input or output signal. STGs can exhibit greater concurrency than burst-mode machines as they allow greater concurrency between inputs and output transitions. Instead of relying on notions of fundamental mode in which the control circuit must stabilize before a new input transition is allowed, STG-based circuits operate on the input-output mode described in Section 7.3. The STG defines when new input transitions are allowed to occur, typically in response to some output transition. The basic requirements for STG specifications to be implementable is that transitions on every signal alternate between rising and falling and the underlying Petri-net be safe.

8.2.1 STG example

As an example, the linear pipeline controller described as a two-state burst mode machine can be reformulated as an STG specification with the same concurrency as the burst-mode diagram illustrated in Figure 8.19 {FIXME}, with a few notable exceptions. First, for simplicity, we excluded the enable signal that in practice this could be implemented with a buffered version of Rreq. Secondly, the STG explicitly shows the causality between the signal transitions that is less obvious in the burst-mode diagram. In particular, the STG shows that the input signal

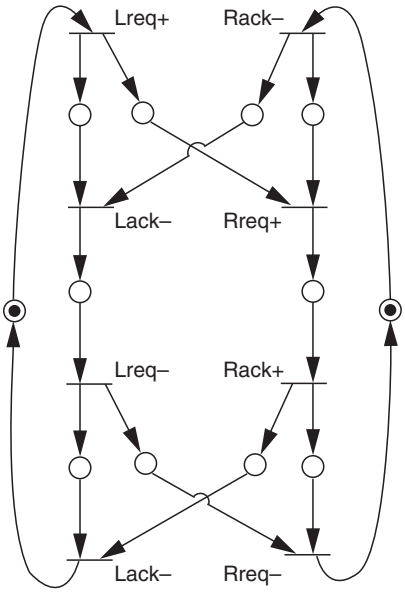

Figure 8.19. STG-based specification of a linear pipeline controller.

transitions on Lreq depends only on the output transitions of Lack and not on any signal in the right environment. Similarly, the input signal transitions on the input Rack depend only on the output transitions of Rreq and not on any transition of the left environment.

Notice also that the initial marking of the STG, indicates the initial state of the controller, which in this case is where Lack = 0, Lreq = 0, Rreq = 0, and Rack = 1, the latter indicating that the right hand environment. As in burst-mode design, an implicit reset signal input may be necessary to guarantee that the circuit initializes in this state.

8.2.2 CAD tools for STG-based controller design

Using STG specifications both bounded-delay circuits as well as speed-independent circuits can be synthesized. In addition, explicit and relative-timing assumptions, if known, can be used to further optimize these circuits [12][13]. Early work in synthesis of STG-based circuits relied on the entire state space of the STG to be enumerated from which binate-covering algorithms can be used to generate two-level logic structures and C-elements [14]. Later work use structural and partial-order techniques to address the complexity issues [15] and more powerful *theory of regions* that enabled more practical multi-level-logic implementations [16]. Much of this work has been made incorporated in the public-domain downloadable tool Petrify developed by Professor Cortadella and his colleagues [17].

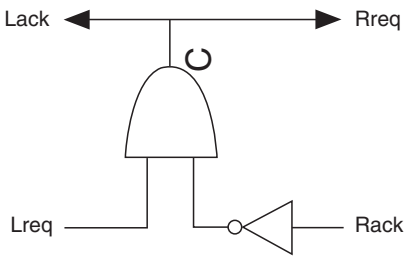

Figure 8.20. A speed-independent implementation of the STG in Figure 8.19.

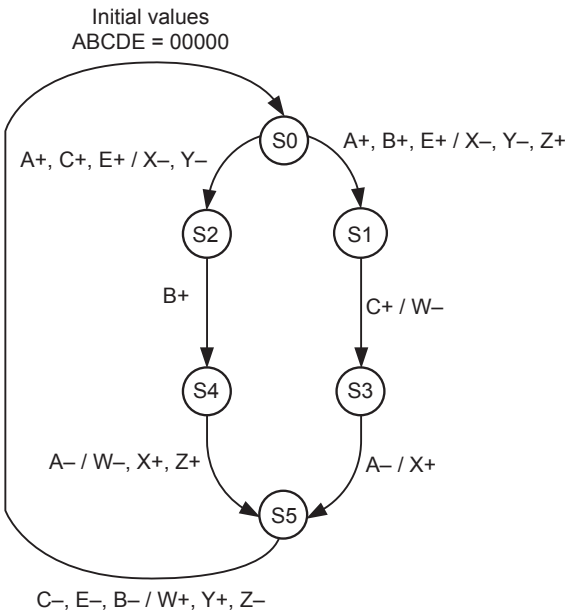

Figure 8.21. Signal transition graph for Exercise 8.1.

A speed-independent implementation of the specification in the above figure, is simply a Muller C-element with an inverted input on Rack that drives both Lack and Rreq, as shown in Figure 8.20.

8.3 Exercises

8.1. State whether the signal transition graph (STG) in Figure 8.21 is in violation of the burst-mode-specification. If so correct the STG.
8.2. State whether the STG in Figure 8.22 is in violation of the burst-mode specification. If so correct the STG.
8.3. State whether the STG in Figure 8.23 is in violation of the burst-mode specification. If so correct the STG.

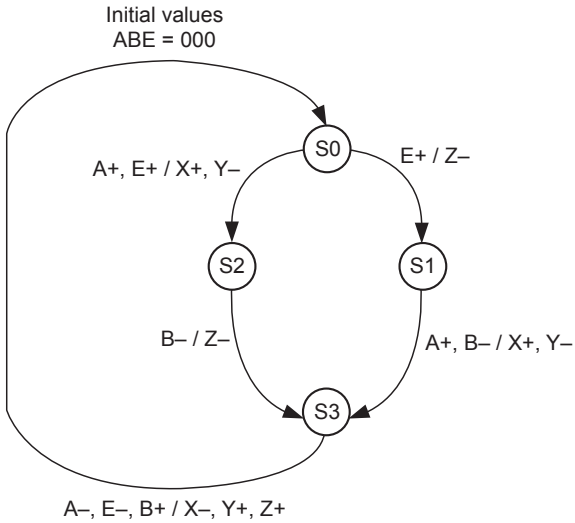

Figure 8.22. Signal transition graph for Exercise 8.2.

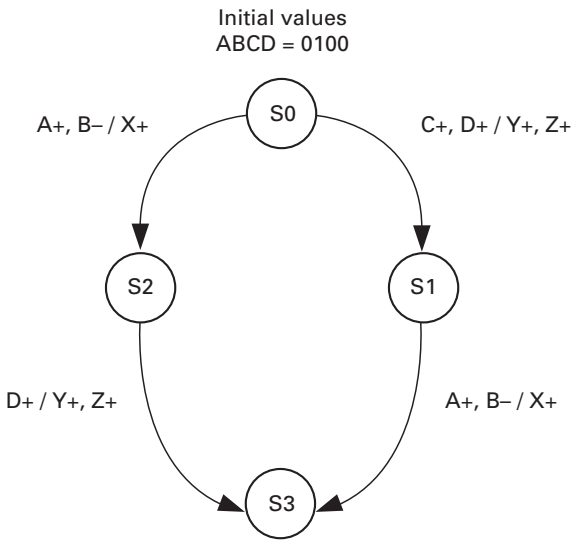

Figure 8.23. Signal transition graph for Exercise 8.3.

References

[1] K. van Berkel, M. B. Josephs, and S. M. Nowick, "Scanning the technology: applications of asynchronous circuits," *Proc. IEEE*, vol. 87, no. 2, pp. 223–233, February 1999.

[2] M. Theobald and S. M. Nowick, "Fast heuristic and exact algorithms for two-level hazard-free logic minimization." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17:11, pp. 1130–1147 (Nov. 1998).

[3] S. M. Nowick and D. L. Dill, "Exact two-level minimization of hazard-free logic with multiple-input changes." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14:8, pp. 986–997 (Aug. 1995).

[4] R. M. Fuhrer and S. M. Nowick, "OPTIMISTA: State minimization of asynchronous FSMs for optimum output logic." *ICCAD: Proceedings of the IEEE International Conference on Computer-Aided Design*, San Jose, CA (Nov. 1999).

[5] R. M. Fuhrer, S. M. Nowick, M. Theobald, N. K. Jha, B. Lin, and L. Plana, "Minimalist: An environment for the synthesis, verification and testability of burst-mode asynchronous machines." Technical Report, CUCS-020-99, Columbia University, NY, 1999.

[6] T. Chelcea and S. M. Nowick, "Robust interfaces for mixed-timing systems." *IEEE Transactions on VLSI Systems*, vol. 12:8, pp. 857–873 (August 2004).

[7] T. Chelcea, A. Bardsley, D. Edwards, and S. M. Nowick, "A burst-mode oriented back-end for the Balsa synthesis system," in *Proc. Design, Automation and Test in Europe* (*DATE*), Mar. 2002, pp. 330–337.

[8] T. Chelcea and S. M. Nowick, "Balsa-cube: an optimising back-end for the Balsa synthesis system." in *Proc. 14th UK Async. Forum*, 2003.

[9] K. Y. Yun and D. L. Dill, "Automatic synthesis of extended burst-mode circuits: Part II (automatic synthesis)," *IEEE Trans. Computer-Aided Design*, vol. 18, no. 2, pp. 118–132, February 1999.

[10] K. Y. Yun and D. L. Dill, "Automatic synthesis of extended burst-mode circuits: Part I (specification and hazard-free implementation)," *IEEE Trans. Computer-Aided Design*, vol. 18, no. 2, pp. 101–117, February 1999.

[11] A. Davis, B. Coates, and K. Stevens, "The Post Office experience: designing a large asynchronous chip," in *Proc. Hawaii Int. Conf. on System Sciences*, vol. I, January 1993, pp. 409–418.

[12] C. Myers, *Asynchronous Circuit Design*. John Wiley & Sons, 2001.

[13] K. Stevens, S. Rotem, and R. Ginosar, "Relative timing," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Apr. 1999, pp. 208–218.

[14] P. A. Beerel, C. J. Myers, and T. H. -Y. Meng, "Covering conditions and algorithms for the synthesis of speed-independent circuits," *IEEE Trans. on Computer-Aided Design*, Mar. 1998.

[15] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, *Logic Synthesis of Asynchronous Controllers and Interfaces*. Springer-Verlag, 2002.

[16] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev 1996. Complete state encoding based on the theory of regions. In *Proceedings of the 2nd international Symposium on Advanced Research in Asynchronous Circuits and Systems* (March 18–21, 1996). ASYNC. IEEE Computer Society, Washington, DC, 36.

[17] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers," in *Proc. XI Conference on Design of Integrated Circuits and Systems*, Nov. 1996.

9 Micropipeline design

One of the earliest and most widely used asynchronous design styles is micropipelines. The basic concept and name was presented in Ivan Sutherland's 1989 Turing award lecture [1]. Since then, the basic design style has been extended in a variety of ways to support more sophisticated control and datapath styles. As illustrated in Figure 9.1, a micropipeline is now characterized as any pipelined design that uses bundled-data channels implemented with a delay line matched to the worst-case delay of single-rail combinational logic fed by a register element that is controlled by an asynchronous control circuit. The setup and hold requirements on the register element are similar to those in synchronous design and are often called *bundling constraints*. Thus, micropipeline design is also known as *bundled-data design*, owing to the use of these bundling constraints (see Chapter 7). In practice, there are several micropipeline variants: the handshaking protocols can be two-phase or four-phase and data-validity schemes can be broad or early. In addition, the register element may be edge-triggered (e.g. it could be a flip-flop) or level-sensitive (e.g. a latch) and the datapath can be designed using static or dynamic logic. The following section first reviews the original templates proposed by Sutherland before describing several extensions.

9.1 Two-phase micropipelines

As originally proposed the storage element is implemented with a capture–pass latch formed by connecting two master–slave flip-flops in parallel [1], as illustrated in Figure 9.2. The upper flip-flop is controlled by C and P and their inverted counterparts $\overline{C}$ and $\overline{P}$. This arrangement can be viewed as a conventional double edge-triggered master–slave flip-flop, in which the C and P signals correspond to a clock and its inverted counterpart. In conventional master–slave flip-flops, C and P switch nearly simultaneously, i.e. the closing of the master latch (capture) and the opening of the slave latch (pass) occur simultaneously. However, in capture–pass latches, the latches become transparent when a pass event occurs (thus releasing the previously captured data) and become opaque when a subsequent capture event occurs. The intervals between pass and capture events may be indefinite.

Each stage of a linear micropipeline without data is formed as shown in Figure 9.3. Note that each stage's capture signal toggles (transitions) only after

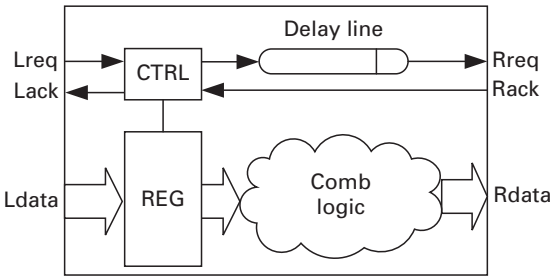

Figure 9.1. Basic organization of a micropipeline stage.

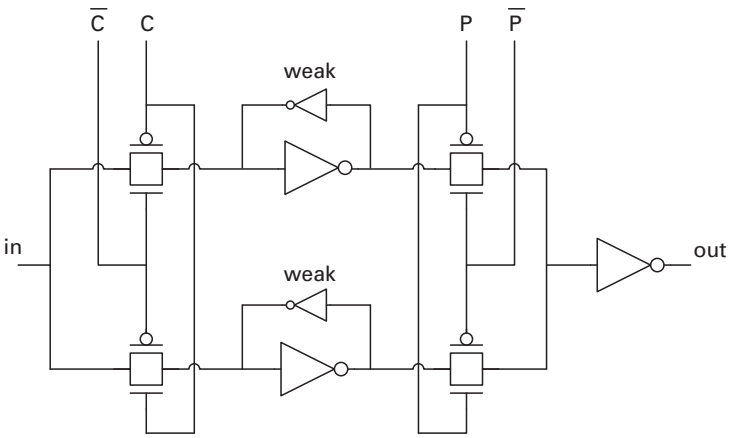

Figure 9.2. Capture–pass latch.

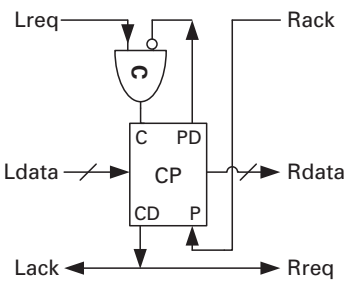

Figure 9.3. Two-phase micropipeline stage (without logic) using capture–pass latches.

the corresponding pass signal toggles, owing to the C-element. When an input request occurs, the stage captures the newly available data and generates a capture-done signal as an amplified and delayed version of the capture signal (not shown in Figure 9.2). Note that the slave latch of the flip-flop, which captures the data, remains transparent for the data to flow through. The capture-done signal is

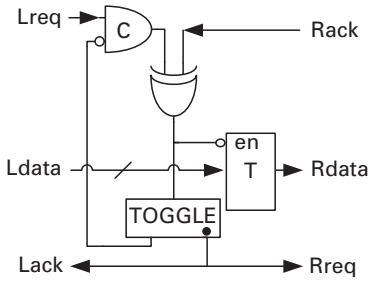

Figure 9.4. Two-phase micropipeline stage (without logic) using transparent latches.

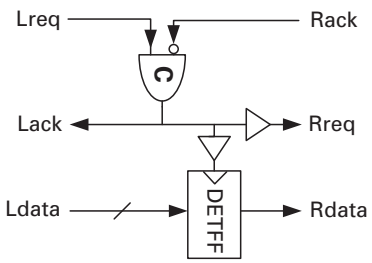

Figure 9.5. Two-phase micropipeline stage (without logic) using DETFFs.

used to acknowledge the previous stage and also to request the next stage to accept the data. When the next stage captures the data and acknowledges, the pass signal toggles, allowing the alternate flip-flop to drive the output. The pass-done signal, an amplified and delayed version of the pass signal, primes the C-element for the next input request.

Because capture–pass latches are large (four transmission gates, a pair of cross-coupled inverters and an output inverter) and consume significant power, the AMULET group from the University of Manchester suggested using transparent latches with level-sensitive enables. While this reduces power, it comes at the expense of latency and throughput; this is largely due to the control overhead associated with the two-to-four-phase interface around the level-sensitive enables. A simple implementation using an XOR and a TOGGLE element is illustrated in Figure 9.4 [3][4].

An alternative two-phase implementation that uses double-edge triggered flip-flops (DETFFs) instead of transparent latches was proposed [5] (see Figure 9.5). The DETFFs yield substantially higher performance but at the cost of increased area and power consumption in comparison with alternative templates. A timing optimization, sometimes called *control kiting* [10], can be used to remove the delay of the buffer driving the DETFFs from the cycle time. In particular, if the delays of the buffers driving the DETFFs of neighboring stages are equal then the input data and the latching signal are equally delayed and the control overhead reduces to that of the C-element delay.

One of the highest-throughput two-phase pipelines that has been developed is the MOUSETRAP pipeline, one stage of which is illustrated in Figure 9.6.

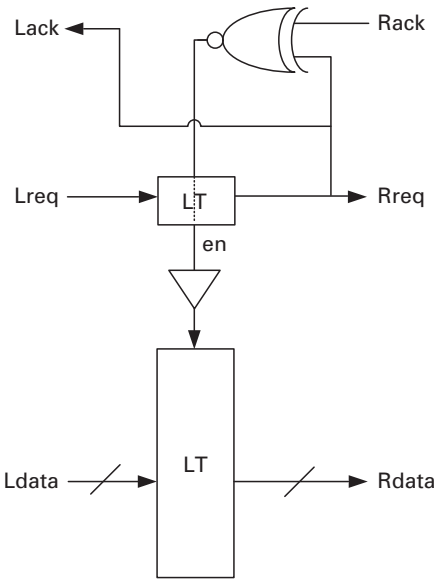

Figure 9.6. Ultra-high-speed MOUSETRAP pipeline stage without logic. The diagonal slashes indicate that Ldata and Rdata are multi-bit signals.

The data and the request are both latched at every stage, with latches that are controlled by a local XNOR gate. The latches are initially transparent, while waiting for data to arrive. Since the requests of successive tokens have alternating values, when a request goes through the latch it will change the value of the XNOR output and the latch is made opaque. When the next stage also fires and the data has been latched in this stage, the XNOR will change value again and the latch is made transparent again, allowing new data to arrive. The protocol has a cycle time of 5–6 transitions and has been simulated to operate at over 2.1 GHz in a 0.18 micron process [6]. This approach is very promising and sophisticated applications of it to more complex architectures that include conditional communication are being actively researched.

Since the design is not just a FIFO, single-rail bundled data logic can be added to the data path between the latches. Moreover, as usual, the request signal is delay-matched with the slowest combinational path in order to avoid latching the values that will not meet the bundling constraint. A significant advantage of this design style is that both control and datapath can be designed from standard gates found in any synchronous library, making adoption by industry easier. The interested reader is directed to [15] for detailed performance and timing analysis along with a description of additional circuit optimizations.

9.1.1 Non-linear pipelines

More complicated pipelines are designed using control circuits that operate on request–acknowledge control signals called *event modules*. Sutherland proposed

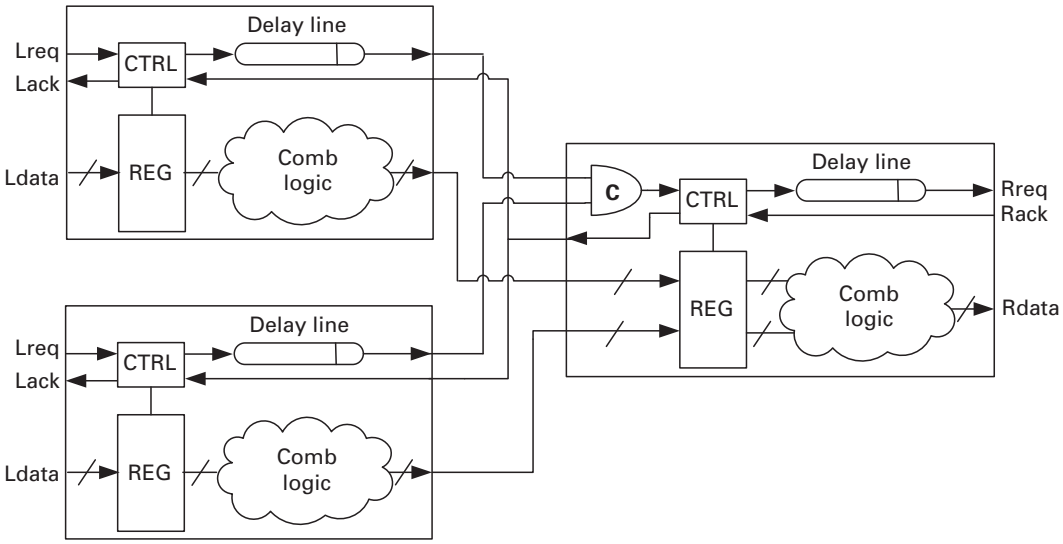

Figure 9.7. Example of the use of a C-element event module in a micropipeline join stage.

a small set of event modules from which a large range of complex systems could easily be designed [1].

The first type of event module to be considered is one that we have already encountered, the C-element. It acts as an AND gate for transition signaling events since only when there are events on both inputs of the C-element will there be an event on the output. This is useful in a pipeline join, as illustrated in Figure 9.7 for two-phase control and registers. The requests of both incoming stages are the inputs to a C-element that feeds the delay line in the join stage. This guarantees that all data inputs to the join stage are valid before the delay line is triggered, ensuring that the corresponding register captures valid outputs.

Another originally proposed event module is the *select* element (see the square in Figure 9.8 with ports A, T, F, S). Depending on the value of a sampled one-bit input, the select element conditionally steers transition-signaling events on its input port to one of two different output ports. This is useful in creating conditionally communicating pipeline stages. As an example consider Figure 9.8, in which, depending on the conditional input, the output token is sent to one of two different destination channels. The acknowledgement events from the two successor stages are mutually exclusive. Thus they can be combined with an XOR gate, which acts as an event module, producing an event on its output if an event occurs on either input. Notice that the select input S must be stable before the input event on terminal A of the select module arrives; this is guaranteed by the length of the delay line.

9.1.2 Resource sharing

The CALL event module has two request–acknowledge pairs associated with two input channels and one request–acknowledge pair associated with an output

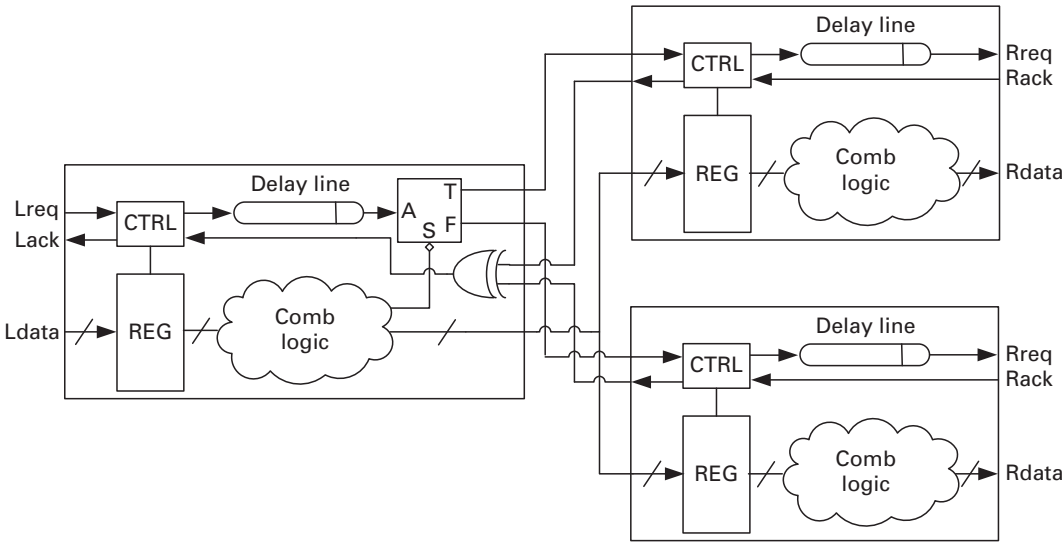

Figure 9.8. The select and XOR event modules in a pipeline stage that conditionally communicates with two successors.

channel, as illustrated in Figure 9.9(a). The events on the input requests are treated as mutually exclusive and transferred to the output request. The module then waits for an acknowledgement on the output channel, which it routes as an acknowledgement to the requesting input channel. This is designed to enable resource sharing among the different input channels.

Using terms from Chapter 2, the CALL module *encloses* the handshake on the output synchronization channel within that of the input channel. This is useful because receipt of the acknowledgement tells the initiator that the called procedure is complete. However, it is also important to understand that in an otherwise pipelined system it can have a profound impact on performance. In particular, the input pipeline may be stalled until the acknowledgement is received.

9.1.3 Arbitration

The final event module proposed in [1] is a two-phase arbiter, as illustrated in Figure 9.9(b). It has two request inputs and two request–acknowledge pairs associated with the two output channels. Similarly to the arbiters discussed in Chapter 2, the arbiter routes the first incoming request to the corresponding output request, non-deterministically choosing the winner when inputs requests arrive simultaneously. After the winner is chosen and the corresponding acknowledge event is received, the arbiter can choose a new winner. Interestingly, the proposed arbiter module does not have corresponding acknowledge signals for the input requests. These must be generated from other signals in the system.

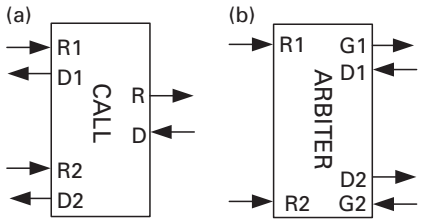

Figure 9.9. Call and arbitration module used for resource sharing.

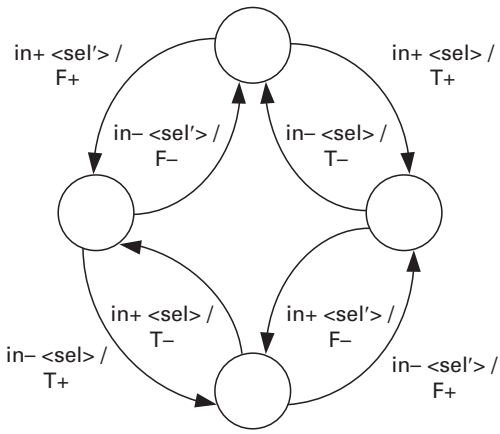

Figure 9.10. Extended burst-mode specification of a select event module. The input signals are separated from the output signals by slashes. The notation <sel> indicates that the signal is sampled and the associated edge can only be taken when the sampled value of sel is 1; likewise an edge labeled <sel′> can only be taken when the sampled value of sel is 0.

9.1.4 Event-module implementations

Initially, hand-designed transistor-level implementations of event modules were proposed. With the advent of both burst-mode and STG-based synthesis, however, a variety of synthesized implementations have demonstrated more efficient designs.

As an example, the extended burst mode format (XBM) specification and corresponding synthesized transistor-level implementation of a toggle is shown in Figures 9.10 and 9.11 [5]. Synthesis tools also enable more complex event modules, which can yield significant reductions in control overhead, to be defined and implemented.

The specification of the burst-mode event module highlights a disadvantage of event-based micropipelines for complex systems. In particular, the toggle has to support input requests from four different states that depend on the current values of the input and output signals. This is more complex than in the case of four-phase controllers, whose input and outputs always reset to a known value. In particular, the two-phase nature implies a symmetric transistor-level implementation that will generally have a PMOS network that is as complex as the NMOS network. Because PMOS transistors are inherently weaker than NMOS

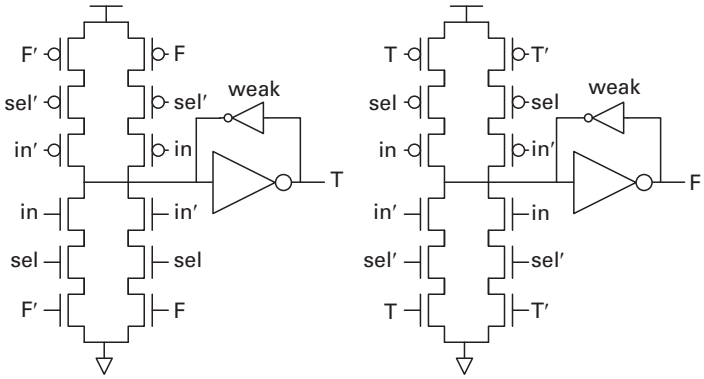

Figure 9.11. Synthesized transistor-level implementation of a select module.

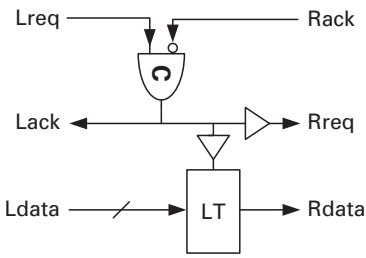

Figure 9.12. Simple four-phase micropipeline without logic.

transistors, two-phase event modules are typically slower than the corresponding four-phase components.

9.2 Four-phase micropipelines

Another advantage of four-phase-based micropipelines over the original two-phase schemes stems from the level-sensitive nature of the latch commonly used as the register element. As mentioned previously, the two-phase micropipelines described above often use TOGGLE and XOR elements as two-phase-to-four-phase converters in order to interface with a latch-based datapath. Adopting a control scheme that is four-phase avoids this significant overhead [4].

A simple four-phase control scheme uses a C-element as the latch controller, as illustrated in Figure 9.12. Again using the terminology of Chapter 2, it is a half buffer because it cannot support distinct tokens on both its input and output channels. Thus N-stage pipelines using this controller can support only $N/2$ tokens. This half-buffer nature is disadvantageous when many tokens need to be stored, as in a FIFO.

Several full-buffer alternatives have been proposed [2]. In particular, a semi-decoupled latch controller has been proposed and is illustrated in Figure 9.13.

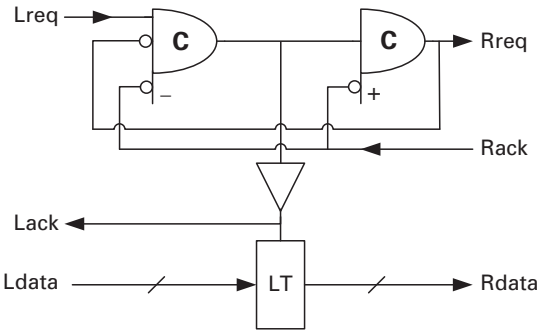

Figure 9.13. Semi-decoupled four-phase latch control.

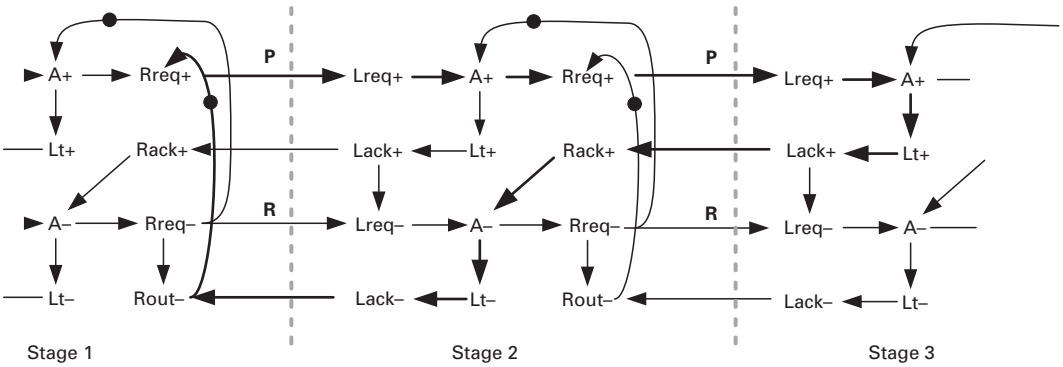

Figure 9.14. Performance bottleneck in semi-decoupled controller based pipelines.

The full-buffer nature enables the use of more area-efficient FIFOs. However, when logic is used, the processing delays associated with the datapath on the input and output sides of the controller are both included in its local cycle time.

This performance bottleneck can be illustrated by examining the marked graph description of three back-to-back semi-decoupled FIFO stages [4], as illustrated in Figure 9.14. In particular, notice that the bold-line cycle contains only one token but two "arcs" labeled **P** between the Rreq+ and Lreq+ transitions. Each arc represents the location in which a matched-delay line is present, suggesting that the cycle time includes two matched-delay line delays. Interestingly, this implies that full-buffer controllers, if not carefully designed, can in principle have the same performance bottlenecks as half buffers.

A higher-performance full buffer is the more complex fully decoupled latch controller [4] illustrated in Figure 9.15. This controller enables the input and output handshakes to reset completely independently, thus requiring only the delay associated with a single matched-delay line to be included in the local

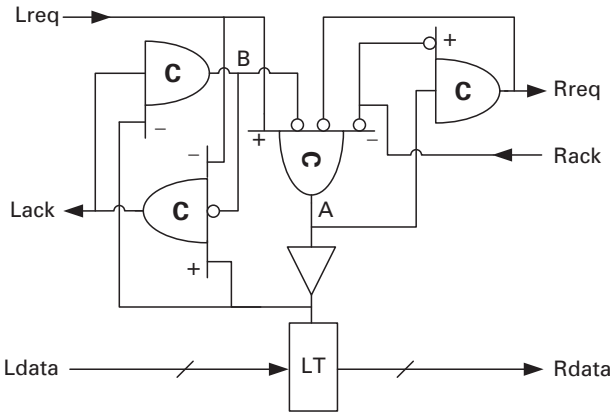

Figure 9.15. Fully decoupled latch controller.

cycle time. This represents a significant benefit in performance. However, note that now the reset delay of the associated delay line becomes a critical element in the local cycle time. Consequently, the use of an asymmetric rather than symmetric delay line becomes important, as otherwise the cycle time would again have effectively two function unit delays in its local cycle time. Several implementations of asymmetric delay lines will be discussed later in this chapter.

As in two-phase micropipelines, four-phase micropipeline fork stages need to wait for all output enable signals to set or reset before setting or resetting the output tokens. The simple solution is to insert a C-element to combine all output enable signals. If the number of fork stages is small then the C-element functionality can be integrated into the circuit element that generates the output request. Similarly, four-phase join stages need to wait for all input data to be set or reset before setting or resetting the input enable. The solution is the same as that in the two-phase join stage, i.e. to combine the requests of all input channels with a C-element to detect the availability of all input data.

More complex pipeline stages that involve channels that conditionally read and write are also possible. In the absence of predefined controllers, synthesis tools such as those described in Chapter 8 can be used to generate the desired control circuits. The biggest design challenge is to ensure that the burst-mode or STG controllers in a network for a complex architecture work together to create a correct design. In particular, each controller's specification identifies when to expect different input transitions, and its implementation may fail if its environment does not adhere to these assumptions.

In the absence of the automated generation of the network of controllers needed for a complex architecture, sophisticated formal verification may be necessary to ensure that the such a network mutually guarantees these assumptions and is deadlock-free [7].

9.3 True-four-phase pipelines

To reduce the overhead associated with delay lines, pipeline control templates that follow the *true-four-phase handshaking* protocol have also been proposed. In particular, these templates wait for the left-hand token to arrive, the left-hand enable to be sent back, and the left-hand token to reset before generating a right-hand token. In other words, they explicitly decouple the control by essentially forcing the handshaking with the left-hand environment to finish before communication with the right-hand environment begins. Consequently the forward latency includes both phases of the delay line, enabling the use of either asymmetric or symmetric delay lines. In particular, the symmetric delay line can be half the size needed for other templates, reducing the area and power consumption overheads.

A proposed implementation of a true-four-phase micropipeline stage (without logic) is shown in Figure 9.16 (from [14]). Upon reset, the internal signals lt and en are low and the acknowledge signals Lack and Rack are high. When a left-hand token arrives, signaled by Lreq+, the internal signal lt is raised and Lack is lowered. The environment then lowers Lreq, causing en and subsequently Rreq to rise, thereby triggering the positive-edge-triggered FF to latch the data token and the transmission of the output token to the right-hand environment. In addition, Rreq+ causes the signal lt to reset and Lack to rise, allowing the left-hand environment to send a new input token. Concurrently, after the right-hand token is consumed, Rack and Rreq will fall in succession. The right-hand environment will finally reassert Rack, allowing en to reassert and thereby enabling the circuit to latch a new input token.

In terms of power consumption, one disadvantage of this type of true-four-phase pipeline is the need for edge-triggered FFs instead of smaller latches having less input capacitance. In fact, adapting these templates for use with latches that are normally opaque is an interesting area of future work, as such templates can combine the low-power benefits of a small symmetric delay line with that of latch-based pipelines.

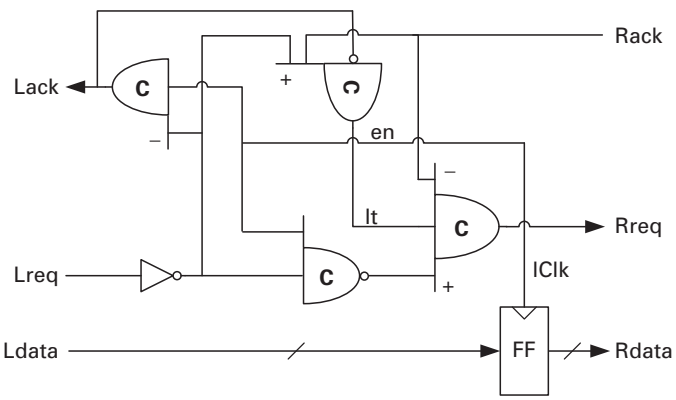

Figure 9.16. True-four-phase micropipeline stage (without logic).

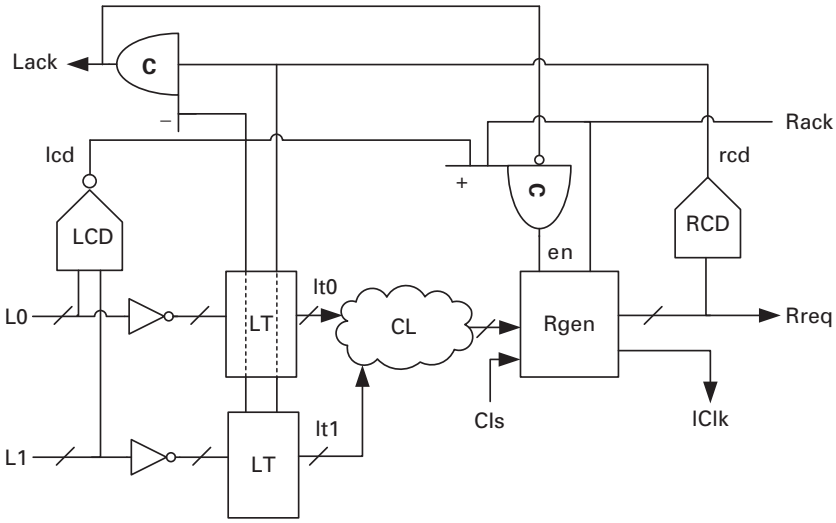

Figure 9.17. A T4PFB control template for join stages: CL is the combinational logic and the modules labeled LT are blatches.

Complex true-four-phase controllers can be supported by extending the one-wire communicating Rreq control signals to 1-of-*N* wires and using 1-of-*N* control templates to generate the local clock signal lClk used in the single-rail datapaths. Each pipe stage can operate on these control tokens and support data-dependent behavior. Specifically, this 1-of-*N* extension includes input and output completion-sensing units internal to the control-and-separate output generation logic for each output rail [14]. An abstract circuit template for a true-four-phase full-buffer (T4PFB) join stage supporting multiple input channels and one output channel is shown in Figure 9.17.

Join stages need to wait for all input tokens to arrive and reset before setting or resetting the input enable. An example of a true four-phase for a join stage implementing the OR of two dual-rail control channel inputs, L1 and L2, is depicted in Figure 9.18. The inverted LCD circuit is shown in Figure 9.18(a); each of the four channel wires are latched with the circuit illustrated in Figure 9.18(b), and the OR functionality is precomputed within the combinational blocks shown in the broken-line boxes in Figure 9.18(c). The left-hand combinational logic (CL) block, driving d_f, is asserted only when the "false" rails of lt1 and lt2 are asserted. The right-hand CL block driving d_t is asserted when either "true" rail of lt is asserted. These two signals are fed into the Rgen circuitry to generate the dual-rail outputs, as illustrated in Figure 9.18(c).

Extending this template to support fork stages with multiple output channels is straightforward. In particular, fork stages need to wait for all output enable signals to set or reset before setting or resetting the output tokens. As with other four-phase templates, one solution is to insert a C-element to combine all output acknowledge signals. Moreover, supporting conditional reading and writing is

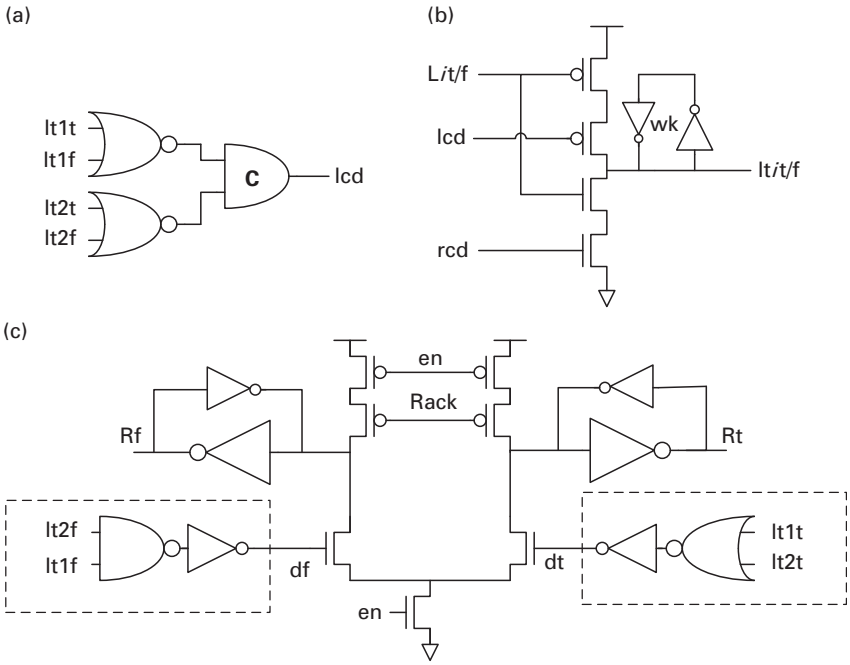

Figure 9.18. True-four-phase circuits implementing the OR of two dual-rail inputs: (a) input completion sensing, (b) one-bit latch, (c) combinational logic and R_gen circuitry (see Figure 9.17).

only slightly more complex. To conditionally read a channel, the associated Lack generation block generates a left-hand enable only if a channel is read. To conditionally write a channel, the Rgen block must conditionally evaluate and handshake with the right-hand enable only when it evaluates its output. For instance, a skip can be implemented by triggering the evaluation of a separate output signal (not routed out of the controller), which acts like an extra output rail, and then immediately sending an acknowledgement back to the left-hand environment without waiting for the right-hand environment.

The interested reader is directed to [14] for a detailed analysis of the associated timing assumptions necessary for correct operation, along with other performance optimizations.

9.4 Delay line design

A delay matching element (delay line) consists of combinational logic whose propagation delay is matched with the worst-case logic delay of some associated block of logic. Generally, a delay line is implemented by replicating portions of the block's critical path.

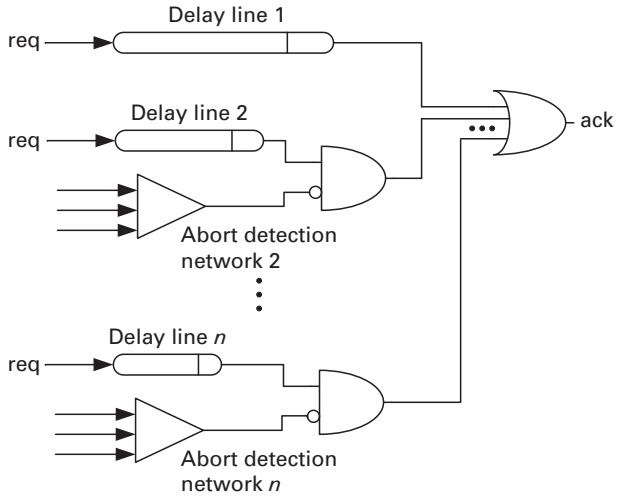

Figure 9.19. Speculative completion-sensing architecture.

To optimize performance for the average case, a more complicated delay line design based on speculative completion sensing can be adopted [8]. The speculative delay line uses data-dependent *abort* signals to select between several delay lines on the basis of the expected delay of the operation, as illustrated in Figure 9.19. To save area, it possible to reuse previous delay elements to generate the next-larger matched delay [8]. However, in the original formulation the input req signal can propagate through the entire delay line independently of the selected delay, thereby potentially expending unnecessary power.

9.4.1 Asymmetric delay line templates

Two improved speculative delay-matching templates that are compact and power saving, one for an asymmetric delay line and one for a symmetric delay line, have also been proposed [13]. The asymmetric template adapted from [9] creates a set of delay line controllers, one per delay element, as shown in Figure 9.20. Each controller functions similarly to an asynchronous split, in that its input signal is routed to one of its output signals on the basis of the Select control line values. If the Select lines indicate that the target delay has been obtained, the controller generates the "done" signal by routing the input to LD_i. Otherwise, it propagates the input signal to the next delay element via NR_i. Because the input signal stops at the target delay element, the power consumption is significantly reduced.

The operation begins with the set phase. When start+ arrives, it propagates to the first asymmetric delay line (ADL) asserting a delayed signal d_0. This delayed signal and the select lines sel provide the input signals for an asymmetric controller

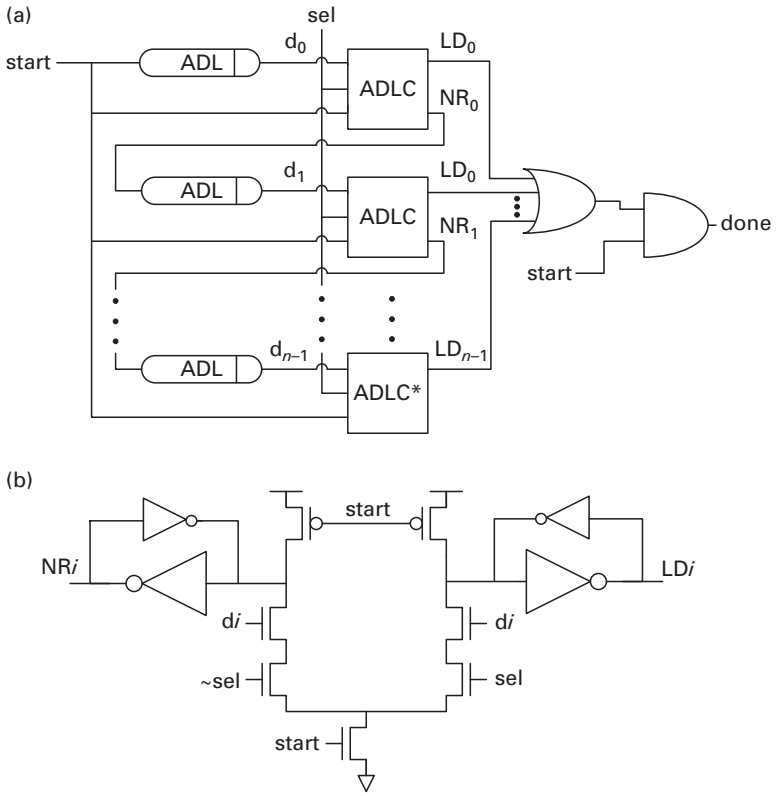

Figure 9.20. Efficient asymmetric delay line implementation.

(ADLC), whose implementation is shown in Figure 9.20(b). This controller decides to assert either a "local done" signal LD_0 or the next request signal NR_0. If one of the local done signals LD_i is asserted then the done signal is asserted, finishing the set phase. Otherwise, a next request signal NR_i+ activates the next delay element. Note that the last controller (ADLC*) is actually not required; it merely generates a local done signal, LD_n.

The reset phase is typically minimized to reduce control overheads when it is used with four-phase micropipeline controllers. This phase begins when the start signal is reset. Since all delay elements are bypassed with an AND gate, the done signal resets quickly, i.e., in a one-gate delays. Simultaneously, the start signal actively resets all delay elements and controllers.

Two timing constraints associated with the delay line must be satisfied. First, the select lines of each controller must be set up and valid before their associated delay signal d_i+ arrives, in order to avoid a wrong routing decision. This constraint is referred to as a *select line setup constraint* [13]. Second, all internal signals must be reset before the next start signal arrives; this is referred to as the *delay line reset constraint*.

9.4.2 Symmetric delay line templates

The symmetric delay line depicted in Figure 9.21 utilizes both the set and reset phases to match the worst-case logic delay. It is well suited to the T4PFB control protocol since it transfers data to the next stage after it has passed through both the set and reset phases of the delay line.

There are two timing constraints associated with the symmetric delay line. First, the *select line setup constraint* described for the asymmetric delay line also applies to the symmetric delay line. Notice, however, that this setup constraint is more stringent than in the asymmetric delay line case because now the matched delay elements are only half as long. In addition, the select lines must be stable until after the end of the reset phase; this is referred to as the *select line hold constraint*.

Satisfying both these constraints, however, is significantly easier than satisfying the reset constraint of the asymmetric delay line. In particular, the lack of a reset constraint allows us to eliminate the final AND gate and alleviates the heavy load of the start signal in the symmetric delay line controller (SDLC) shown in Figure 9.21(a). The symmetric delay line is also approximately half the length of the asymmetric delay line, saving both area and power. These advantages make the use of a symmetric template very attractive.

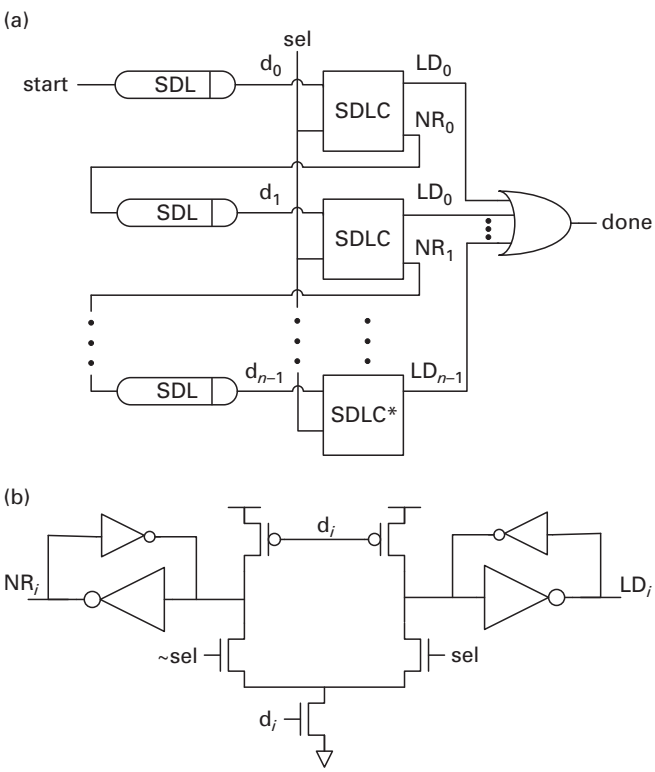

Figure 9.21. Efficient symmetric delay line implementation.

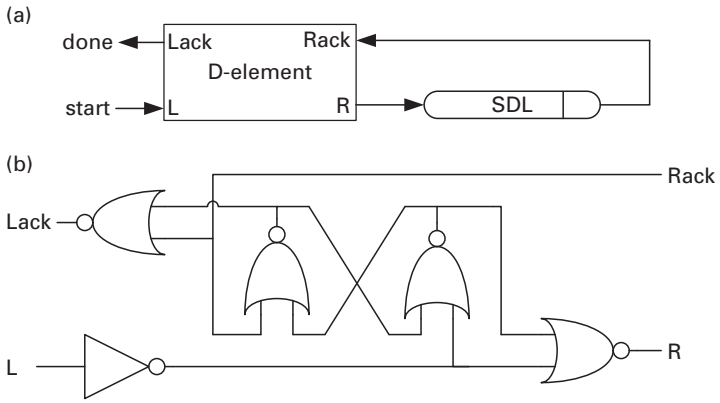

Figure 9.22. Power-efficient asymmetric delay line.

9.4.3 Power-efficient asymmetric delay line

A power-efficient asymmetric delay line can be constructed using a combination of a symmetric delay line and a D-element [11][12], as illustrated in Figure 9.22(a). The D-element operates as follows. After receiving a left-hand request it completes a full handshake on the right-hand environment before acknowledging the left-hand environment, enabling the use of a symmetric delay line on its right-hand environment. In the reset phase, the D-element, shown in Figure 9.22(b), can reset in four gate delays. The forward latency includes both phases of the delay line plus a small delay from the D-element (six gate delays in total). Additionally, the overhead is independent of the delay line delay, but it is still large owing to the combined overhead from the controller and the reset delay from the D-element (four gate delays).

In comparison with other asymmetric delay lines, this delay line can reduce both the area and the power by approximately half. However, owing to the large forward latency, it is not suitable for shallow pipeline stages.

9.5 Other micropipeline techniques

In many templates an attempt has been made to combine the simplicity of the micropipeline technique with the performance advantage of dynamic datapaths [17][18][19]. In most of these templates, the control is four-phase and the datapath components generate a data-dependent done signal, which removes the need for delay lines. These techniques can more readily support data-dependent delay and are more robust to process variations than single-rail techniques, which have to rely on a matched delay line. However, this robustness comes at the cost of the increased area and, often, increased switching activity associated with dual-rail logic. One promising approach is to combine static and dynamic datapaths, using a 1-of-*N* dynamic datapath only for the critical path. This has been used effectively

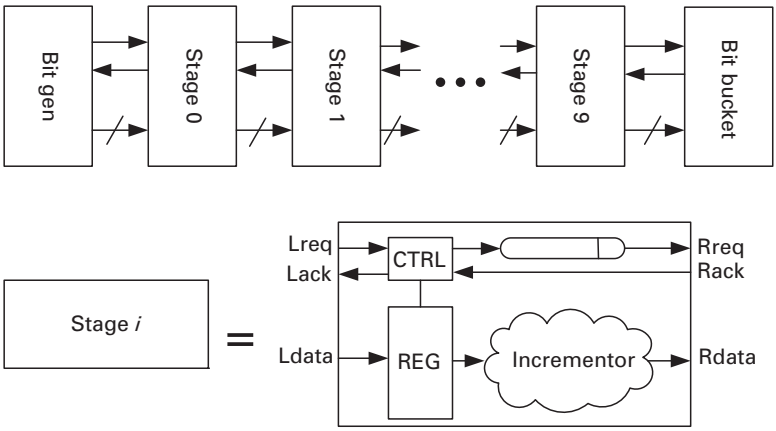

Figure 9.23. Ten-stage incrementor pipeline.

in adders in which the carry chain is implemented in domino dual-rail and the less critical sum logic is implemented in static logic. Area is thus saved and the complexity of the required completion tree is reduced [7]. A hybrid approach was also used in an iterative asynchronous divider and square root circuit in which static *selector* circuits act as the interface between dynamic dual-rail and static single-rail logic [21]. In this latter work, it was shown that dual-rail datapaths can actually save power, despite switching one output every cycle, if the significant glitching activity that would otherwise be present in the static logic alternative can be avoided.

Finally, an advanced micropipeline design style that has been proposed is the single-track two-phase controller known as GasP [16]. This will be discussed in Chapter 13, along with other single-track templates.

9.6 Exercises

9.1. Design a non-linear two-phase bundled-data pipeline for the implementation of Euclid's GCD algorithm described in Section 4.3. Use capture–pass latches and identify all non-linear pipeline stages and the associated control elements. Optionally, design each hardware component behaviorally in a hardware description language and use simulation to verify correctness. Include in your model estimated integral delays for each hardware component. Try to use a VerilogCSP testbench, and design shims to interface the bundled-data design to the CSP testbench.

9.2. Repeat Exercise 9.1 using any four-phase micropipeline template. Compare the cycle times of the two and four-phase designs.

9.3. Consider the 10-stage linear pipeline depicted in Figure 9.23. Assume an eight-bit datapath, that each incrementor has a delay of 30, that the bit

generator produces random eight-bit data tokens with zero delay, and that the bit bucket consumes data tokens with zero delay. Also, assume that all primitive gates have pin-to-pin delays equal to 1. Design a two-phase micropipeline based on the transparent latch-based register illustrated in Figure 9.3. Design the circuit with the smallest symmetric delay lines (having a chain of inverters of even length) necessary for correct operation. Design and simulate your gate-level circuit using a hardware description language such as Verilog or VHDL and include a behavioral model of the incrementor block. Add behavioral performance monitors to the circuit to measure the average cycle time. What is the average cycle time of the circuit?

9.4. Repeat Exercise 9.1 using the MOUSETRAP micropipeline scheme.
9.5. Repeat Exercise 9.1 using the simple four-phase micropipeline scheme.
9.6. Repeat Exercise 9.1 using the semi-decoupled micropipeline latch controller.
9.7. Repeat Exercise 9.1 using the fully decoupled micropipeline latch controller.
9.8. Design an asymmetric delay line with inverters and distributed NAND gates tied to the input signal. Repeat Exercise 9.7 with this asymmetric delay line.
9.9. Verify through simulation that the simple four-phase controller is a half buffer by making the bit bucket arbitrarily slow and counting the number of tokens that can enter the pipeline.
9.10. Repeat Exercise 9.9 with a semi-decoupled latch controller and verify that it is a full buffer.
9.11. Replace the symmetric delay line in Exercise 9.7 with the power-efficient asymmetric delay line shown in Figure 9.22. How much shorter is the delay line needed? What is the revised cycle time?
9.12. Design an efficient speculative completion-sensing circuitry for each incrementor. Replace the delay in the two-phase controller in Exercise 9.1 with your speculative delay design and find the average cycle time improvement.

References

[1] I. E. Sutherland, "Micropipelines," *Commun. ACM*, vol. 32, no. 6, pp. 720–738, June 1989.
[2] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods, "AMULET1: a micropipelined ARM," in *Proc. IEEE Computer Conf. (COMPCON)*, March 1994, pp. 476–485.
[3] N. C. Paver, The design and implementation of an asynchronous microprocessor, Ph.D. thesis, Department of Computer Science, University of Manchester, 1994.
[4] S. B. Furber and P. Day, "Four-phase micropipeline latch control circuits," *IEEE Trans. VLSI Systems*, vol. 4, no. 2, pp. 247–253, June 1996.
[5] K. Y. Yun, P. A. Beerel, and J. Arceo, "High-performance two-phase micropipeline building blocks: double edge-triggered latches and burst-mode select and toggle circuits," *IEE Proc. Circuits, Devices and Systems*, vol. 143, pp. 282–288, October 1996.
[6] M. Singh and S. M. Nowick, "MOUSETRAP: high-speed transition-signaling asynchronous pipelines," *IEEE Trans. VLSI*, vol. 15, no. 6, pp. 684–698, June 2007.

[7] K. Y. Yun, P. A. Beerel, V. Vakilotojar, A. Dooply, and J. Arceo, "The design and verification of a low-control-overhead asynchronous differential equation solver," *IEEE Trans. VLSI*, vol. 6, no 4, pp. 643–655, December 1998.
[8] S. M. Nowick, "Design of a low-latency asynchronous adder using speculative completion," *IEE Proc. Computers and Digital Techniques*, vol. 143, no. 5, pp. 301–307, September 1996.
[9] K. Kim, P. A. Beerel, and Y. Hong, "An asynchronous matrix–vector multiplier for discrete cosine transform," *in Proc. Int. Symp. on Low Power Electronics and Design*, July 2000, pp. 256–261.
[10] C. Molnar, I. Jones, W. Coates, J. Lexau, S. Fairbanks, and I. Sutherland, "Two FIFO ring performance experiments," *Proc. IEEE*, vol. 87, no. 2, pp. 297–307, February 1999.
[11] A. Bystrov and A. Yakovlev, "Ordered arbiters," *Electronics Lett.*, vol. 35, no. 11, pp. 877–879, 1999.
[12] A. J. Martin, "Programming in VLSI: from communicating processes to delay-insensitive circuits," in C. A. R. Hoare, ed., *Developments in Concurrency and Communication, UT Year of Programming Series*, Addison-Wesley, 1990, pp. 1–64.
[13] S. Tugsinavisut, Y. Hong, D. Kim, K. Kim, and P. A. Beerel, "Efficient asynchronous bundled-data pipelines for DCT matrix–vector multiplication," *IEEE Trans. VLSI Systems*, vol. 13, no. 4, pp. 448–461, April 2005.
[14] S. Tugsinavisut, The design and synthesis of concurrent asynchronous systems, Ph.D. thesis, University of Southern California, 2005.
[15] M. Singh and S. M. Nowick, "MOUSETRAP: ultra-high-speed transition-signaling asynchronous pipelines," in *Proc. Int. Conf. on Computer Design*, September 2001, pp. 9–17.
[16] I. E. Sutherland and S. Fairbanks, "GasP: a minimal FIFO control," in *Proc. ASYNC*, pp. 46–53, 2001.
[17] C. Farnsworth, D. A. Edwards, and S. S. Sikand, "Utilising dynamic logic for low power consumption in asynchronous circuits," in *Proc. Symp. on Advanced Research in Asynchronous Circuits and Systems*, November 1994, pp. 186–194.
[18] S. B. Furber and J. Liu, "Dynamic logic in four-phase micropipelines," in *Proc. 2nd Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, March 1996, pp. 11–16.
[19] C.-S. Choy, J. Butas, J. Povazanec, and C.-F. Chan, "A new control circuit for asynchronous micropipelines," *IEEE Trans. Computers*, vol. 50, no. 9, pp. 992–997, September 2001.
[20] T. Chelcea, G. Venkataramani, and S. C. Goldstein, "Self-resetting latches for asynchronous micropipelines," in *Proc. Design Automation Conf.*, June 2007, pp. 986–989.
[21] G. Matsubara, and N. Idem, "A low power zero-overhead self-timed division and square root unit combining a single-rail static circuit with a dual-rail dynamic circuit," in *Proc. 3rd Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, April 1997, pp. 198–209.

10 Syntax-directed translation

As discussed in Chapter 3, a common means of describing asynchronous design is with a high-level language that includes primitives for communication along channels, as described in Hoare's *Communicating Sequential Processes* [1][2]. Syntax-directed translation is a synthesis process that generates an asynchronous circuit from such a high-level language specification by translating each language construct in the specification into a hardware component. Thus the structure of the circuit corresponds closely to the syntax of its program specification.

Phillips pioneered this approach in the early 1990s with the development of the Tangram language and compiler [10]–[12]. An intermediate set of components called handshake circuits formed the building blocks of Tangram implementations. Initially, quasi-delay-insensitive (QDI) versions of each handshake component were developed [11] and, subsequently, single-rail versions; these were faster, smaller, and consumed significantly lower power [14]–[16]. This approach, referred to as *VLSI programming*, has been commercialized by an offshoot of Phillips called Handshake Solutions. A similar language and academic synthesis tool called Balsa has been developed by Manchester University [17]–[21].

The benefit of the syntax-directed approach is that it provides a tighter connection between the input language and the resulting circuits. Designers can easily map circuit bottlenecks back to the circuit design and improve the language-level specification, quickly exploring the design space by altering parallelism, pipelining, and resource sharing. The perceived disadvantage with the syntax-directed approach is the lack of optimality. However, peephole optimizations, akin to compiler optimizations, can identify common handshake circuit structures and replace them where necessary with known efficient implementations.

This chapter begins with a review of the Tangram language and its commercial incarnation Haste. A description of syntax-directed translation into a handshake circuit then follows. After that we discuss both dual-rail and single-rail implementations of handshake components, highlighting their key differences. The chapter closes with a review of successful chip designs using the syntax-directed approach, with an emphasis on its potential benefits and limitations.

10.1 Tangram

A Tangram program has the form P. (T) or L | P. (T), where P. is a set of external ports, T is a command, and L is an optional list of definitions used in P. and T. The list P defines the ports of *handshake channels* through which the program communicates with its environment. The ports can be of input or output type, to receive on send data, respectively, or they can be undirected, to support synchronization. An output port can be connected to many input ports, supporting broadcast communication. The program is defined by T.

In this section we first describe the data types for variables in Tangram and then discuss the variety of different commands that make up a program.

10.1.1 Data types

Tangram defines channels and variables and also their types, e.g. Boolean, integer, range, and structures [11]. In Haste, the type set also includes bit vectors, arrays, and enumerate types as well as both signed and unsigned data types [22]. Unlike other hardware description languages, support for explicit delays or other types of time is not included. In addition, Haste does not support C-like types of float, double or real.

10.1.2 Primitive commands

The primitive commands supported in Tangram are:

- *Assignment*, x: = E. The value of the expression E is assigned to variable x.
- *Input command*, A?x. A token on port A is received and its value is placed in variable x. The types of variable x and port A must be the same.
- *Output command*, A!E. The value of the expression E is sent as a token on port A.
- *Synchronization command*, B∼. All computations connected to the channel associated with port B are to be synchronized.

10.1.3 Composite commands

Tangram defines more complex commands as the inductive composition of primitive commands. The composition has several forms [10][11]:

- *Sequential composition*, R;S. First R is executed then S. Ports in both commands must have the same direction.
- *Concurrent composition*, R||S. The commands R and S are executed in parallel; the composite command terminates after both R and S terminate. They communicate only via channels. Thus when one command assigns an expression to a variable x, the other command cannot access x.
- *"While" looping*, do G then C od. While the guard expression G evaluates to true, repeat the command C.
- *Infinite repetition*, forever do C od. Infinitely repeat command C.

- *Conditional statements*, if B then C else D fi. If the Boolean expression B evaluates to true then execute C else execute D.
- *Block command*, |[D|S]|. Port and variable declarations in D apply to the ports in S but are concealed from the environment.

As an example, the greatest common divisor program based on Euclid's algorithm is shown below [4].

```
  int = type [0..255]
& gcd: main proc (in? chan ≪int, int≫ & out ! chan int)
begin x, y: var int
| forever do
       in?⟨⟨x,y⟩⟩    # get input values for x and y
                     # from channel in
       #  while x and y differ do the following
     ; do x<>y then
          # if x smaller than y subtract x from y
          # otherwise subtract y from x
          if x<y then y:= y−x
                 else x:= x−y
          fi
    od
          # output the resulting value of x on channel out.
     ;    out ! x
    od
end
```

Notice that the functionality of this Tangram program is equivalent to the abstract description of the greatest common divisor (GCD) specification presented in Chapter 4.

The Haste program enhances this set of commands with more sophisticated constructs, including support for input and output probes on channels and a mechanism for the user to specify whether shared resources require arbitration [22].

10.2 Handshake components

Several handshake components were described in Chapter 2, including a sequencer with a parallel component. The compiled handshake circuit for the GCD specification above is shown in Figure 10.1 [4].[1]

This circuit includes a handshake component "do," which has one passive-synchronization port, one active-input port, and one active-synchronization port.

[1] As mentioned in Chapter 2, for consistency throughout the book, when depicting an abstract channel in our figures we omit the open circle associated with the passive port typically seen in other publications on handshake circuits.

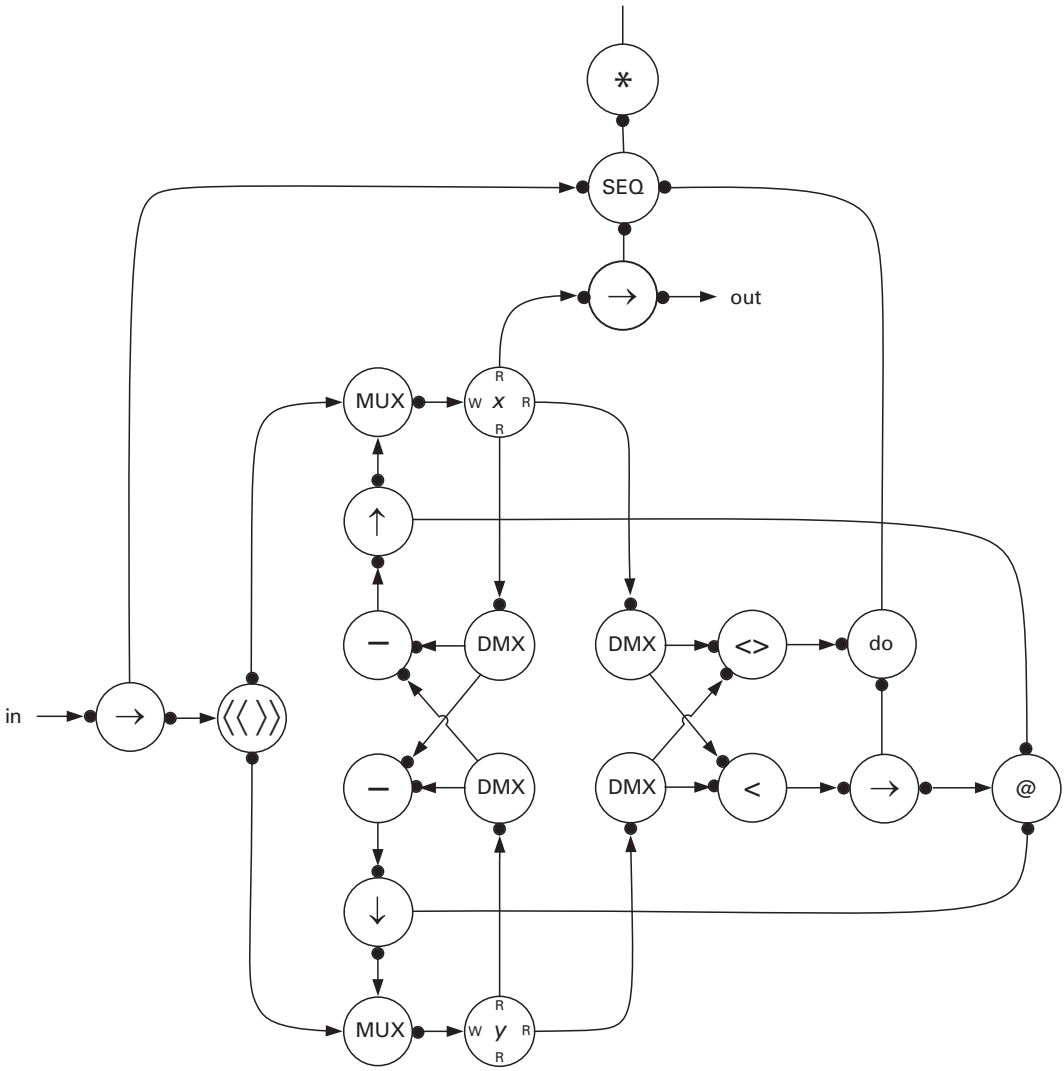

Figure 10.1. Handshake circuit for GCD program [4].

Upon the arrival of a request at its passive-synchronization port, the do component receives Boolean data on its input port. If the value is a 1, it executes a handshake on its active synchronization port and repeats by reading new data on its input port. If the value read is a 0, however, it terminates by acknowledging its passive-synchronization port. In the GCD application, the Boolean input value comes from a comparator labeled "< >" and the active synchronization port is attached to a transferer that eventually triggers the update of variable x or variable y.

The GCD handshake circuit also contains multiplexer (MUX) and de-multiplexer (DMX) handshake components, which are responsible for routing datapath

values. The multiplexer element connected to variable x accepts data inputs from the environment along one passive port and from inputs internal to the loop along its second passive port. It is the responsibility of the control to ensure that these two communications are mutually exclusive, as is the case here. The demultiplexing element simply forks the input data to multiple output channels. The handshake component, labeled "⟪ ⟫," splits the input channel 2-tuple into two distinct channels for x and y and an "@" case component implements the if–then–else statement, triggering the update of either x or y on the basis of the transferred output value of the comparator labeled "<."

10.3 Translation algorithm

The syntax-directed translation algorithm for a Tangram program can be described graphically using the structural induction of the program text. As mentioned earlier, a Tangram program has the top-level structure P.(T). If, for example, P is a single synchronization port a, the implementation has the form illustrated in Figure 10.2, where T is a compound command that will be expanded.

In one of the simplest programs possible, T is a synchronization on a channel a. In this case, the translation can be depicted as in Figure 10.3. The empty circuit is called a *connector* and encloses the handshake on its active port within the handshake on its passive port. Consequently, upon being initiated by the special channel Start the program handshakes on channel a and then terminates by acknowledging Start.

When T is a do-while-loop in which the synchronization channel a is referenced in a command C, the expansion is as depicted in Figure 10.4; G and C are expanded subsequently, depending on their structure.

When T is of the form R;S, where R and S both access the synchronization channel a, the expansion is as depicted in Figure 10.5(a) and includes a *mixer* element, indicated by a vertical line, that merges the mutually exclusive synchronization requests from R and S to channel a. Similarly, when T is of the form R||S then the common synchronization on channel a is handled using a "join" handshake component, as illustrated in Figure 10.5(b). In the join component, the handshake on channel a is enclosed by both the handshakes on its two passive ports.

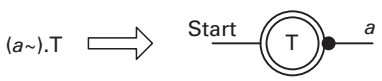

Figure 10.2. Graphical translation of top-level program.

Figure 10.3. Translation of a trivial program synchronizing on channel a.

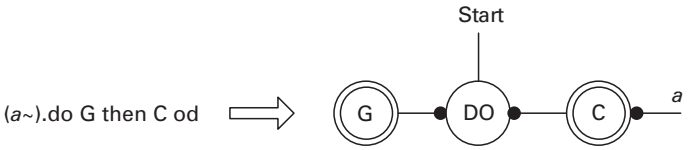

Figure 10.4. Translation of the "do-while-loop" command.

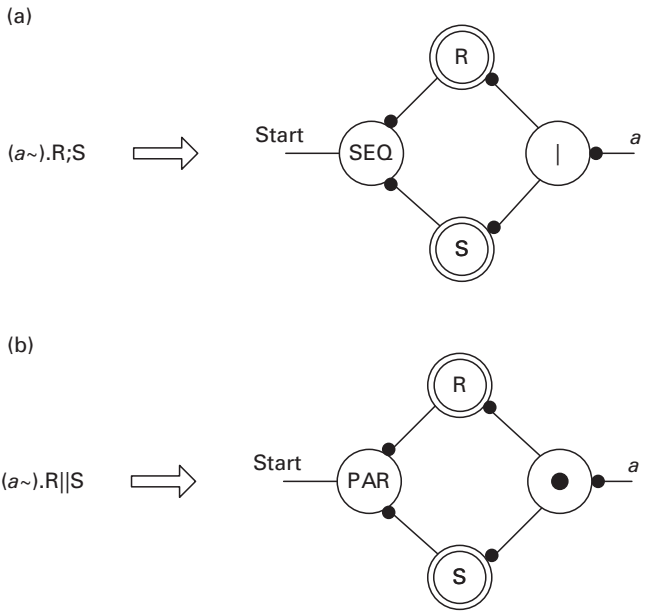

Figure 10.5. Translation of sequential and parallel commands. The double circles represent a compound command (see Figure 10.2).

A detailed expansion for the ternary semaphore Tangram program

```
(A~, B~). A~; forever do [B~ || A~] od
```

is shown in Figure 10.6 [10]. Notice that, as part of the last transformation step, the connector component is removed because it is functionally redundant.

A more formal treatment of this translation process can be found in [11].

10.4 Control component implementation

As mentioned in Chapter 2, most efficient implementations of handshake components use four-phase handshaking. Examples of four-phase implementations of the repeater, join, and mixer components are shown in Figure 10.7. (Note that the basic QDI design technique for such small components was originally developed by Martin and his students throughout the 1980s and 90s; see e.g. [3].)

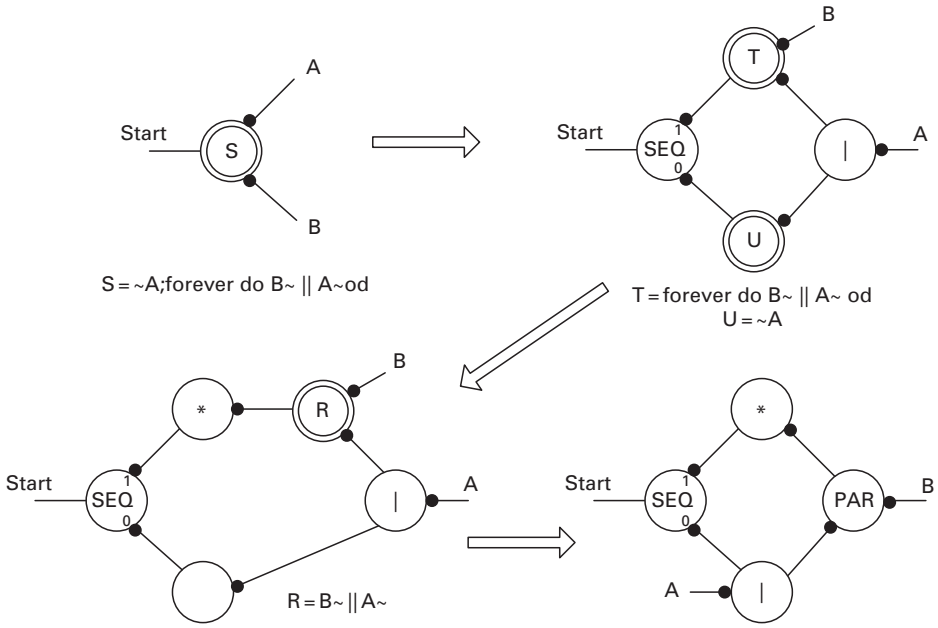

Figure 10.6. Sequence of syntax-directed translation steps for a simple Tangram program.

Sequencing and parallel operators rely on enclosed handshaking. A four-phase implementation of an S-element (also known as a Q-element [3]) that employs enclosed handshaking is shown in Figure 10.8. Initially Areq, Aack, Breq, and Back are all 0 and the output of the inverting C-element is 1. When Areq rises, it performs a complete four-phase handshake on B, before driving Aack high. (The inverting C-element goes low after Breq goes high.) This causes Areq to reset, which returns the output of the inverting C-element to a 1, which causes Aack to fall, returning the S-element to its initial state.[2]

The application of an S-element in four-phase sequencer and in parallel components is shown in Figures 10.9(a), (b).

As an example of how these components fit together, the implementation of the final handshake circuit in Figure 10.6 is illustrated in Figure 10.10.

10.5 Datapath component implementations

Both QDI and bundled-data implementations of datapath components have been developed. Initially, QDI implementations were created because there was a need for the robustness that they can provide, but bundled-data implementations lead

[2] Note that it is also possible to build an S-element with a symmetric rather than asymmetric C-element. However, the symmetric C-element would lead to SEQ and PAR elements that are not self-initializable, as defined in Section 10.7, and is thus less useful.

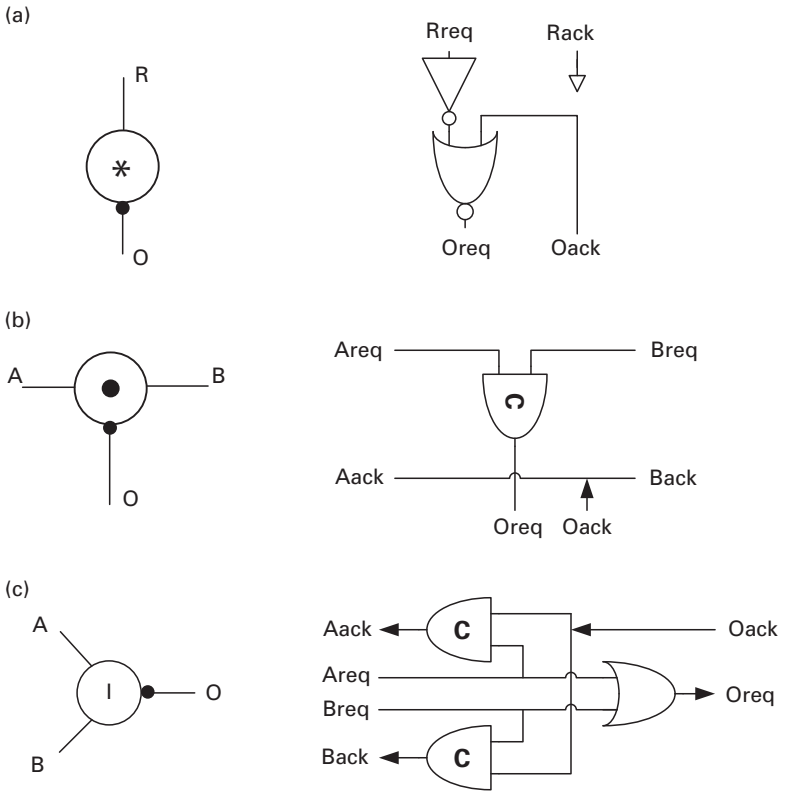

Figure 10.7. Four-phase implementations of (a) the repeater, (b) the join, and (c) the mixer components.

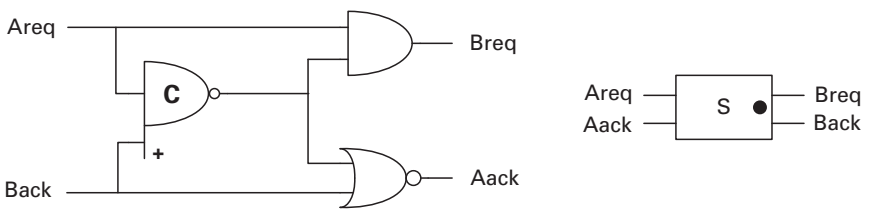

Figure 10.8. Implementation and symbol for an S-element.

to smaller, lower-power, and faster implementations. To understand the differences, we consider the compiled handshake components associated with the assignment statement $z := x \otimes y$, where $\otimes$ is an abstract binary operation, illustrated in Figure 10.11. We assume that the variable z is also written elsewhere in the program, thereby requiring the multiplexer component. Using this approach we will discuss both the QDI and the bundled-data implementations of the variable, transferer, and datapath multiplexer components and how they work together to implement this type of assignment.

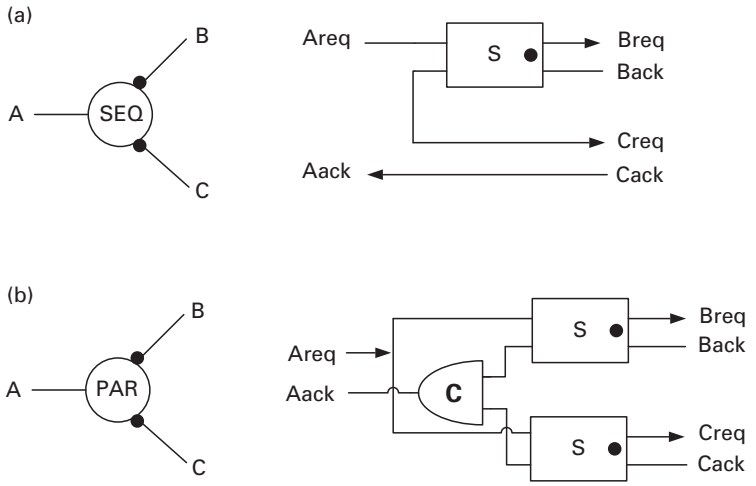

Figure 10.9. Four-phase implementations of SEQ and PAR components.

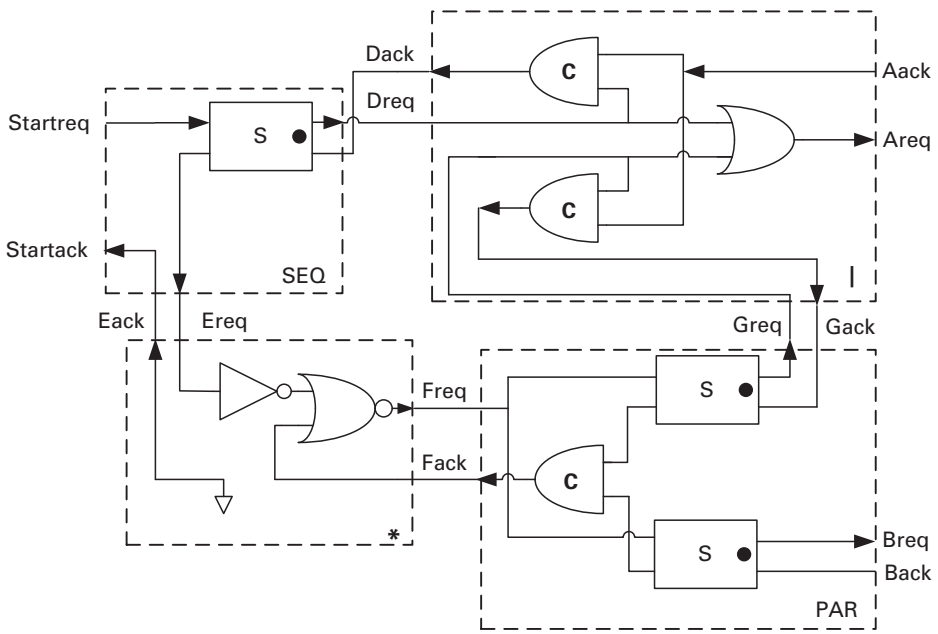

Figure 10.10. Implementation of the final compiled handshake circuit in Figure 10.6.

10.5.1 QDI implementations

Quasi-delay-insensitive arithmetic units in Tangram use dual-rail weak-conditioned implementations of the datapath. In the University of Manchester implementation of Balsa, delay-insensitive minterm synthesis (DIMS) was initially used. In this approach each minterm of each output is implemented with a

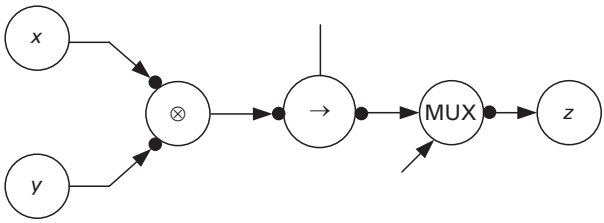

Figure 10.11. Compiled handshake circuit for a simple assignment statement.

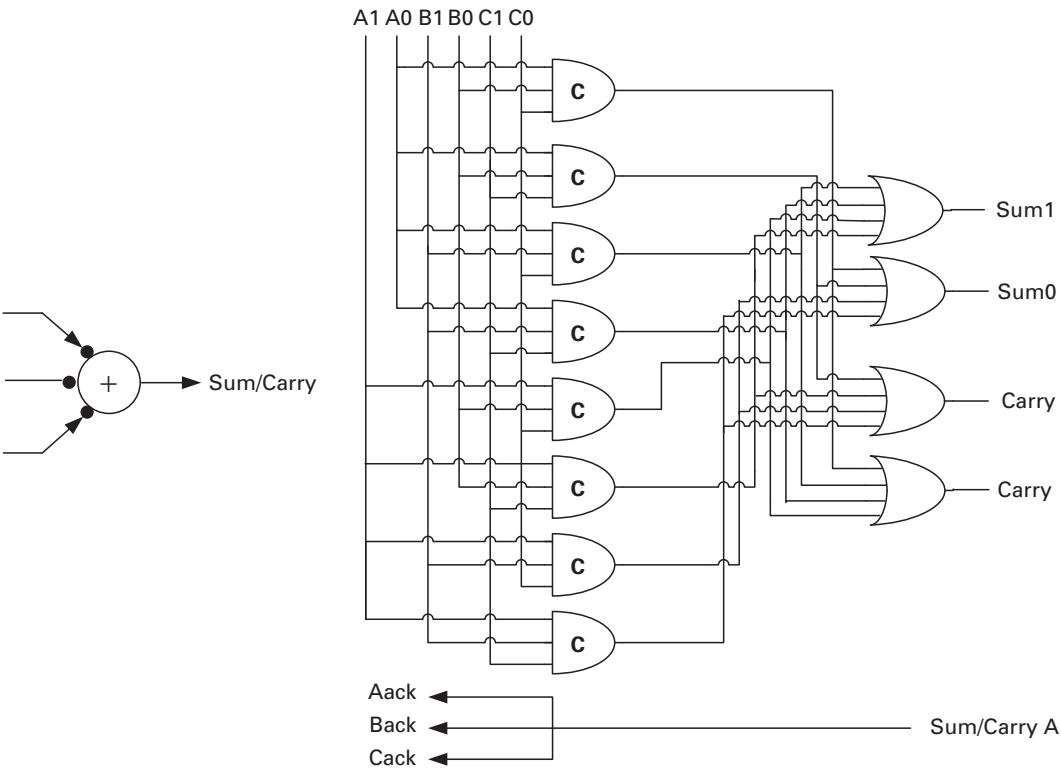

Figure 10.12. Delay-insensitive minterm synthesis implementation of a full adder.

C-element. Each C-element is then mutually exclusively triggered and all minterms associated with an output are safely combined with an OR gate. As an example, a DIMS implementation of a full-adder operator is shown in Figure 10.12.

Alternatively, speed-independent logic decomposition theory can be used to develop more efficient multi-level speed-independent combinational logic [20] [21]. Dual-rail inputs and outputs are typical, although 1-of-*N* logic blocks can also be used. The latter often reduce switching activity but can sometimes require more complex logic. For example, using 1-of-3 logic in kill–propagate–generate (KPG)-based adders can be very efficient (see e.g. the KPG tree-adder in

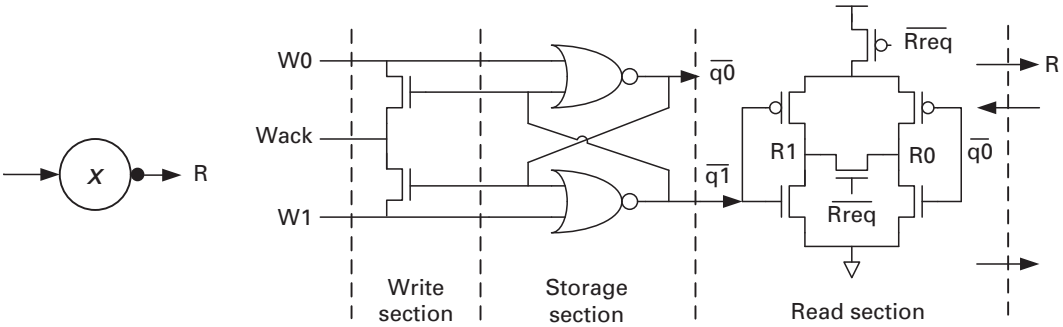

Figure 10.13. A 1-bit dual-rail handshake variable.

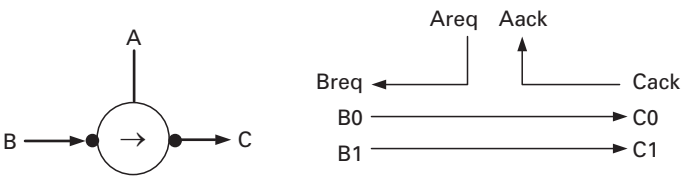

Figure 10.14. A 1-bit dual-rail transferer.

Chapter 13), whereas experiments with 1-of-4 datapaths have produced unclear power benefits [20].

Dual-rail datapaths require dual-rail variables. A one-bit variable consists solely of a one-bit dual-rail latch, as depicted in Figure 10.13 [12]. A multi-bit variable requires a number of such latches and a C-element tree to combine the "write" acknowledgements for the bit. Extensions to 1-of-4 variables can be found in [20].

A 1-bit dual-rail transferer can be implemented just with wires, as illustrated in Figure 10.14 [11]. Extending this to multi-bit dual-rail datapaths simply involves adding an additional pair of true and false wires per additional datapath bit. This means that the request and acknowledge signals are shared among all bits.

Notice that the B channel is a *pull channel* (see Chapter 2) and responds to a rising Breq by producing valid data on its dual-rail wires. The C channel, however, is a push channel and initiates the data transferer by placing valid data on its dual-rail wires. The transfer is initiated when Areq rises and it completes when Aack falls.

A dual-rail one-bit multiplexer is shown in Figure 10.15. Notice that there are two OR gates that combine the mutually exclusive input data rails into corresponding output data rails and two OR gates that act as completion-detection units for each input channel. Extending this to a multi-bit multiplexer involves adding two additional OR gates per bit in the forward path and enhancing the completion-detection units for multiple dual-rail channels. In particular, as mentioned in Chapter 7, a dual-rail completion detection unit consists of one two-input OR gate per dual-rail bit, which feeds a multi-level C-element tree.

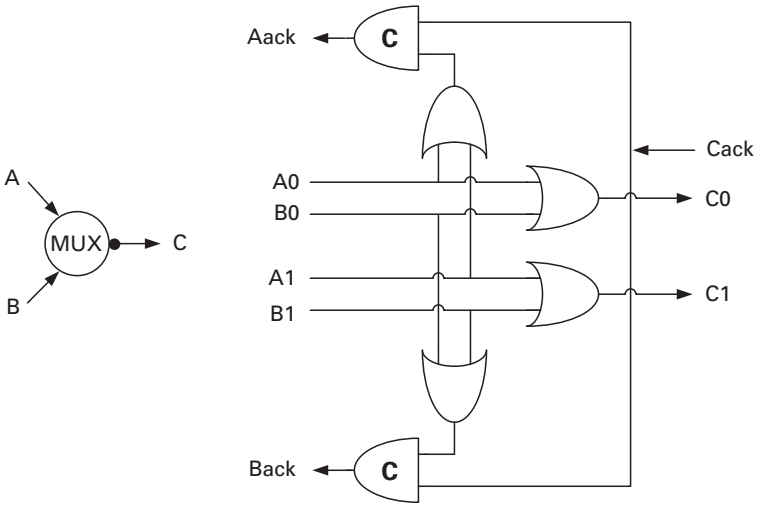

Figure 10.15. A 1-bit dual-rail multiplexer.

Now all the component associated with the assignment statement in Figure 10.11 have been described, we will discuss their operation in this context. The assignment is triggered when the transfer element requests data through the Boolean operator from both the *x* and *y* variables. The variables present valid dual-rail data to the dual-rail operator. The results propagate through the transferer and multiplexer elements and individual bits get written into their respective one-bit latches in the variable *z*. The completion of these individual "writes" propagates through a C-element tree in the variable, thus generating its acknowledgement. The acknowledge signal propagates backward through the C-element in the multiplexer and finally back to the transferer. The transferer then resets its request signal, which propagates through the Boolean operator and resets the variables *x* and *y*. This in turn resets their individual data wires to the neutral state, which propagates through the Boolean operator, through the transferer, through the multiplexer, and finally arrives at the latches. Each one-bit latch resets its acknowledgement, which then again propagate through the C-element tree to reset the variable's acknowledge signal. The reset of the acknowledge signal then propagates through the multiplexer before finally arriving at the transferer, completing the entire assignment.

It is important to note that if the assignment operation is triggered by the sequencer described in Figure 10.8 then this entire four-phase handshake must complete before the next operation in the Tangram program can begin to execute. Thus, the entire reset phase of the assignment becomes part of the program's critical path. This is in contrast with pipeline templates, in which the reset of the multi-level dual-rail datapath and the completion-sensing overhead is largely hidden from the critical path. This will be covered in Chapters 11 through 13.

Dual-rail datapaths in Tangram also suffer from area and power overheads. Dual-rail datapaths have twice as many wires as the single-rail alternatives and generally have about twice the number of transistors. Moreover, unlike single-rail datapaths, which incur no switching if the input data does not change from cycle to cycle, each dual-rail channel incurs two switching events every cycle owing to the embedded four-phase protocol. This incurs substantial power penalties.

10.5.2 Single-rail implementations

Single-rail bundled-data datapaths have been proposed as an alternative implementation of datapath handshake components as a way of addressing the overheads associated with dual-rail design. As mentioned in Chapter 7, single-rail datapaths are identical to those used in synchronous design (they both have one wire per bit) and can make use of synchronous logic synthesis and optimization tools. The single-rail datapath is coupled with request and acknowledgement wires to form the single-rail datapath handshake component with active inputs. This is illustrated for a multi-bit adder in Figure 10.16. Interestingly, the same circuitry also implements a multi-bit adder with passive inputs. The only difference is what initiates the communication – the acknowledgement or the request.

Single-rail variables can also employ synchronous storage elements. In particular, single-rail handshake variables use synchronous latches [16], as illustrated in Figure 10.17. A requirement for the single-rail handshake variable to operate properly is that the latches close before the data-validity period of the input data completes. We require this "write" data to follow the broad data-validity protocol, meaning that the data is valid from when the request goes high to when the acknowledgement goes low. This guarantees that the data is valid until after the latch closes. In addition, note that the "read" acknowledgement is immediately generated upon a "read" request. The reason is that in Tangram the "read" and "write" actions on variables are guaranteed to be mutually exclusive; thus, upon a "read" request, the "read" output data is guaranteed to be stable.

Many types of standard latch can be used in a single-rail handshake. Two common latches are shown in Figure 10.18. The latch shown in Figure 10.18(b)

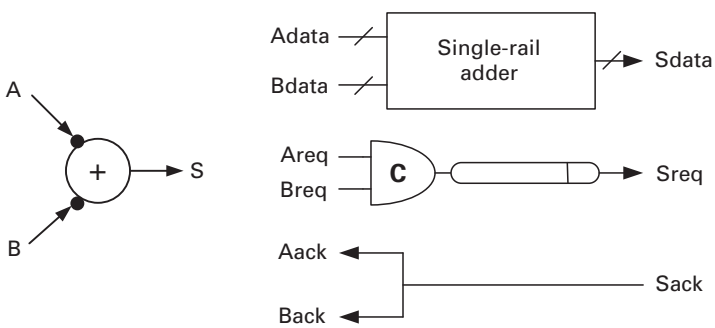

Figure 10.16. Bundled-data handshake component for a multi-bit adder.

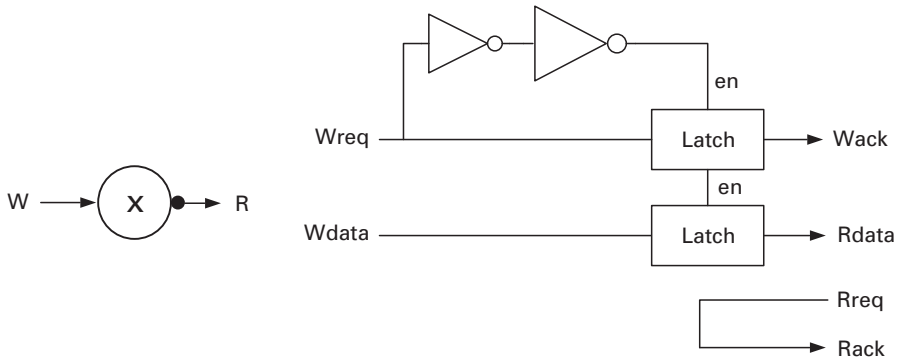

Figure 10.17. Single-rail handshake variable.

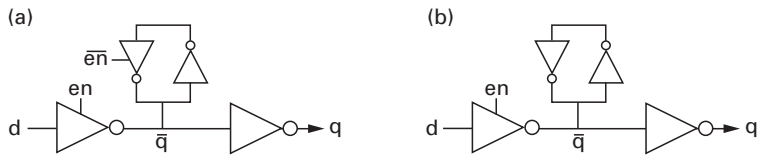

Figure 10.18. Two different implementations of a one-bit latch.

has less capacitive load on the enable signal than its alternative, shown in Figure 10.18(a), but it has a staticizer on the internal $\bar{q}$ signal that must be overcome during a write. Both latch implementations are substantially smaller and lower-power than the dual-rail counterpart shown in Figure 10.13.

The data validity scheme for the pull output channel of the latch deserves some additional discussion. The data is valid as soon as the latch closes upon a write action and becomes invalid at the beginning of a second write action. Because the write and read actions on a handshake variable are guaranteed to be mutually exclusive, the read data-validity signal is provided by the read acknowledge signal going high (Rack+); the data is then valid through the reset of the acknowledgment (Rack−) until a subsequent write action occurs. This is shorter than the broad protocol discussed in Chapter 2, which requires the data to be valid until after the next read request (Rreq+). Consequently, this protocol is called the *reduced-broad-validity protocol*.

The wire-only transferer shown in Figure 10.19 converts the reduced-broad-validity protocol on its input channel B into a broad validity protocol on the push channel on C. That is, the data on C is valid from Creq+ to some time after Cack− (which enables a subsequent write on B).

A single-rail multiplexer is illustrated in Figure 10.20. The datapath driving the n single-rail output rails is similar to a synchronous, MUX shown on the left for comparison. The select signals As and Bs are generated via cross-coupled NOR gates, which store the select signal value in between handshakes. The output request on channel C is generated with the same AND–OR circuit as the datapath

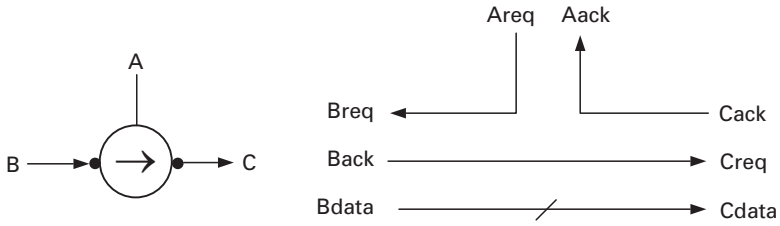

Figure 10.19. Wire-only implementation of a single-rail transferer.

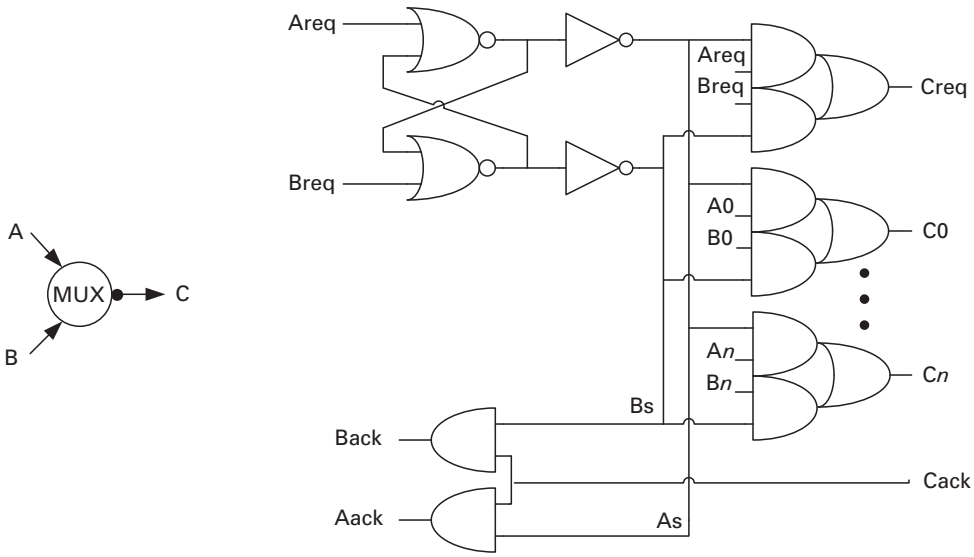

Figure 10.20. Single-rail two-input *n*-bit multiplexer.

bits and goes low when the input requests go low. Two AND gates are responsible for routing channel C's acknowledge signal back to the requesting channel on the basis of the stored select signal. Note that if the input data channels satisfy the broad validity protocol then the data on the output channel C also satisfies the broad-data validity protocol, as needed by the single-rail latches described above. More specialized versions optimized to fast-forward the output request can be found in Peeters' thesis [16].

As with the dual-rail implementations, we now discuss the operation of these single-rail component implementations in the context of the assignment circuit shown in Figure 10.11. The assignment is triggered by the transferer, which routes the request through the datapath element to the single latches. Because the variables are guaranteed to have valid data, the data is known to be valid at the outputs and the requests can be converted into acknowledgements by a simple wire. The acknowledgement then goes through the symmetric delay line in the datapath operator. The result passes through the transferer, sets up the select signal in the

multiplexer, and continues on to the destination variable, opening its latches. The latches raise the acknowledge signal, which propagates back to the transferer. The transferer then lowers its request, which propagates through the single-rail operator and resets the read request to the latches. The data is left untouched but the read acknowledge signal is reset. This reset goes through the single-rail variable for a second time and propagates through the transferer to the multiplexer. In particular the request to the multiplexer resets, which causes the output request of the multiplexer to fall. The internal select signals of the multiplexer, however, maintain their state owing to the cross-coupled NOR gates. This falling request propagates to the output variable, which responds by latching the output data and resetting its acknowledgement. The acknowledge signal propagates back through the multiplexer and terminates the entire sequence at the transferer.

This protocol has multiple advantages. First, the sum of the two phases of the delay line can be matched against the delay of the datapath operator. This means that a symmetric delay line of approximately half the delay of the datapath operator can be used. This saves area and power compared with bundled-data schemes, in which only one phase of the delay line is matched against the entire pipeline. In comparison with a dual-rail datapath, the overhead includes some margin on the delay lines but does not involve any completion sensing or propagation of neutral data through the datapath and multiplexers. The second advantage is that during the first half of the four-phase protocol the select lines to the multiplexer are set up and the latch is opened in preparation for the data. This implies that glitches that occur earlier do not propagate through the multiplexer, saving some glitching power. Third, this enables the latches to be closed normally, without an overly complex write protocol involve both the opening and the closing of the latch when a write request transition is sensed. Because all phases of the four-phase protocol do useful work, Peeters called this protocol the *true-four-phase protocol*.

One performance disadvantage common to both the single- and dual-rail implementations is the delay between the writing of the source variables and the triggering of the transferer to start the assignment process. Attempts to reduce this overhead by creating efficient sequencers and reducing the size of the delay line have provided an interesting area of future work [23]. That said, a side effect of this delay overhead is that most glitches in the datapath are likely to have settled before the multiplexer select lines have been set, reducing the glitching power through the multiplexer and subsequent datapath components.

10.6 Peephole optimizations

Two types of peephole optimization have been formalized by van Berkel [40]. In the first "equals replace equals", i.e. cheaper handshake components replace more complex handshake components. The removal of the connector in Figure 10.6 is an example of such an optimization but, given that the connector is implemented with wires only, the advantages evaporate during the layout phase [11]. A more

pertinent example is a tree of mixer, sequencer, or parallel components. Balancing such trees can improve average performance with no other cost or change in external behavior. In addition, optimized implementations of multi-way sequencers have also been explored [32].

In other cases, peephole optimization involves replacement by cheaper subcircuits with "almost equal" behavior, when the effect on the external behavior cannot be observed by any possible environment. This is called *refinement-in-context*. An important refinement-in-context in single-rail implementations of datapath components is the removal of redundant C-elements associated with Boolean operators in assignments. Because of the mutual exclusivity between reads and writes in the pull protocol, the C-element associated with binary operators is driven by a single source. For example, in the single-rail implementation of the handshake implementation shown in Figure 10.11, the C-element associated with the binary operand $\otimes$ is redundant. The two requests come from a single source component, in this case the transferer in, and the associated control delay associated with the two inputs is not required for correctness. Consequently, the C-element can be removed and the delay line remains triggered by the transferer.

10.7 Self-initialization

All VLSI circuits are designed to be driven to a known valid state upon power-up of the IC. Synchronous circuits achieve this using a special reset signal that forces specific flip-flops to a known state. Many asynchronous pipeline templates similarly require a reset signal to go to every pipeline stage and force the internal state-holding elements. Tangram circuits embody a more elegant approach by being *self-initializable* [11]. They enter a known initial state when the electrical inputs of the circuit are forced low – *no extra reset circuitry is needed*.

This self-initialization property comes from the fact that the *activity graph* of a handshake circuit is acyclic and all component realizations have two key properties. The activity graph is the directed graph obtained from a handshake circuit by replacing the components by nodes and by introducing a directed arc from one node to another if there is a corresponding handshake channel for which the component of the first node is active and that of the second node is passive. Its acyclic nature can be proven using properties of the translation algorithm [11]. For single-rail datapaths, the two handshake component properties are as follows: (i) when the (request) inputs of the passive ports are low, the (request) outputs of all active ports go low; (ii) when all inputs (active acknowledge signals and passive requests) are low, all outputs go low [16]. Initialization is caused by a ripple effect from the primary inputs to all parts of the circuit. At each step, there exists either a component all of whose passive request input ports are low, which therefore lowers its active request outputs, or a component all of whose inputs are low, which therefore lowers all its outputs.

As an example, the initialization sequence for the circuit shown in Figure 10.10 is shown in Figure 10.21. After the lowering of the Startreq signal (at time 0),

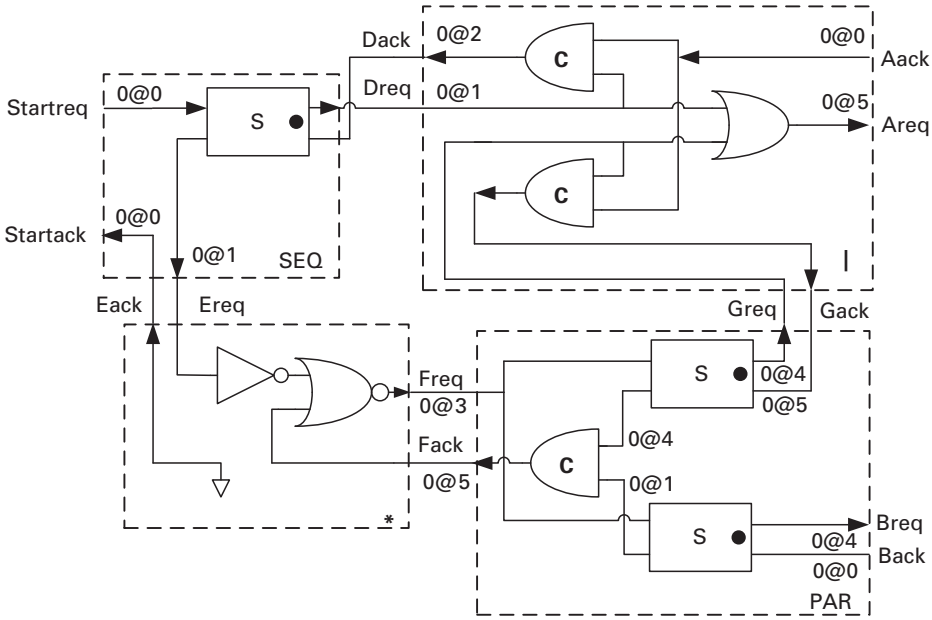

Figure 10.21. Reset sequence for the circuit illustrated in Figure 10.10. The symbols 0@0 mean "low at time 0," etc.

which is the active input to the SEQ component, the two active SEQ outputs go low (at time 1). The Ereq signal is the only active input to the repeater component; thus its active output request Freq goes low (at time 3). This is the only active input of the PAR component, which thus lowers both its output requests (at time 4). One of these outputs, Greq, is the last input to the mixer to go low, which forces both its outputs, Aack and Gack, to go low (at time 5). Concurrently Freq goes low, and this causes the unlabeled request output of the top S-element in the PAR block to go low (at time 4). This forces the PAR block's C-element finally to reset, causing Fack to go low (at time 5). Note that the reset ripple effect starts with a sequence of active output requests that go low and then completes as the remaining outputs go low. On the basis of this ripple effect, it has been shown that the length of time taken by the initialization process is proportional to the longest acyclic path in the activity graph [11].

A more general formalization using *weak and strong initialization* properties of handshake components leading to self-initialization (which capture properties of QDI datapaths) is also described in [11].

10.8 Testability

Quasi-delay-sensitive circuits have the natural testability advantage that most stuck-at faults on gate outputs cause such circuits to deadlock [24]. However,

stuck-at faults on gate inputs as well as other types of faults, such as bridging faults, can cause more unpredictable behavior. Moreover, this testability advantage does not carry over to single-rail datapaths. Both scan [30] and current-based IDDQ testing [29] have been explored to test Tangram circuits. This section focuses on the more recently proposed scan-chain approach, which involves adding design-for-test (DfT) circuitry that makes the Tangram circuit look synchronous in test mode and adopts synchronous full-scan solutions with the goal of getting close to 100% stuck-at-fault coverage [26]–[28].

In particular, asynchronous circuits can be characterized as combinational logic circuits with feedback loops of two types. First, there are the feedback loops implicit in asynchronous memory elements such as C-elements. Second, there are explicit cycles among the gates. Both types of cycle cause problems with synchronous test analysis and automatic test pattern generation (ATPG) tools designed for synchronous flip-flop or latch-based circuits in which the combinational logic is acyclic. One approach to address this issue is to break all loops with scan-chain latches or flip-flops acting as buffers during normal operation. Essentially this enables the asynchronous circuit to look like a synchronous circuit during test.

Adding latches to break explicit cycles among the gates in Tangram circuits is straightforward since the number of these loops tends to be relatively small. However, given the large number of C-elements in a Tangram circuit, making scannable C-elements has been accorded special attention [27]. A naïve approach involves building C-elements out of combinational standard cell logic with a feedback path, as described in Chapter 7, and inserting the scan elements in the feedback path. This has three disadvantages. The first is that the scan elements increase the delay of the feedback path, making it harder to ensure the fundamental-mode assumption necessary for correct operation. The second is that the scan circuitry delays the output of the C-element, incurring a performance penalty. The third is that since the number of C-elements necessary in the control of Tangram circuits is large, as mentioned above, the resulting impact in area is quite significant.

Beest *et al*. proposed a more elegant approach to break C-elements, illustrated in Figure 10.22, in which a MUX element is placed in the feedback latch and a scan latch is attached to the C-element output [28]. During normal operation the scan enable (SE) connects the MUX output to the output of the C-element function block, completing the feedback path. During the scan-in operation, the output signal Z changes as the MUX is set to pass the scan input to the output. During the subsequent capture cycle, the scan enable is lowered and the output of the C-element value is stored in the additional latch. The properties of the C-block ensure that the lowering of the scan enable signal does not change its output value during the capture cycle, so that it is properly stored by the attached latch. In particular, if on the one hand the C-block is initialized in a *set on reset state* then its output value is independent of the feedback signal and thus guaranteed not to change when SE is lowered. Thus the value of the C-block will propagate to the output Z and the input of the attached latch. If, on the other hand, the C-block is in a *state-retention state* then its output signal value is the same as

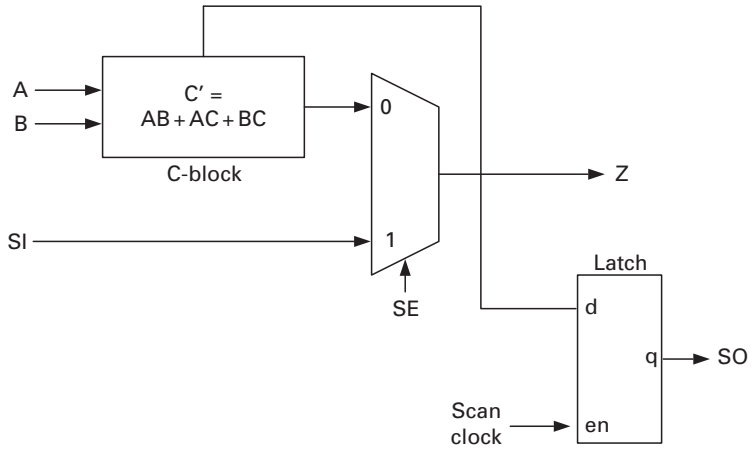

Figure 10.22. MUX-based scanned C-element [28].

the value on the feedback line. This means that the C-block output is the same as the last scanned-in value SI as well as the same as the value of Z. Thus, when the scan enable is lowered, this value is also input to the attached latch. In both cases the value of Z after initialization is properly stored in the latch. This argument, however, is based on the assumption that the inputs A and B to the C-block do not change during the capture cycle, so that the C-element remains in the same state as that in which it is initialized. This assumption is guaranteed by connecting the multiplexers of MUX-based C-elements connected in series to different scan enable signals, since these are never simultaneously active during any test [28].

Once all the cycles and C-elements have been broken with latches, a Tangram design looks similar to a latch-based synchronous circuit, the difference being that single-rail datapaths have locally generated latch enables. Given that the latch enables are relatively easy to control with a modest amount of test circuitry [27], the synchronous techniques used for latch-based circuits can be readily applied to these datapaths.

In particular, one of the most robust forms of testing synchronous latch-based synchronous designs is level-sensitive scan design (LSSD). An LSSD element contains two latches L1 and L2. These latches are connected in a scan chain where the output of an L2 latch is connected to the scan input of the next L1 element. The L1 latch implements the observability (read) function required for testing, whereas the L2 latch is used to control (i.e. write) the data of a feedback loop and the next element in a chain of such elements. During testing, a two-phase non-overlapping clock is applied to the two latches, and the circuit will behave as a normal clocked circuit. During functional mode the L1 element operates as the normal datapath latch. The operation during testing is independent of the rise time, fall time, or minimum delay of the circuit.

In the LSSD approach the L2 latches are added for scan purposes and are not otherwise needed during normal operation. Thus they represent DfT overhead

that should preferably be minimized. An improvement upon this method is called the L1L2* method [25][26], in which a subset of the L2 latches can be made from existing latches in the datapath as long as different functional blocks are tested independently. In particular, the read input latches to each functional block must be of the same latch type and the write outputs latches of a functional block must be of the same latch type. This allows all inputs to a functional block to be read on the same clock phase and latched on the same clock phase. The scan chain can thus be created by sequencing latches of different types to form master–slave alternating latch types. Optimal methods for ordering the scan chains for single-rail datapaths (including adding dummy L2 latches where necessary) that minimize test time have also been developed [31].

This work has demonstrated that the LSSD approach can yield high test coverage using synchronous scan and ATPG tools, with test-area overheads between 20% and 40% for a variety of large Tangram designs [28]. Future enhancements have been identified that are expected to reduce the overhead further. However, finding a comprehensive DfT strategy that can approach the 5% to 10% area overhead typical in synchronous design remains a research challenge.

10.9 Design examples

We will discuss four representative large design examples, two of which were implemented with Tangram and two implemented with Balsa. The latter is described in detail in the Ph.D. thesis of W. B. Toms [20].

10.9.1 Tangram digital compact cassette error corrector

The first example is a digital compact cassette (DCC) error corrector design. In the DCC player, parity symbols are recorded on tape to protect the audio information against tape errors. During play mode, a Reed–Solomon error detector accepts code words of 24 or 32 symbols (eight bits each), including four or six parity symbols. Each symbol has an associated one-bit erasure flag indicating the known error status of that symbol.

Let e, r, and p denote the number of errors, erasures, and parity symbols respectively. The detector is required to output the positions and values of the e errors and the r erasures, given that $2e + r \leq p$. For a formal specification of the detector see [33]. The DCC application specifies a performance of 3000 code words of length 24 and 2300 code words of length 32 per second. This implies an average input rate of 145 600 symbols per second.

Two synchronous IC baselines exist. The first operates on a 6.14 MHz clock and uses a ROM-based centralized controller and a small RAM to store intermediate results. The ROM and RAM together account for more than 50% of the power dissipation. In the second, a successor of the first, the clock frequency is halved

Table 10.1. Single-rail, dual-rail, and synchronous DCC error correctors [16]

Quantity	1-rail 1 (IC)	1-rail 2 (Layout)	2-rail 1 (IC)	2-rail 2 (Layout)	Sync 1 (IC)	Sync 2 (IC)
Number of transistors (1000s)	21.8	20.3	44.0	30.8	—	—
Core area (mm^2)	4.5	3.9	7.0	5.9	3.4	3.3
Time (ms)	50	40	83	55	—	—
Power (mW)	0.5	0.35	2.3	0.8	12	2.7

and the RAM eliminated. Furthermore, by means of clock gating (in the enabling of the ROM), the ROM storage power is further reduced to 6%–10% of the entire design.

Two single-rail designs were generated, the first to a working IC and the second to a layout that removed unnecessary single-to-dual-rail interfaces, included more peephole optimizations, and was substantially improved. Two dual-rail designs were also generated. The first was a QDI design and used a dedicated cell library. The second used some timing assumptions (extended isochronic forks), a generic cell library, and an optimized Tangram program that reduced the activity factor for correct code words by about 50%. All asynchronous designs do exhibit data-dependent delay but they all easily meet the target performance specification with worst-case data. The DfT strategy in both single-rail designs was based on a partial-scan approach that does not achieve 100% stuck-at-fault coverage but has only 4% area overhead. Design-for-test circuitry was not added to the dual-rail designs.

Table 10.1 shows that while the single-rail circuit is 20% larger than the synchronous version it consumes only 15% of the power. Compared to the best double-rail version, the best single-rail version is 33% smaller, 25% faster, and requires 50% less power.

The dramatic power savings of the single-rail circuit in comparison to the best synchronous design can be attributed to two main sources. The first is the use of single-rail latches over flip-flops, which, in synchronous circuits, has been shown to yield a 20% savings in power. The second is that distributed and controlled data movement and storage can provide even lower power consumption than optimal gated clocking. In particular, the best synchronous design operates at 3.1 MHz, which is over 30 times the input symbol rate of about 150000 symbols per second. This implies that with perfect clock gating a large fraction of registers are gated over 95% of all clock cycles. But achieving perfect clock gating is difficult because the clock enable circuitry can be quite complex and, even during gated cycles, a significant amount of power is still consumed in the shared clock tree as well as in the individual gating logic. This residual power consumption can be substantial and is completely avoided by the distributed nature of asynchronous control.

Table 10.2. Single-rail and dual-rail implementations of the SAMIPS design [20]

	Single-rail	Dual-rail
Area (mm)	0.9	2.1
Speed (MIPS)	15.8	6.8
Energy (μJ)	0.69	2.63

10.9.2 Balsa SAMIPS – an asynchronous MIPS R3000 processor

The synthesizable asynchronous micro-processor without interlocked pipeline stages (SAMIPS) is an asynchronous implementation of the MIPS R3000 processor and consists of a five-stage pipeline implemented using the Harvard architecture with two memory ports. The five stages are: (1) an *instruction–fetch* stage, which fetches instructions and updates the PC; (2) an *instruction–decode* stage, which contains the instruction decoder and thirty-two 32-bit general-purpose registers and which implements control forwarding to resolve data hazards within the pipeline; (3) an *execute* stage, which contains the ALU (including a multiplier–divider), a shifter, the forwarding unit, and a functional unit to determine whether branches are taken; (4) a *memory* stage, which interacts with the memory unit; and (5) a *write-back* stage, which writes the data back to the general-purpose registers and supplies data to the forwarding unit if there is a data hazard.

Estimated sets of figures for single- and dual-rail implementations are analyzed in [20] and illustrated in Table 10.2. The dual-rail area and energy overhead values can be explained by the large dual-rail completion-sensing and storage elements. In particular, the completion and storage categories account for 56% of the transistors and 76% of the energy consumption in the dual-rail SAMIPS, as compared with 21% and 26% in the bundled-data implementation. The performance penalty of the dual-rail design can be attributed to its channel-wide completion sensing of valid and neutral data and its propagation of neutrality through the dual-rail datapath that arises in the enclosed-handshaking-based implementation of assignment statements, as described in subsection 10.5.1. No comparison with a synchronous baseline was made, however.

10.9.3 Balsa SPA – an asynchronous ARM V5T processor

A synthesizable processor amulet (SPA) core is an ARM V5T compatible core designed to reduce the susceptibility of a circuit to power-analysis attacks by eliminating data-dependent changes in the power consumption of the circuit. In order to make the operation data-independent, SPA was implemented in dual-rail, where the number of transitions required to transmit a data word is the same for all data words. The logic was implemented using "balanced" DIMS, where "dummy loads" are added to make the OR gate networks equal for all outputs. The SPA core also employed *return-to-zero storage,* to eliminate power consumption differences

Table 10.3. Single-rail and dual-rail implementations of the SPA design [20]

	Single-rail	Dual-rail
Area (mm)	1.4	2.6
Speed (MIPS)	4.3	3.8
Energy (μJ)	0.78	4.2

that arise when data sometimes overwrites identical data and sometimes overwrites different data.

The SPA core is a three-stage pipelined Harvard architecture design that includes (i) a fetch unit that fetches instructions from memory and controls the flow of the processor, (ii) two instruction decoders, one for regular ARMV5 instructions and another for the compressed 16-bit Thumb® instructions, and (3) an execute unit consisting of a memory access unit, a register bank, and a functional-execution unit.

Estimated sets of figures for single- and dual-rail SPA implementations are analyzed in [20] and are illustrated in Table 10.3. As with the SAMIPS design, the substantial area overhead for the dual-rail version can be explained by the large completion-sensing and storage elements. They account for 52% of the transistors and 64% of the energy consumption of the dual-rail SPA, as compared with 14% and 26% for the bundled-data implementation. The performance penalty of the dual-rail design is still significant (13% slower) but is not as great as in the SAMIPS case. As in the SAMIPS study, no comparison with a synchronous baseline was made.

10.9.4 Haste ARM996HS: a commercially available asynchronous ARM core

The ARM996HS is a commercially available asynchronous ARM9E core developed in a partnership between ARM and Handshake Solutions [34]. It is based on the ARMV5TE instruction set architecture, which includes the 16-bit Thumb® and 32-bit ARM instruction sets. The Harvard-architecture core is based on a five-stage integer pipeline with a fast 32-bit multiply–accumulate block. It has separate instruction and data memories, each up to 4 MB. Dual synchronous buses provide the instruction and data interfaces. Specific security enhancements for the ARM996HS core include a memory protection unit (MPU) and the provision of non-maskable interrupts (NMI).

The ARM996HS operates up to a maximum speed of 77 MHz. For comparison purposes, a synchronous ARM968E-S counterpart was synthesized for 100 MHz in the same Sage-X 0.13 micron TSMC process and run at 77 MHz. A performance, power, and area comparison can be made using Table 10.4. It shows that the areas of the two cores are similar and the power efficiency of the asynchronous ARM is close to three times better than its synchronous counterpart. An important secondary advantage of the asynchronous ARM is the 2.4 times lower peak throughput, which results in substantially lower electromagnetic interference. In particular, the asynchronous ARM reduces peaks by 25 dB in the wireless spectrums, ensuring that there

Table 10.4. Commercially available ARM 9E cores [34]

	ARM996HS	ARM968E-S
Area (gate count)	89K	88K
Frequency (equiv. MHz)	50 (worst-case) 77 (nominal-case)	100
Performance (DMIPS)	54 (worst-case) 83 (nominal-case)	107
Power (mW/equiv. MHz)	0.045	0.13

is no interference with the common GSM and Bluetooth receivers. A significant disadvantage of the asynchronous version, however, is that it has a lower performance than that achievable by synchronous alternatives and thus can be used only for relatively low-performance applications.

10.10 Summary

Tangram QDI circuits are very robust to process variation and provide efficient distributed control of data movement and storage, enabling different portions of a chip to operate at their ideal data rates. Thus Tangram circuits consume power only where and when needed. The dual-rail DCC error corrector demonstrates substantially lower power than comparable synchronous designs. However, the performance and area overheads provide substantial room for improvement. In fact, single-rail Tangram designs consume less than half the area and less than half the power of their dual-rail counterparts and have significantly higher performance. Moreover, using conservatively designed delay lines, the single-rail designs have demonstrated robustness characteristics initially attributed to only QDI designs and they operate correctly over a large range of voltage supplies. In addition to providing significant low-power benefits to a variety of applications, the designs also have far smaller peak currents and thus substantially lower electromagnetic emissions.

One of the most powerful reasons why the Tangram approach obtains these advantages is that the syntax-directed CAD flow enables low-level circuit bottlenecks to be mapped easily to specific characteristics of the high-level language specification. This enables an efficient exploration of different architectural choices, including alternatives in parallelism and pipelining, and the resulting tradeoffs in achievable power and performance. The one disadvantage of Tangram is that the proprietary nature of the language creates a barrier of adoption to synchronous designers accustomed to using the existing hardware description languages such as Verilog and VHDL. In addition, the circuits demonstrated have yielded relatively low performance. Finding methods to reduce the performance overhead associated with handshake circuits is an area of active research.

It may also be worthwhile to emphasize that dual-rail Tangram implementations are slower than their single-rail static counterparts and this fact is somewhat counter-intuitive. In both synchronous design and other asynchronous templates designed for fine-grain pipelining, the opposite is true. In particular, compared with

synchronous counterparts, a factor 2 improvement in speed is typically associated with dual-rail logic when implemented with dynamic logic families such as dual-rail domino. Moreover, dual-rail implementations can avoid the extra margin associated with conservatively set delay lines and flip-flop on latch setup times [6]. In Tangram, however, these raw latency improvements appear to be overshadowed by the overhead associated with the completion logic of multi-bit dual-rail channels and the reset delay through the dual-rail logic. We will explore how other asynchronous pipeline templates overcome these limitations in later chapters.

10.11 Exercises

10.1. Identify the isochronic forks in the S-element shown in Figure 10.8. Explain what goes wrong if the isochronic fork assumption is violated.
10.2. Create an un-optimized handshake circuit for a two-stage buffer. Then, using single-rail implementation, implement all the handshake components and analyze the cycle time of the circuit.
10.3. Attempt to use peephole optimizations to transform the two-stage buffer into something similar to a micropipeline-based implementation. Show your working and include an analysis of the new cycle time. Show how the optimized circuit remains initializable.
10.4. Implement the greatest common divisor (GCD) circuit in Figure 10.1 using a single-rail implementation of all handshaking components. Optimize it using peephole optimizations.

References

[1] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.
[2] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, August 1978.
[3] A. J. Martin, "Programming in VLSI: from communicating processes to self-timed VLSI circuits," in *Proc. UT Year of Programming Institute on Concurrent Programming*, March 1987, Addison-Wesley, 1991.
[4] J. Kessels and A. Peeters, "DESCALE: a design experiment for a smart card application consuming low energy," in *Principles of Asynchronous Circuit Design, A Systems Perspective*, J. Sparso and S. Furber, eds., Kluwer Academic, 2001.
[5] K. van Berkel, F. Huberts, A. Peeters, "Stretching quasi delay insensitivity by means of extended isochronic forks," in *Proc. 2nd Working Conf. on Asynchronous Design Methodologies*, May 1995, pp. 99–106.
[6] N. Weste and D. Harris, *CMOS VLSI Design, A Circuits and Systems Perspective (3rd Edition)*, Addison Wesley, 2005.
[7] J. M. Rabaey, *Digital Integrated Circuits – A Design Perspective*, Prentice-Hall, 1996.
[8] C. L. Seitz, "System timing," in *Introduction to VLSI Systems*, C. A. Mead and L. A. Conway, eds., Addison-Wesley, 1980.

[9] J. Kessels and A. Peeters, "The Tangram framework: asynchronous circuits for low power," in *Proc. Asia and South Pacific Design Automation Conf.*, pp. 255–260, February 2001.
[10] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij, "The VLSI-programming language Tangram and its translation into handshake circuits," in *Proc. European Conf. on Design Automation (EDAC)*, 1991, pp. 384–389.
[11] K. van Berkel, *Handshake Circuits: an Asynchronous Architecture for VLSI Programming,* International Series on Parallel Computation, vol. 5, Cambridge University Press, 1993.
[12] K. van Berkel, R. Burgess, J. Kessels, M. Roncken, F. Schalij, and A. Peeters, "Asynchronous circuits for low power: a DCC error corrector," *IEEE Design & Test of Computers,* vol. 11, no. 2, Summer 1994. pp. 22–32.
[13] H. van Gageldonk, K. van Berkel, A. Peeters, D. Baumann, D. Gloor, and G. Stegmann, "An asynchronous low-power 80C51 microcontroller," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, April 1998, pp. 96–107.
[14] A. Peeters and K. van Berkel, "Single-rail handshake circuits," in *Proc. 2nd Working Conf. on Asynchronous Design Methodologies,* May 1995, pp. 53–62.
[15] K. van Berkel, R. Burgess, J. Kessels, *et al.*, "A single-rail re-implementation of a DCC error detector using a generic standard-cell library," in *Proc. 2nd Working Conf. on Asynchronous Design Methodologies*, May 1995, pp. 72–79.
[16] A. M. G. Peeters, *"Single-rail handshake circuits,"* Ph.D. thesis, Eindhoven University of Technology, 1996.
[17] A. Bardsley and D. A. Edwards, "The Balsa asynchronous circuit synthesis system," in *Proc. Forum on Design Languages*, September 2000.
[18] A. Bardsley and D. A. Edwards, "Synthesizing an asynchronous DMA controller with Balsa," *J. Systems Architecture*, vol. 46, pp. 1309–1319, 2000.
[19] A. Bardsley, "Implementing Balsa handshake circuits," Ph.D. thesis, Department of Computer Science, University of Manchester, 2000.
[20] W. D. Toms, "Synthesis of QDI datapath circuits," Ph.D. thesis, Department of Computer Science, University of Manchester, February 2006.
[21] W. B. Toms and D. A. Edwards, "Efficient synthesis of speed-independent combinational logic circuits," in *Proc 10th Asia and South Pacific Design Automation Conf.*, 2005, pp. 1022–1026.
[22] A. Peeters and M. de Wit, "The HASTE manual, version 3.0," Handshake Solutions, www.handshakesolutions.com, 2007.
[23] L. Plana and S. M. Nowick, "Concurrency-oriented optimization for low-power asynchronous systems," in *Proc. 1996 Int. Symp. on Low Power Electronics and Design*, August 1996, pp. 151–156.
[24] P. A. Beerel and T. H.-Y. Meng, "Semi-modularity and testability of speed-independent circuits," *Integration, VLSI J.*, vol. **13**, no. 3, pp. 301–322, 1992.
[25] S. DasGupta, P. Goel, R. Walther, and T. Williams, "A variation of LSSD and its implications on design and test pattern generation in VLSI," in *Proc. IEEE Test Conf.*, 1982.
[26] K. van Berkel, A. Peeters and F. te Beest, "Adding synchronous and LSSD modes to asynchronous circuits," in *Proc. 8th Int. Symp. on Asynchronous Circuits and Systems*, April 2002, pp. 161–170.

[27] F. te Beest, A. Peeters, M. Verra, K. van Berkel and H. Kerkhoff, "Automatic scan insertion and test generation for asynchronous circuits," in *Proc. Int. Test Conf. 2002 (ITC'02)*, p. 804.
[28] F. te Beest and A. Peeters, "A multiplexer based test method for self-timed circuits," in *Proc. 11th IEEE Int. Symp. on Asynchronous Circuits and Systems (ASYNC 2005)*, March 2005, pp. 166–175.
[29] M. Roncken, "Defect-oriented testability for asynchronous ICs," *Proc. IEEE*, vol. **87**, February 1999, pp. 363–375.
[30] M. Roncken, "Partial scan test for asynchronous circuits illustrated on a DCC error corrector," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, 1994, pp. 247–256.
[31] M. Roncken, E. Aarts, and W. Verhaegh, "Optimal scan for pipelined testing: an asynchronous foundation", in *Proc. Int. Test Conf.*, October 1996, pp. 215–224.
[32] A. Bailey and M. B. Josephs, "Sequencer circuits for VLSI programming," in *Proc. 2nd Working Conf. on Asynchronous Design Methodologies*, 1995.
[33] T. Verhoeff, "Z Specification for DCC error decoder," Technical Report, Eindhoven University of Technology, 1993.
[34] Handshake Solutions website, http://www.handshakesolutions.com.

11 Quasi-delay-insensitive pipeline templates

As mentioned in Chapter 7, in delay-insensitive (DI) design it is assumed that the delays of the composite gates and wires can be unbounded and thus the circuits will work correctly for any arbitrary set of time-varying gate and wire delays [4][5]. This is the most conservative and robust delay model, but it has been shown that it is not very practical because very few DI circuits exist [6]. Therefore the notion of quasi-delay-insensitive (QDI) circuits has been developed [3][7]. These circuits work correctly regardless of the values of the delays in the gates and wires, except for those associated with wire forks designated *isochronic*. By definition, the difference in the times at which a signal arrives at the ends of an isochronic fork is assumed to be less than the minimum gate delay. If these isochronic forks are guaranteed to be physically localized to a small region, this assumption can be easily met and the circuits can be practically as robust as DI circuits. This chapter covers a variety of QDI templates designed with pipelined handshaking. Note, however, that the QDI model is also used in circuits that implement enclosed handshaking (see Chapter 8) and has been extended to include the assumption of isochronic propagation through a number of logic gates [8].

11.1 Weak-conditioned half buffer

The first QDI template we will cover is the weak-conditioned half buffer (WCHB). Figure 11.1(a) illustrates the high-level WCHB template for a linear pipeline with left- (L) and right-hand (R) 1-of-N channels and Figure 11.1(b) illustrates its gate-level implementation assuming dual-rail channels ($N = 2$). In Figure 11.1(b) the WCHB is optimized for improved cycle time by using the inverting C-element outputs to drive the output completion-detection logic rather than the outputs of the subsequent inverters. In the WCHB dual-rail buffer, L0 and L1 and R0 and R1 identify the false and true dual-rail inputs and outputs, respectively; Lack and Rack are active-low acknowledge signals. Note that the staticizers that are used to hold the state at the outputs of the C-elements are not shown explicitly.

The operation of the buffer is as follows. After the buffer has been reset, all data lines are low and the acknowledgement lines, Lack and Rack, are high. When data arrives, one input rail goes high and the corresponding C-element output goes low, lowering the left-hand acknowledge signal Lack. After the data has been

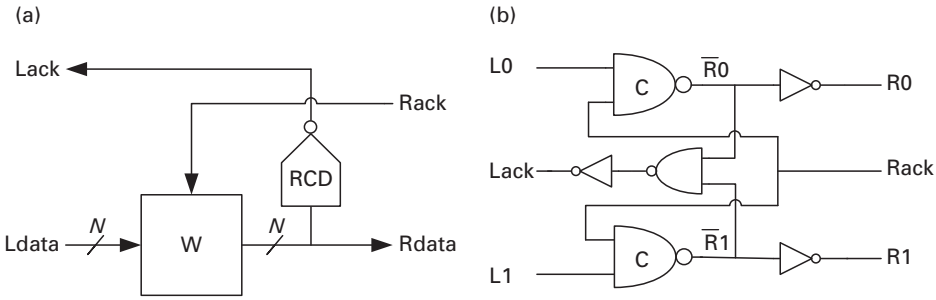

Figure 11.1. Weak-conditioned half buffer W: (a) block diagram and (b) gate-level implementation.

propagated to the outputs through one of the inverters, the right-hand environment will assert Rack low, acknowledging that the data has been received. Once the input data resets, the template raises Lack and resets the output.

The distinguishing feature of the WCHB template is that the validity and neutrality of the output data R implies the validity and neutrality of the corresponding input data L. As mentioned in Chapter 7 this is called *weak-conditioned logic* [1], and we will review its advantages and disadvantages later.

Notice in Figure 11.1 that Rack forks to two C-elements. This fork must be isochronic in order to guarantee correct operation. The Rack line starts high, goes low, and then returns to high while processing the first token; this is initiated by, for example, a rising transition from L0. If the low transition of Rack does not reach the second C-element by the time a rising transition from L1 occurs due to the arrival of a second token, the C-element may experience a runt pulse at its output. Note this can only happen if the forked wire can experience multiple transitions in transit, which assumes what is called a *pure delay model*. In contrast, in an *inertial delay model* input pulses shorter than the delay of the wire are not propagated.

Since the L and R channels cannot simultaneously hold two distinct data tokens, this circuit is said to be a *half buffer* or to have *slack* of 1/2 [3].[1] The behavior of the three-stage pipeline shown in Figure 11.2 is illustrated in Figure 11.3 using a marked graph in which we have assumed that only 0 data tokens traverse through the pipeline. Notice that Figure 11.3 does not include any model of the environment and consequently the transitions associated with the primary inputs and outputs are left dangling. Moreover, we focus on 0 data tokens because modeling the inherent non-determinism associated with a mixture of 0 data tokens and 1 data tokens traversing through the pipeline is complex and obfuscates the parallelism in the pipeline template.

[1] In Figures 11.2–11.6, and in certain other figures in this chapter, subscript notation for signals is used for convenience.

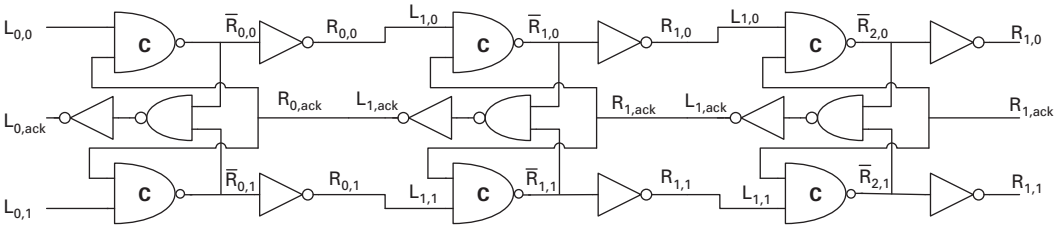

Figure 11.2. A three-stage WCHB pipeline.

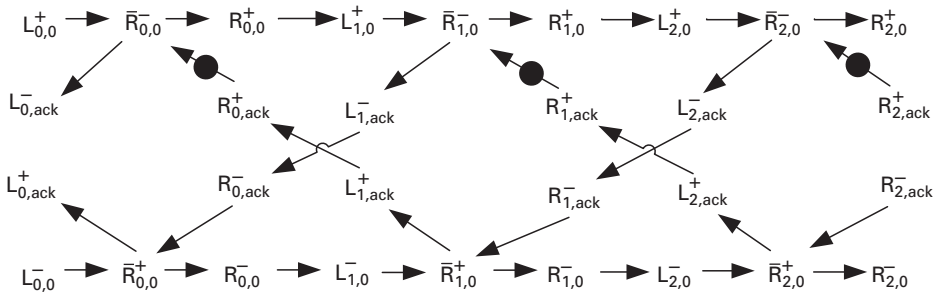

Figure 11.3. Marked graph showing the behavior of the three-stage WCHB pipeline in Figure 11.2.

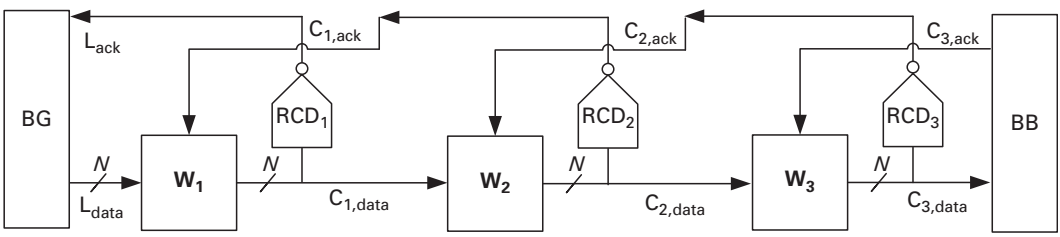

Figure 11.4. A block diagram of the three-stage WCHB pipeline surrounded by an abstract bit-generator and bit-bucket environment. The right-hand completion detectors are denoted RCD_i.

To analyze the pipeline parallelism independently of specific data values, we sometimes use an abstract block diagram of the pipeline and a corresponding marked graph. In particular, a block diagram of the three-stage pipeline is shown in Figure 11.4. Here, we also include an abstract notion of the left-hand and right-hand environments, which are shown as a bit generator (BG) and bit bucket (BB). These two environments may be in reality arbitrarily complex but, for the purpose of this pipeline, BG simply responds to the left-hand pipeline acknowledgements by providing new data and BB responds to the pipeline output tokens by consuming them and acknowledging the pipeline. The corresponding marked graph is shown in Figure 11.5. An advantage of this marked graph is that the cycle-based

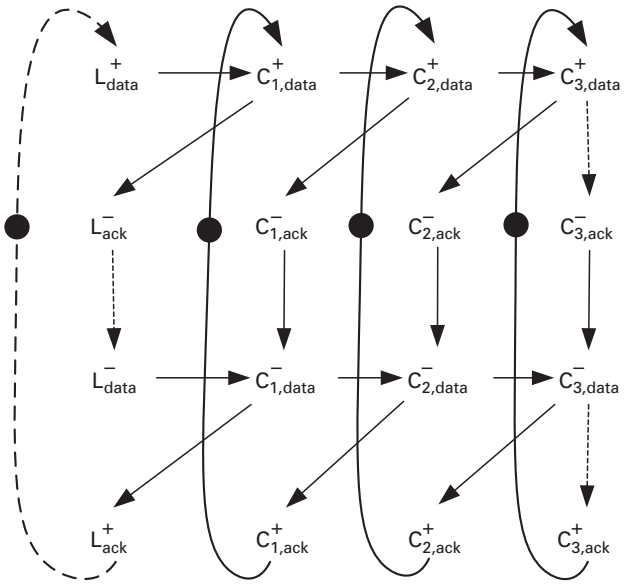

Figure 11.5. Marked graph of the abstract three-stage WCHB pipeline in its environment.

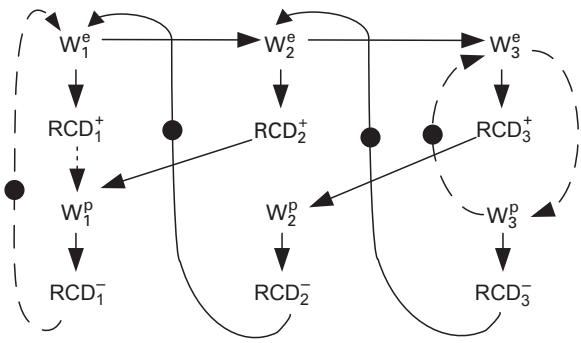

Figure 11.6. Marked graph showing a functional view of three-stage WCHB pipeline in its environment.

performance analysis based on the techniques described in Chapters 4 and 5 can include the response time of the left- and right-hand environments.

Another abstract view of the performance of such pipelines can be obtained by creating a marked graph in which the nodes represent block diagram events rather than signal transitions. This is illustrated in the marked graph in Figure 11.6 in which the evaluation and precharge events of the ith weak-conditioned function block are denoted $\mathrm{W}^{\mathrm{e}}_i$ and W^{p}_i, respectively, and the identifications of the valid and neutral data by the right-hand completion detector are denoted RCD^+_i and RCD^-_i, respectively.

Analyzing the longest cycle in the marked graph shown in Figure 11.6, we can see that the local cycle time involves the delays from three evaluations,

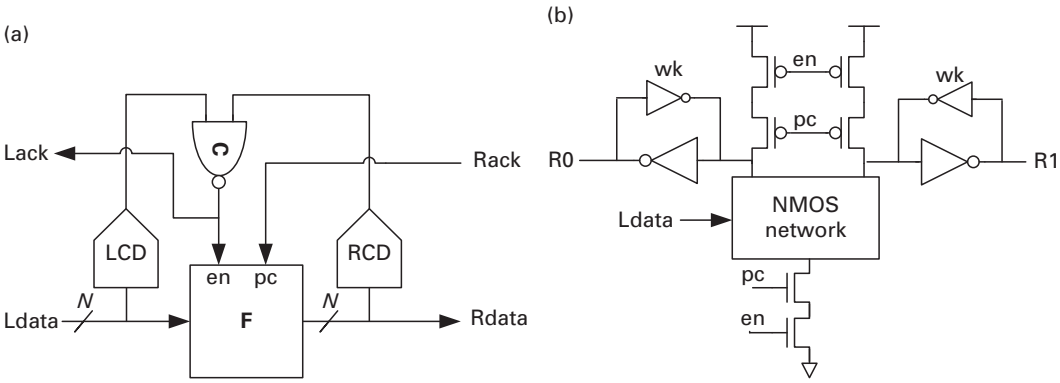

Figure 11.7. Precharged half-buffer template: (a) block diagram and (b) dual-rail domino F block.

one precharge, and two completion detectors. The evaluation and precharge delays are generally two transitions and the completion detectors are as small as one transition (for a single input and output channel, in which we account for the fact that the RCD inputs are taken before the datapath output inverters), giving rise to a 10-transition cycle time. This is faster than for the buffers designed with most other QDI templates. For WCHB circuits with more than one input channel, the datapath contribution to the cycle time does not change (when counting transitions) but the delay of the completion detectors increases logarithmically. Nevertheless, the complexity of WCHB circuits is typically limited to what can feasibly be implemented in a single level of weak-conditioned logic.

In particular, a closer look at Figure 11.1 shows that more complex weak-conditioned function blocks can be implemented by changing the C-elements appropriately. However, this means increasing the complexity of both the NMOS and PMOS stacks of the C-element. The two-input C-element shown in Figure 11.1 already has two PMOS transistors in series and a general rule of thumb is not to exceed that number. Therefore implementing any complex function with the WCHB leads to a poor performance.

An attractive alternative is to replace the C-elements with gates in which the number of PMOS transistors in series is independent of the function implemented. In particular, using one level of domino logic in place of the C-elements is a promising approach; this is the core idea underlying the precharged half buffer template described next.

11.2 Precharged half buffer

The block-level template of the precharged half buffer (PCHB) is illustrated in Figure 11.7(a). Like the WCHB template, the PCHB template has an

output completion detector, denoted as RCD (right-hand completion detection). However, unlike the WCHB, the validity and neutrality of the input channels are checked using a distinct input completion detector, denoted as LCD (left-hand completion detection). Moreover, the function block F need not be weak-conditioned and instead can be implemented using one level of domino logic with two control inputs en and pc, as illustrated in Figure 11.7(b). The signal pc is driven by the right-hand acknowledgement and the signal en is driven with the inverting C-element combination of the two completion detectors.

Note that, as in the WCHB template, the inputs to the domino inverters are used to drive the RCD in order to improve throughput (not shown).

The PCHB template is quasi-delay-insensitive and works on the assumption that the wire fork between the LCD and function block F is isochronic. In particular, the falling transition of the input data should arrive at the data inputs to the F block before it propagates through the LCD and C-element and causes a rising transition of the en input to the F block. Otherwise, if Rack is already 1 then the function block may incorrectly re-evaluate with old data.

Because the function block need not be weak-conditioned, it can evaluate before all the inputs have arrived (if the logic allows). However, the template only generates an acknowledgment signal Lack after all the inputs have arrived *and* the output has evaluated. In particular, the output of the inverting C-element that combines the LCD and RCD outputs, driving the signal en, also drives the Lack output signal.

A few minor aspects of this template should also be pointed out. First, because the C-element is inverting, the acknowledgment signal is an active-low signal. Second, the Lack signal is often buffered using two inverters before being sent out. Often, another two inverters are also added in order to buffer the internal signal en that controls the function block.

Figure 11.8 illustrates a detailed implementation of a PCHB with dual-rail input and output channels and includes explicitly the buffering of the acknowledge and en signals. For simplicity, these buffering inverters will typically not be shown in the remaining figures of this book.

Figure 11.9 illustrates the handshaking protocol using the marked graph for an abstract three-stage PCHB pipeline. As in the WCHB marked graph, the broken arrows represent the actions of the bit generator and bit bucket. Notice that the evaluation of the domino logic depends on four precursors: the input data is valid; the LCD indicates that the previous data has gone neutral; the RCD indicates that the previous output data has gone neutral; and the acknowledge signal has gone high. Similarly, the precharge of the domino block depends on three precursors: the LCD indicates that the input is valid; the RCD indicates that the output is valid; and the acknowledge signal has gone low and so indicates that the output data has been consumed by the next stage. In particular, the dependency between the data-input falling and the data-output falling has been removed and, instead, the input falling is a precursor for the output's

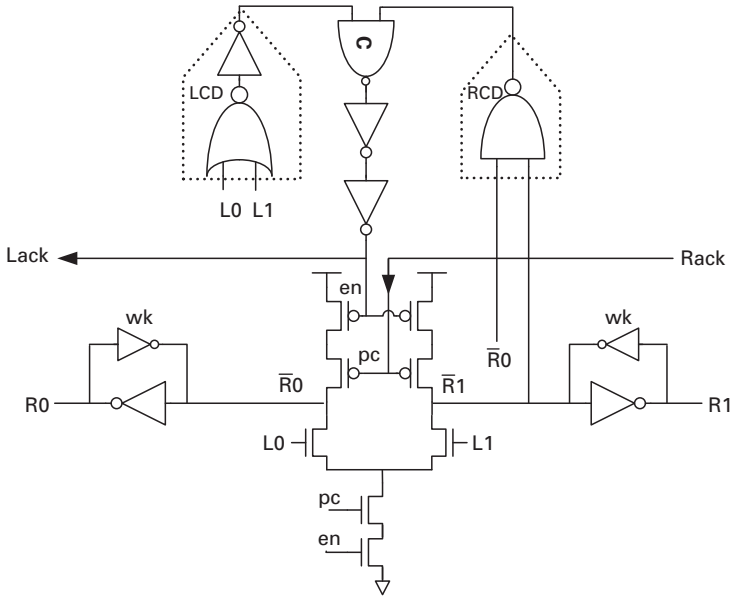

Figure 11.8. The detailed implementation of a 1-of-2 PCHB with buffered acknowledgement.

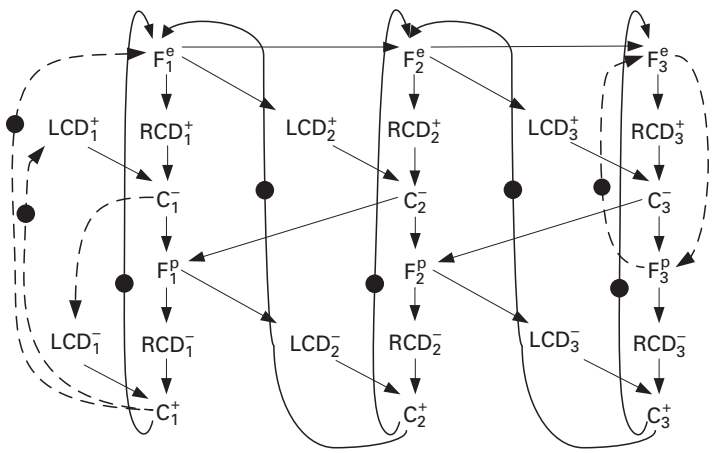

Figure 11.9. Marked-graph behavior of a three-stage PCHB pipeline.

re-evaluating. From the marked graph it is possible to derive the pipeline's analytic local cycle time:

$$T_{\mathrm{PCHB}} = 3t_{\mathrm{eval}} + 2t_{\mathrm{CD}} + 2t_{\mathrm{c}} + t_{\mathrm{prech}}.$$

During to the extra buffering, the cycle time is typically at least 14 gate delays or *transitions* and, for more complex instances with multiple inputs and output channels, is often 18.

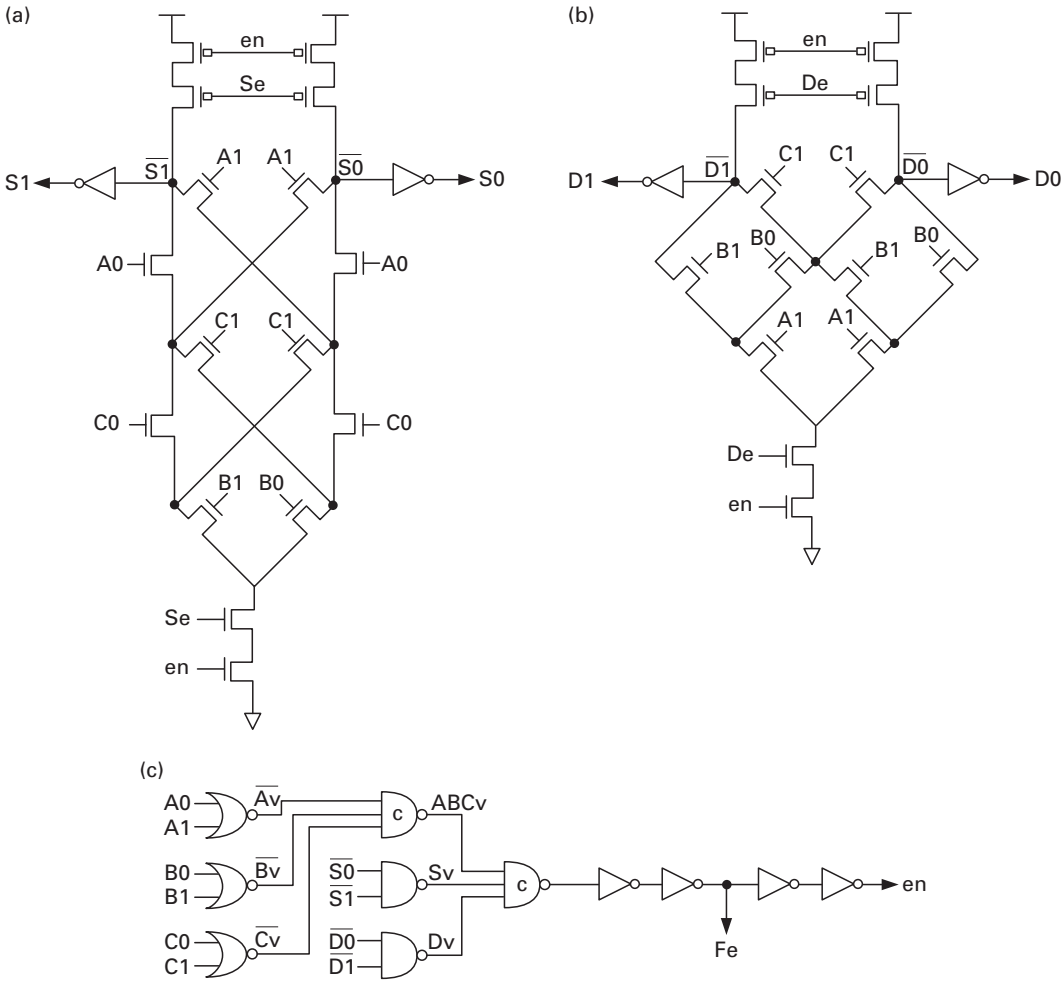

Figure 11.10. Precharged half-buffer full adder transistor-level diagram: the logic for (a) the sum, (b) the carry-out, and (c) the enable and acknowledgement.

11.2.1 PCHB full adder

As an illustration of a PCHB template that supports multiple input channels, we will consider the detailed implementation of a full adder shown in Figure 11.10.

The ABCv node is asserted when all the data inputs A, B, and C are valid. When the functions evaluate, the dynamic nodes before the inverters will be lowered and the output after the output inverters will be pulled high. The completion of function evaluation is detected with the two NAND gates and therefore the Sv and Dv nodes are asserted high. The input and output completion detection is merged with a C-element, and the internal en signal and the signal Fe sent to acknowledge the A, B, and C inputs are generated. (Since the PCHB template uses an active-low acknowledge signal, the latter is often denoted with an "e.") The Se

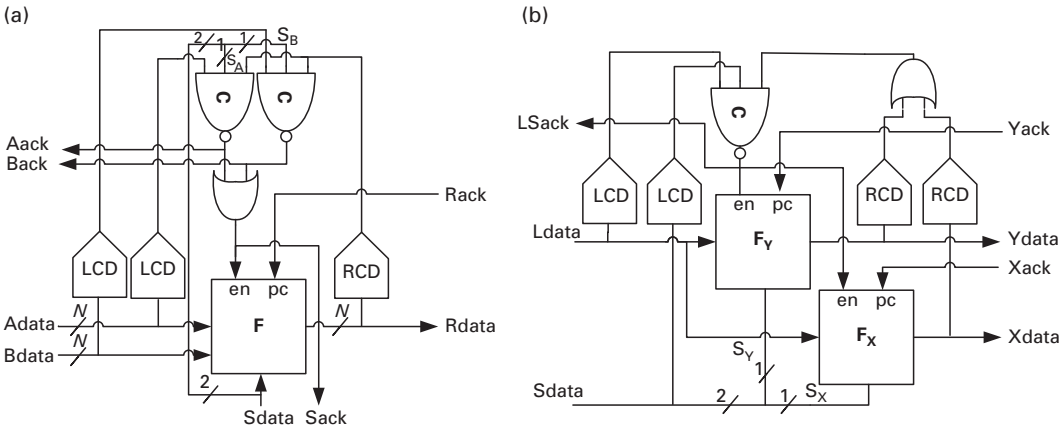

Figure 11.11. Conditional communication using PCHB: (a) reading of the input channels and (b) conditional writing of the output channels.

signal is used as the pc (precharge control) signal for the sum (S) dynamic gate, and the De signal is used as the pc signal for the carry (D) dynamic gate.

11.2.2 Conditional reading and writing

The high-level PCHB block diagrams for the template supporting the conditional reading of input channels and writing to output channels are shown in Figure 11.11.

The conditional reading template, illustrated in Figure 11.11(a), uses an input completion detector to detect the validity of each of the conditional inputs A and B. Depending on the condition, which is encoded in the dual-rail S input, only one input will be sent to the output. Once the output validity is detected by the RCD, the consumed data and select channels are acknowledged.

The conditional writing template illustrated in Figure 11.11(b), however, has one function block for each possible output; only one of these will evaluate and which one it will be is determined by the data on the select channel. The RCDs of each output channel are mutually exclusively high (i.e. only one will fire) and are ORed together before feeding an inverting C-element that drives a shared acknowledgement wire for the data and select channels.

As an example, the detailed implementation of a two-way split using the PCHB template is shown in Figure 11.12.

The L input channel is sent to either A or B depending on the select channel S. Only one acknowledge signal, SLe, is generated for both the L and S channels, since L and S will always be consumed and acknowledged unconditionally. However, in the two-way merge illustrated in Figure 11.13, either the token on channel A or that on channel B is consumed (depending on the token on the control channel M) but not both, and separate acknowledge signals Ae and Be must be generated. The acknowledge signal Me is derived from the acknowledge signals Ae and Be, since the channel M must be acknowledged after a token on

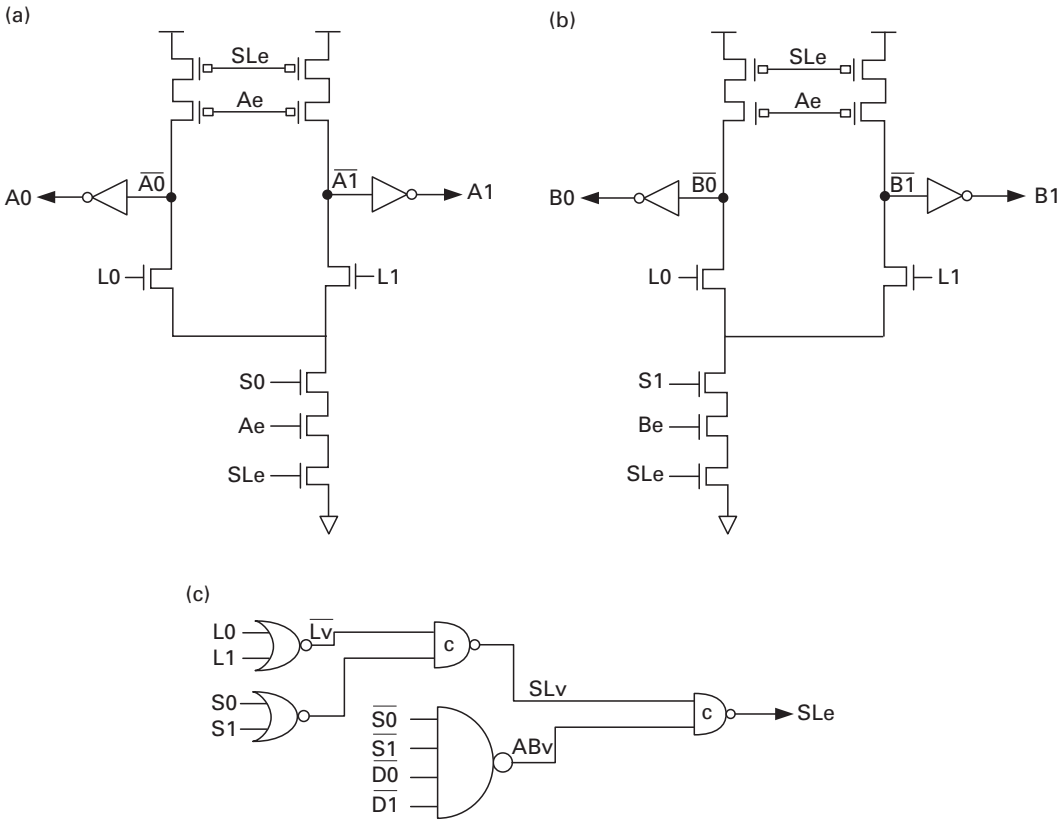

Figure 11.12. Precharged half-buffer split transistor-level diagram: the logic for (a) the A output channel, (b) the B output channel, and (c) the enable and acknowledgement.

either A or B is consumed. In these split and merge examples, the internal enable signal is the same as the acknowledge signals Me and SLe and is not buffered using two inverters. This is often necessary to limit the cycle time because, as the number of input and output channels increases, the LCD and RCD blocks become more complex and increase their impact on the cycle time.

11.2.3 PCHB reset

As with all systems, upon global reset the entire system should go to a known initial state. The general methodology for resetting an asynchronous pipelined system implemented with many templates covered in this book, including the PCHB, involves the use of a global reset signal to lower all the acknowledge signals to 0 during reset. Since the outgoing acknowledge signal is the same as the internal enable signal and since the incoming acknowledge signal from the next stage will also go low, the domino logic units in all stages will precharge and their output data will reset. Once reset is de-asserted, all acknowledge signals will then be allowed to return to 1 and the system will be ready to operate.

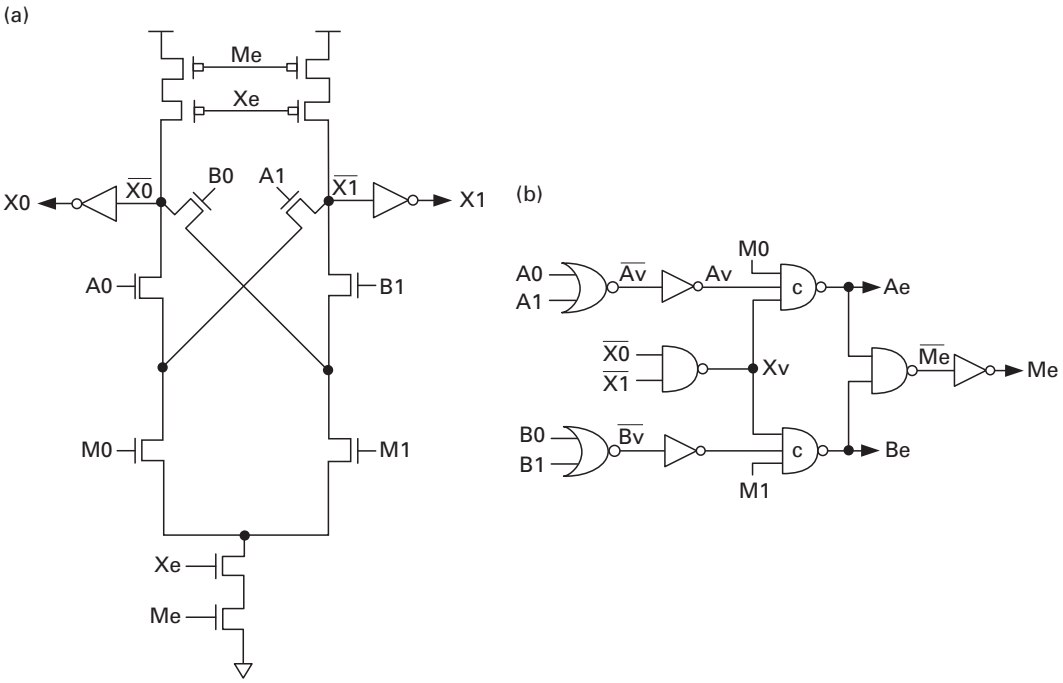

Figure 11.13. Precharged half-buffer merge transistor-level diagram: the logic for (a) the X output channel, (b) the enable and acknowledgement.

Forcing the acknowledge signals (Lack and Rack) to go low upon an active-high reset can be implemented in a number of ways. The effects on cycle time, area, and power are to be considered when different options for such an implementation are being compared. Figure 11.14 illustrates one approach using the PCHB buffer shown in Figure 11.8.

The sequence of signal transitions during reset is illustrated in Figure 11.15. The signal Reset is an active high signal. When Reset is 0, $\overline{\text{Reset}}$ will be 1 and has no effect on the output of the NAND gate. However, when Reset is set to 1 then $\overline{\text{Reset}}$ will be 0, and this will force the output of the NAND to go high, driving Lack low through the inverter. Since Rack will go low through the same process in the next buffer, the domino logic will reset, as illustrated in Figure 11.15(d). Once all acknowledge signals have reset to 0, the reset process is completed by setting Reset back to 1, as illustrated in Figure 11.15(e). The NAND gate will then act as an inverter, enabling normal operation.

11.2.4 PCHB register

A register is a building block used to store data for later use. With a simple register we can save data, read data, or do both at the same time. There are a number of ways to

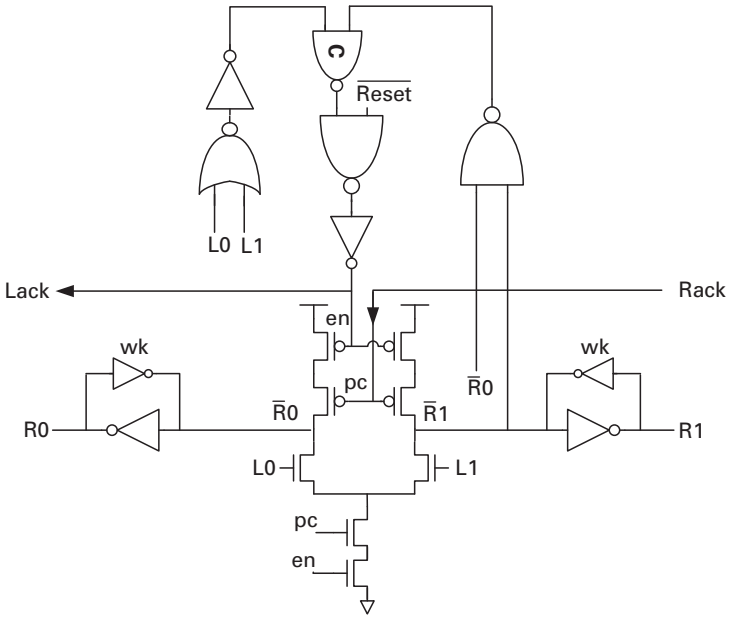

Figure 11.14. Reset circuitry for a PCHB buffer.

implement a register using asynchronous templates. One possible implementation is using the FSM loopback structure covered in Chapter 2. A more compact option is to use state-bit structures, which store the data internally rather than looping it around with feedback buffers. In this section we will present such a register implementation.

We will consider a 1-of-4 encoded PCHB-based register that has L and R as input and output channels and C as a control channel. If C is 0 then the internally stored data is sent to the output channel R. If C is 1 then the input token on L is consumed and internally stored. Finally, if C is 2 then the token on L is consumed, stored internally, and sent to the output channel R.

A top-level diagram consisting of the input channels L and C and the output channel R is given in Figure 11.16(a). The implementation of the register is more complex than most other units covered; therefore we show its implementation hierarchically. Figure 11.16(b) illustrates the top-level partitioning. The REGISTER_DATA block stores the data internally and implements the output logic, whereas the REGISTER_CTRL block implements the conditional handshaking. Finally, Figure 11.16(c) illustrates the partitioning of the REGISTER_DATA block. Here there are four logic networks, labeled Rail *i*, for each of the four data rails of the output channel R, two STATE_BIT blocks which store the 1-of-4 input tokens as two 1-of-2 data bits, and a local control circuit labeled REGISTER_DATA_CTRL.

The state-bit structure deserves some attention because of its unusual implementation and is thus illustrated in Figure 11.17. After reset, the s0 output will

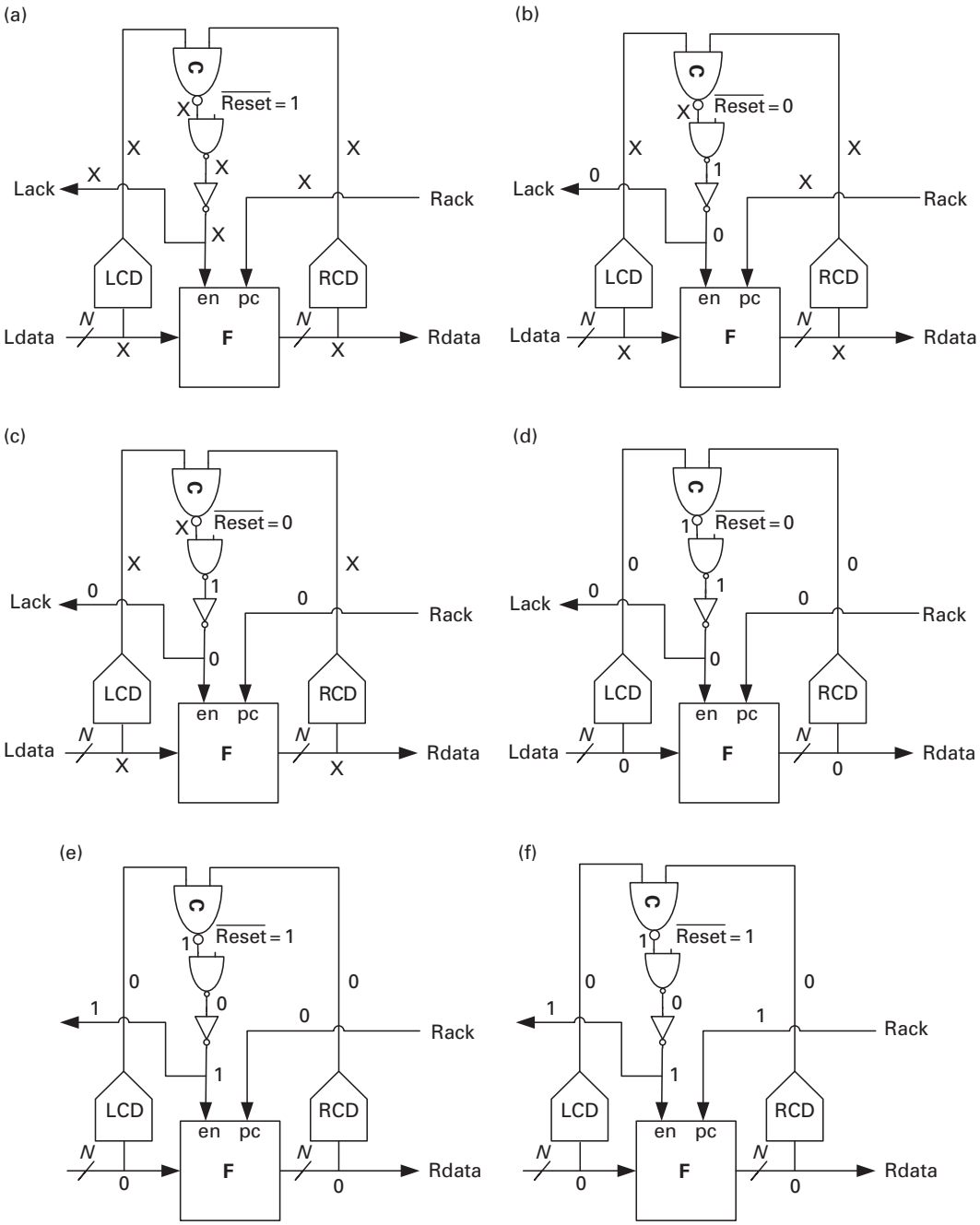

Figure 11.15. Reset sequence in the PCHB buffer in Figure 11.14.

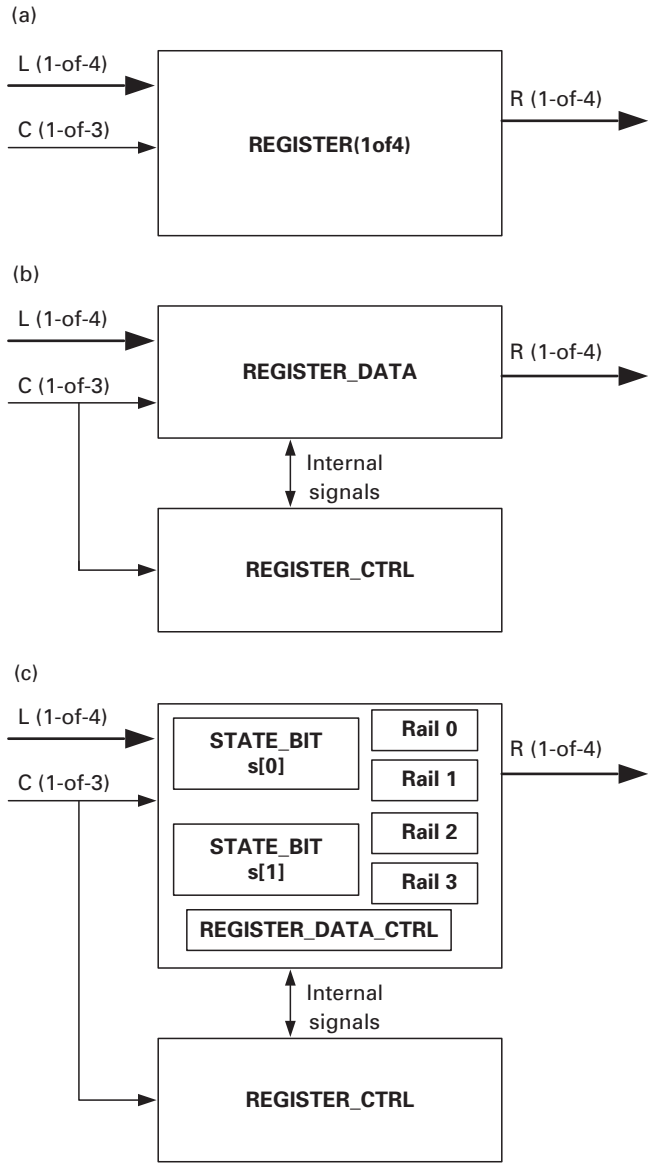

Figure 11.16. Top-level block diagram of the 1-of-4 register.

be high and the s1 output will be low, indicating that the stored state is 0. The stored state will only change when the internal signal en (generated by the REGISTER_CTRL block) is subsequently lowered. The details of the pull-down NMOS network for s0 and s1 depend on how the state-bit structure is to be used and will vary from application to application. Consider the case where the inputs to the NMOS networks are such that $\overline{s1}$ goes low and $\overline{s0}$ remains high. Once en goes low, s1 will be pulled high. Because $\overline{s0}$ remains high, as s1 goes high

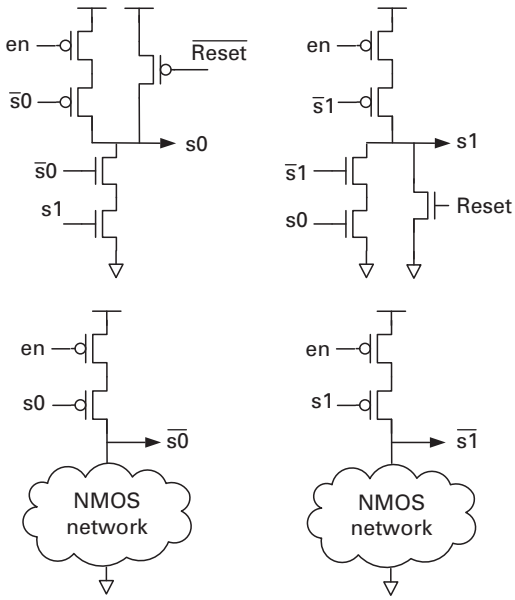

Figure 11.17. Abstract structure for a state bit s used in a register.

s0 will be pulled low, changing the stored state from s0 = 1 and s1 = 0 to s0 = 0 and s1 = 1.

The 1-of-4 register illustrated in Figure 11.16 uses two 1-of-2 state-bits s[0] and s[1] to store the input 1-of-4 input data L. In particular, an NMOS network controlled by the inputs C, L, and the currently stored variables s[0] and s[1] are used to change the state by pulling down one of the four state-bit rails, as illustrated in Figure 11.18. The output data R is similarly constructed and converts these two 1-of-2 bits to a 1-of-4 output channel, as illustrated in Figure 11.19.

The REGISTER_DATA_CONTROL block is depicted in Figure 11.20. Its function is to check the validity of all input and output channels. Notice that in the case of a write command the controller uses the existence of a fake fifth rail R4 to indicate that the R channel is valid. This simplifies the REGISTER_CTRL block, because now the R validity signal rv and the combined R and state validity signal rsv are always asserted independently of a read or write.

The high-level REGISTER_CONTROL block is illustrated in Figure 11.21. Its function is to generate the enable signal en that controls the evaluation of both the state-bit update and the read channel R, as well as the acknowledgements to both the C and L channels. (Recall that the use of the subscript "e" on the acknowledgement wire indicates that the acknowledgements are active-low.) Notice that the C channel is acknowledged only after the right-hand channel R and state bits s (see Figure 11.16) become valid, indicating that the read–write command has been fully executed. Similarly L is acknowledged upon a completed write, i.e. when doL_1 is asserted (see Figure 11.21).

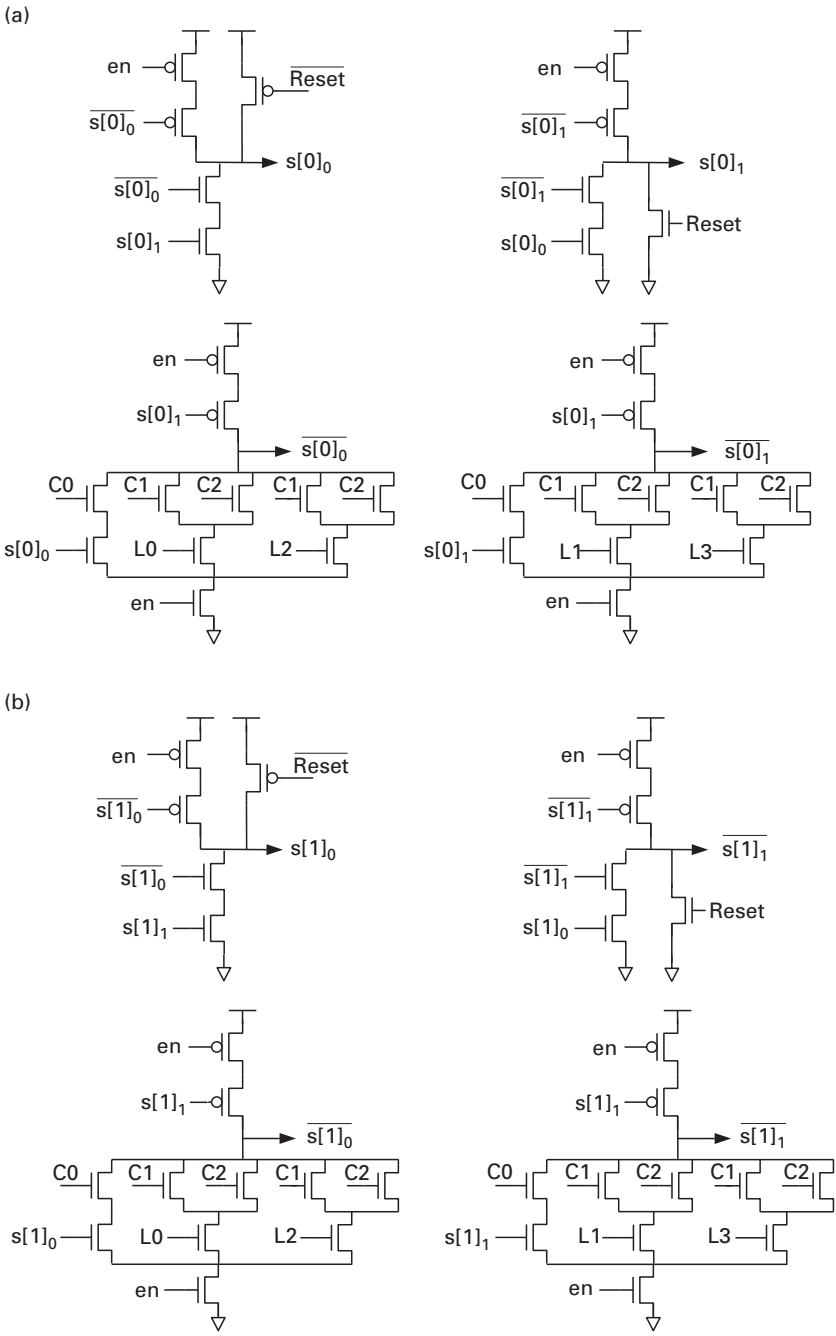

Figure 11.18. Detailed implementation of the dual-rail state bits (a) s[0] and (b) s[1].

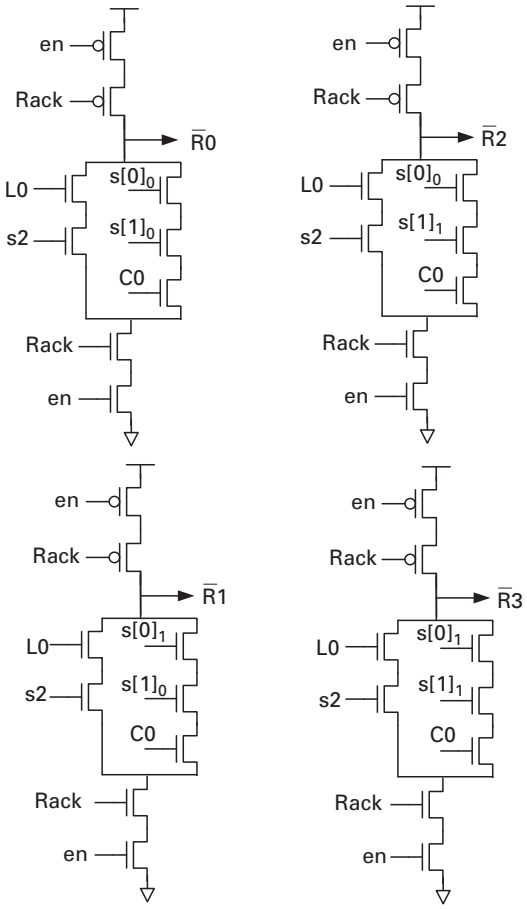

Figure 11.19. Detailed implementation of the output logic of a 1-of-4 register.

11.3 Precharged full buffer

The PCFB template, shown in Figure 11.22, is more complicated than the PCHB template, having two asymmetric C-elements, denoted together as aC, rather than one regular C-element. As mentioned in Chapter 7, asymmetric C-elements are similar to C-elements but have inputs that are required only for the falling or rising transition. In particular, an input to an asymmetric C-element that is denoted with a plus (minus) is required to be a 1 (0) for the C-element's output to rise (fall). An input to an asymmetric inverting C-element that is denoted with a plus (minus) is required to be a 0 (1) for the C-element's output to rise (fall). An input with no denotation is involved in both transitions.

The two state variables enable the PCFB to be more concurrent than the PCHB since they allow the L and R handshakes to reset in parallel. To see this, consider the behavior of a PCFB in a three-stage pipeline, as illustrated in the marked

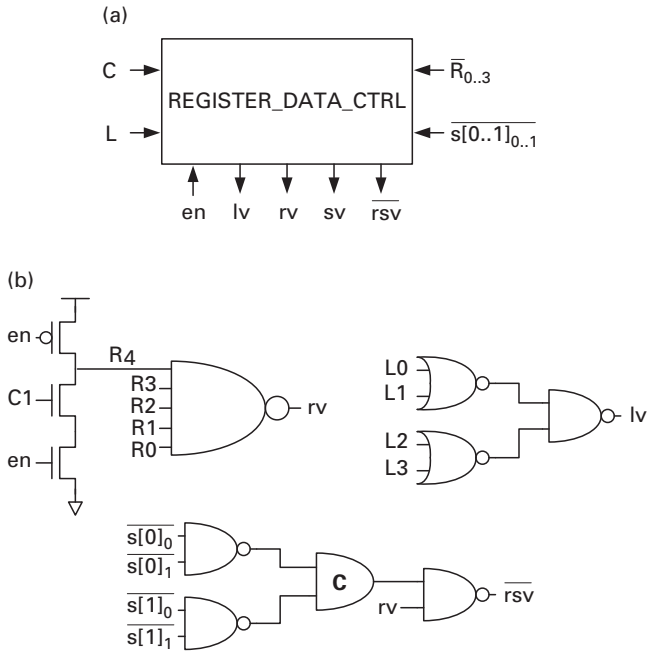

Figure 11.20. REGISTER_DATA_CTRL: (a) block diagram, (b) gate-level implementation.

graph shown in Figure 11.23. The key difference from a PCHB is that acknowledge signals can reset high without waiting for the corresponding stage to precharge. Consequently, PCFB buffers have a slack value equal to 1. Moreover, there is no cycle of transitions that includes three evaluations. In particular, the analytical local cycle time is:

$$T_{\mathrm{PCFB}} = 2t_{\mathrm{eval}} + 2t_{\mathrm{CD}} + 3t_{\mathrm{c}} + t_{\mathrm{prech}}.$$

The time t_{CD} is at least two transitions, one C-element takes one transition, and the other takes two transitions, yielding a typical cycle time equal to at least 12 transitions. Of course, the buffers on Lack and enable (not shown) increase this number.

11.4 Why input-completion sensing?

A join is a pipeline stage with multiple input channels whose data is somehow combined into a single output channel. A fork is a pipeline stage with one input channel and multiple output channels. Complex forks and joins can involve conditionally reading from or writing to channels on the basis of the value of a control channel that is unconditionally read, as in a merge or split.

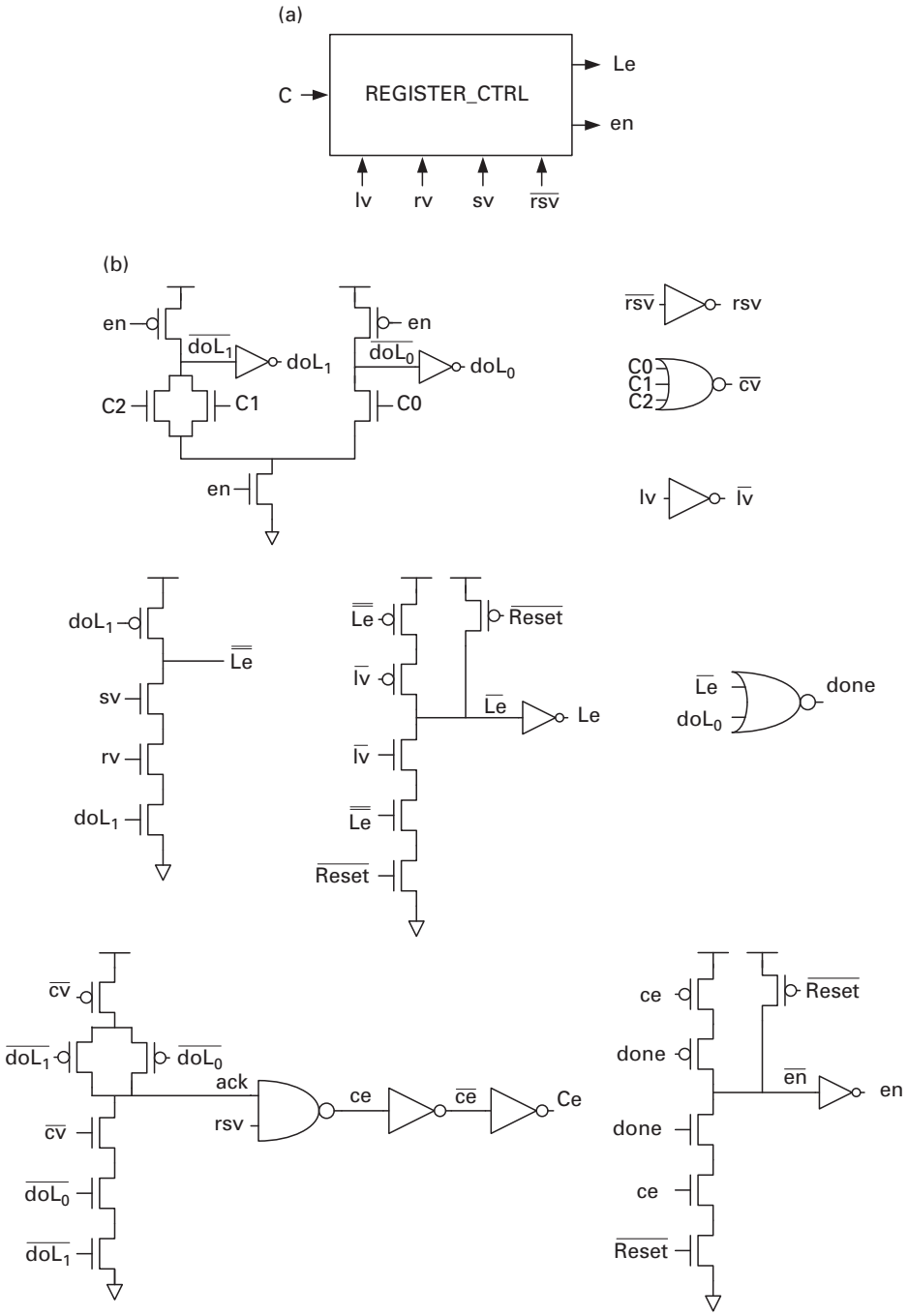

Figure 11.21. REGISTER_CTRL: (a) block diagram, (b) detailed implementation.

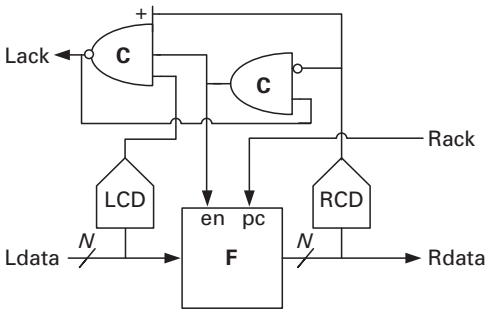

Figure 11.22. Precharged full-buffer template.

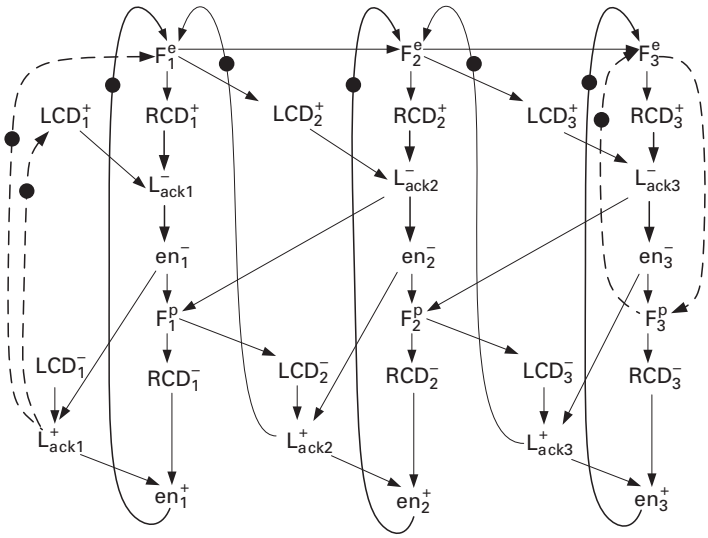

Figure 11.23. Marked graph behavior of a three-stage PCFB pipeline.

Since a fork has multiple output channels, it must receive an acknowledge signal from them all before it precharges. A join, however, receives inputs from multiple channels and must broadcast its acknowledge signal to all its input stages.

A join acts as a synchronization point for data tokens. The acknowledgement from the join should be generated only when all the input data is valid. Otherwise a stage A, feeding a join, that is particularly slow in generating its data token may receive an acknowledge signal when it should not, violating the four-phase protocol. Moreover, if the acknowledge signal is de-asserted before the slow stage A generates its token then the token is not consumed by the join, as it should be. In fact, this token may cause the join to generate an extra token on its output, thereby corrupting the intended synchronization. In addition, neutrality should be checked to guarantee that the previous stages have precharged before the acknowledge signal is de-asserted.

The templates that we will present in this section check validity and neutrality in different ways. Because the function block in the WCHB template (see Section 11.1) is weak-conditioned, the output-completion detector implicitly checks the validity and neutrality of the input data token. In the WCHB template the weak-conditioned function block is a simple C-element. However, for more complex, non-linear, pipelines, the weak-conditioned function blocks require complex NMOS and PMOS networks. This results in slower forward latency and larger transistor sizes and thus motivates the use of an explicit left-hand completion detector and the PCHB template. As an example for comparison, a weak-conditioned and pre-charged OR leaf cell with dual-rail input channels A and B and dual-rail output channel C is shown in Figure 11.24.

LCD–RCD merging

One optimization that can be applied to the PCHB and PCFB templates is to merge the LCD of one stage with the RCD of the other by adding an additional request line to the channel. This is shown in Figure 11.25 for a PCHB template.

The request line indicates the assertion or de-assertion of the input data, as in a bundled-data channel. However, in contrast with a bundled-data channel, the data is sent using 1-of-N encoding, yielding what we call a 1-of-N+1 *channel*. The request line, at least from the channel point of view, may appear redundant but in fact it enables the removal of the input-completion detector, thereby saving area and reducing capacitance on the data lines. Moreover, the request line does not significantly impact performance, the template is still quasi-delay-insensitive, and the communication between stages remains delay-insensitive.

We next cover two 1-of-N+1 QDI templates that intelligently reduce concurrency in order to reduce the stack size of the function blocks, thereby improving performance.

11.5 Reduced-stack precharged half buffer (RSPCHB)

The motivation for a reduced-stack PCHB template is to eliminate the need for an enable signal en from the domino function-block control. The underlying observation is that this enable signal is only needed to support a concurrency that typically does not improve performance.

More specifically, in the PCHB template the outputs of the LCD and RCD are combined, using a C-element, to generate the acknowledgement signal Lack. This supports the integration of the handshaking protocol with that for the validity and neutrality of both input and output data, which removes the need for the function block to be weak-conditioned; however, the use of an en signal is still required.

In particular, in the case of a join the non-weak-conditioned function block may generate an output as soon as one input channel provides data. In response, the RCD of the join will assert its output. Meanwhile, any subsequent stage can receive

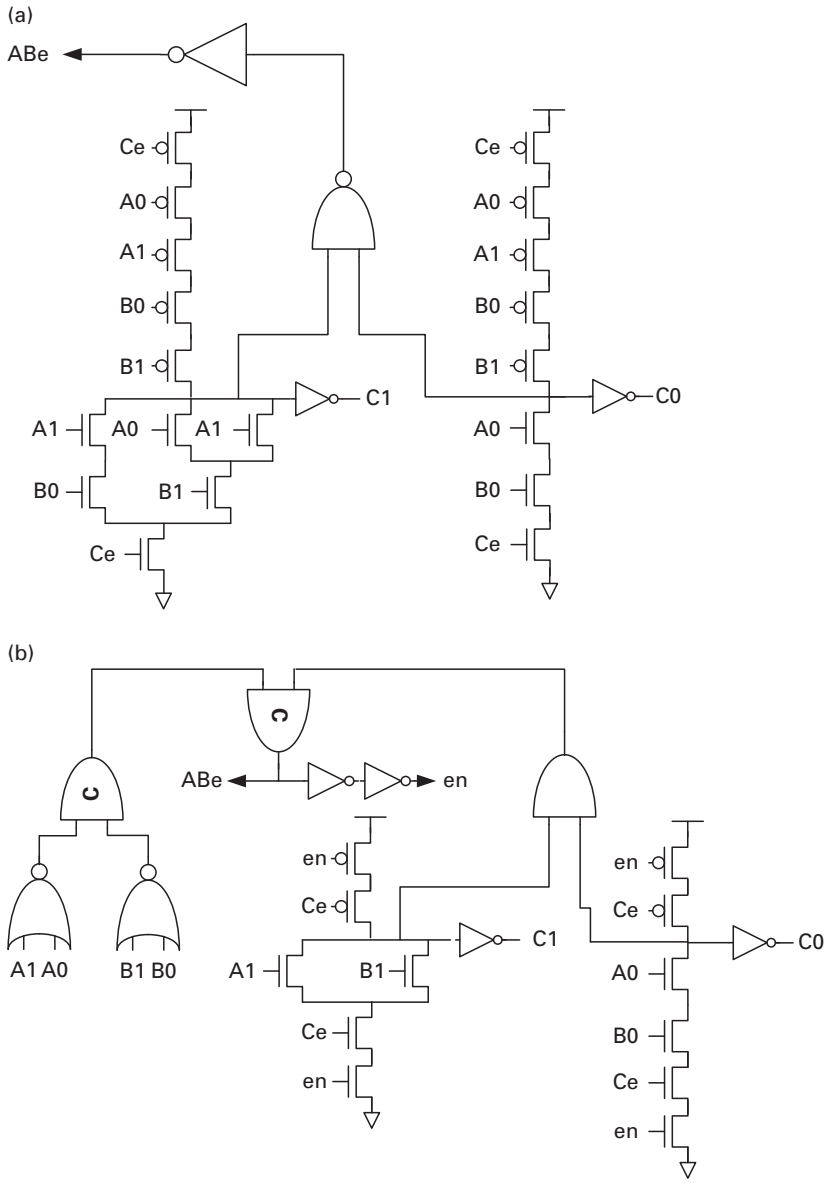

Figure 11.24. A dual-rail OR gate implemented in (a) WCHB and (b) PCHB templates.

this data, evaluate, assert (send high) both its LCD and RCD outputs, and assert its acknowledge signal. Although the join can receive this acknowledgement, it will not precharge until en is asserted. In fact, the en signal delays the precharge of the function block until after the acknowledgement to the input stages has been asserted. This delayed precharging is critical to the correct operation of the circuit because it prevents the RCD from de-asserting until after it has helped to assert the left-hand acknowledgement.

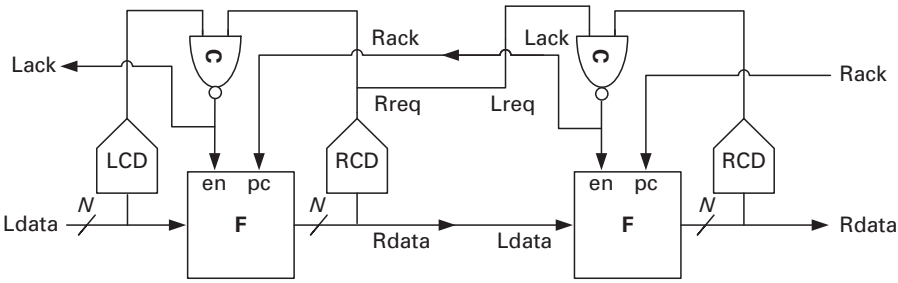

Figure 11.25. Optimized PCHB for a 1-of-N+1 channel.

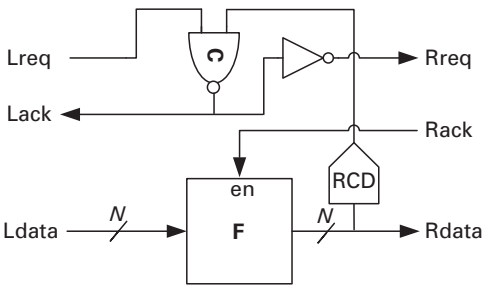

Figure 11.26. Quasi-delay-insensitive RSPCHB pipeline template.

If, however, the generation of the acknowledge signal from any stage subsequent to the join can be delayed in some way until all the input data to the join has arrived and been acknowledged, then the en signal can be safely removed. In fact, such a delay in acknowledgement would not generally impact performance because the join is the performance bottleneck for the subsequent stages. Therefore, this added concurrency is typically unnecessary.

In this section we review a different pipeline template, which reduces this unnecessary concurrency allowing the elimination of the internal en signal, thereby reducing the transistor stack sizes in the function block. We refer to this QDI pipeline template, illustrated in Figure 11.26, as a *reduced-stack precharged half buffer* (RSPCHB) [11]. The marked graph for this RSPCHB is shown in Figure 11.27, and a specific form of this template for a single dual-rail buffer is shown in Figure 11.28(b). Notice that here the RCD like the PCHB, is optimized, by tapping its inputs before the output inverter and using a NAND gate instead of an OR gate.

The unique feature of the RSPCHB is that it derives the request line from the output of the C-element rather than from the RCD. (In particular, since the output of the C-element is active-low and the request line is active-high, the output of the C-element is sent through an inverter before driving Rreq.) The impact of this change is that the assertion and de-assertion of Rreq are delayed until after all Lreqs have been asserted and de-asserted.

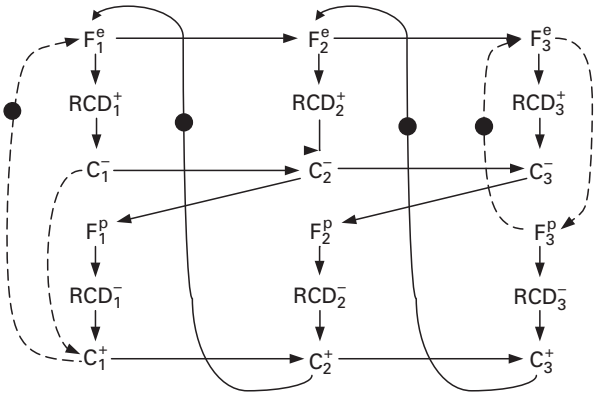

Figure 11.27. The marked graph behavior of a three-stage linear pipeline of the RSPCHB in Figure 11.26.

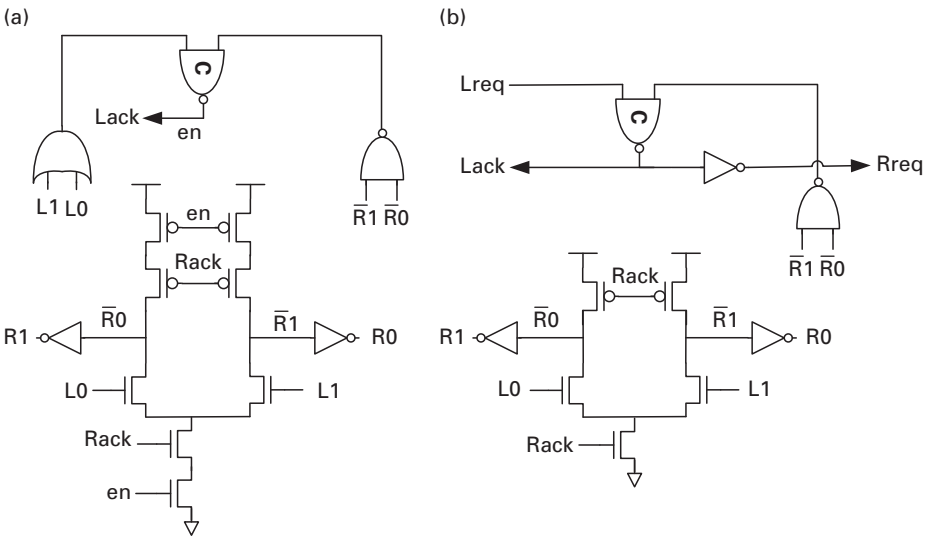

Figure 11.28. Implementation of 1-of-2 buffers using (a) the PCHB template and (b) the RSPCHB template.

As a consequence, the acknowledgement from a subsequent stage of the join will be delayed until well after its data inputs and outputs have become valid. More specifically, the stage will delay the assertion of its acknowledge signal until all Lreqs have been asserted, and this can occur arbitrarily later than the associated data lines becoming valid. This extra delay, however, typically has no impact on the steady-state system performance because the join stage is the bottleneck, waiting for all its inputs to arrive before generating its acknowledgement. In fact, this change yields a template with no less concurrency than a WCHB.

The advantage of this type of generation of the request line is that the function block does not need to be guarded by the enable signal. In particular, it is sufficient

to guard the function block solely by the Pc signal because the latter now properly identifies when the inputs and outputs become valid. Namely, the function block is allowed to evaluate when Pc is de-asserted, and this occurs only after all the input and output data lines have been reset. Similarly, it is allowed to precharge when Pc is asserted, and this occurs only after all the input and output data lines have become valid.

The RSPCHB is still QDI; however, the communications along the input channels to the joins become QDI instead of DI (other channels remain DI). In particular, the requirement that must be satisfied is that the data should reset before the join stage enters a subsequent evaluation cycle. If we assume that the fork between the function block, the RCD, and the next stage is isochronic [9] then this requirement is satisfied. In particular, the data line at the receiver side is then guaranteed to reset before the request line Rreq resets because only after the data lines reset can the RCD trigger the C-element and subsequently Rreq. An analytical expression for the timing margin associated with this isochronic fork assumption can be derived from marked graph shown in Figure 11.27. In particular, the delay difference between the resetting of the data and the associated request line should be less than

$$T_{\text{margin}} = 2t_{\text{inv}} + t_{\text{CD}} + 3t_{\text{c}}$$

This margin is between six and eight gate delays depending on the buffering and can be achieved easily with modern timing-driven place-and-route tools. Notice that this timing assumption only applies to the input channels of join stages, because non-join stages must receive both valid data and a valid Lreq before generating valid output data or a valid Rreq.

The analytical cycle time of the RSPCHB can be derived from the marked graph shown in Figure 11.27:

$$T_{\text{RSPCHB}} = \max(3t_{\text{eval}} + 2t_{\text{CD}} + 2t_{\text{c}} + t_{\text{prech}}, t_{\text{eval}} + 2t_{\text{CD}} + 4t_{\text{c}} + t_{\text{prech}}).$$

With bubble shuffling, RSPCHBs and PCHBs have equal numbers of transitions per cycle. The advantage of RSPCHBs is that the lack of an LCD and the reduced stack size of the function block, which reduces the capacitive load, yields significantly faster overall performance. The cost of this increase in performance is that it requires one extra communicating wire between stages. For purposes of comparison, Figure 11.28 illustrates both a PCHB and an RSPCHB encoded as 1-of-2 implementing a buffer function.

A fork can be implemented easily either by using a C-element to combine the acknowledge signals from the forking stages or by combining them by means of increasing the stack size of the function block. Similarly a join can be implemented by combining the request lines in the C-element and forking back the acknowledge signal.

11.5.1 Conditional reading and writing RSPCHB

High-level PCHB-template block diagrams supporting conditional reading of input channels and writing to output channels are shown in Figure 11.11(a),(b).

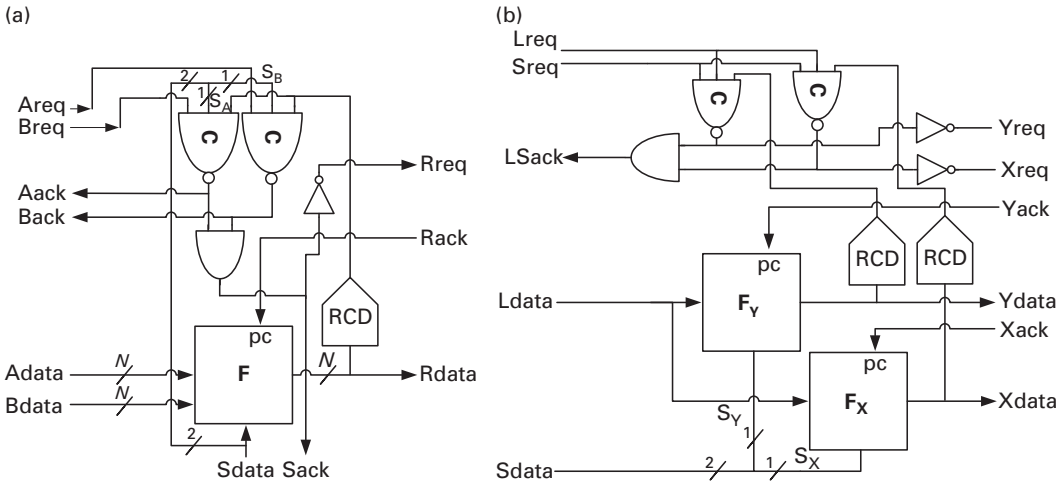

Figure 11.29. Conditional (a) join and (b) split using an RSPCHB.

Now we consider the slightly more complicated RSPCHB template for a conditional join, in which a control channel S is used to select the input channel that is to read and write a given read data token to a single output channel. This is illustrated in Figure 11.29(a). The template has one C-element per input channel, each of which generates the associated acknowledge signal. Each C-element is triggered by not only the RCD output but also the corresponding control channel bit. The collection of C-elements are simply ORed to generate the Lreqs, because the former are mutually exclusive. This template can be easily extended to handle more complex conditionals in which multiple inputs can be read for some values of the control.

The template for a conditional split is shown in Figure 11.29(b). Here, the functional block, the RCD, and the C-element are repeated for each output channel. The select data lines ensure that only one function block evaluates. All C-elements are combined using an AND gate to generate the acknowledgement for the select channel. (The reason is that both the C-element outputs and the acknowledge signal are active-low.) This template can easily be extended to handle the above-mentioned generation of multiple outputs in response to some values of the control.

A common example of a conditional fork is a *skip*, in which, depending on the control value, the input is consumed but no output is generated. The implementation has a skip output acting as an internal $N + 1$ output rail that is not externally routed but is triggered upon the skip control value. Note that a skip in which no control value generates an output has the same functionality as a bit bucket [10].

11.5.2 RSPCHB register

While the register illustrated for the PCHB can be modified to be implemented using an RSPCHB or any other template, in this subsection an alternative implementation

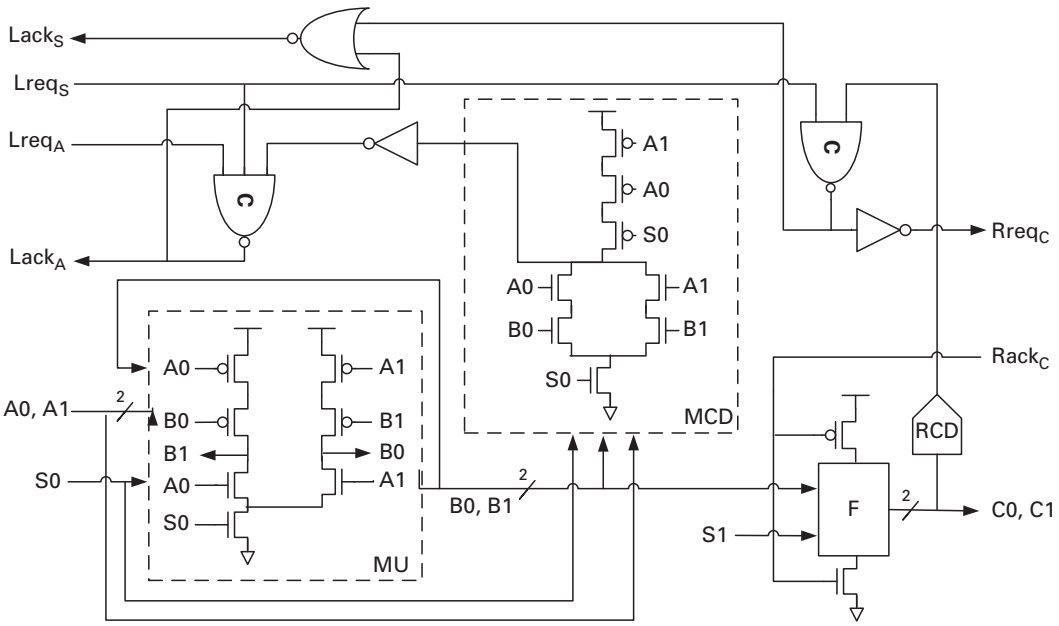

Figure 11.30. An RSPCHB one-bit memory.

is presented. Figure 11.30 shows a one-bit memory implemented using an RSPCHB template. It has input and output channels A and C and an internal storage channel B. An input control channel S determines whether the memory should read or write. When S0 is high, on the one hand, the memory stores the value at the input channel A to the internal dual-rail storage B. When S1 is high, on the other hand, the memory is read, that is, the stored memory value is written to the output channel C. For a write, both the input data and control channels are acknowledged while for a read only the control channel is acknowledged.

The write and read operations are as follows. After reset, the state stored in the dual-rail *memory unit* MU (which is similar to that in [3]) is initialized to some value and one rail of the internal dual-rail signal B is high. When a token on A arrives and S0 is high, one rail of B is asserted high, storing the data on A. The *memory completion detector* MCD detects that the value in the memory has been updated and asserts its output. The output of the MCD as well as the request lines from the data and control channels drive a C-element, which generates the acknowledge signal Lack$_A$. When S1 is high, however, the internal data stored in B is sent to the output channel C. When an acknowledgement is received from the output channel C, the outputs are reset but the data stored remains unchanged. The control channel S is acknowledged for both write and read operations, using an AND gate driven by the two C-element outputs.

Notice that the memory is actually implemented by merging two RSPCHB units. The first is used to store data (i.e. write) and the second is used to send the data to the outputs (i.e. read). The MCD detects the completion of the write operation and resets when all inputs are lowered.

The MCD can be simplified by replacing the PMOS transistors driven by A0 and A1 with a PMOS transistor driven by Lack_A. However, this requires that the delay difference between the data lines of channel A and its associated request line is not long enough to cause short-circuit current. This restriction can be removed by controlling the NMOS stack, adding one more NMOS transistor driven by the Lack_A signal. The overall benefit, however, is not clear.

11.5.3 Loops using RSPCHB

One can see from the simple buffer shown in Figure 11.28(b) that the delay from the left-hand request Lreq to the right-hand request Rreq is two gate delays, consisting of the delays from a simple two-input C-element and an inverter. This closely matches the delay through the domino logic datapath. For a more complex gate with multiple input and output channels, however, the delay from Lreq to Rreq will increase and for some cells will be larger than the delay through the domino logic. It is important to realize that, for RSPCHB designs with loops, this delay may become the performance bottleneck [12].

In particular, the fact that the data moves forward faster than its request signal causes a problem. Given a loop with L stages, assuming that both the assertion of the data and the request signal happen to be synchronized at stage 0, let the data move forward faster than the request signal, at a rate t_d/t_r, where t_d is the forward latency of the data and t_r is the forward latency of the request signal ($t_\text{r} \geq t_\text{d}$). Pipeline stages in the loop that have evaluated will be able to re-enter the evaluation phase only after the request signal is asserted, allowing them to precharge and then subsequently be de-asserted. Therefore, as the data attempts to overtake the request signal around the loop (which may take many loop iterations), it will stall while waiting for the request signal of the subsequent stage to de-assert. From this point on, the forward latency of the data will slow down to match the forward latency of the request signal. Figure 11.31 demonstrates the slowdown in loop performance for a large, 33-stage, loop. The loop consists of 33 pipeline stages, where $t_\text{d} = 2$, $t_\text{r} = 4$, and $t_\text{c} = 18$ (the cycle time of each pipeline stage); simulation results of this loop show that the long-term average cycle time is 33×4 unit delays.

One solution to prevent this negative effect is to insert *request breakers* into the loop. Request breakers are pipeline stages that generate an Rreq signal as soon as they evaluate rather than waiting for an Lreq signal from the previous stage. A request breaker, shown in Figure 11.32, is a modified PCHB that not only generates a request signal from the output of the output completion sensing but also handles the left-hand side handshake.

The period at which request breakers must be inserted into the loop depends on the L number of stages in the loop, the forward latency t_r of the request signal, the forward latency t_d of the data, and the cycle time t_c of the stages used. We will compute the number of stages N needed for the data to overtake the request. In other words, the request signal has arrived at stage N and the data is about to be

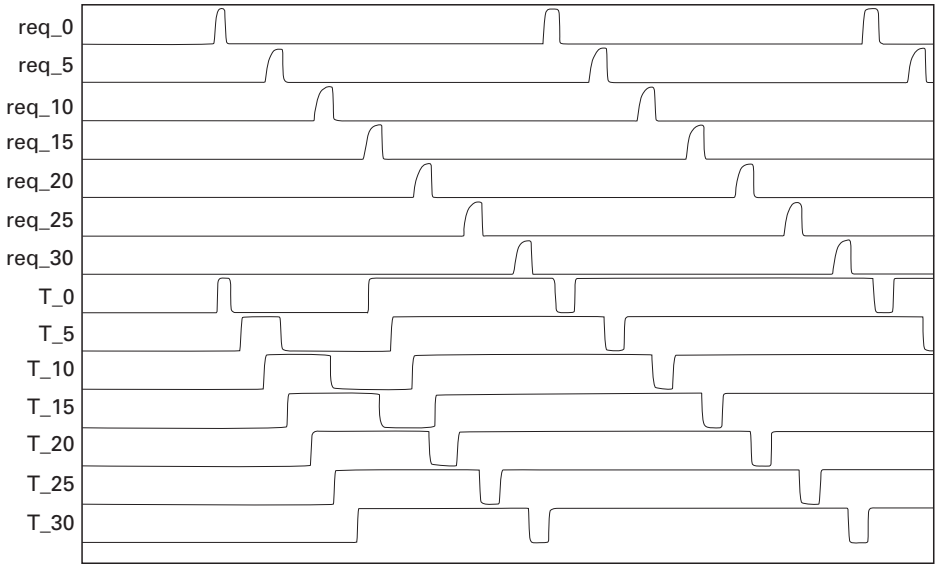

Figure 11.31. Performance slowdown in a 33-stage loop using an RSPCHB.

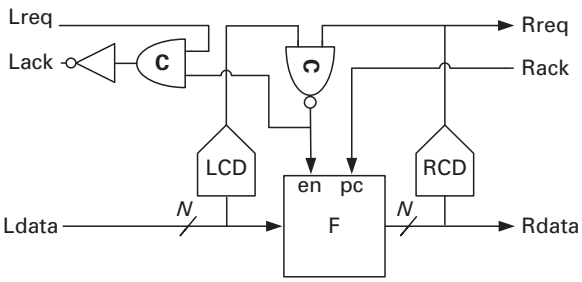

Figure 11.32. Request breaker for RSPCHB loops.

stalled at stage $L + N - 2$ (with loop index $(L + N - 2)\%L = (N - 2)\%L$). More specifically, the time it takes for the request signal to enter stage N plus the cycle time for the stage to complete its precharge and re-enter its evaluation phase should be larger than the time it takes for the data to go through the loop, enter stage $N - 2$, and stall. Thus,

$$(L + N - 2)t_d = t_c + Nt_r,$$

and so

$$N = ((L - 2)t_d - t_c)/(t_r - t_d).$$

The above formula suggests that N is 22 in our 33-stage-loop example. This means that a request breaker should be inserted every 22 stages to prevent a slowdown, as shown in Figure 11.33. Figure 11.34 shows the waveforms for a loop with the

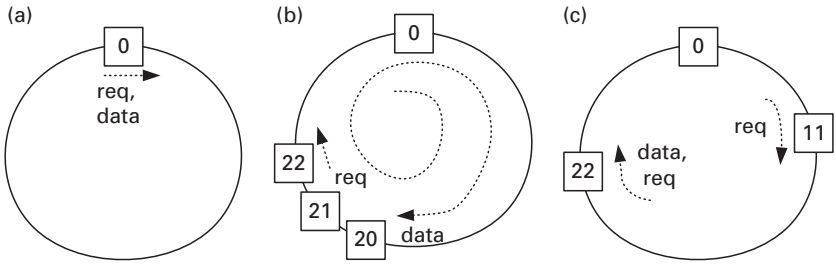

Figure 11.33. (a) The loop at time $t=0$ when data and request are synchronized. (b) Data travels around the loop and stalls when the request arrives at stage 22 and the data loops around and stalls at stage 20. (c) Request breakers are inserted every 22 stages to prevent the stall.

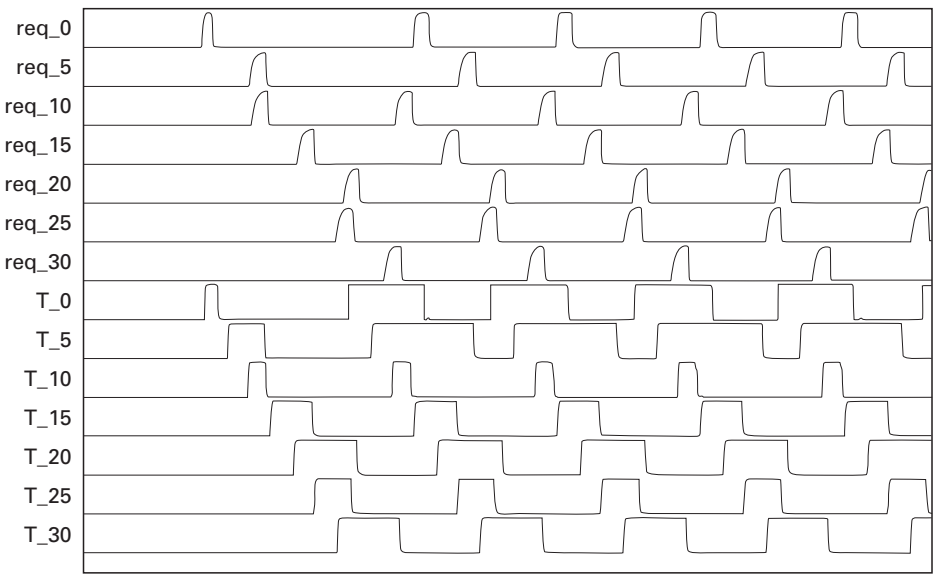

Figure 11.34. Performance improvement using request breakers in a 33-stage RSPCHB loop.

breakers inserted, confirming that a slow down is avoided. Note that if N is larger than L, only one request breaker is needed.

11.6 Reduced-stack precharged full buffer (RSPCFB)

This section presents a 1-of-(N+1) QDI full-buffer pipeline template constructed by merging an RSPCHB with a modified WCHB. A schematic illustration of this reduced-stack precharged full buffer (RSPCFB) [11] is shown in Figure 11.35 and a more detailed implementation for dual-rail data is shown in Figure 11.36.

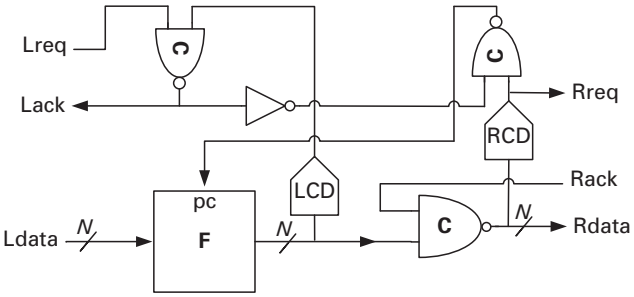

Figure 11.35. Abstract RSPCFB.

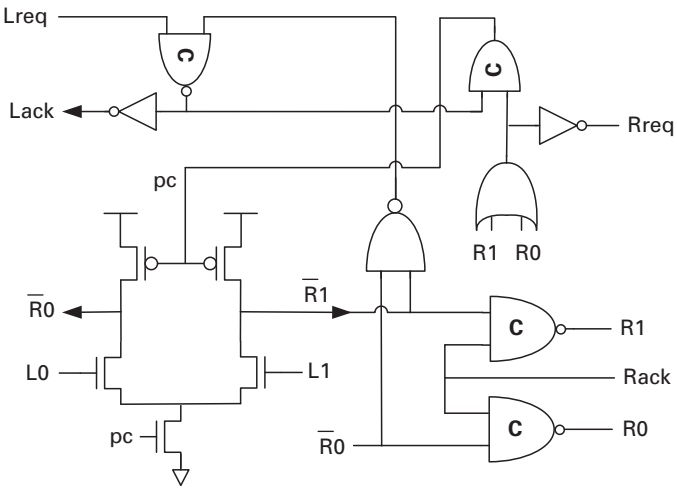

Figure 11.36. Detailed implementation of a 1-of-2 RSPCHB buffer.

The RSPCFB has two new features. First, the inverters from both half buffers have been removed to keep the forward latency of the new template at a two-gate delay. The inverters between the two half buffers can safely be removed because the RSPCHB has little gate load and the wire load can be minimized by placing and routing this template as a single unit. The output inverters are only necessary if the unit is driving a significant load and can be added as necessary. (However, a staticizer, not shown, is still necessary.) Second, the WCHB has to be modified to accept an input request signal and generate an output request signal. This input request signal drives a C-element whose other input is the RCD output. This C-element then triggers the internal acknowledgement to the RSPCHB part of the circuit instead of to the RCD alone. In addition, the output request signal is driven from the RCD output. One other difference is that the request signal is now active-low because the inverters have changed location (this is known as *bubble shuffling* [9]).

The circuit operates as follows. The completion detector of the RSPCHB part, referred to as the internal completion detector (ICD), detects the evaluation of the

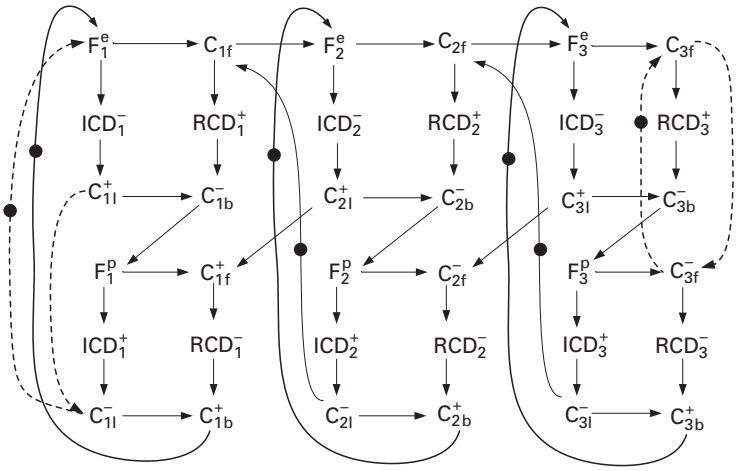

Figure 11.37. The marked graph behavior of a three-stage linear pipeline of an RSPCFB.

function block and asserts its output. The output of the ICD drives the C-element, which generates the acknowledge signal Lack to the previous stage after all the request lines associated with the data have also arrived. If the next stage is ready to accept new data, the acknowledgement signal Rack should already be de-asserted, allowing the C-elements in the forward path to pass the data to the next stage. Subsequently, the WCHB's RCD will assert its output, thus asserting the request signal to the next stage. The output of the RCD also drives the C-element C_b, which asserts the internal acknowledgement back to the RSPCHB part, allowing the function block to precharge. When the acknowledge signal Rack is de-asserted, the C-element in the forward path will de-assert its outputs. This will trigger the WCHB's RCD to de-assert Rreq and the C-element C_b to de-assert the internal acknowledgement back to the RSPCHB, thereby enabling the function block to re-evaluate.

Notice that the Rreq of the RSPCFB is taken from the output of the RCD instead of the C-element, unlike in the RSPCHB. The reason is that the WCHB part of the circuit has weak-conditioned logic, which will not reset until all inputs, including inputs from the RSPCHB part, have reset. This implicitly avoids the problem of preventing the assertion of the acknowledge signal back to the RSPCHB part, which delaying Rreq solved in the RSPCHB case. The advantage of this is that Rreq can be generated earlier. The disadvantage is that the timing margin on input channels to joins is reduced to 5–7 gate delays, depending on the buffering.

The RSPCFB has a minimum of 10 transitions per cycle, less than the RSPCHB, which has 12 transitions. The analytical cycle time, using the marked graph in Figure 11.37, can be expressed as

$$T_{\mathrm{RSPCFB}} = \max(2t_{\mathrm{eval}} + 2t_{\mathrm{CD}} + 4t_{\mathrm{c}}, t_{\mathrm{eval}} + t_{\mathrm{prech}} + 2t_{\mathrm{CD}} + 4t_{\mathrm{c}}).$$

The RSPCFB can be extended to handle non-linear pipeline structures in the same way as the RSPCHB, without any additional timing assumptions.

11.7 Quantitative comparisons

In this section we review HSPICE simulations, designed for comparison purposes, of the precharge and reduced-stack precharge-based templates [12]. Simple digital simulations tend not to reflect accurately the delay differences of pull-up or pull-down logic consisting of different numbers of transistors.

Since the goal in [12] was comparison, no attempt was made to fine-tune the transistor sizing to achieve optimum performance. In particular, all transistors were sized in order to achieve a gate delay roughly equal to that of a small inverter (WNMOS = 0.8 μm, WPMOS = 2 μm, and L = 0.24 μm) driving a same-sized inverter. For the purposes of this comparison, wire delay was also ignored.

For the half buffers, i.e. the PCHB and the RSPCHB, a linear dual-rail pipeline of buffers with 60 stages was constructed to achieve a static slack of 30, which means that it can hold 30 distinct data tokens. For the full buffers, i.e. the PCFB and the RSPCFB, 30 stages were used to achieve the same static slack. All pipelines could hold 30 distinct tokens.

HSPICE simulations were performed using a 0.25 Taiwan Semiconductor Manufacturing Co. (TSMC) process with a 2.5 V power supply at 25 °C. Figure 11.38(a) shows throughput versus token number triangles for the half buffers and Figure 11.38 (b) shows corresponding plots for the full buffers. The triangles for the PCHB and PCFB are indicated by the broken lines. Approximately 15 distinct points were obtained per pipeline for the triangle graphs using the HSPICE simulation. One key result obtained from this simulation is the dynamic slack (see Chapter 4) of each pipeline, which is the number of tokens required to achieve maximum throughput [2][3].

The PCHB achieves maximum throughput 772 MHz with dynamic slack 7.3. The RSPCHB is faster, with maximum throughput 920 MHz and dynamic slack 8.25. The throughput improvement is approximately 20%. For the full buffers, the PCFB achieves maximum throughput 707 MHz and dynamic slack 3.7. The RSPCHB is faster, with maximum throughput 1000 MHz and dynamic slack 5.9. The speed improvement is approximately 40%; however, owing to the C-elements in the forward path of the RSPCFB, the forward latency is about 15% slower. In both the half and full buffers, a higher dynamic slack was achieved. This means that the reduced-stack templates support more system-level concurrency and higher stage utilization.

Notice that although the PCFB has 12 and the PCHB 14 transitions per cycle, the PCFB is slower. This is partially due to the heavier load on the internal wiring in the PCFB compared with the PCHB. Clearly, careful transistor sizing and buffering can improve the performance of all pipeline templates; nevertheless, we expect the relative performances to remain approximately the same.

11.8 Token insertion

Sometimes it is necessary to initialize a channel with a token value. Figure 11.39 illustrates a 1-of-2 encoded buffer that inserts a logic 0 on its output channel upon

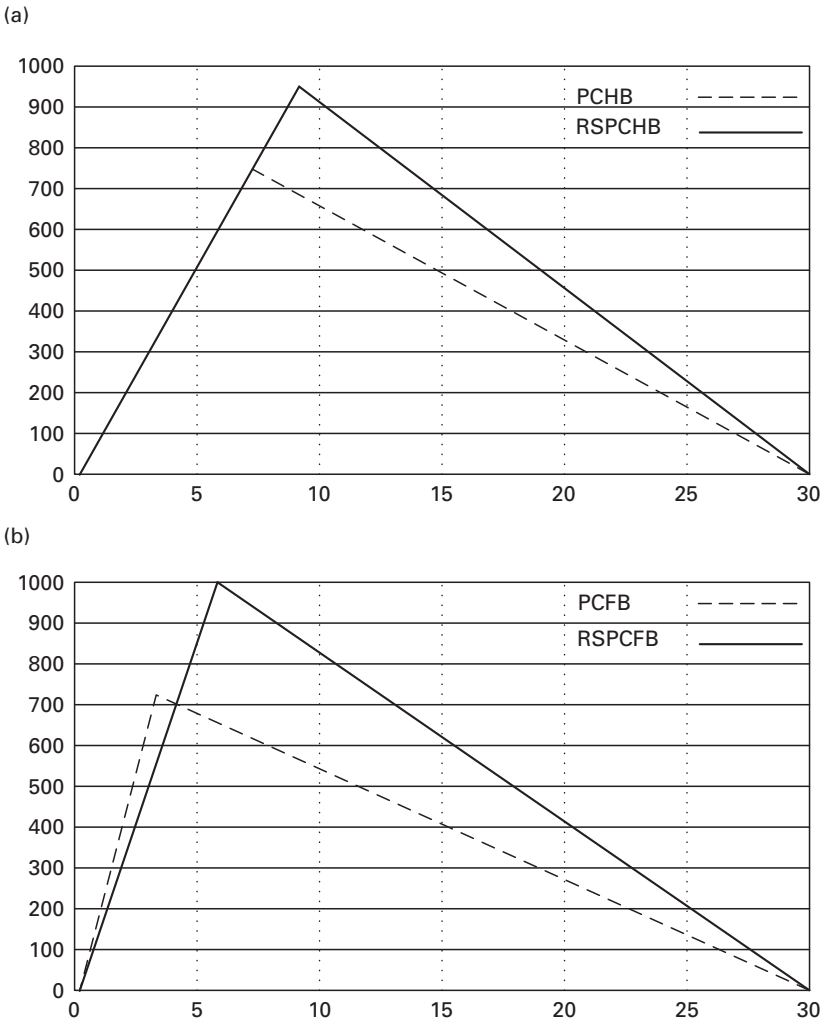

Figure 11.38. Throughput versus token number for (a) the PCHB and RSPCHB and (b) the PCFB and RSPCFB linear pipelines.

reset. Note that, by applying additional logic to the logic-1 rail R1 rather than the logic-0 rail R0, a 1 instead of a 0 can be inserted. The illustrated scheme also scales to 1-of-4 data encoding.

While the R1 output is, as usual, driven by an inverter, the rail that is pulled high upon reset, R0, is driven by a NAND2 gate. During normal operation both data rails, R0 and R1, are pulled up by the corresponding input rails, L0 and L1, as in a WCHB template. And finally there is additional circuitry to force a data value on the outputs and acknowledge the previous signal as a result of the reset.

The reset and token insertion operation can be described as follows. As Reset is an active high signal, at the resetting of the circuit Reset will go to 1 and $\overline{\text{Reset}}$ will go to 0, pulling up the $\overline{x}$, $\overline{y}$, $\overline{R1}$, and $\overline{R0}$ signals. When $\overline{y}$ and $\overline{R0}$ go to 1, R0 goes to 0.

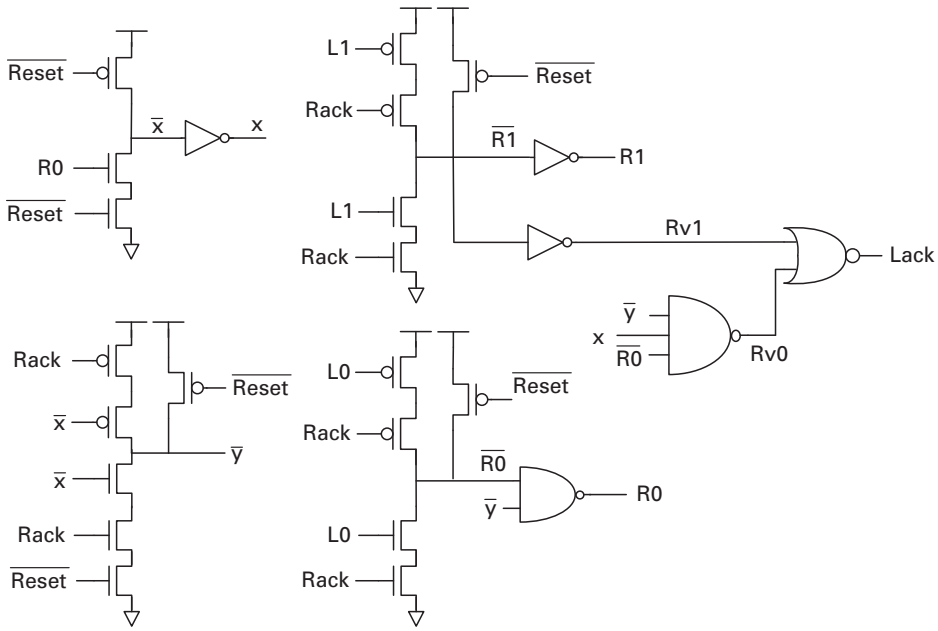

Figure 11.39. A TOKEN_BUF_1-of-2.

In addition, R1 is directly driven by $\overline{R1}$, therefore R1 will also reset to 0. Since x is a 0, the NAND3 output Rv0 will go to 1, forcing the acknowledge signal Lack also to reset. Since Rack is tied to the Lack value of the following stage, which also will reset, Rack will also go to 0. At this point all output rails and acknowledge signals are 0, just like the signals coming out of a regular PCHB-based stage, as illustrated in Figure 11.40.

However, when the Reset signal is de-asserted to 0, setting $\overline{\text{Reset}}$ to 1, the structure of TOKEN_BUF will yield a different state from the PCHB. Since $\overline{x}$ was 1, once $\overline{\text{Reset}}$ is 1 and Rack goes back to 1, its expected initial value, $\overline{y}$, will be pulled down to 0. This forces the NAND3 output to stay high and consequently the Lack output signal to remain low. When node $\overline{y}$ goes low this causes R0 to go low also, via the NAND2, causing node $\overline{x}$ to go low as well. In fact, node $\overline{x}$ will stay 0 forever unless another reset happens. This sequence of node changes is illustrated in Figure 11.41. From this state, $\overline{y}$ will go high after the following stage consumes the token provided by this stage and Rack is lowered. Moreover, $\overline{y}$ will remain high until another reset occurs, because the NMOS stack is gated by the reset signal. Once $\overline{y}$ is set to 1 and x is set to 1, the circuit will be equivalent to that illustrated in Figure 11.42.

The resulting circuit above is very similar to a dual-rail WCHB. The NAND2 now acts as an output inverter, and the NAND3 gate also acts as an inverter. The Lack signal is thus effectively driven by two inverters followed by a two-input NOR gate rather than a two-input NAND gate followed by an inverter, as shown in Figure 11.1. In both cases, the functionality is the AND of the inverted outputs $\overline{R0}$ and $\overline{R1}$.

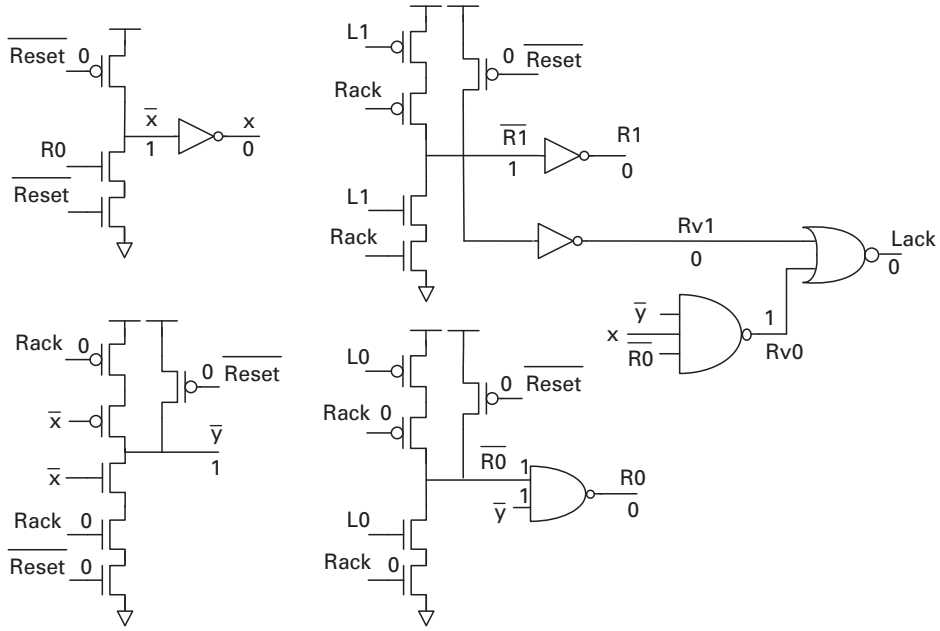

Figure 11.40. TOKEN_BUF when reset is asserted.

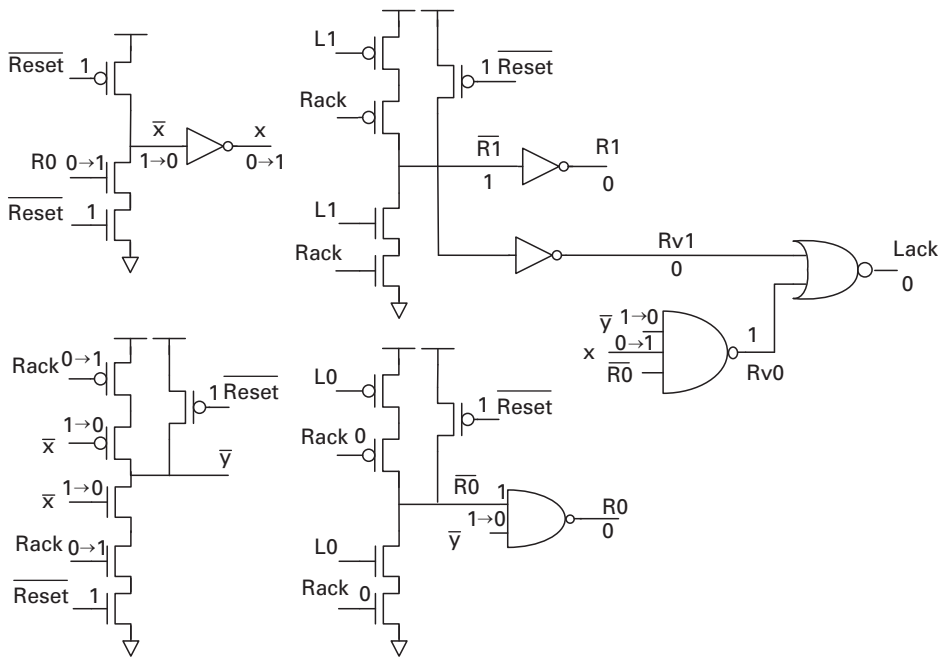

Figure 11.41. TOKEN_BUF immediately after reset is de-asserted.

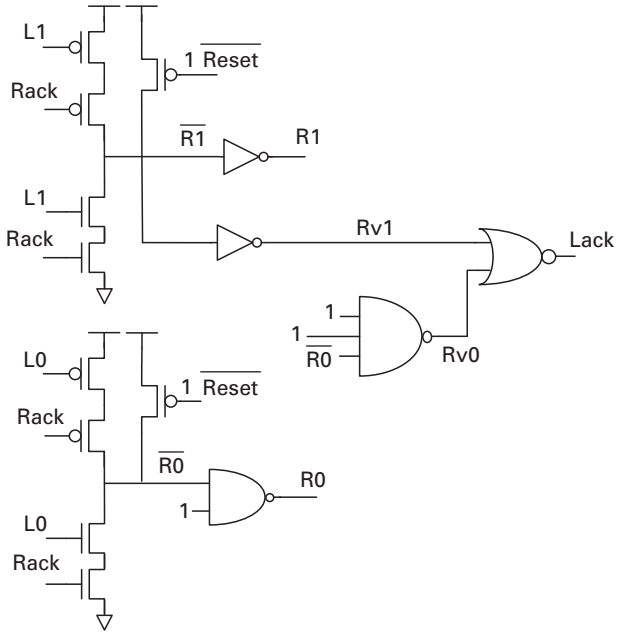

Figure 11.42. TOKEN_BUF after reset under normal operation.

11.9 Arbiter

As mentioned in Chapter 2, arbiters are often required to manage access to a shared resource. In this section we will present a two-way QDI pipelined arbiter. When two independent channels assert a request, the problem that the arbiters must solve is which request is granted first. The main building block of an arbiter is the *mutual-exclusion* element, which is used for making a non-deterministic choice between two inputs. If only one request is available then the choice is obvious, and the latency through the arbiter is a constant. However, if two requests arrive at the same time (or close enough) then the time it takes for the arbiter to make a decision is potentially unbounded due to its metastability.

The two-way arbiter covered in this section consists of two parts, the core and the metastability filter, as illustrated in Figure 11.43.

The core part of the arbiter implements the acknowledgement generation and the cross-coupled portion of the mutual exclusion element and is illustrated in Figure 11.44. The classic metastability filter [1] is illustrated in Figure 11.45.

Let us review its operation. In the first scenario, assume that there is only one request. In this case the request line of, for example, the L0 1-of-1 input channel will go high. The acknowledge signals L0ack and L1ack are both high and the other request line L1req is low. After reset (the reset transistors not shown), $\overline{r0}$ and $\overline{r1}$are high and r0, r1 and the output signals W0, W1 are low. Now $\overline{r0}$ will go low while $\overline{r1}$ remains high. The filter will raise the r0 output and, since Wack is high, the signals will propagate through the C-element and the output inverter and the W0 rail will go

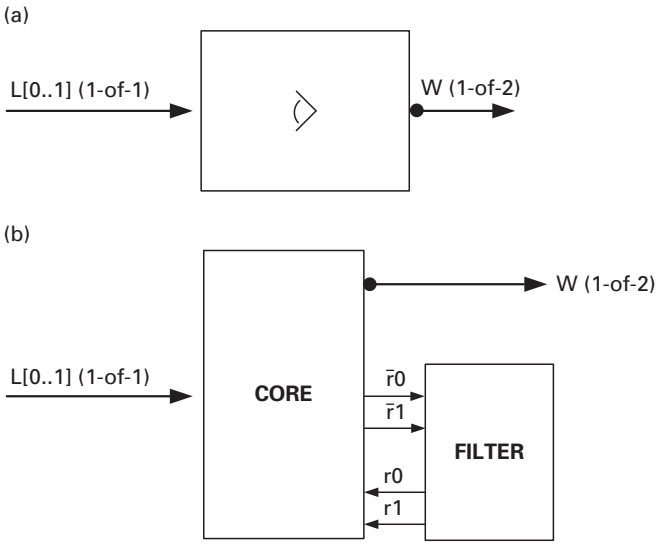

Figure 11.43. A QDI pipelined arbiter: (a) symbol, (b) partitioning.

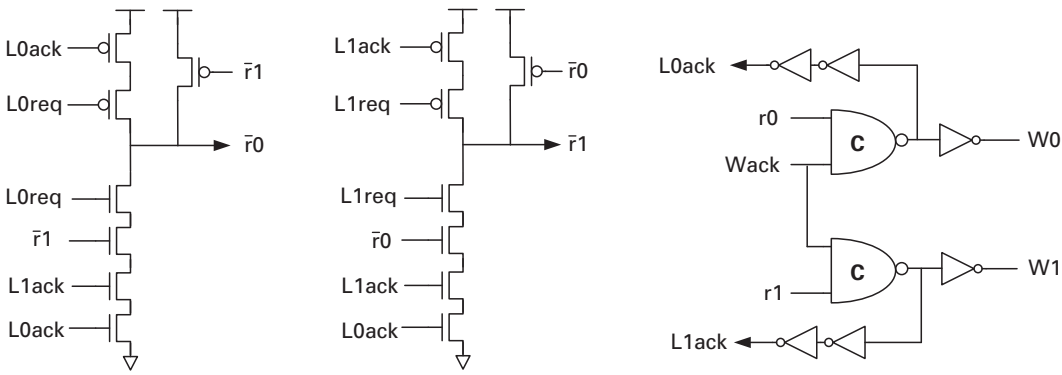

Figure 11.44. The QDI pipelined arbiter core.

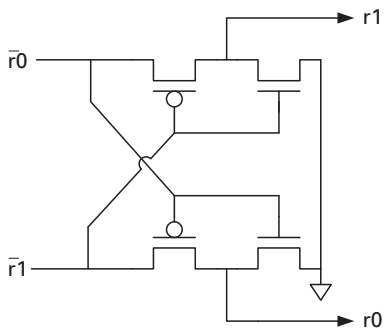

Figure 11.45. Arbiter metastability filter.

high. In addition, the acknowledge signal L0ack will go low, declaring the L0 request channel the winner and granting it access to the resource that is shared. In this scenario the latency is four transitions, unlike most other QDI templates.

If two requests arrive at the same time (or sufficiently close), however, then the latency can be unbounded owing to the time it takes to complete metastability resolution. In particular, in the case when both requests L0req and L1req go high, both $\overline{r0}$ and $\overline{r1}$ will try to go low. However, as $\overline{r0}$ goes low, it will try to pull $\overline{r1}$ high and as $\overline{r1}$goes low it will try to pull $\overline{r0}$ high. After some, possibly a long, time the slight variations due to noise in the actual implementation will allow the filter to pull one of the outputs high and as a result only one of the requests is granted. Either r0 or r1 will go high, therefore only one of the inputs will be acknowledged. The remainder of the handshaking protocol remains unchanged.

11.10 Exercises

11.1. Consider an unconditional cell with 1-of-2 inputs A and B and a 1-of-2 output channel Z. The output Z implements Z = NAND(A, B).

Draw the transistor–gate-level diagrams for WCHB, PCHB, and RSPCHB implementations of this cell. Explain why PCHB and/or RSPCHB may be preferred over WCHB implementation.

If you also wanted an AND cell, do you need to change the design? If not, explain how you can implement an AND cell without changing the design.

11.2. Consider a leaf cell that takes two 1-of-4 inputs A and B and produces a 1-of-2 output C. If the number of 1s in the binary form of the two inputs is odd then it produces a 0 and if the number of 1s is even then it produces a 1. (For example, if a 1-of-4 L[0] is 3 then the binary form is 2′b11 and if a 1-of-4 L[1] is 1 then the binary form is 2′b01. In this case the total number of 1s is odd and the output should be 0.)

Draw the transistor–gate-level diagrams for PCHB and RSPCHB implementations of this cell.

11.3. Consider a leaf cell where A and B are 1-of-2 input channels, F is a 1-of-3 encoded input control channel, and X and Y are 1-of-2 output channels. It has the following functionality:
if (F = 0) then X = XOR(A, B) and Y = A,
if (F = 1) then Y = AND(A, B) and X = B,
if (F = 2) then X = B and Y = A.

Draw the transistor–gate-level diagrams for PCHB and RSPCHB implementations.

11.4. Consider a leaf cell with conditional output behavior as follows: A is a 1-of-2 input channel, S is a 1-of-2 input channel, and O is a 1-of-2 output channel; if S is a 1 then the token on A is consumed and copy of the value is sent on O. If S is a 0, the token on A is consumed but no data is sent on S.

Draw the transistor/gate-level diagrams for implementations using the WCHB, PCHB, and RSPCHB templates.

11.5. Consider a leaf cell with a special conditional input behavior as follows: A is a 1-of-2 input channel, S is a 1-of-2 input channel, and O is a 1-of-2 output channel. If S is a 1, the token on A is consumed and a copy of the data is sent on O. If S is a 0, a valid token on A is awaited and then a token is sent along O but A is not acknowledged (i.e. the token is not consumed). In this way, the token on A is conditionally re-used, implementing a type of memory.

Draw transistor/gate-level diagrams for implementations that follow the PCHB and RSPCHB templates. More specifically, these templates may need to be modified slightly to accommodate this specialized input-conditional behavior in which you always wait for the token but conditionally acknowledge (i.e. consume) it.

References

[1] C. L. Seitz, "System timing," in *Introduction to VLSI Systems*, C. A. Mead and L. A. Conway, eds., Addison-Wesley, 1980.
[2] T. E. Williams, "Self-timed rings and their application to division," Ph.D. thesis, Stanford University, 1991.
[3] A. Lines, "Pipelined asynchronous circuits," Ph.D. thesis, California Institute of Technology, 1998.
[4] J. C. Ebergen, "Translating programs into delay-insensitive circuits," Ph.D. thesis, Dept of Mathematics and Computer Science, Eindhoven University of Technology, 1987.
[5] J. T. Udding, "A formal model for defining and classifying delay-insensitive circuits," *Distributed Computing*, vol. 1, no. 4, pp. 197–204, 1986.
[6] A. J. Martin, "The limitations to delay-insensitivity in asynchronous circuits," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, 1990, pp. 263–278.
[7] A. J. Martin, "Compiling communicating processes into delay-insensitive VLSI circuits," *Distributed Computing*, vol. 1, no. 4, pp. 226–234, 1986.
[8] K. van Berkel, F. Huberts, and A. Peeters, "Stretching quasi delay insensitivity by means of extended isochronic forks," in *Proc. Conf. on Asynchronous Design Methodologies*, May 1995, pp. 99–106.
[9] A. J. Martin, "Programming in VLSI: from communicating processes to delay-insensitive circuits," in *Proc. Conf. on Developments in Concurrency and Communication*, UT Year of Programming Series, 1990, pp. 1–64.
[10] A. M. Lines, Private communication, 2001.
[11] R. O. Ozdag and P. A. Beerel, "High-speed QDI asynchronous pipelines," in *Proc.* ASYNC'02, April 2002, pp. 13–22.
[12] R. O. Ozdag, *"Template-based asynchronous circuit design,"* Ph.D. thesis, University of Southern California, November 2003.

12 Timed pipeline templates

It is possible to achieve higher performance with pipelined templates by applying timing assumptions rather than designing them to be quasi-delay-insensitive. In fact, designing templates with assumptions on the relative order of certain signal transitions can not only speed up the operation of the circuit but also lower area and power consumption.

12.1 Williams' PS0 pipeline

Figure 12.1 shows one stage of Williams' PS0 pipeline [1][2], one of the earliest proposed timed pipeline templates. The pipeline stage consists of a dual-rail function block F and a completion detector. The output of the completion detector is fed back to the previous stage as the acknowledge signal. The completion detector checks the validity or absence of data at the outputs. There is no input-completion detector.

The function block is implemented using domino logic. The precharge and evaluation control input Pc of each stage comes from the output of the next stage's completion detector. The precharge logic can hold its data outputs even when its inputs are reset, therefore it also provides the functionality of an implicit latch. Each completion detector verifies the completion of every computation and precharge of its associated function block.

The operation of the PS0 pipeline is quite simple. Stage N is precharged when stage $N + 1$ finishes evaluation. Stage N evaluates when stage $N + 1$ finishes reset. This protocol ensures that consecutive data tokens are always separated by bubbles also known as holes.

The complete cycle of events for a pipeline stage is derived by observing how a single data token flows through an initially empty pipeline. The sequence of events from one evaluation by stage 1 to the next such evaluation is: (i) stage 1 evaluates, (ii) stage 2 completes, (iii) stage 2's completion detector detects completion of evaluation, and then (iv) stage 1 precharges. At the same time, after completion of step (ii), (iii′) stage 3 evaluates, then (iv′) stage 3's completion detector detects completion of evaluation and initiates the precharge of stage 2. After these steps, (v) stage 2 precharges and finally (vi) stage 2's completion detector detects the completion of precharge, thereby releasing the precharge of stage 1 and enabling it

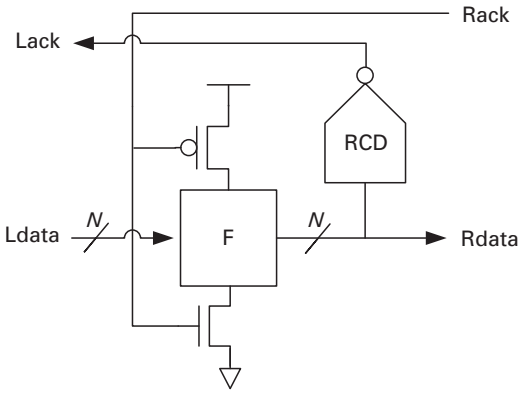

Figure 12.1. A stage of Williams' PS0 pipeline.

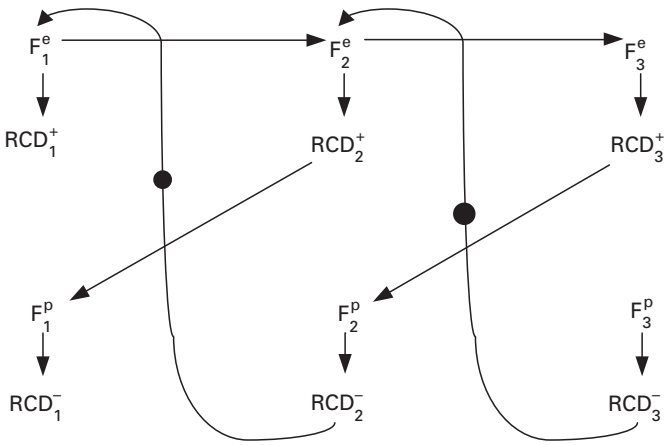

Figure 12.2. The marked graph for the PS0 pipeline.

to evaluate once again. Thus there are six events (some compound) in the complete cycle for a stage, from one evaluation to the next.

The protocol for a PS0 pipeline stage is captured by the marked graph for a four-stage pipeline illustrated in Figure 12.2. From the marked graph, it is possible to derive the pipeline's analytical cycle time:

$$T_{\mathrm{PS0}} = 3t_{\mathrm{eval}} + 2t_{\mathrm{CD}} + t_{\mathrm{prech}}.$$

In particular, as illustrated in this marked graph, the PS0 pipeline has a timing assumption that the data precharges before the corresponding acknowledgement is reset. This corresponds to the relative timing assumption among successive stages i, $i + 1$, and $i + 2$

$$t_{\mathrm{prech},i} + t_{\mathrm{CD},i} \leq t_{\mathrm{eval},i+2} + t_{\mathrm{CD},i+2} + t_{\mathrm{prech},i+1} + t_{\mathrm{CD},i+1},$$

which must be verified during physical design.

12.2 Lookahead pipelines overview

The lookahead pipeline (LP) designs use Williams' PS0 as a starting point and apply certain optimizations to its protocol that are targeted towards reducing the overall cycle time. The basic strategy is anticipating the arrival of certain critical events and optimizing the template accordingly.

In these lookahead pipelines, developed by researchers at Columbia University, two specific optimizations are used, *early evaluation* and *early done*. In early evaluation, a pipeline stage uses control information not only from the subsequent stage but also from stages *further down* the pipeline. This information is used to give the stage a headstart on its evaluation phase. In the second optimization, early done, a stage signals to its previous stage when it is *about to* precharge or evaluate, instead of after it has completed those actions. This information is used to give the pipeline stage a headstart on its evaluation phase as well as on its precharge phase. The net result of applying these two optimizations is a significant reduction in pipeline cycle time, and consequently a dramatic increase in throughput, with no net increase in latency. The remainder of this chapter presents three new pipeline designs, LP3/1, LP2/2, and LP2/1, in detail. The LP3/1 pipeline uses early evaluation, the LP2/2 pipeline uses early done, and the LP2/1 is a hybrid that combines both optimizations [3][4].

12.3 Dual-rail lookahead pipelines

12.3.1 The LP3/1 pipeline

In the LP3/1 dual-rail lookahead pipeline design style, as mentioned above, an *early evaluation* protocol is used, in which a pipeline stage receives control information not only from the subsequent stage but also *from its successor*. As a result, LP3/1 pipelines have shorter cycles than Williams' PS0 pipelines: a complete cycle for a stage of an LP3/1 pipeline consists of four events, as opposed to six events for a stage of a PS0 pipeline. The new pipeline derives its name from the fact that three out of the four events in every stage's cycle fall in its evaluation phase, and one event falls in its precharge phase. Using this terminology, Williams' PS0 would be 3/3.

Figure 12.3 shows an implementation of the LP3/1 pipeline template. The key difference is that, unlike the PS0 stages, an LP3/1 stage has two control inputs. The first control input, pc, comes from the next stage, as in PS0. The second control input, eval, comes from the completion detector two stages ahead. This second input is the key to achieving a shorter cycle time.

The key idea is that stage N can evaluate as soon as stage $N+1$ has started precharging, instead of waiting until stage $N+1$ has completed precharging. This idea can be used because a dynamic logic stage undergoing precharge is insensitive to changes on its inputs. Therefore, as soon as stage $N+1$ begins to precharge, stage N can proceed with its next evaluation. Now, since stage $N+1$ begins

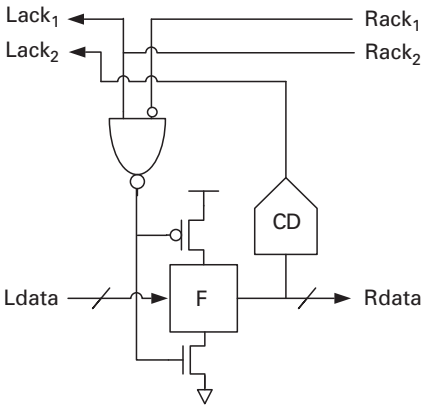

Figure 12.3. The LP3/1 pipeline.

precharging when stage $N + 2$ completes evaluation, the new condition for evaluation is: *evaluate N when N + 2 completes evaluation*. The condition for precharge remains unchanged: *precharge N when N + 1 completes evaluation*. Therefore, stage N needs inputs from both the completion detector of $N + 1$ as well as from that of $N + 2$. Figure 12.3 shows how the two control inputs are combined inside the implementation of one stage. Evaluation is enabled when eval is asserted high, or pc is de-asserted low, or both. The former condition, eval = high, corresponds to stage $N + 2$ completing its computation (i.e. stage $N + 1$ *starting* its precharge; see Figure 12.3). The latter condition, pc = low, is identical to the evaluation condition of PS0. Precharge is enabled when both pc is asserted high and eval is de-asserted low.

In LP3/1, there are two distinct control inputs for a stage N, i.e. the outputs from stages $N + 1$ and $N + 2$. The precharge phase of stage N begins after stage $N + 1$ has finished evaluating (pc is asserted high), much like PS0. However, this phase is now shortened: precharge terminates when stage $N + 2$ has finished evaluating (eval is asserted high). In contrast, in PS0 precharge terminates only when stage $N + 1$ has finished precharging. At this point stage N enters its evaluate phase. In LP3/1, the evaluate phase continues until two distinct conditions hold, which drive the stage into the next precharge: (a) stage $N + 1$ has completed evaluation (as in PS0, pc is asserted high), and (b) stage $N + 2$ has completed precharging (eval is de-asserted low). The NAND gate in Figure 12.3 (with a bubble on its eval input) directly implements these two conditions.

During LP3/1's evaluate phase, the early eval signal from stage $N + 2$ may be *non-persistent:* it may be de-asserted low even before stage N has had a chance to evaluate its new data! However, one-sided timing constraints are imposed to insure a correct evaluate phase: pc = low will arrive in time to take over control of the evaluate phase, which will then be maintained until stage N has completed evaluating its inputs (as in PS0).

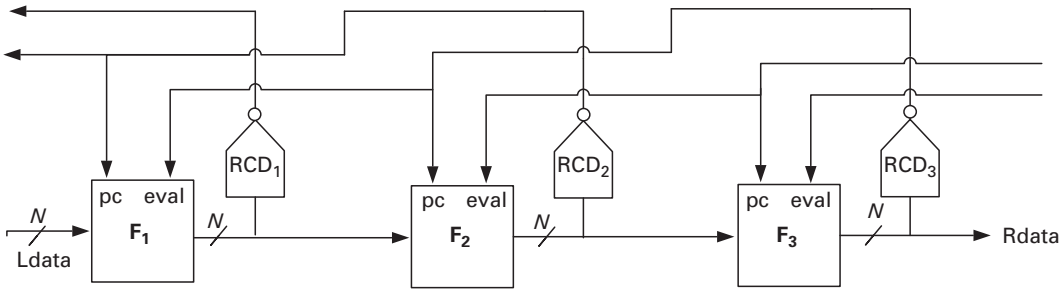

Figure 12.4. A three-stage LP3/1-based linear pipeline.

The complete cycle of events for a stage, from one evaluation until the next, can be derived from Figure 12.4: (i) stage 1 evaluates, (ii) stage 2 evaluates, (iii) stage 2's completion detector detects the completion of precharge, and then (iv) stage 1 precharges. At the same time, after completing step (ii), (iii′) stage 3 evaluates, and (iv′) stage 3's completion detector detects the completion of stage 3's evaluation, thereby enabling two subsequent events, the precharge of stage 2 and the next evaluation of stage 1 ("early evaluation"). Thus, there are only four events in the complete cycle for a stage from one evaluation to the next, which is two less than the six events in PS0. This reduction has been brought about by the elimination of the two events of stage 2's precharge phase from stage 1's cycle.

The cycle time of the pipeline is therefore

$$T_{\mathrm{LP3/1}} = 2t_{\mathrm{eval}} + t_{\mathrm{CD}} + t_{\mathrm{NAND}},$$

where t_{NAND} is the delay through the NAND gate for the early evaluation signal. Thus the LP3/1 cycle time is $t_{\mathrm{prech}} + t_{\mathrm{CD}} - t_{\mathrm{NAND}}$ shorter than that of the PS0.

The per-stage forward latency is simply the evaluation delay of a stage, as in the PS0:

$$L_{\mathrm{LP3/1}} = t_{\mathrm{eval}}.$$

12.3.2 The LP2/2 pipeline

The key feature of the dual-rail LP2/2 pipeline is that a pipeline stage is now allowed to signal its previous stage when it is about to evaluate (or precharge) instead of after it has completed that action. Thus, this pipeline uses an *early done* protocol. The LP2/2 pipelines have shorter cycle times than PS0 and, like LP3/1, the cycle of an LP2/2 stage consists of four events; however, unlike LP3/1, the stages of an LP2/2 pipeline have only one control input as opposed to two, thereby reducing the loading on the completion detectors.

Figure 12.5 shows a block diagram of an LP2/2 pipeline. The stages are similar to those used in PS0, but with one key difference: the completion detectors are placed before their functional blocks. The idea is to let the previous pipeline stage know when the current stage is about to evaluate (or precharge).

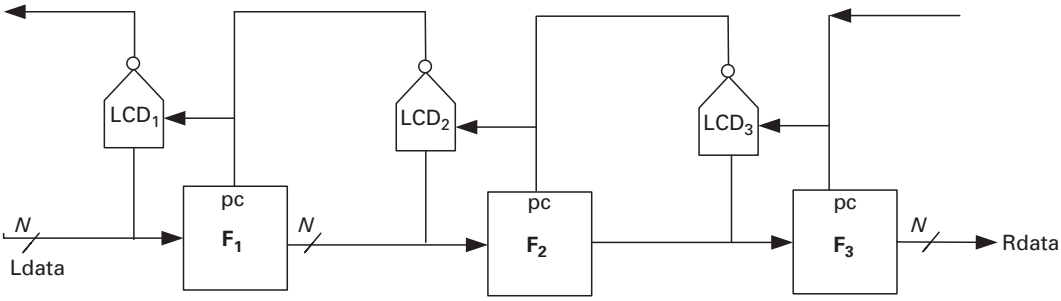

Figure 12.5. A three-stage LP2/2 based linear pipeline.

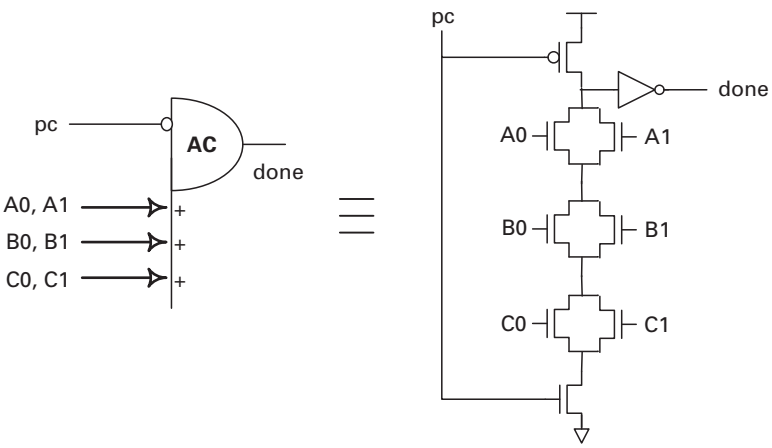

Figure 12.6. The LP2/2 completion detector.

A modified completion detector is needed in order to generate the "early done" signal. The completion detector now requires an extra input, the stage's control input pc. The functionality of the completion detector is as follows. The completion detector asserts done (done goes high) when the stage is about to evaluate: the stage is enabled to evaluate (pc is de-asserted low), and it has valid dual-rail inputs. The completion detector de-asserts done (done goes low) when the stage is about to precharge: pc is asserted high. Thus, the done signals are produced in parallel with the precharge or evaluation by the associated function block, instead of after its completion. Note that these conditions are asymmetric: a single condition (pc is asserted high) enables the stage to precharge and its completion detector to indicate that precharge is complete.

This new completion detector is implemented using an asymmetric C-element, as illustrated in Figure 12.6. From the figure, it is clear that this particular asymmetric C-element is a degenerate special case: it can be regarded as simply a precharged dynamic gate, which de-asserts done (low) whenever pc is asserted high.

As before, a complete cycle of events for stage 1 can be traced; see Figure 12.5. From one evaluation to the next, it consists of four events: (i) stage 1 evaluates, (ii) stage 2's completion detector detects the "early done" of stage 2's evaluation (in parallel with stage 2's evaluation), thereby asserting the precharge control of stage 1, and then (iii) stage 1 precharges. At the same time, after completing step (i), (ii′) stage 2 evaluates, (iii′) stage 3's completion detector detects the "early done" of stage 3's evaluation (in parallel with stage 2's evaluation), thereby asserting the precharge control of stage 2, and (iv) stage 2's completion detector detects the "early done" of stage 2's precharge (in parallel with stage 2's precharge), thereby enabling stage 1 to evaluate once again in the next step.

Thus, the cycle time of the pipeline is

$$T_{\mathrm{LP2/2}} = 2t_{\mathrm{eval}} + 2t_{\mathrm{CD}},$$

which is $t_{\mathrm{eval}} + t_{\mathrm{prech}}$ shorter than that of PS0. The latency is identical to that of PS0 and LP3/1:

$$L_{\mathrm{LP2/2}} = t_{\mathrm{eval}}.$$

12.3.3 The LP2/1 pipeline

The LP2/1 is basically a hybrid. It combines the "early evaluation" of LP3/1 and the "early done" of LP2/2. Consequently, an LP2/1 pipeline has the shortest analytical cycle time of the three LP design styles: a cycle for a stage consists of only three events. Figure 12.7 shows the implementation of an LP2/1 pipeline.

Each stage uses information from two succeeding stages, as in LP3/1, and also employs early completion detection, as in LP2/2. A complete cycle of events for stage 1 can again be traced in the figure. From one evaluation to the next it consists of three events: (i) stage 1 evaluates, (ii) stage 2's completion detector detects the "early done" of stage 2's evaluation (in parallel with stage 2's evaluation), thereby asserting the precharge control of stage 1, and then (iii) stage 1 precharges. At the same time, after the completion of step (i), (ii′) stage 2 evaluates, and (iii′) stage 3's completion detector detects the "early done" of stage 3's evaluation, thus enabling the evaluation of stage 1 in the next step. Thus, the cycle time is

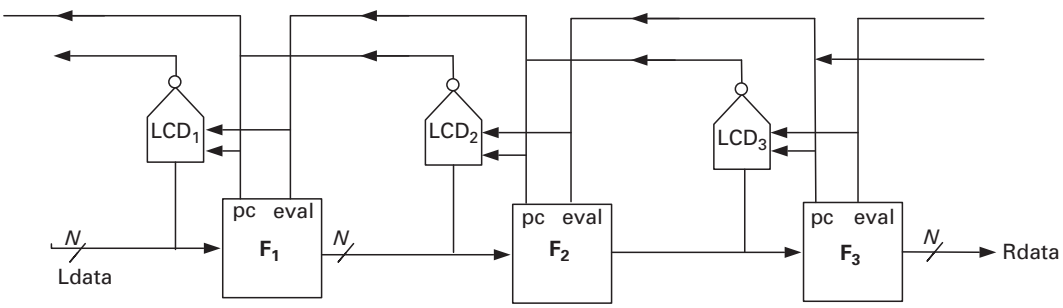

Figure 12.7. A three-stage LP2/1-based linear pipeline.

$$T_{\mathrm{LP2/1}} = 2t_{\mathrm{eval}} + t_{\mathrm{CD}} + t_{\mathrm{NAND}},$$

which is $t_{\mathrm{eval}} + t_{\mathrm{prech}} + t_{\mathrm{CD}} - t_{\mathrm{NAND}}$ shorter than that of PS0. Once again, the latency is identical to that of PS0:

$$L_{\mathrm{LP2/1}} = t_{\mathrm{eval}}.$$

12.4 Single-rail lookahead pipelines

12.4.1 The LPSR2/2 pipeline

The LPSR2/2 can be thought of as a derivative of LP2/2, or of PS0, adapted to a single-rail bundled datapath.

Figure 12.8 illustrates a single LPSR2/2 template stage and Figure 12.9 illustrates a three-stage pipeline implemented with the LPSR2/2. Each pipeline stage has a function block and a control block. The function block alternately

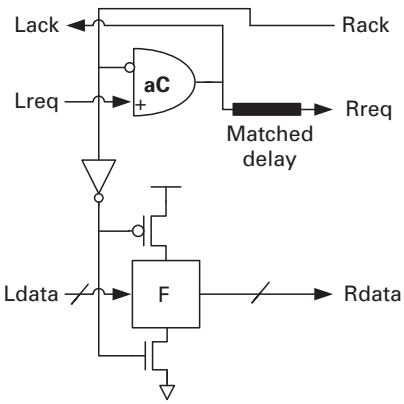

Figure 12.8. A single LPSR2/2 stage.

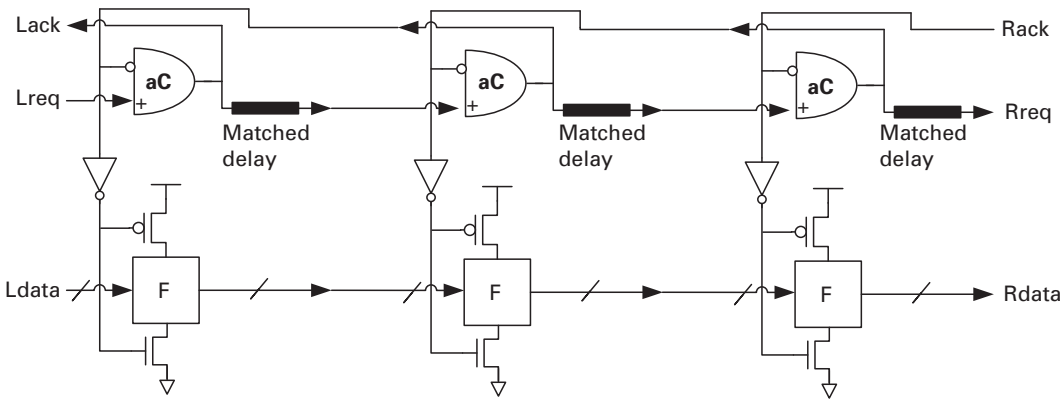

Figure 12.9. A three-stage LPSR2/2-based linear pipeline.

evaluates and precharges. The control block generates the bundling signal to indicate the completion of evaluation (or precharge). The bundling signal is passed through a suitable delay, allowing time for the dynamic function block to complete its evaluation (or precharge). This signal is communicated to two stages: (i) to the previous stage, to indicate "done", and (ii) to the next stage, to indicate a request (req) (i.e. for valid data).

The pipeline protocol is very similar to that of PS0. When a stage has finished evaluating, it tells the previous stage to precharge. Similarly, when a stage has finished precharging, it tells the previous stage to evaluate. In addition, the done signal is passed forward to the next stage, indicating that the evaluation (or precharge) is complete. However, there are two subtle optimizations that take advantage of the innate property of dynamic logic. The first is aimed at reducing the cycle time; the second is aimed at decreasing latency.

In the first optimization the done signal for the previous stage is tapped off before rather than after the matched delay. In spirit, this optimization is similar to the "early done" of LP2/2. The same justification applies. For footed dynamic logic, it is safe to indicate the completion of precharge as soon as the precharge cycle begins: during precharge, the stage is effectively isolated from changes at its inputs. Likewise, for a dynamic stage, it is safe to indicate completion of evaluation as soon as the stage begins to evaluate on valid inputs; once the stage has evaluated, its outputs are effectively isolated from a reset at the inputs. This early tap-off optimization has a significant impact on the pipeline performance: the cycle time is reduced by an amount equal to two matched delays.

In the second optimization an early precharge-release is allowed. In dynamic logic, unlike static logic, the function block can be precharge-released before new valid inputs arrive. Once data inputs arrive, the function block starts computing its data outputs. Similarly, once the matched bundling input arrives, the bundling output (req) is also generated. Thus, in this design the precharge-release of the function block is completely decoupled from the arrival of the inputs. In contrast, in other recent pipeline designs the function block is precharge-released only after the bundling input has been received [5]; this latter requirement typically adds extra gates to the critical forward path of the pipeline.

In LPSR2/2 the optimization results in a reduction in the forward latency. A complete cycle of events for a stage in LPSR2/2 is quite similar to that in PS0; see Figure 12.9. From one evaluation of stage 1 to the next, the cycle consists of four events: (i) stage 1 evaluates, (ii) stage 2 evaluates, (iii) stage 3 evaluates, asserting the precharge input for stage 2, and finally (iv) stage 2 precharges, enabling stage 1 to evaluate once again. The following notation is used for the various delays associated with this pipeline:

$T_{\mathrm{LP2/1}} = 2t_{\mathrm{eval}} + t_{\mathrm{CD}} + t_{\mathrm{NAND}}$,
t_{eval} = time for a stage evaluation,
t_{gC} = delay of the control block (generalized C-element),
t_{delay} = magnitude of the matched delay.

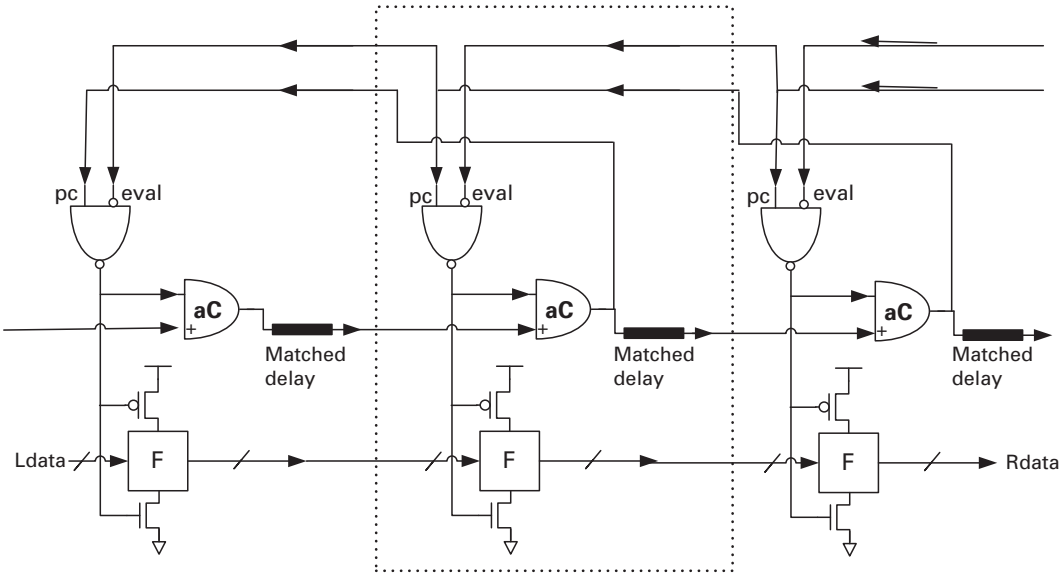

Figure 12.10. A three-stage LPSR2/1-based linear pipeline.

For ideal operation, we assume that t_{delay} is no larger than necessary: $t_{\text{delay}} = t_{\text{eval}} - t_{\text{gC}}$. In this notation the delays of steps (i) and (ii) in the cycle traced above are each t_{eval}. The delays of steps (iii) and (iv) are each t_{gC}. Therefore, the pipeline cycle time is

$T_{\text{LPSR2/2}} = 2 = 2t_{\text{eval}} + 2t_{\text{gC}}$, the per-stage forward latency of the pipeline,

thus

$$L_{\text{LPSR2}}/2 = t_{\text{eval}}.$$

12.4.2 The LPSR2/1 pipeline

The LPSR2/1 can be thought of as a derivative of LP2/1, or LP3/1, adapted to a single-rail bundled datapath.

Figure 12.10 shows the structure of the LPSR2/1 pipeline. Each stage has a function block and a control block identical to those of the first single-rail design. However, a stage receives control inputs not only from the subsequent stage (pc) but also from its successor (eval). Much like LP2/1 and LP3/1, the second control input is used for "early evaluation."

The sequencing of control is very similar to that in LP2/1 or LP3/1. A complete cycle of events, from one evaluation of stage 1 to the next, consists of three events: (i) stage 1 evaluates, (ii) stage 2 evaluates, and finally (iii) stage 3 evaluates, triggering the "early evaluation" of stage 1. Thus, the cycle time is

$$T_{\text{LPSR2}} = 1 = 2t_{\text{eval}} + t_{\text{gC}} + t_{\text{NAND}}.$$

The analytical cycle time is somewhat better than that of LPSR2/2, because $t_{\text{NAND}} < t_{\text{gC}}$. Once again, the forward latency $L_{\text{LPSR2/1}} = t_{\text{eval}}$.

12.5 High-capacity pipelines (single-rail)

The LPHC pipeline combines both asynchronous and self-resetting features. A novelty is that it *decouples* the control of pull-up and pull-down. A dynamic gate is now controlled by two separate inputs, pc and eval. Note that the pc and eval signals of this section are different from the pc and eval signals of LPSR2/1 pipelines. Whereas the latter pc and eval were first combined using a NAND gate and then used inside the dynamic gate, here pc and eval are used directly as two separate gate inputs.

Using these signals, unlike in traditional approaches, a stage is driven through three distinct phases in sequence: Evaluate, Isolate, and Precharge. In the Isolate phase, a stage holds its outputs stable irrespectively of any changes at its inputs. As a result, adjacent pipeline stages are capable of storing distinct data items. Figure 12.11 shows a block diagram of a three-stage lookahead pipeline high-capacity (LPHC)-based pipeline.

Each stage consists of three components: a function block, a completion detector, and a stage controller. Much like LPSR2/1, the function block alternately produces data tokens and bubbles for the next stage, and the completion detector indicates completion of the stage's evaluation or precharge. The third component, the stage controller, generates the decoupled signals pc and eval that control the function block and completion detector. Figure 12.12 shows one gate of a function block in a pipeline stage.

The pc input controls the pull-up stack and the eval input controls the "foot" of the pull-down stack. Precharge occurs when pc is asserted low and eval is de-asserted low. Evaluation occurs when eval is asserted high and pc is de-asserted high. When both signals are de-asserted, the gate output is effectively isolated from the gate inputs; this is the Isolate phase. To avoid a short circuit, pc and eval are never simultaneously asserted. As in the earlier design, an asymmetric C-element aC is used as a completion

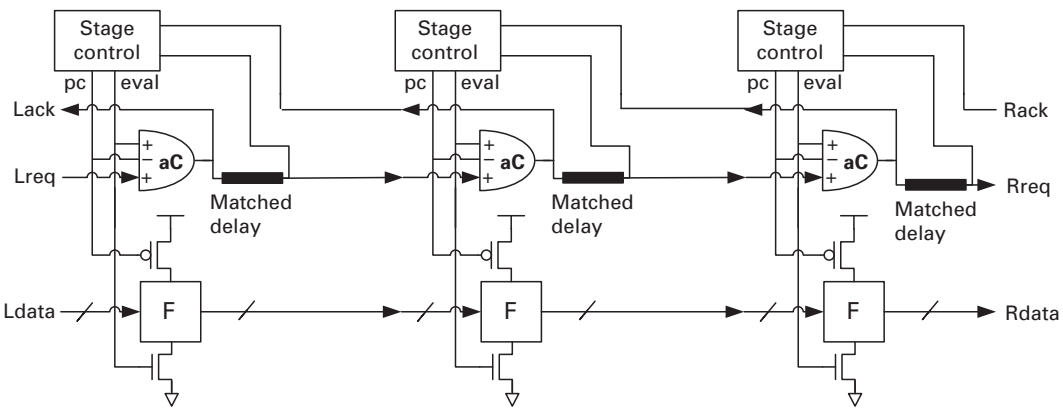

Figure 12.11. A three-stage high-capacity-based linear pipeline.

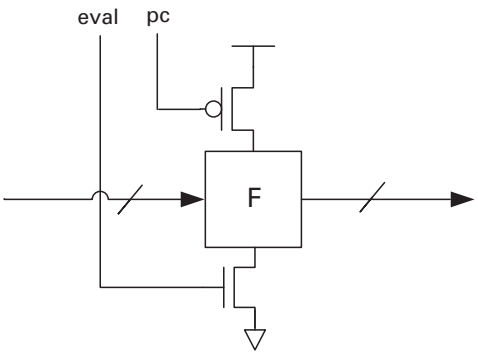

Figure 12.12. Details of an HC pipeline-stage function block.

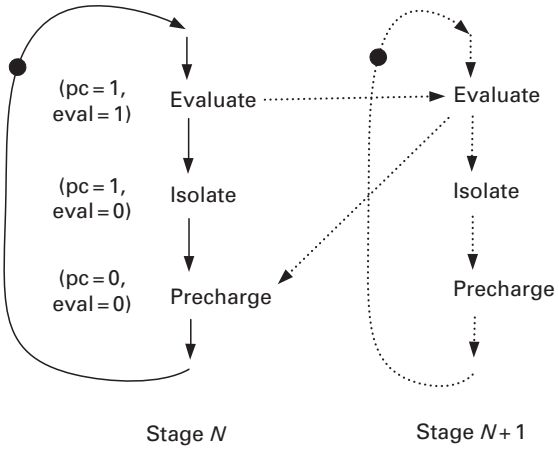

Figure 12.13. Sequence of phases in an HC stage cycle.

detector. The detector's output is set when the stage has begun to evaluate, i.e. when two conditions occur: the stage is in evaluate phase (eval is high) and the previous stage has supplied valid input (the completion-detector output of the previous stage is high). Its output is reset when the stage is enabled to precharge (pc is asserted low). Thus, Precharge will immediately reset the completion detector's output, while Evaluate will set the detector's output only if valid data inputs have also arrived. The aC element output is again fed through a matched delay, which (combined with the completion detector) matches the worst-case path through the function block. As indicated earlier, for a gate-level pipeline the matched delay is often unnecessary: the aC delay already matches the function block delay. The resulting completion signal T (of the second stage in Figure 12.11) is fed in turn to three components: (i) the previous stage's controller, indicating the current stage's state, (ii) the current stage's controller (through the matched delay), and (iii) the next stage (through the matched delay). The stage controller will be discussed shortly, after we have presented the desired protocol.

A pipeline stage simply cycles through the three phases, as shown in Figure 12.13. After it completes its Evaluate phase, it then enters its Isolate phase and

subsequently its Precharge phase. As soon as precharge is complete, it re-enters the Evaluate phase, completing the cycle. The novelty of the approach is seen in the protocol that governs the interaction between stages. Unlike the PS0 and LPSR2/1 pipelines, there is now *only one explicit synchronization point* between stages. Once stage $N + 1$ has completed its Evaluate phase, it enables the previous stage N to perform its entire next cycle, i.e. to precharge, isolate, and evaluate a new data item. In contrast,Williams' PS0 uses two explicit synchronization points between adjacent stages: for the start of evaluation and for the start of precharge. Likewise, the LPSR2/1 design uses two explicit synchronization points, but they are signaled from two distinct stages, $N + 1$ (to start precharge) and $N + 2$ (to start evaluation). As usual, there is one additional implicit synchronization point, the dependence of stage $N + 1$'s evaluation on its predecessor N's evaluation. A stage cannot produce new data until it has received valid inputs from its predecessor. The synchronization points are shown by causality arcs in Figure 12.13. The introduction of the Isolate phase is the key to the protocol. Once a stage finishes evaluation, it immediately isolates itself from its inputs by a self-resetting operation – regardless of whether this stage will soon be allowed to enter its precharge phase. Subsequently, not only can its predecessor precharge, it can even safely evaluate the next data token since the current stage will remain isolated. There are two benefits of this protocol: higher throughput, since stage $N + 1$ can evaluate the next data item even before N has begun to precharge; and higher capacity for the same reason, since adjacent pipeline stages are now capable of simultaneously holding distinct data tokens without requiring separation by spacers.

A complete cycle of events for stage N can be traced in Figure 12.11. From one evaluation by N to the next, the cycle consists of three operations: (i) stage N evaluates, (ii) stage $N + 1$ evaluates, which in turn enables stage N's controller to assert the precharge input (pc = low) of N, and finally (iii) stage N precharges, the completion of which, passing through stage N's controller, enables N to evaluate once again (eval is asserted high). We assume that no extra matched delays are required for the gate-level pipeline, i.e. that the completion detector and other delays already match the gate's evaluate and precharge. Then, in the notation introduced earlier, the delay of step (i) is t_{eval}, the delay of step (ii) is $t_{\text{aC}} + t_{\text{NAND3}}$, and the delay of step (iii) is $t_{\text{prech}} + t_{\text{INV}}$. Here, t_{NAND3} and t_{INV} are the delays through the NAND3 and the inverter, respectively, of Figure 12.14. Thus, the pipeline cycle time is

$$T_{\text{LPHC}} = t_{\text{eval}} + t_{\text{prech}} + t_{\text{aC}} + t_{\text{NAND3}} + t_{\text{INV}}.$$

A stage's latency is simply the evaluation delay of the stage:

$$L_{\text{LPHC}} = t_{\text{eval}}.$$

State variable LPHC pipelines require a one-sided timing constraint for correct operation. The signal ok2pc (see Figure 12.14) goes high once the current stage has evaluated and the next stage has precharged (Rreq, Rack = 10). Subsequently, Rack goes high as a result of evaluation by the next stage. For correct operation, ok2pc must complete its rising transition before Rack goes high:

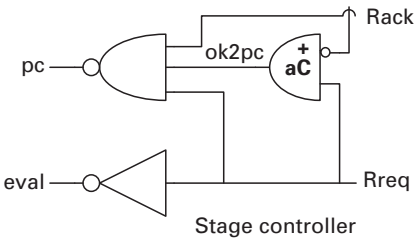

Figure 12.14. Stage controller implementation.

$$t_{\text{ok2pc}''} < t_{\text{eval}} + t_{\text{INV}}.$$

In practice, this contraint is very easily satisfied.

Precharge width As in LPSR2/1, an adequate precharge width must be achieved. This constraint is partly enforced by the bundling constraint: the aC-element and the (optional) matched delay must together have a greater delay than the worst-case precharge time of the function block. Hence, the Rreq input to the NAND3 in Figure 12.14 will be maintained appropriately. However, there is one additional constraint on the precharge width: the Rack input to the same NAND3 must not be de-asserted. Suppose that Rack has just gone high. At this point, stage F_1's NAND3 starts the precharge of F_1 (in Figure 12.5). Concurrently, Rack will only be reset after a path through F_3's aC-element, F_2's NAND3 and aC-element, and F_1's NAND3:

$$t_{\text{NAND3}} + t_{\text{prech}N} \leq t_{\text{aC}} + t_{\text{NAND3}} + t_{\text{aC}} + t_{\text{NAND3}}.$$

Assuming that all stages are similar, this constraint becomes

$$t_{\text{prech}N} \leq t_{\text{aC}} + t_{\text{aC}} + t_{\text{NAND3}}.$$

This final constraint is also easily satisfied.

12.6 Designing non-linear pipeline structures

As described in earlier chapters, the basic assumption in linear pipelines is that each pipeline stage has a single input channel and a single output channel. Non-linear pipeline stages, however, may have multiple input and output channels. This section presents an overview of the challenges involved in designing non-linear pipelines using timed templates. In particular we address the challenges presented by synchronization with multiple destinations (for forks) and synchronization with multiple sources (for joins). Subsequent sections provide our detailed solutions for each of the three pipeline styles reviewed above; we then briefly describe how these solutions are extended to channels that are *conditionally read* or *written* [7].

12.6.1 Slow and stalled right-hand environments in forks

Consider an abstract two-way fork in which the forking stage S1 drives stages S2 and S3. For correct operation, S1 must receive (and recognize) acknowledgements from

both S2 and S3. A problem is that S2 and S3, and the subsequent stages of each, may be operating largely independently of each other. One stage may become arbitrarily stalled, thus potentially stalling its acknowledge signal from either S2 or S3.

If the pipeline templates designed for linear pipelines were naïvely extended to a datapath with a fork, in the expectation that S1 will synchronize on all the acknowledge signals from the forked stages when combined using a C-element, then the resulting pipeline may malfunction.

In particular, the acknowledge signals generated in many linear pipeline structures are *non-persistent*. That is, it is assumed that after a stage asserts its acknowledgement the precharge of the previous stage is fast. Therefore, it does not explicitly check for the completion of that precharge before de-asserting the acknowledge signal. We call this restriction or assumption the *fast-precharge constraint*. In the case of a non-linear pipeline, however, if exactly one of stages S2 or S3 is slow or stalled then the acknowledge signal of the fast stage may be de-asserted before S1 has a chance to precharge, causing deadlock. In other words, in this situation S1 violates the fast-precharge constraint. We call this problem the slow or stalled right-hand environment (SRE) problem. In particular, Williams' classic PS0 pipelines [1] and the recent lookahead and high-capacity pipelines all have this problem.

We propose two general solutions. In the first only the immediate stages after a fork are modified, so that, even after precharging, they maintain the assertion of their acknowledge signal and are explicitly prevented from re-evaluating until after the forking stage is guaranteed to have precharged. The key is to modify the stages after a fork to guarantee that their acknowledgements are properly received while still ensuring that these stages satisfy the fast-precharge constraint.

In the second solution every pipeline stage is modified in such a way that it maintains the assertion of its acknowledge signal until after its predecessor stages are guaranteed to have precharged. In other words, in this solution the entire pipeline is modified so as to remove the fast-precharge constraint, implicitly solving the SRE problem. This solution must be applied to all stages because otherwise an unmodified stage may operate as though its predecessors satisfy the fast-precharge constraint, which may not be the case.

12.6.2 Slow and stalled left-hand environments in joins

The second challenge mentioned at the start of this section is one of synchronization with multiple input channels, as is needed in a join. In a two-way join structure for an abstract pipeline, the data from each of the input stages, say S1 and S2, must be consumed by the join stage, say S3. The data outputs of S1 and S2 are gathered together and presented to S3 as its inputs. Subsequently, S3 sends an acknowledgment to both S1 and S2 once it has consumed the input data. Thus, a two-way join represents a synchronization point between the outputs of two senders.

A problem can arise if the logic implementation of stage S3 is "eager," i.e. S3 may produce an output after consuming one but not both of its data inputs (see [2]). For example, if S3 contains a dual-rail OR function that evaluates eagerly (i.e. as soon as

one high input bit arrives) then after evaluation it will send an acknowledge signal to both S1 and S2, even though one of them may not have produced data at all. As a result, if one of the input stages is particularly slow or stalled then it may receive an acknowledge signal from S3 too soon. This can cause the insertion of a new unwanted data token at the output of the slow stage and thus corrupt the synchronization between the stages. We call this the *stalled left-hand environment (SLE) problem*.

One solution is to allow the join stages to have eager function blocks but still to ensure that the generation of the acknowledge signal occurs only after data from all the input stages has been consumed. This solution has been used extensively in quasi-delay-insensitive templates [6].

12.7 Lookahead pipelines (single-rail)

Handling the joins in single-rail lookahead pipelines is straightforward, and was initially proposed in [4]. The join stage receives multiple request inputs (Lreqs), all of which are merged together in the asymmetric C-element (aC) that generates the completion signal. In particular, each additional request is accommodated by adding an extra series transistor in the pull-down stack of the aC-element. The aC will only acknowledge the input sources after all the Lreqs are asserted and the stage has evaluated.

To handle forks, however, a C-element must be added to the forking stage to combine the acknowledge signals from its immediate successors. In addition, the other stages of the pipeline must also be modified to overcome the stalled right-hand environment problem of subsection 12.6.1. As indicated, the problem is that the acknowledge signal from the immediate successor of a fork stage can be regarded as a pulse that may be de-asserted before its predecessor forking stage has precharged, causing deadlock. The following subsections give two distinct solutions for handling such forks arising in the LPSR2/2 template.

12.7.1 Solution 1 for LPSR2/2

In the first solution, the immediate successor stages of the forking stages are modified so as to latch their Lack acknowledge signals and delay their re-evaluation until all predecessors have precharged. For LPSR2/2, this solution is implemented by modifying the Lack logic and the control of the foot transistor, as shown in Figure 12.15. We assume that the forked stage has just evaluated and the acknowledge signal Lack has just been asserted. At this time, the right-hand environment will assert Rack, causing the output of the latch, X, to be asserted ($X = 0$, i.e. active-low), effectively latching the non-persistent acknowledge signal. The X output is held low even when Rack is de-asserted. In particular, X is de-asserted ($X = 1$) only after the done signal goes low (which is caused by Lreq going low), implying that the input forking input stage has precharged. Effectively, the foot transistor now prevents re-evaluation until after X goes low, delaying re-evaluation until all inputs (including any slow inputs) are guaranteed to have precharged.

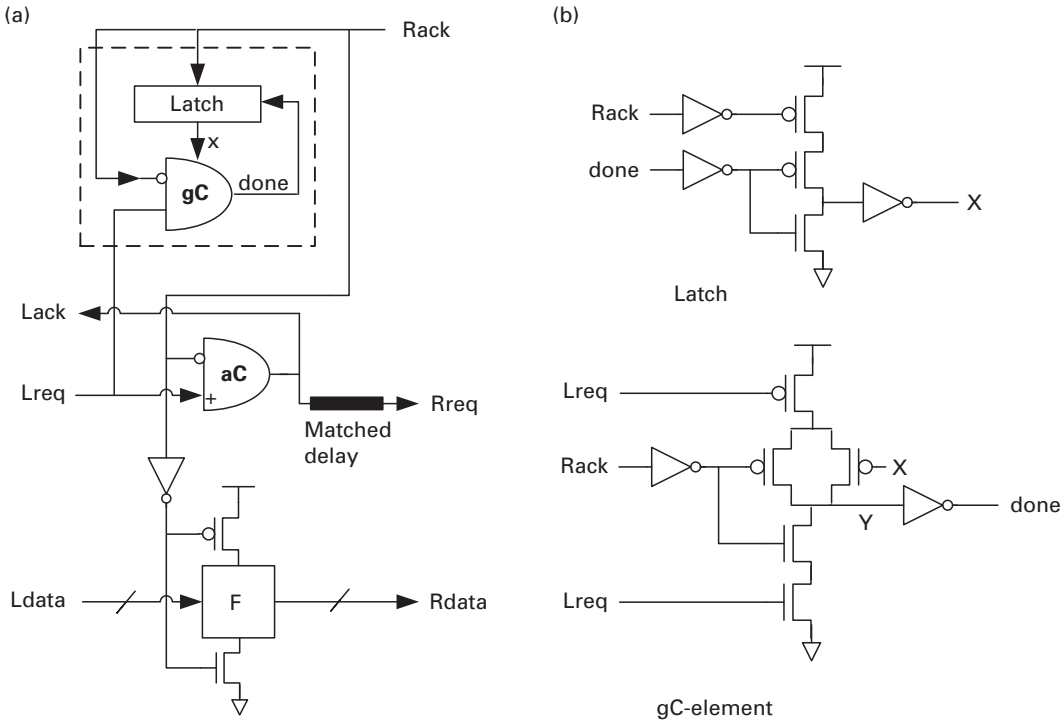

Figure 12.15. (a) Modified first stage after the fork. (b) Detailed implementation of the gates in the broken-line box.

These modifications ensure that even late acknowledgements from a stage S3 immediately after a fork are guaranteed to be properly received while S3 still satisfies the fast precharge constraint, thereby solving the SRE problem.

The only new timing assumption that this template introduces compared with the original LPSR2/2 is that the Rack pulse width must be long enough for it to be properly latched. This pulse-width assumption, however, is looser than the original timing assumption, which remains; i.e. the pulse width must be longer than the stage's precharge time.

12.7.2 Solution 2 for LPSR2/2

In the second solution each stage is modified so that it does not de-assert its acknowledge signal until all input stages are guaranteed to have precharged. This solution can be implemented using the modified LPSR2/2 template shown in Figure 12.16, in which the asymmetric C-element is converted to a symmetric C-element. As suggested earlier, this modification removes the fast-precharge constraint, implicitly solving the SRE problem.

12.7.3 Pipeline cycle time

For the first solution, the cycle-time expressions do not change if the additional acknowledge signals simply increase the stack height and do not add additional

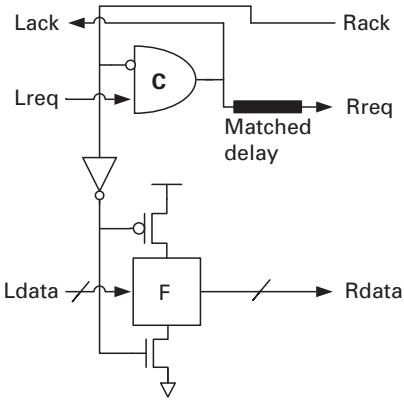

Figure 12.16. The LPSR2/2 pipeline stage with a symmetric C-element.

gates. For multi-way forks and joins, however, the cycle time will increase owing to the additional C-elements needed to combine them. For the second solution, the cycle time becomes

$$T_{\mathrm{LPSR2/2}} = \max(2t_{\mathrm{eval}} + 2t_{\mathrm{gC}}, t_{\mathrm{eval}} + t_{\mathrm{prech}} + 2t_{\mathrm{gC}}).$$

12.8 Lookahead pipelines (dual-rail)

This section extends the dual-rail lookahead pipeline LP3/1 to handle forks and joins. Since both the stalled left-hand environment (SLE) and the stalled right-hand environment (SRE) problems of Section 12.6 can arise in dual-rail pipelines, detailed solutions are presented for both forks and joins.

12.8.1 Joins

Unlike LPSR2/2, the LP3/1 pipeline has no explicit request line and thus may not function correctly unless it is modified to handle the SLE problem in joins. Our proposed solution still allows the use of eager function blocks; however, it ensures that no acknowledgement is generated from a stage until after all its input stages have evaluated.

In particular, in our solution request signals are added to the input channels of the joins and fed into the join stage's completion detector, as illustrated in Figure 12.17. The join's completion detector now delays asserting its acknowledge signal not only until the function block has finished computing but also until the input stages have all completed evaluation; thereby the left-hand environment problem is solved. Note that the additional request signals are taken from the outputs of the preceding stages' completion detectors. While this modification does not affect the latency of the pipeline, the analytical cycle time changes to

$$T_{\mathrm{LP3/1}} = 2t_{\mathrm{eval}} + 2t_{\mathrm{CD}} + t_{\mathrm{NAND}}.$$

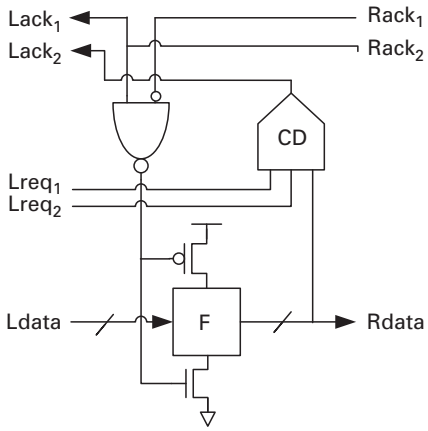

Figure 12.17. The LP3/1 pipeline with a completion detector modified to handle joins.

12.8.2 Forks

As in the case of the single-rail lookahead pipeline, LPSR2/2, we propose two solutions for the slow or stalled right-hand environments. These solutions are similar in essence to the solutions for the single-rail case but are adapted to dual rails.

The implementation of solution 1 (see subsection 12.7.1) is very similar to that for LPSR2/2, as shown in Figure 12.18. First, the completion detector (CD) is modified in such a way that the acknowledge signal is de-asserted only after the forking stage has precharged. In addition, the re-evaluation of the function block is delayed until the forking stage has precharged, using a decoupled foot transistor controlled by the Y signal.

In the second solution, a request line is added to all LP3/1 channels and de-assertion of the acknowledge signal ($Lack_1$ in this case) is delayed until after all immediate predecessors have precharged, as shown in Figure 12.19. The request line is generated via a C-element that combines the incoming request line(s) and the output of the completion detection. The output of this C-element becomes the new $Lack_1$. Because the C-element de-asserts its acknowledge signal only after Lreq is de-asserted, the fast precharge constraint is removed, solving the SRE problem.

For solution 1, compared with the original LP3/1 template the cycle time is slightly increased to

$$T_{\mathrm{LP3/1}} = 2t_{\mathrm{eval}} + 3t_{\mathrm{CD}} + t_{\mathrm{prech}}.$$

For solution 2, the cycle time increases to

$$T_{\mathrm{LP3/1}} = t_{\mathrm{eval}} + 3t_{\mathrm{CD}} + t_{\mathrm{NAND}}.$$

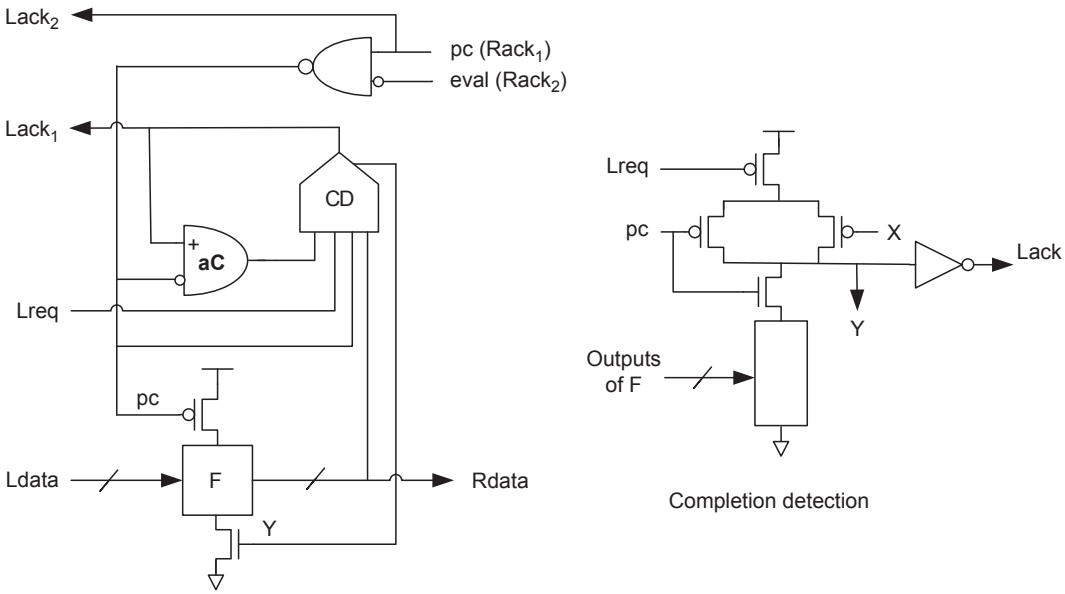

Figure 12.18. (a) Modified first stage after the fork. (b) Detailed implementation of the additional gates.

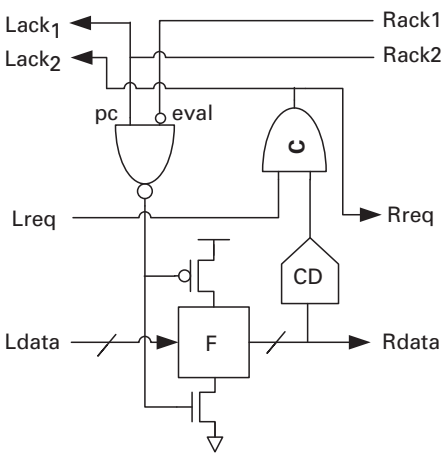

Figure 12.19. The LP3/1 stage with a C-element.

12.9 High-capacity pipelines (single-rail)

Since the high-capacity pipeline template uses single-rail encoding, it has a request line associated with the data and thus does not have the slow or stalled left-hand environment problem in joins. However, because the acknowledge signals in high-capacity pipelines are also non-persistent (effectively, they are timed pulses), they do have a slow or stalled right-hand environment problem in forks.

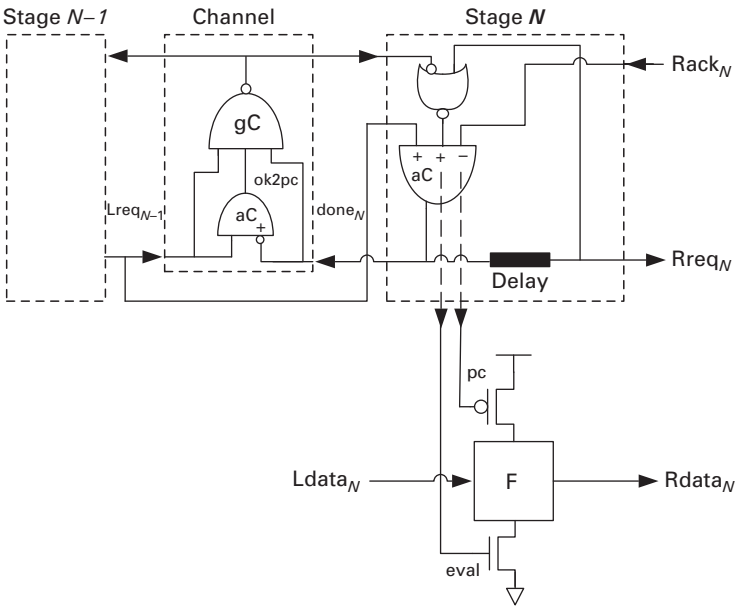

Figure 12.20. New high-capacity stage.

In a simple modification to the original stage controller of the high-capacity pipeline, illustrated in Figure 12.20, de-assertion of the acknowledge signal is delayed until after the request line goes low, thus removing the fast-precharge constraint and solving the SRE problem using solution 2 (see subsection 12.7.2).

In particular, by replacing the NAND3 gate by a state-holding generalized C-element, the acknowledge signal Rack triggers only the assertion of the precharge control signal pc. The de-assertion of pc is caused by the input request signal Rreq's going low. Thus, pc remains asserted until after precharge is completed and is unaffected when the acknowledge signal from the next stage is de-asserted. Furthermore, the inverter is replaced by a NOR2 gate with an additional input, in order to delay the stage's re-evaluation until after the stale input data is reset.

In the new version of the HC pipeline stage the state variable ok2pc belongs to the channel between stages $N-1$ and N. The reasoning is as follows. The function of the state variable is to keep track of whether stages $N-1$ and N are computing the same token or distinct (consecutive) tokens; the precharge of $N-1$ is inhibited if the tokens are different. If there are two stages, $(\mathrm{N}-1)^{(\mathrm{A})}$ and $(\mathrm{N}-1)^{(\mathrm{B})}$, supplying data for stage N then we propose two separate state variables, one to keep track of whether stages $(N-1)^{(\mathrm{A})}$ and N have the same token, and the second to keep track of whether stages $(N-1)^{(\mathrm{B})}$ and N have the same token. Similarly, if stage N has two *successors*, $(N+1)^{(\mathrm{A})}$ and $(N+1)^{(\mathrm{B})}$, we propose two distinct state variables, one each for the pair N, $(N+1)^{(\mathrm{A})}$ and the pair N, $(N+1)^{(\mathrm{B})}$.

The asymmetric C-element that implements the state variable ok2pc is pulled out of the stage controller and placed in between stages $N-1$ and N (i.e. it is moved into the channel). In addition, the generalized C-element is also moved into the channel to avoid extra wiring.

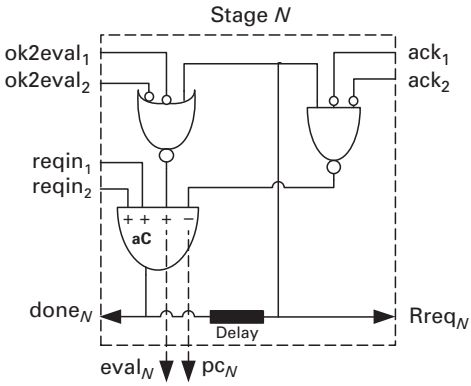

Figure 12.21. Implementation when stage N is both a fork and a join.

12.9.1 Handling forks and joins

Figure 12.21 shows the implementation of a template for stage N for the case where stage N is both a fork as well as a join. Multiple occurrences of reqin, ok2eval, and ack are handled by simple modifications to the linear pipeline in Figure 12.20.

Multiple reqins Each additional reqin is handled by adding a single series transistor to the asymmetric C-element that makes up the completion generator, similarly to how joins were handled for LPSR2/2 in Section 12.7. Hence, the done signal is generated only after all the input data streams have been received.

Multiple ok2evals Each additional ok2eval is handled by adding it as an extra input to the NOR gate that produces the eval signal. Consequently, the stage is enabled to evaluate (i.e. eval is asserted) only after all the ok2eval signals are asserted, i.e. after all the senders have precharged.

Multiple acks These are handled by ORing them together. Since the acks are all asserted low, the OR gate output goes low only when all the acks have been asserted, thus ensuring that precharge occurs only after the stage's data outputs have been absorbed by all the receivers. The OR gate is actually implemented as a NAND with bubbles (inverters) on the ack inputs. This NAND has an additional input – the stage's completion signal – whose purpose is to ensure that, once precharge is complete, pc is quickly cut off. Otherwise, pc may be de-asserted slightly after eval is asserted, causing a momentary short circuit between supply and ground inside the dynamic gates.

12.9.2 Pipeline cycle time

If only joins are present, the cycle time is only slightly increased. Compared with the cycle time obtained in [4], the new cycle time equation has a NOR delay instead of an inverter delay, and a gC delay instead of a NAND3 delay:

$$T_{\mathrm{LPHC}} = t_{\mathrm{eval}} + t_{\mathrm{prech}} + t_{\mathrm{aC}} + t_{\mathrm{gC}} + t_{\mathrm{NOR}}.$$

If forks are also present then the cycle time increases by the delay of the OR gate needed to combine the multiple acknowledgments:

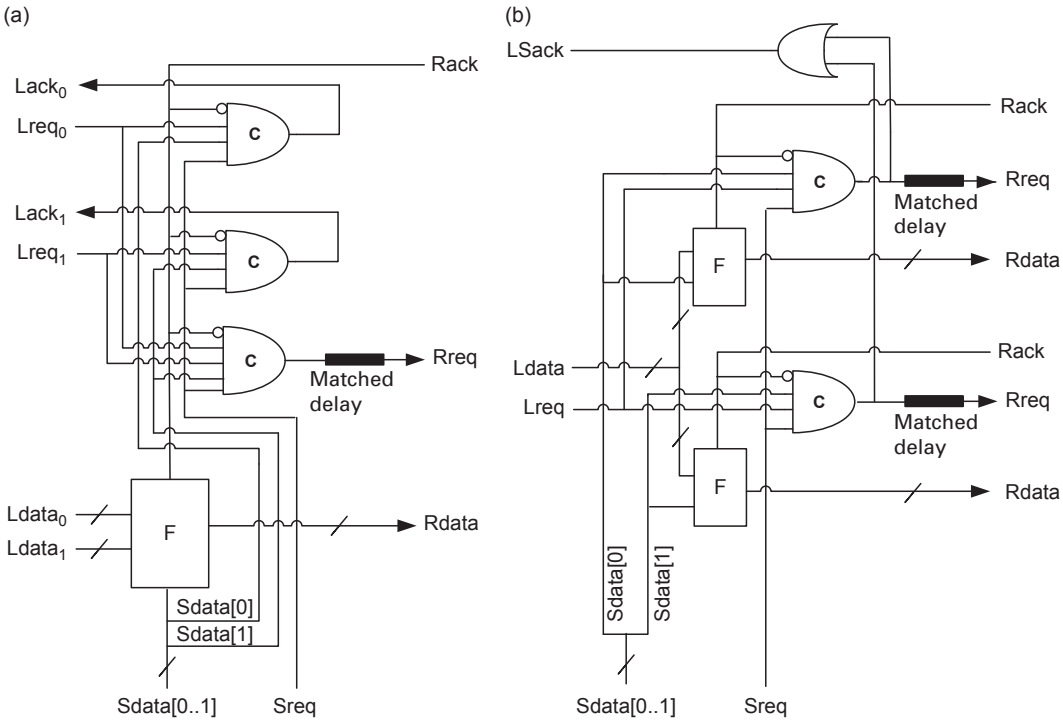

Figure 12.22. Conditional (a) read and (b) write.

$$T_{\mathrm{LPHC}} = t_{\mathrm{eval}} + t_{\mathrm{prech}} + t_{\mathrm{aC}} + t_{\mathrm{gC}} + t_{\mathrm{NOR}} + t_{\mathrm{OR}}.$$

12.10 Conditionals

Other complex pipeline stages allow the conditional reading and writing of data and can have internal states. This section briefly covers the implementation of these cells for the LPSR2/2 template; however, a similar approach can also be applied to the other pipeline styles.

Figure 12.22(a) shows a conditional read, in which the stage reads only one input channel; this is determined by depends the value of the select channels. Only the channel that is read is acknowledged. Figure 12.22(b) shows a conditional write, in which the stage reads the input channel and outputs the data (i.e. writes) to only one output channel; which one depends on the value of the select channel. It receives an acknowledge signal only from the output channel where the data is written. Note that the C-elements are symmetric only for the Rack input; they are asymmetric for all others.

Figure 12.23 shows a one-bit memory implemented using a LPSR2/2 template. The primary input and output channels are denoted by A and C, respectively. Additionally, B is the internal storage and S is an input control channel that selects the write or read operation. When S0 is high, the memory stores the value at the

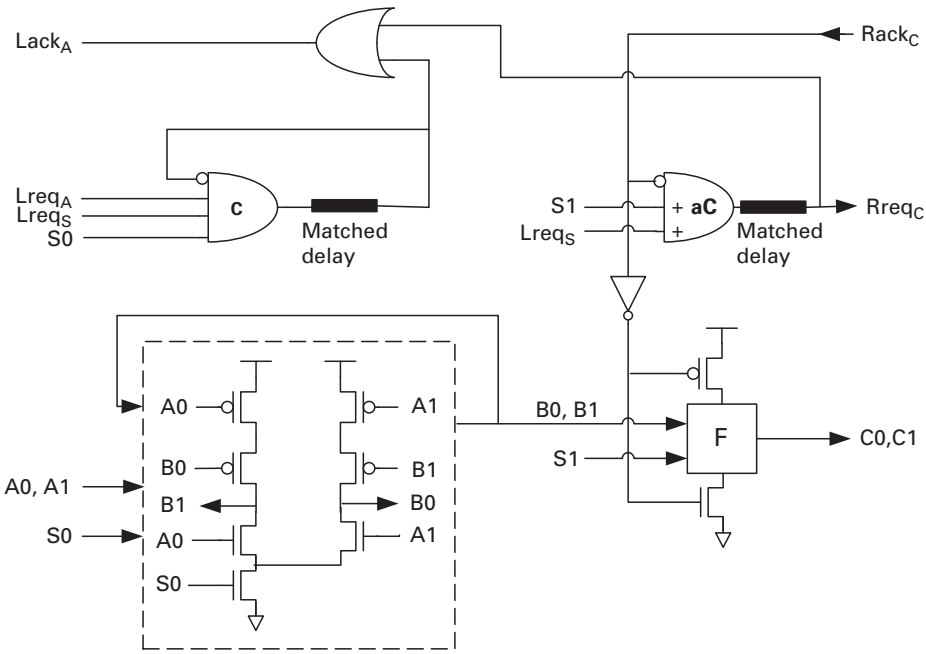

Figure 12.23. A one-bit LPSR2/2 memory.

input channel A to the internal storage B. Both the input A and the select channels are acknowledged. The the data storage implementation is shown in the broken-line box (similar to B). Assuming that there is already data stored, one dual-rail bit of B is high and the other is low. When an input A is applied and S0 is high, first both rails are lowered and then one is asserted high, thereby storing the data. The C-element, which generates the acknowledge signal of the input channel Lack_A through a matched delay line, is reset using its own output, since it does not receive an acknowledgement from an output. The delay of the delay line is matched to the delay of writing the internal node B.

When S1 is high, however, the internal data stored in B is sent to the output channel C. When an acknowledge signal is received from the output channel C, the outputs are reset but the data stored remains unchanged.

12.11 Loops

In the simple LPSR2/2 buffer, the delay from the left-hand request Lreq to the right-hand request Rreq consists of a simple non-inverting asymmetric C-element (implemented using two gates) and an externally added delay to match the data. For a more complex gate with multiple input and output channels, however, the delay from Lreq to Rreq will increase and for some cells will be larger than the delay through the domino logic. It is important to realize that, for loops

implemented with multiple input and output channels, this delay may become the performance bottleneck.

In particular, the fact that the data moves forward faster than its request signal causes a problem. Given a loop with L stages, assuming that both the assertion of the data and the request signal happen to be synchronized at stage 0, let the data move forward faster than the request signal at a rate t_d/t_r, where t_d is the forward latency of the data and t_r is the forward latency of the request signal ($t_r \geq t_d$). Pipeline stages in the loop that have evaluated will be able to re-enter the evaluation phase only when the request signal is asserted, allowing them to precharge, and then subsequently de-asserted. Therefore as the data attempts to overtake the request signal around the loop (which may take many loop iterations), the data will stall while waiting for the request signal of the subsequent stage to de-assert. From this point on, the forward latency of the data will slow down to match the forward latency of the request signal.

The LPSR2/2 modified according to solutions 1 and 2 and the LP3/1 pipeline modified according to solution 2 will still have this problem, since the Rreq signal is generated in the same way and therefore the delay between the Lreq and Rreq signals increases as the number of input and output channels increases. A detailed solution is given in Chapter 11.

12.12 Simulation results

HSPICE simulations were performed using a 0.25 Taiwan Semiconductor Manufacturing Co. (TSMC) process with a 2.5 V power supply at 25 °C. The purpose of these simulations was to quantify the performance overhead of using the fork–join structures described earlier, in comparison with that for linear pipelines. Hence, as before, no attempt was made to fine-tune the transistor sizing to achieve optimum performance. In particular, all transistors were sized to achieve a gate delay roughly equal to that of a small inverter (widths $W_{NMOS} = 0.8$ μm and $W_{PMOS} = 2$ μm and length $L = 0.24$ μm) driving a same-sized inverter. For the purposes of this comparison, the wire delay μ has been ignored also.

The simulation results for all the linear and non-linear pipelines discussed in this chapter are presented in Table 12.1. The original linear pipelines appear under the Sol1 heading in the linear1 row because solution 1 involves only modifying the

Table 12.1. Cycle times (ns) of the original linear pipelines and the proposed non-linear pipelines

	LPSR2/2		LP3/1		LPHC
	Sol1	Sol2	Sol1	Sol2	Sol2
Linear1	0.99	—	1.20	—	—
Linear2	—	1.06	—	1.28	0.93
Fork	1.23	1.29	1.41	1.45	1.20
Join	1.05	1.10	1.27	1.34	1.01

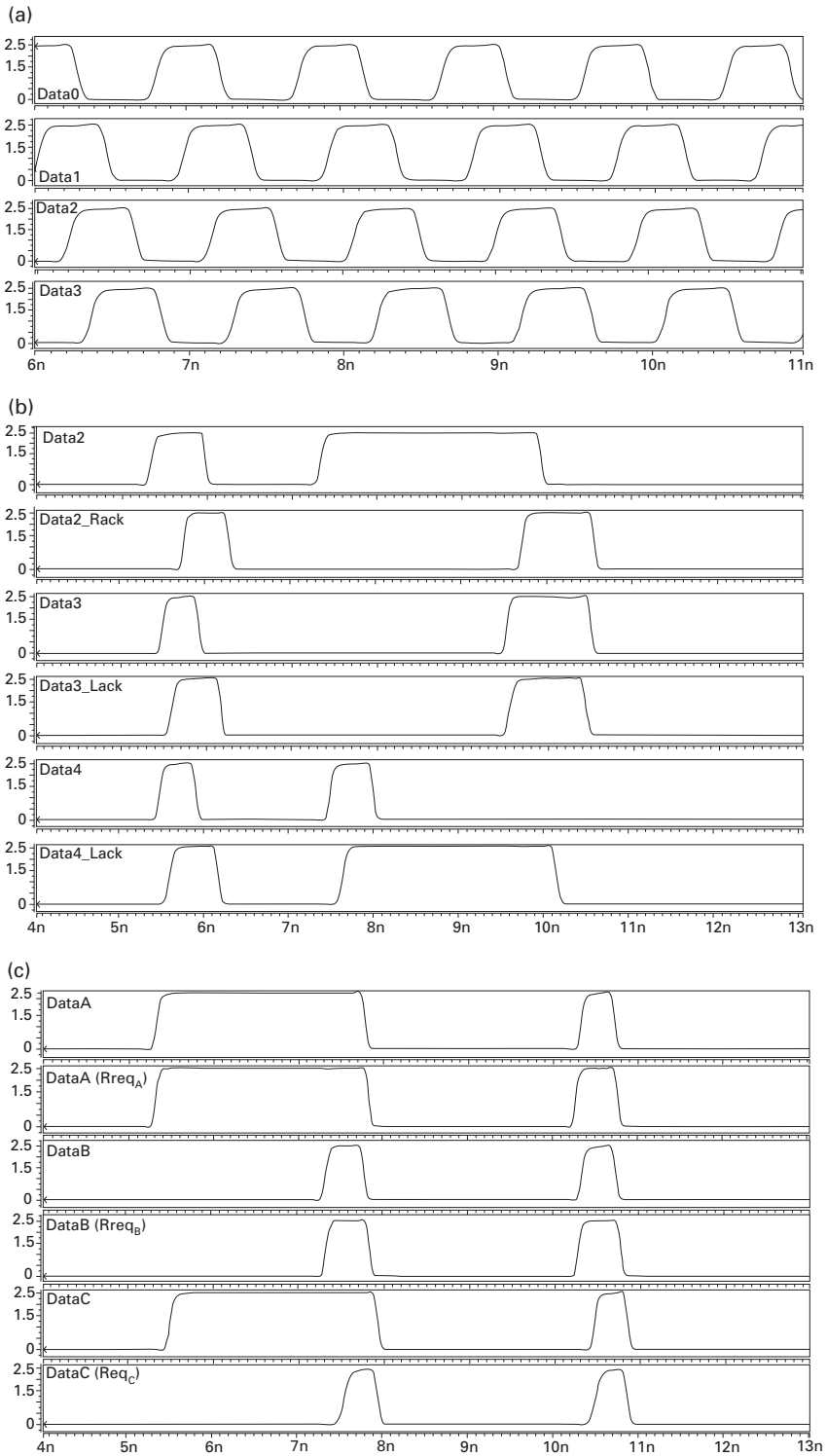

Figure 12.24. HSPICE waveforms: (a) linear pipeline, (b) two-way fork, and (c) Two-way join. The horizontal axis numbers are nanoseconds.

first stages after a fork and forks do not exist in linear pipelines. The linear2 row and Sol2 column gives the cycle times for linear pipelines, where each stage has been modified according to solution 2. Note that while the joins add only ~5% to the cycle time, the forks increase the cycle time by ~20% because of the additional C-element needed.

The waveforms in Figure 12.24(a) show the data signal for an LPSR2/2 one-bit linear pipeline. Note also that the cost difference of the more robust solution 2 and solution 1 is generally less than 5%. Figure 12.24(b) shows waveforms for a fork with a slow right-hand environment channel called Data4 and Figure 24(c) shows waveforms for a join with a slow left-hand environment channel called DataB.

12.13 Summary

In this chapter, we have reviewed new high-speed asynchronous circuit templates for non-linear dynamic pipelines, including forks, joins, and more complex configurations in which channels are conditionally read and/or written. Two sets of templates arise from adapting the LPSR2/2 and LP3/1 pipelines and one set of templates arises from adapting the LPHC pipelines. Timing analysis and HSPICE simulation results demonstrate that forks and joins can be implemented with a roughly 5%−20% performance penalty over linear pipelines. All pipeline configurations have timing margins of at least two gate delays, enabling a good compromise between speed and ease of design.

References

[1] T. E. Williams, "Self-timed rings and their application to division," Ph.D. thesis, Stanford University, 1991.

[2] T. E. Williams and M. A. Horowitz, "A zero-overhead self-timed 160 ns 54 b CMOS divider," *IEEE J. Solid-State Circuits*, vol. 26, no. 11, pp. 1651–1661, November 1991.

[3] M. Singh and S. M. Nowick, "High-throughput asynchronous pipelines for fine-grain dynamic datapaths," in *Proc. Int. Symp. Advanced Research in Asynchronous Circuits and Systems*, April 2000, pp. 198–209.

[4] M. Singh and S. M. Nowick, "Fine-grain pipelined asynchronous adders for high-speed DSP applications," in *Proc. IEEE Computer Society Workshop on VLSI*, April 2000, pp. 111–118.

[5] M. Singh, J. A. Tierno, A. Rylyakov, S. Rylov, and S. M. Nowick, "An adaptively-pipelined mixed synchronous–asynchronous digital FIR filter chip operating at 1.3 gigahertz," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, April 2002, pp. 84–95.

[6] A. Lines, *"Pipelined asynchronous circuits,"* Master's thesis, California Institute of Technology, 1995.

[7] R. Ozdag, P. Beerel, M. Singh, and S. Nowick, "High speed non-linear asynchronous pipelines," 2002, pp.1000, Design, Automation and Test in Europe Conference and Exhibition (DATE′ 02).

13 Single-track pipeline templates

13.1 Introduction

The single-track protocol, introduced in Chapter 2, was initially proposed by van Berkel and Blink [2] as a communication method for bundled-data pipeline controllers. As an example, Figure 13.1 shows a single-track synchronization channel implemented with a single wire between a sender and a receiver. In this example, the sender sends a synchronization token by driving the channel wire high and the receiver detects the token and resets the channel by resetting the wire low. This is then detected by the sender, which may send a second token by raising the channel wire high again and thus repeating the entire handshaking process. An alternative form of this protocol is also possible in which the sender drives the communication wire low and the receiver drives it high.

Unlike other two-phase protocols, the single-track protocol returns the communication wire to its initial state. This enables single-track templates to react to only one type of transition, avoiding the need for the complex PMOS transistor networks that are a source of area and power inefficiencies for templates that use two-phase transition signaling, i.e. a non-return-to-zero protocol. Moreover, compared with four-phase protocols the single-track protocol has fewer wire transitions, only two instead of four, to complete a handshake, leading to improved power consumption per transmitted bit. Finally, because this protocol does not use an acknowledge wire it requires fewer wires than protocols that use distinct request and acknowledge wires.

However, the single-track protocol requires some timing assumptions with respect to the active phase of the sender and receiver transistors to avoid driving overlap (i.e. short circuit) and to guarantee that the channel wire is driven fully to power and ground.

In [2], three possible ways to drive the channel wire are described:

- In a *dynamic handshake* (Figure 13.2(b)), both sender and receiver actively drive the wire only during the transitions. In this case, the stored charge may leak away to an indeterminate value if the channel is not frequently used. Consequently, staticizer circuits must be added to the channel to hold the charge. If the staticizers are not strong enough, however, the voltage may still be pulled to a indeterminate or wrong value when subjected to crosstalk noise.
- In a *static handshake* (Figure 13.2(c)), the wire floating time is bounded. The sender is responsible for actively holding the wire low until it raises it and the

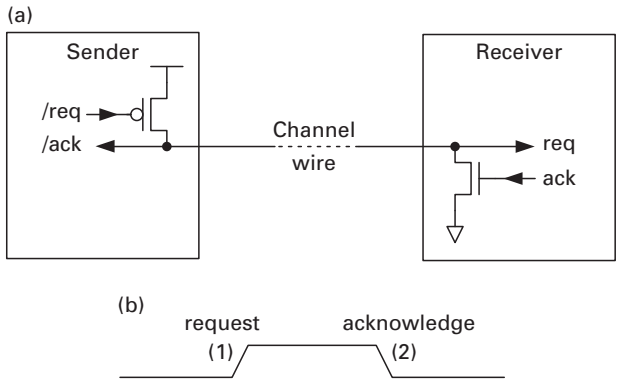

Figure 13.1. Single-track (a) block diagram and (b) channel wire waveform. The slashes before req and ack in the left-hand part of (a) indicate that req and ack are active-low.

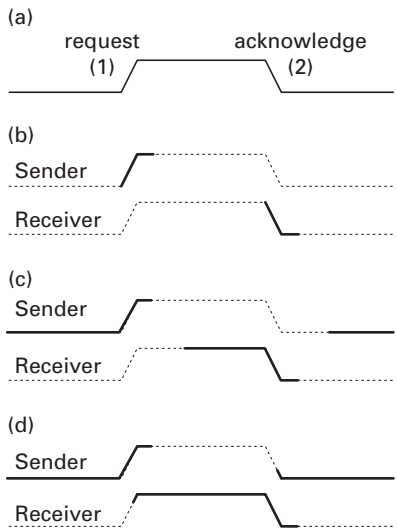

Figure 13.2. Typical single-track (a) waveform and the sender–receiver active drive shape for (b) dynamic, (c) static, and (d) overlap handshakes.

receiver is responsible for holding the wire high until it consumes the token and resets the channel low. Depending on the implementation, there may be a period of time after a transition during which neither receiver nor sender actively drives the wire. This time must be short enough not to affect the functionality.

- The *overlap handshake* (Figure 13.2(d)) is a special case of the static handshake in which the channel wire is continuously driven and both the sender and receiver help to complete each other's transitions. That is, the receiver is responsible for helping to drive the wire high after the high transition is initiated by the sender and the sender is responsible for helping to drive the wire low after the channel reset is initiated by the receiver.

In practice the driving timing is usually made longer than the transition timing, in order to guarantee a full swing of the voltage level, and the difference between the static and overlap driving strategies then becomes a matter of the timing of when the other stage will start helping to drive. Therefore, we will simply refer to both strategies as the *static single-track* protocol.

In the remainder of this chapter we present single-track template implementations and discuss their performance and tradeoffs. Sections 13.2 and 13.3 present the GasP and pulsed logic single-track templates. Section 13.4 gives the single-track full-buffer (STFB) template, and Section 13.6 gives an STFB standard-cell library implementation. In Section 13.7 we describe how a conventional (synchronous) standard-cell flow was used to implement the evaluation design presented in Section 13.8. The chapter closes with a discussion of the STFB performance and some open issues regarding the single-track protocol in Section 13.9.

13.2 GasP bundled data

One of the most aggressive single-track templates is GasP [3][4][5]. GasP is a bundled data channel where the request and acknowledge wires are merged into a single-track wire. For GasP, a low level signals the presence of a request token in the control channel, while acknowledging it is done by driving the wire back high.

Figure 13.3 shows a GasP circuit where, after reset, L, R, and A are high (the non-active level for GasP). When L is driven low by the left-hand environment, the self-resetting NAND will fire, driving A low. This will restore L, activate the data latches, and drive R low, propagating the signal and avoiding re-evaluation until after R is restored high by the right-hand environment. The self-resetting

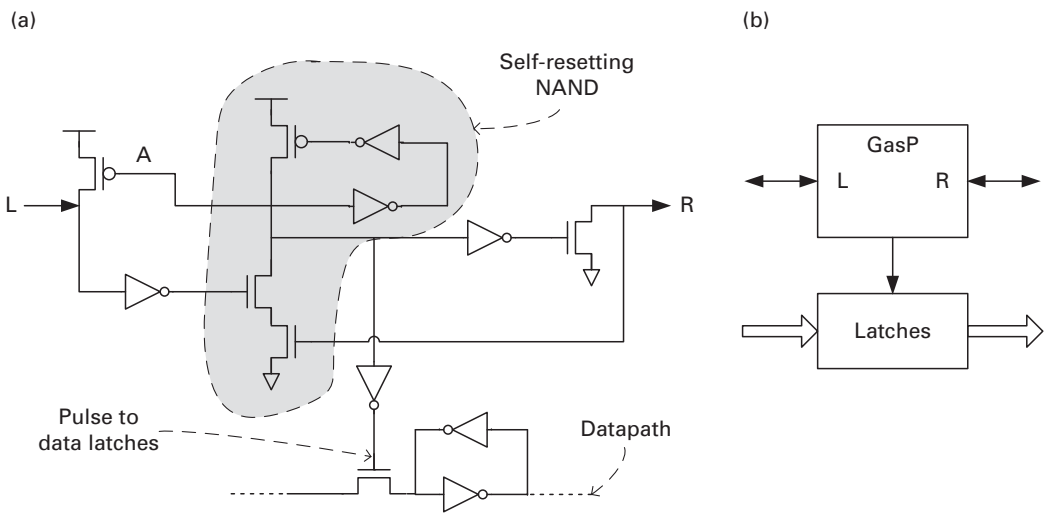

Figure 13.3. GasP circuit: (a) schematic and (b) block diagram.

NAND will restore itself by driving A high after three transitions. The output of the NAND controls the latches in a parallel single-rail datapath.

GasP circuits require four transitions to forward data and two transitions to reset, i.e. two transitions to move a bubble backwards. Regarding the forward latency of the four transitions, approximately two transitions are required for the latency through the latches in the datapath and to satisfy the setup and hold times, leaving approximately two transitions for computation in the datapath. Note that the control circuit itself makes up the delay line and that it is the datapath designer's responsibility to pipeline it so as to match the control circuit delay while satisfying all setup and hold times and the time margin due to process variations.

Owing to its high performance, the timing assumption associated with stability of the datapath before latching (the bundled-data timing constraint) implies that GasP requires full-custom design as well as new CAD flows that automatically verify this constraint.

13.3 Pulsed logic

Single-track asynchronous pulsed logic (STAPL), proposed by Nystrom [20]–[22], is a template for the implementation of logic stages communicating though 1-of-*N* encoded single-track channels. Similarly to GasP, STAPL uses self-reset circuits to determine the active timing of the reset and output transistors.

Figure 13.4 shows the block diagram and the template schematic of the dual-rail STAPL. Notice that there are two types of pulse generator, one for the output driving transistors and another for the input reset transistors. After all the necessary input tokens have arrived, one side (S0 or S1) of the NMOS transistor stack

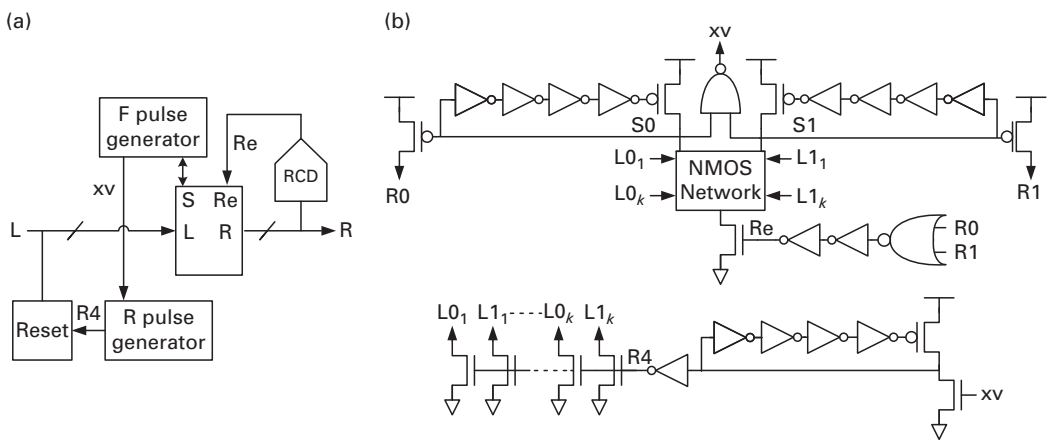

Figure 13.4. STAPL (a) block diagram and (b) template schematic. In (a) the acknowledge signal is denoted Re.

will go low, driving high one of the output signals (R0 or R1) and triggering the reset pulse by the signal xv.

The pulse generators utilized on STAPL use five inversions in the loop, which indicates that this template has a 10-transition cycle time, at least, and, since from the NMOS transistor stack to the output there are two transitions, the STAPL has a two-transition forward latency.

Similarly to GasP, STAPL offers no time margin between the drive and reset phases of two consecutives stages. This may not be an issue if the proper transistor sizing is performed, as shown in subsection 13.6.5. However, while requiring a similar number of components and offering similar performance to QDI, STAPL uses a single-track protocol, which is less robust than QDI.

The STFB template presented in the following sections offers GasP-like performance with much smaller area than the QDI template.

13.4 Single-track full-buffer template

In order to keep the same cycle time as GasP (six transitions) while using 1-of-*N* data encoding, the single-track full-buffer (STFB) template, proposed by Ferretti, Beerel, and Ozdag [6]–[10], does not use pulse generators as in STAPL or a self-rest NAND as in GasP. For STFB stages, when the output token is issued a stage resets and disables itself. Then, when the output token is consumed (removed from the channel), the stage is enabled again. This combination of output detection and timing generation simplifies the template and allows a very small cycle time, as described below.

Figure 13.5 shows a typical block diagram for an STFB cell. When there is no token in the right-hand channel (R) (i.e. the output channel is empty), the right-hand environment completion-detection block (RCD) asserts the signal B, enabling the processing of the next token. In this case, when the next input token arrives at the left-hand channel (L) it is processed, thus lowering the state signal S; this creates an output token on the right-hand channel R and causes the state-completion-detection block (SCD) to assert the signal A, removing the token from the left-hand channel through the Reset block. The presence of the output token

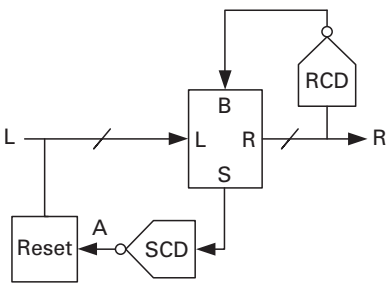

Figure 13.5. Typical STFB block diagram.

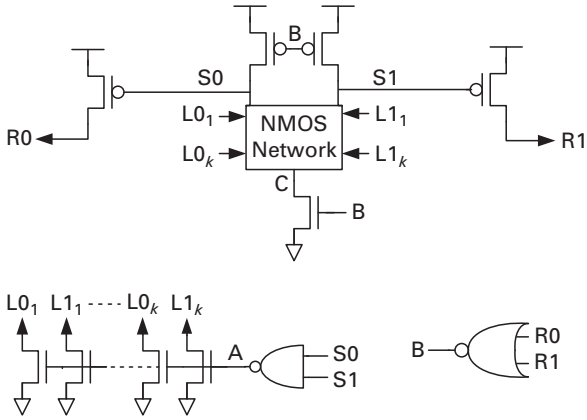

Figure 13.6. Simplified dual-rail STFB template.

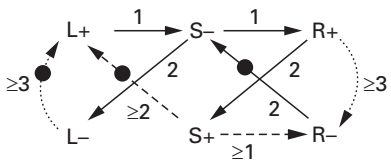

Figure 13.7. Marked graph behavior of STFB template.

on the right-hand channel resets the signal B, preventing the stage from firing while the output channel is busy.

Figure 13.6 shows a simplified schematic of the STFB dual-rail template. The NOR gate in this figure is the RCD, the NAND gate is the SCD, and the NMOS transistor stack defines the cell's main function. Note that the NMOS transistor stack (the *N*-stack) is designed to be semi-weak-conditioned in that it will not evaluate until all the expected input tokens arrive. This combination of functionality and input completion detection removes the need for a left-hand environment completion-detection (LCD) circuit, thus reducing the template's complexity, size, and cycle time. Other alternatives that separate the *N*-stack and the LCD functionality, such as the quasi-delay-insensitive (QDI) template and the *non-weak-conditioned STFB template* [6][10], are bigger and slower and should only be used where the application requires them.

The cycle time of the STFB template is six transitions and the forward latency is two transitions. This implies that the peak pipeline throughput can be achieved with just three stages per token, which allows high pipeline occupancy and the implementation of high-performance small rings (loops of pipeline stages). The full-buffer characteristic of the STFB stage refers to each stage's capacity of holding one token. The STFB template is very flexible and can be expanded to different functionalities [10][6].

Figure 13.7 shows the timed marked graph for the presented templates. The notation " + " and "–" represents the rising and falling of the signals, respectively.

The left- and right-hand environments drive the dotted arrows and the broken-line arrows represent the timing constraints. The arrows are annotated with delays in terms of numbers of transitions. The arrows from S+ to L+ and to R− reflect a relative timing assumption that the source and sink transitions must be sufficiently separated in time to avoid a hazard. In particular, if S+ occurs too late then a short-circuit current occurs on both the input and output wires.

As can be deduced from the timed marked graph, the STFB template has somewhat tight timing constraints. In particular, when counting CMOS transitions, the timing margin between the tri-stating of an output wire (one transition after S+) and the earliest time at which the environment can reset the wire (R−) is zero. Moreover, the timing margin between the tri-stating of an input wire (two transitions after S+) and the earliest time at which the left-hand environment can drive the wire (L+) is also zero. In particular, if these margins are violated, significant short-circuit current may occur during the transitioning of the line. In addition, it is assumed that three transitions are sufficient to fully discharge or charge a line. To accommodate these constraints, the channel load needs to be bounded. This is achieved by limiting the wire lengths of the channels, and it can be easily verified after the placement and routing phase that they are satisfactory. Moreover, automated static timing analysis tools that will further improve the design robustness and sign-off process are under development [28][29].

13.4.1 Static single-track full-buffer (SSTFB) template

One possible implementation of the static single-track (SST) line driver is shown in Figure 13.8. The active drivers are M1 and M10. The additional transistors M2 and M11 ensure that there is no contest during transitions of the wire, allowing M3 and M12 to be as large as desired to combat coupling noise. Therefore, each side of the wire has complementary "drive-and-hold" circuits.

Notice that M3 and M12 are used to drive the channel wire continuously. Consider first the case in which the sender side S is high and A is low. In this case, the line can be low (for example after reset) or high (a token is stalled on the channel). While it is low M2 and M3 actively keep the line low, whereas when the

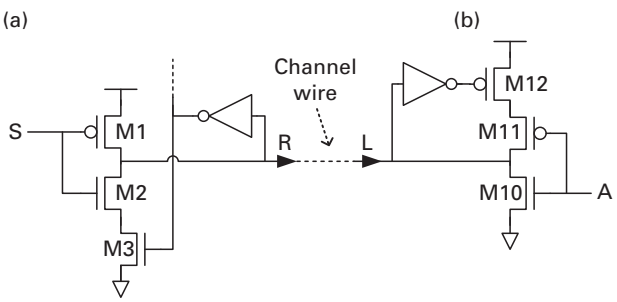

Figure 13.8. Static single-track channel driver implementation: (a) sender and (b) receiver "drive-and-hold" circuits.

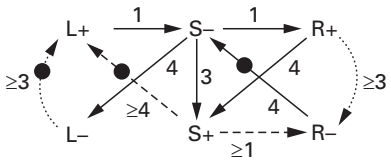

Figure 13.9. Ten-transition STFB marked graph.

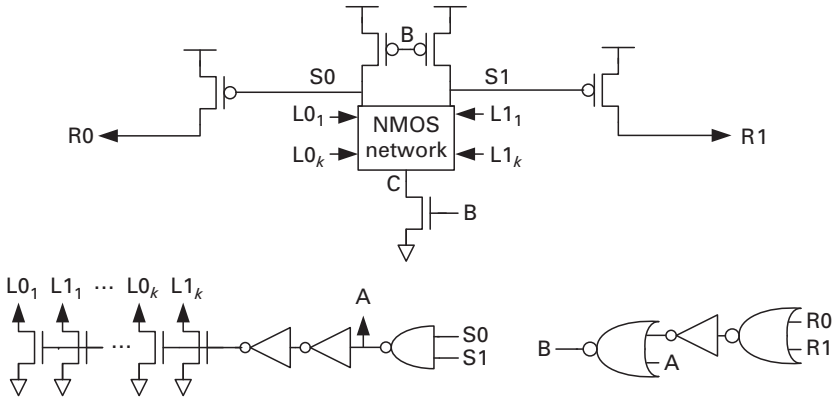

Figure 13.10. Ten-transition STFB template.

line is high M11 and M12 actively keep the line high. Conversely, if the sender side S is low and A is low then M1 actively drives the wire high. Lastly, if A is high then M10 actively drives the wire low. Thus, in all cases there is a strong path from the wire to a power supply.

13.4.2 The 10-transition template

Figure 13.9 shows the marked graph for the alternative 10-transition STFB template in Figure 13.10. This template offers a self-reset three-transition active output (S− to S+ is the period): one transition corresponds to a margin for R+ to hold S+, and two transitions correspond to a margin between the drive and reset phases of the output and input single-track wires.

As can be seen, in the 10-transition STFB template, two more gates (two transitions) are added to the A and B signal paths, and the signal A is used to "self-reset" the states S0 and S1 by lowering B. Once the output has a token, the B signal is held low even after A is restored low. These extra transitions increase the template cycle time to 10 transitions, while the active and reset phase are still three transitions long, which results in a two-transition (two-gate-delay) margin on each side of the template drive (input–output).

The price for these margins is a slightly more complex circuit compared with the six-transition template. Also, for latency-critical "token-limited" systems, the 10-transition STFB template offers the same performance as the six-transition one. Moreover, a 10-transition STFB template is still much better than most QDI

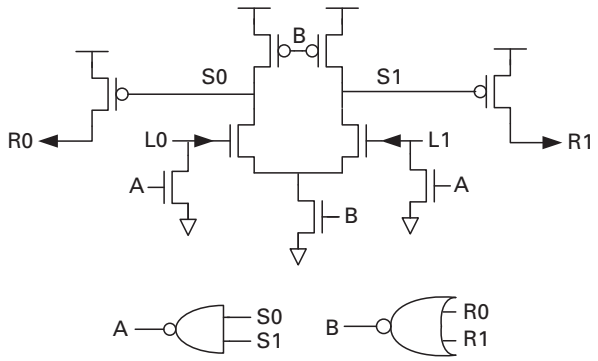

Figure 13.11. Simplified dual-rail STFB buffer.

templates, which require 14 to 18 transitions per cycle, and it can be used in conjunction with the six-transition templates since the active and reset phases have the same duration (three transitions) on both templates.

For complex stages with many inputs and outputs, as in 1-of-4 channels for example, the 10-transistion template may need some of the added inverters in the SCD and RCD to be changed to NAND or NOR gates to allow the easy handling of multiple tracks. For example, a 10-transistion 1-of-4 STFB stage could have two two-input NOR gates connected through a two-input NAND gate in order to perform the RCD function.

13.5 STFB pipeline stages

The following diagrams represent the basic STFB pipeline stages. Many details, explained later in this chapter, are simplified (no Reset signal, staticizers, or transistor sizing etc. are shown) for clarity.

13.5.1 STFB buffer

Figure 13.11 illustrates a dual-rail STFB buffer template, where the *N*-stack is simply implemented with two NMOS transistors. This stage just replicates the incoming token to the output channel. It may be used to improve the drive strength (i.e. in the middle of a long wire) and to add capacity by holding one more token in the pipeline (i.e. adding slack).

13.5.2 STFB fork

Figure 13.12 illustrates a dual-rail STFB fork stage, whose operation consists of replicating the incoming data to two different channels if all output channels are ready. Otherwise, the input data must wait.

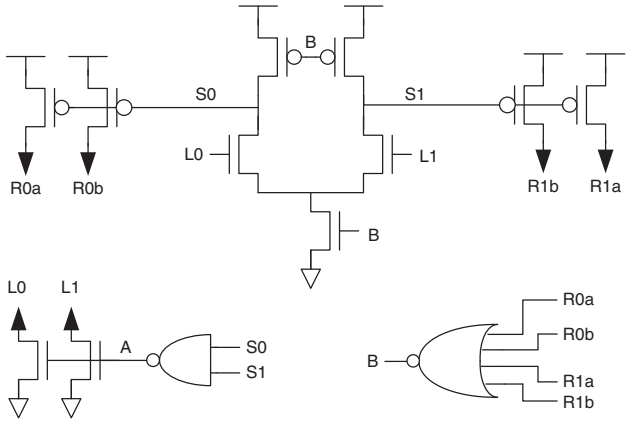

Figure 13.12. Simplified dual-rail STFB fork.

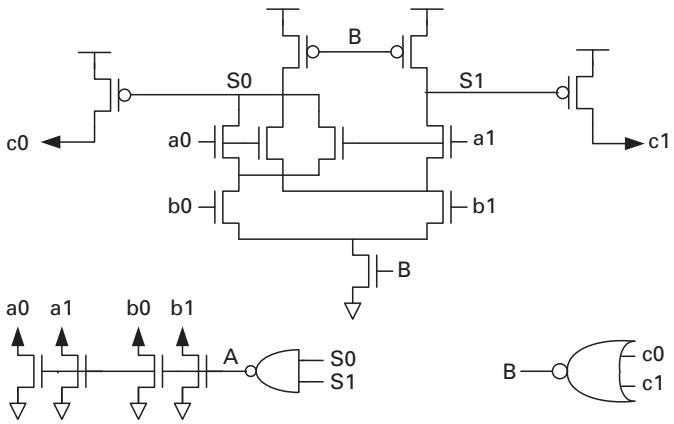

Figure 13.13. Simplified dual-rail STFB join (here, an AND function).

13.5.3 STFB join

Figure 13.13 illustrates a dual-rail STFB join stage, whose operation consists of consuming two incoming data from two different channels and generating an output token if both input tokens have arrived and the output channel is ready. In this example, the join operation is an AND function.

This functionality of the STFB join stage is defined by the arrangement of the NMOS transistors in the *N*-stack. Different functions (OR, XOR, etc.) may be implemented by simply rearranging the *N*-stack transistors accordingly.

13.5.4 STFB merge

The merge operation consists of choosing one of the incoming tokens a and b, on the basis of the value of a control token C. If the output path is busy, the input and control must wait. After forwarding the selected data, the control token is also consumed.

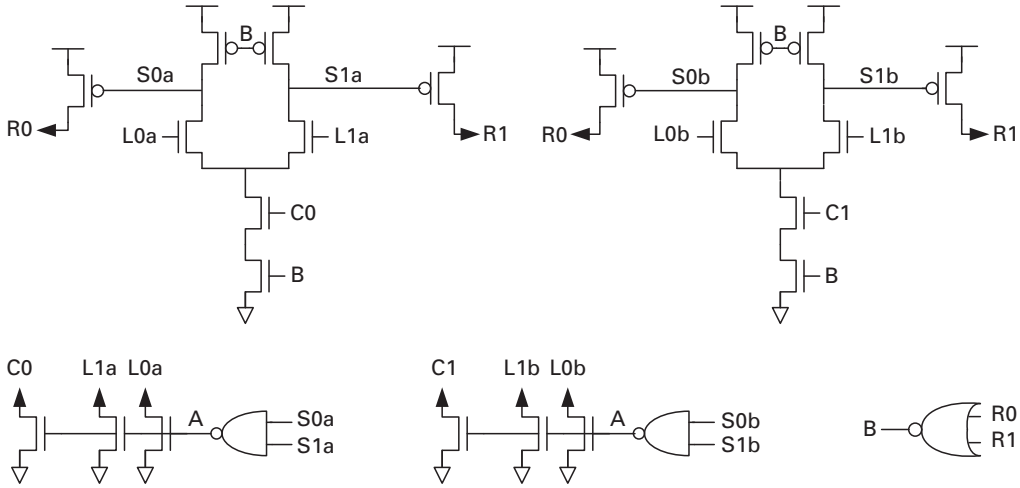

Figure 13.14. Simplified STFB merge.

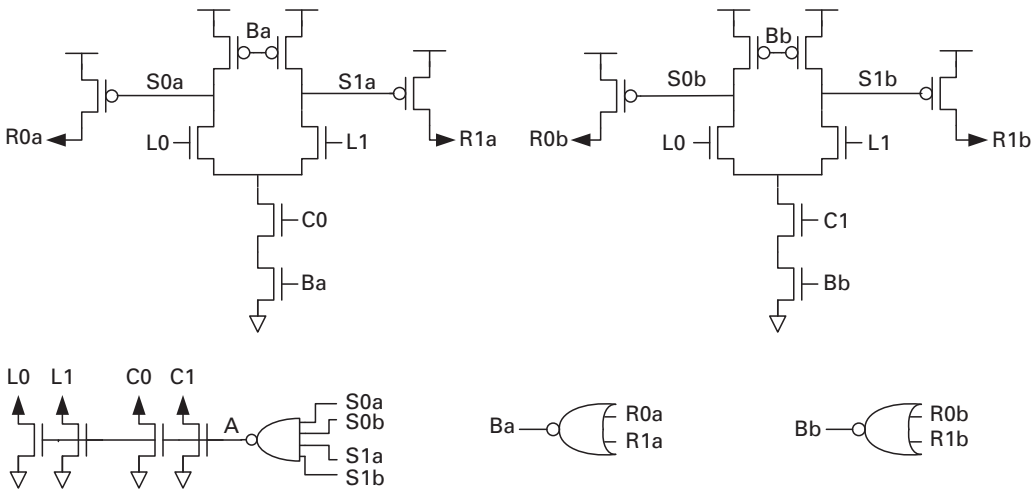

Figure 13.15. Simplified STFB split.

Figure 13.14 shows a two-to-one merge. If C = 0 then the La signals are directed to R and if C = 1 then the Lb signals are directed to R.

13.5.5 STFB split

The split operation consists of forwarding incoming tokens to one of two output channels on the basis of the value of a control channel C. If the chosen output path is busy, the data must wait.

Figure 13.15 shows a one-to-two STFB split circuit. In this example, if C is low then L is directed to Ra and if C is high then L is directed to Rb. Interestingly, the

STFB split allows a token to be forwarded to one channel even if the other channel is busy, which increases the level of parallelism.

13.5.6 STFB arbiter

The single-track full-buffer arbiter (STFB_ARBITER) uses a conventional single-rail mutual exclusion (MUTEX) circuit surrounded by single-track interfaces.

Figure 13.16 shows a common implementation of the MUTEX circuit (see e.g. [26]). The transistor sizing and the choice of NAND gate (NAND2B_28_12) were derived from the STFB library [15]. The operation of the MUTEX is as follows: initially the two inputs R1 and R2 are low, which causes both the outputs X1 and X2 of the set–reset (SR) flip-flop (implemented with two NAND gates) to go high. Therefore, both the MUTEX outputs G1 and G2 are low.

If one request arrives at a time, for example, if req1 goes high then the operation of the MUTEX is trivial: for req1 high, X1 goes low, which makes G1 high as expected.

If both requests go high at the same time, the SR flip-flop may take some time to decide which input wins (this is known as the *metastability* of the circuit [26]), and X1 and X2 may assume some intermediate value (for example, $V_{dd}/2$). The transistors in front of the SR flip-flop will keep both outputs (G1 and G2) low while X1 and X2 have similar voltage levels. In other words, G1 or G2 will be driven high only after the SR flip-flop is out of the metastable state.

Now we have the basic blocks to build the STFB_ARBITER. We just need to add logic to drive and hold the dual-rail single-track output, as shown in Figure 13.17.

Notice that the NReset signal initializes the input channels, req1 and req2, low. The output channels, out1 and out2, are initialized by the next STFB stage connected to these signals. If one input arrives (for example, if the single-track channel req1 goes high) then the respective MUTEX output (G1) also goes high, driving its state (S1) low; this fires the STFB_POUT transistor (at top right), inserting an output token (out1), and removes the input token (req1). Notice that, if one output channel is busy (for example, out1) and a new request arrives at the other side (on req2) then the other output channel (out2) can operate normally. This means that the STFB_ARBITER above is "non-blocking," which is a useful performance characteristic for an arbiter.

Figure 13.18 shows the STFB_ARBITER transition graph for a single input cycle that requires eight transitions (the broken arrows represent the number of transitions of the neighboring stages). (Note that the transitions G− and S+ are not shown in the figure because they are not critical to the arbiter's cycle time.)

Figure 13.19 shows the STFB_ARBITER timed Petri net for the system (with most places omitted for simplicity). If both inputs req1 and req2 are continuously sending requests, then when a token arrives in the place p, only the transition G1+ or G2+ that was not most recently fired is now enabled to fire. In other words, the place p will alternate between servicing G1+ and G2+. Consequently, the cycle time reduces to five transitions per request (10 to service both inputs), as suggested by the boldface path in Figure 13.19.

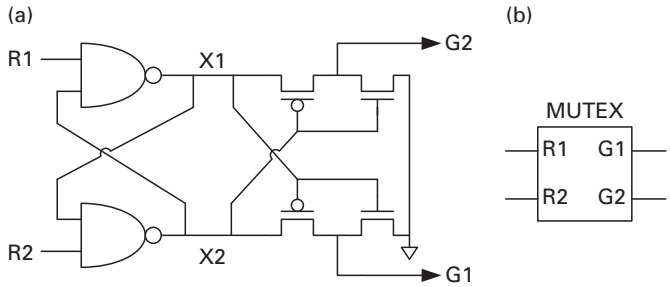

Figure 13.16. MUTEX circuit, (a) schematic and (b) symbol.

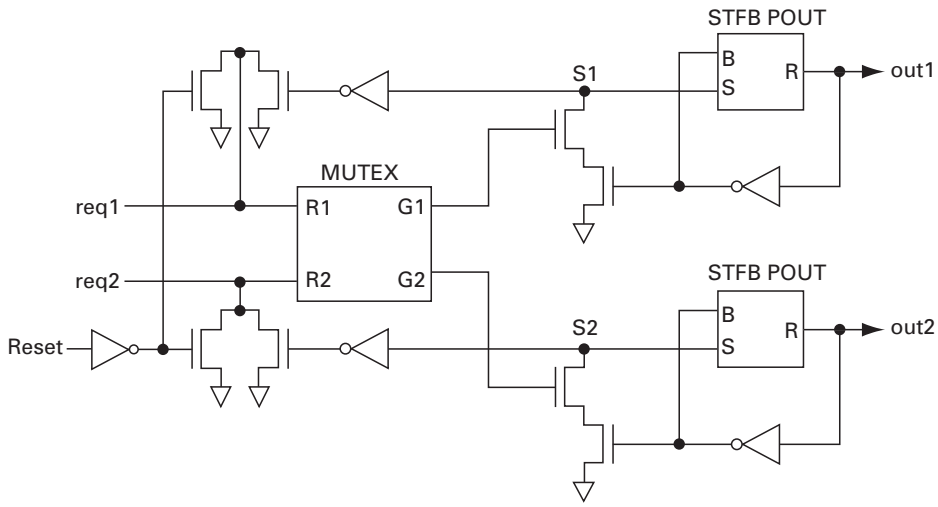

Figure 13.17. STFB_ARBITER diagram.

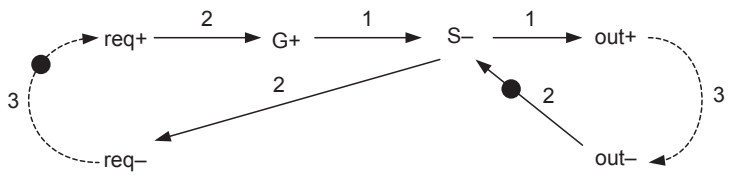

Figure 13.18. STFB_ARBITER timed marked graph for a single request.

13.5.7 STFB kpg adder

The cell STFB3_KPG2_KPG2 utilized in the 64-bit asynchronous prefix adder described below is an example of a cell implementation using 1-of-3 single-track channels, with a forked output. The forked output allows the cell to generate two output tokens in two different channels as if it had an embedded fork stage.

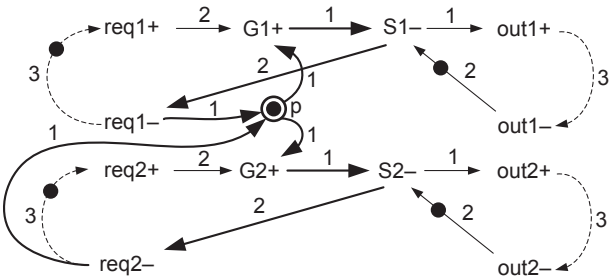

Figure 13.19. STFB_ARBITER timed Petri net for multiple requests.

The KPG (kill, propagate, or generate) carry operation uses two previous kpg results to compute the next KPG output (which is forked), and it can be described by the following equations:

$$\begin{aligned} \text{Ka} &= \text{Kb} = \text{Rk} + \text{Rp.Lk}, \\ \text{Pa} &= \text{Pb} = \text{Rp.Lp}, \\ \text{Ga} &= \text{Gb} = \text{Rg} + \text{Rp.Lg}, \end{aligned}$$

where (Ka, Pa, Ga) and (Kb, Pb, Gb) are the two 1-of-3 output channels and (Rk, Rp, Rg) and (Lk, Lp, Lg) are the two 1-of-3 input channels.

Figure 13.20 shows the STFB3_KPG2_KPG2 cell. In this cell, the NMOS transistor stack implements the KPG equations. One of the states Sk, Sp, and Sg fires on the basis of the values of the input tokens. Then two sets of output driver transistors are activated simultaneously, which generates two tokens, one for each output channel. In addition, the "embedded fork operation" requires that both output channels must be empty, which raises the signals Ba and Bb, in order to allow the *N*-stack to fire again.

13.5.8 Shared channels

As shown in Section 13.1, the communicating stages read and drive both sides of a single-track channel. Therefore, the STFB template was designed to have "point-to-point" channels, which means that there are no forked wires. This characteristic is also utilized to size the cells properly, as explained in Section 13.6.

In order to keep the "point-to-point" channel property, a common channel can be used, as shown in Figure 13.21, where merge stages are utilized to multiplex four channels into one, and split stages are utilized to demultiplex. Notice that the rings on each side will sequentially select the input channels to route the data to the respective output channel and that buffers may be added to the channel if necessary. Similar MUX and DEMUX structures are utilized in the design considered below in Section 13.8 to load or unload 64-bit data through an eight-bit port.

In addition, with some modifications the STFB template allows another possibility for collecting and distributing data: shared channels.

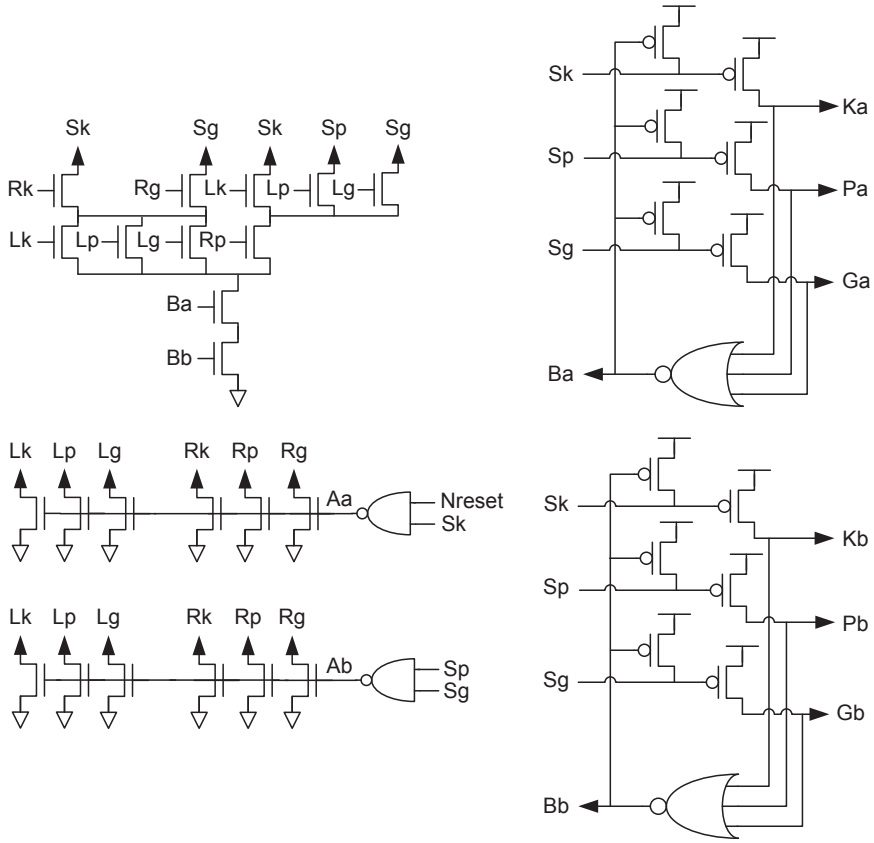

Figure 13.20. STFB_KPG cell with an embedded copy, referred to as STFB3_KPG2_KPG2.

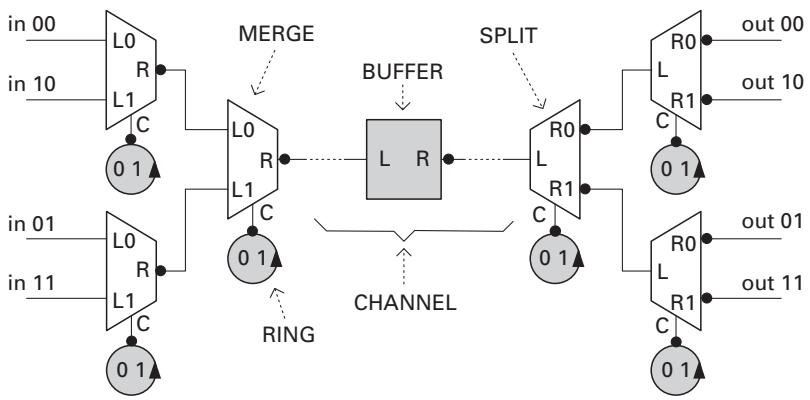

Figure 13.21. Multiplexer to demultiplexer shared-channel approach.

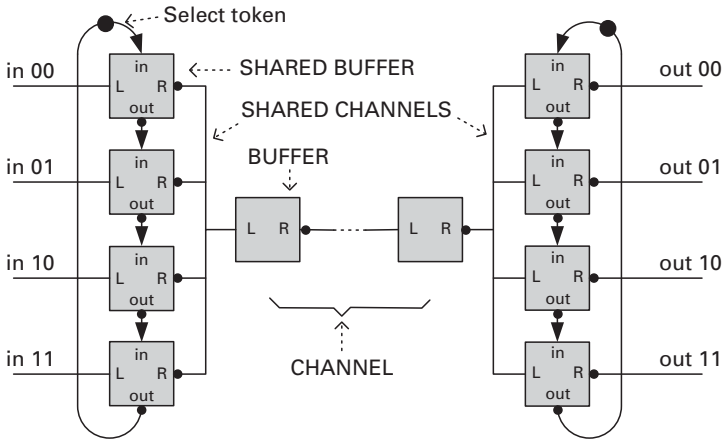

Figure 13.22. Forked-wire shared-channel approach.

Figure 13.22 exemplifies the use of shared channels, where the shared buffers allow their outputs and inputs to be connected in a shared-channel format (with forked wires). The buffer modifications are basically the input–output staticizer, which is needed to take into account the possibility of another buffer to drive the channel, and the in–out control channels, which are single-rail single-track channels required for selection of the input–output pair of shared buffers. The select token must be initialized, usually after reset, in order to synchronize both sides and ensure mutually exclusive access to the channel.

One approach for the stages connected to the shared channel is simply to remove the staticizers inside the stages and add one in the shared channel. This means that the stages connected to the shared channel would have only to contest the staticizer to send or remove a token, not each other.

In addition, in order not to degrade the system's performance, the output and input transistor drivers must be properly sized, taking into consideration the shared channel length and the load of the other stages connected to it.

In [1], van Berkel and Bink mention the possibility of implementing "multiple active and passive subscribers" (multiple senders and receivers), which basically generalizes the concept of shared channels. We could represent this by connecting together (without the common channel connection) the shared channels shown in Figure 13.22 and adding some other form (different from the select ring shown) of selection that indicates which stage is actively reading and which is writing to the bus (shared channel).

13.5.9 Bit generators and buckets

A bit generator creates a data token every time the channel is empty, while a bit bucket consumes unwanted tokens. Both are used during the testing and operation of STFB-based circuits. Examples of an STFB bit generator and a dual-rail bit bucket are shown in Figure 13.23.

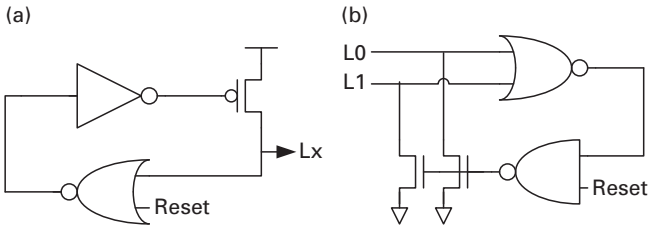

Figure 13.23. Single-track full-buffer (a) bit generator and (b) bucket.

13.5.10 Token insertion

A token buffer Tok Buf is a buffer that, during initialization, inserts a token in the pipeline in order to initialize it (otherwise, there would be just empty channels after reset). For the STFB cells, in order to save area and simplify the circuit, the input channels are forced to zero during reset (not the output, as is commonly found in other templates and for synchronous designs). In order to have a token inserted in the input channel, one input wire is forced high during reset.

Figure 13.24 shows a simplified Tok Buf diagram, where the input L1 is forced high during reset.

Note that during reset (NReset is low) the NMOS transistor stack is disabled to prevent it from firing before the reset is released (NReset goes high). This is necessary because the token is inserted before the buffer (in the input channel) and the output channel may be cleared (emptied) during the reset, which sets the B signal high. After the reset signal is de-asserted, the token buffer behaves as a conventional STFB.

13.6 STFB standard-cell implementation

In order to use the STFB template in a standard-cell place and routing flow, it is necessary to implement a STFB cell library as described below for the Taiwan Semiconductor Manufacturing Co. (TSMC) 0.25 μm process. In this case, the process parameters are taken into account in order to design the cells accordingly. Similar steps should be taken for other processes.

13.6.1 Transistor-sizing strategy

An important characteristic of the STFB architecture is that all the channels are point-to-point channels. This means that there are no forked wires and the channel load is a function of the wire length and the next-stage input capacitance. Consequently, since the fanout is always 1, the variance on output load is even more dominated by variations in the wire lengths than in synchronous designs. Therefore, this initial version of the library, presented here as an example of a real

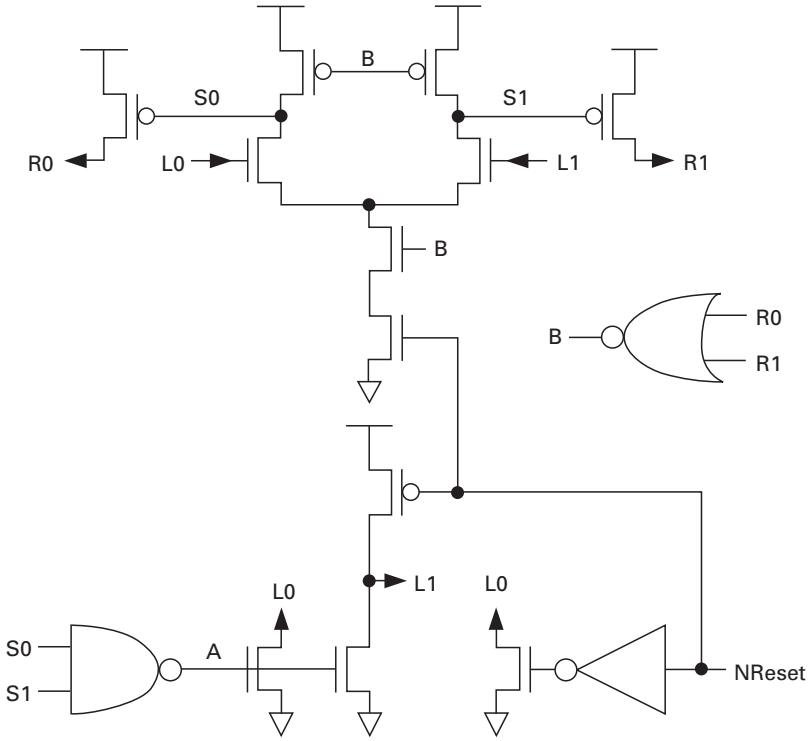

Figure 13.24. Simplified Tok Buf.

implementation, adopts a single-size strategy for each STFB function. The chosen size can drive, with reasonable safety and adequate performance, a buffer load through a wire up to 1 mm long and with width 0.4 μm and spacing 0.5 μm (as found in a TSMC 0.25 μm process). Although the TSMC 0.25 μm process allows somewhat smaller transistors, it was chosen since the minimum NMOS transistor has width 0.6 μm. Then the minimum PMOS, with the same strength as the minimum NMOS, is 1.4 μm wide. To balance the rise and fall slopes of each node, the sizes of all the PMOS and the NMOS networks are implemented with a ratio equal to approximately 1.4/0.6. Also used for sizing was a known practical scaling rule that one inverter can drive efficiently four to five times its own input load. Therefore, to drive a 1-mm-long line, the load capacitance would require an 8× driver transistor (a driver transistor with eight times the strength of the minimum transistor). This means that the PMOS driver and the NMOS reset transistor need 10 μm and 5 μm widths respectively. Scaling the load drive, the *N*-stack needs to have the strength of at least twice the minimum-size transistors in order to activate the respective PMOS drive transistors and the SCD inputs. This means that the width of each NMOS transistor in the *N*-stack should be at least $1.2k$ μm, where k is the number of transistors in the path driving the state to ground.

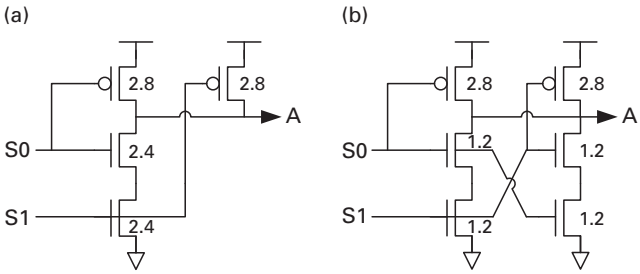

Figure 13.25. Sub-cell NAND2B_28_12: (a) conventional diagram and (b) implemented balanced input diagram.

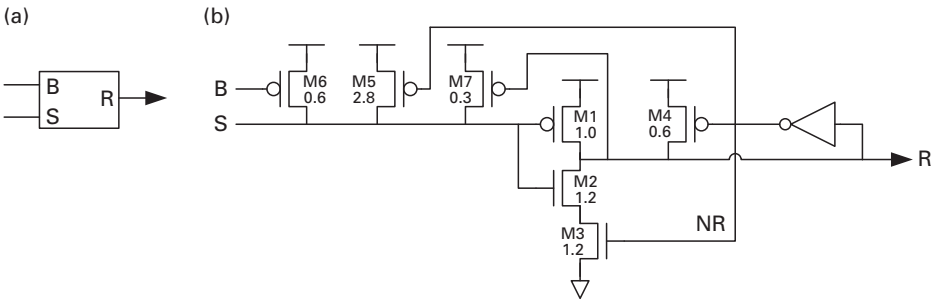

Figure 13.26. Sub-cell STFB_POUT (a) block diagram and (b) schematic.

Symmetrized transistor stacks are utilized to perform the SCD and RCD functions inside the cell. Figure 13.25 shows the symmetrization of a two-input NAND gate (note that the widths are in μm and all lengths are 0.24 μm). This approach minimizes the influence of the data in the cell timing behavior.

13.6.2 Output sub-cell STFB_POUT

The output driver sub-cell STFB_POUT is utilized in all STFB cells of this library. It includes the staticizer structure and three PMOS transistors, which restore the state input S high, as illustrated in Figure 13.26. (Again widths are in μm and all lengths are 0.24 μm.) If the output channel is empty then the B signal is high, R is low, and NR is high. At the same time, M2 and M3 hold R low. When S is driven low, the output driver PMOS transistor M1 drives the output R high, which makes the minimum size inverter drive NR low, deactivating M3 and activating M4 and M5. The RCD (not shown) will also make the B signal fall, activating M6. The transistor M4 will hold the line high while M5 and M6 drive S back high, turning off M1. The transistors M6 and M7 prevent leakage and charge-sharing. This template also improves robustness to charge sharing in the *N*-stack, because this output sub-cell has a low switching threshold voltage for the S signal. The transistors M2 and M3 shift the activation threshold of S low (to around 60% of V_{dd}).

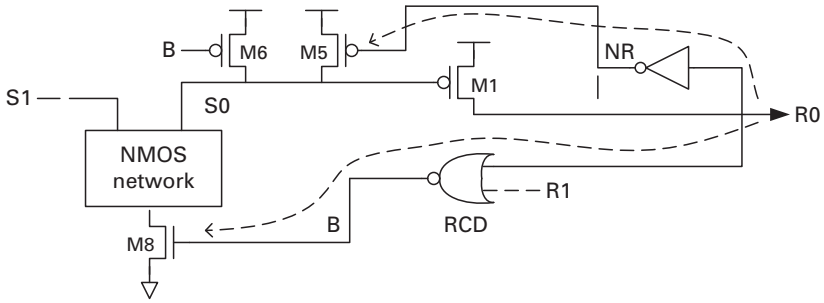

Figure 13.27. Simultaneous activation of B and NR.

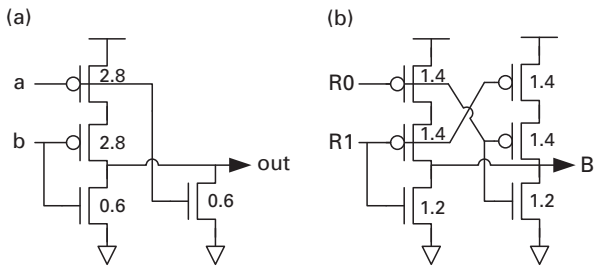

Figure 13.28. (a) Conventional 2-input NOR and (b) balanced RCD.

Notice that M5, controlled by the staticizer inverter (the NR signal), quickly asserts S after its output rail is driven high. This enables M6 to be smaller, thereby reducing the load on the B signal and enabling a very fast cycle time.

13.6.3 The RCD sizing

The NOR gate in the STFB template (the RCD) can also be implemented as a symmetrized gate. Its function is to drive the B signal low no later than the signal NR goes low in order to disable the *N*-stack and restore the signal S, as shown in Figure 13.27. This timing assumption is satisfied by reducing the load connected to the RCD output ($W_{M6} = 0.6$ μm, which is good enough to prevent *N*-stack charge sharing) and by transistor sizing, as shown in Figure 13.28, where the NMOS transistors of the balanced RCD are 1.2 μm wide, while for a regular minimum-sized NOR gate we would use 0.6 μm.

13.6.4 Input channel reset transistors

In the STFB template, the input token is consumed by driving the input channel wires low. This is achieved when the signal A generated by the SCD block activates a set of 5-μm-wide NMOS transistors connected to each input wire. Also, to initially reset the entire circuitry, a global NReset (active-low reset) signal

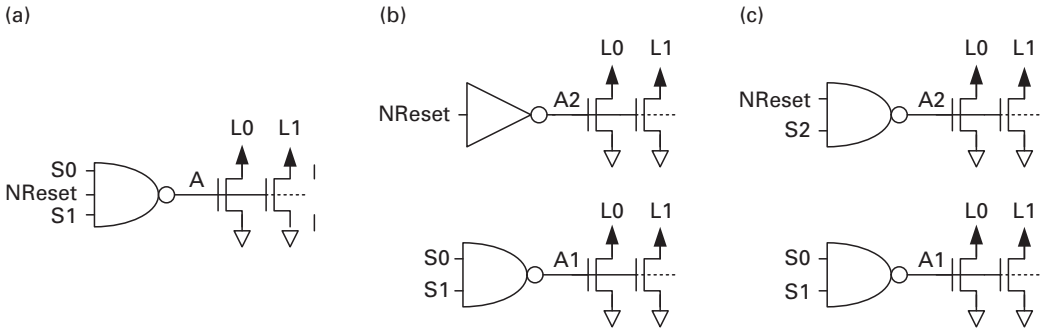

Figure 13.29. SCD and reset (a) initially proposed and the implemented (b) 1-of-2 and (c) 1-of-3.

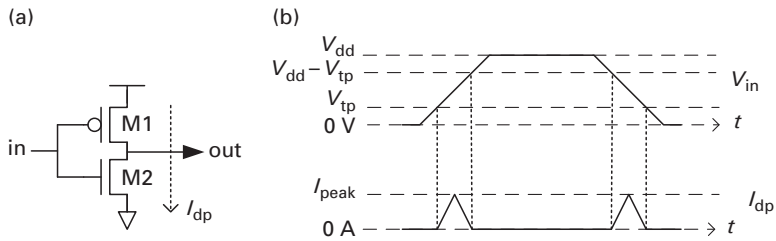

Figure 13.30. (a) Inverter and (b) direct-path current in a conventional CMOS inverter.

is used to force all channels low. Originally, this signal was simply added as one input to the SCD block [10]. Figure 13.29(a) shows the initially proposed three-input SCD. Figures 13.29(b), (c) show the implemented reset structure, which uses two-input NAND gates, allowing a smaller load on the states (S0, S1, S2) and offering a better performance of the SCD for dual-rail and 1-of-3 channels. Notice that the added transistors can share the same drain connections, resulting in only a marginal increase in area and input capacitance for the STFB stage.

13.6.5 Direct-path current analysis

A perceived problem with STFB designs is the amount of direct-path current, also known as short-circuit current, caused by violations of the timing constraint associated with the tri-stating of a wire before the preceding or succeeding stage drives it. Figure 13.30 shows a conventional CMOS inverter where, during the rise time t_r and fall time t_f of the input voltage V_{in}, both transistors will be briefly active, allowing a direct-path current from V_{dd} to ground. Since it has an approximately triangular shape, we can estimate the direct-path current as $I_{dp} = I_{peak}/2$ [1].

For our STFB pipeline stages, the NMOS transistor gate is connected to a signal A and the PMOS transistor gate is connected to Sx (one of the states). Figure 13.31

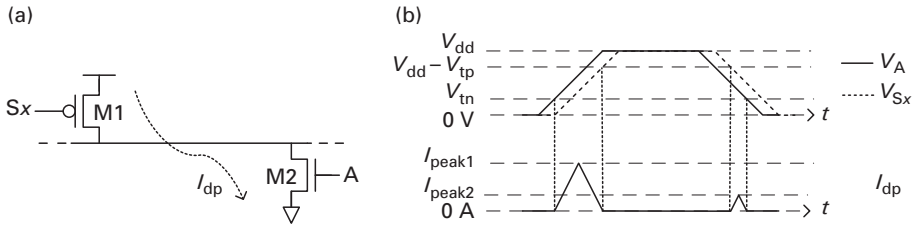

Figure 13.31. (a) Single-track full-buffer output–input drivers and (b) direct-path current if $V_A \neq V_{Sx}$.

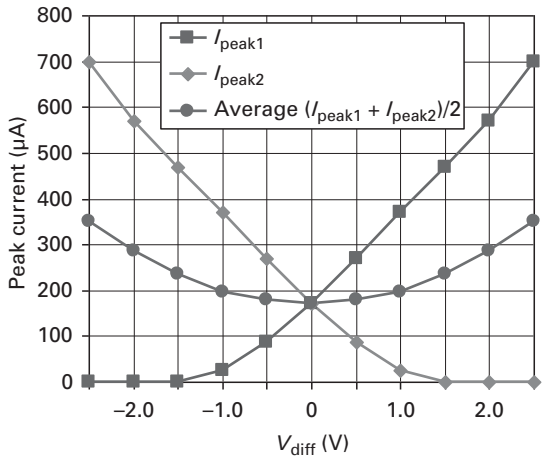

Figure 13.32. Peak direct-path current versus the PMOS–NMOS gate voltage difference.

shows the direct-path current if V_A occurs earlier than V_{Sx}. If the voltage difference $V_{diff} = V_A - V_{Sx}$ is zero, the STFB stage I_{dp} is similar to a conventional inverter. However, if one of the voltage transitions occurs ahead of the other, i.e. V_{diff} is different from zero, we may observe a higher peak current during one transition and a smaller peak current during the next transition, or vice versa.

Figure 13.32 shows the peak direct-path current versus the PMOS–NMOS gate voltage difference V_{diff} during an input rise and fall edge ($V_{diff} = V_A - V_{Sx}$). These values were obtained from DC HSPICE simulation analysis using typical parameters with transistors that were double our minimum size. Notice that, if V_A and V_{Sx} have similar shapes (similar width, rise, and fall times), the average peak current is close to the inverter peak current as long as $V_{diff} < 1$ V.

Simulations using HSPICE also show that the direct-path current of the STFB templates is no worse than that for an inverter driving the line, and that the timing assumption associated with the tri-stating of one stage before the other drives the line is not a hard constraint to meet. For the STFB stages, the time difference between V_A and V_{Sx} is bounded by the wire-length constraint (subsection 13.6.1) to ensure correct operation.

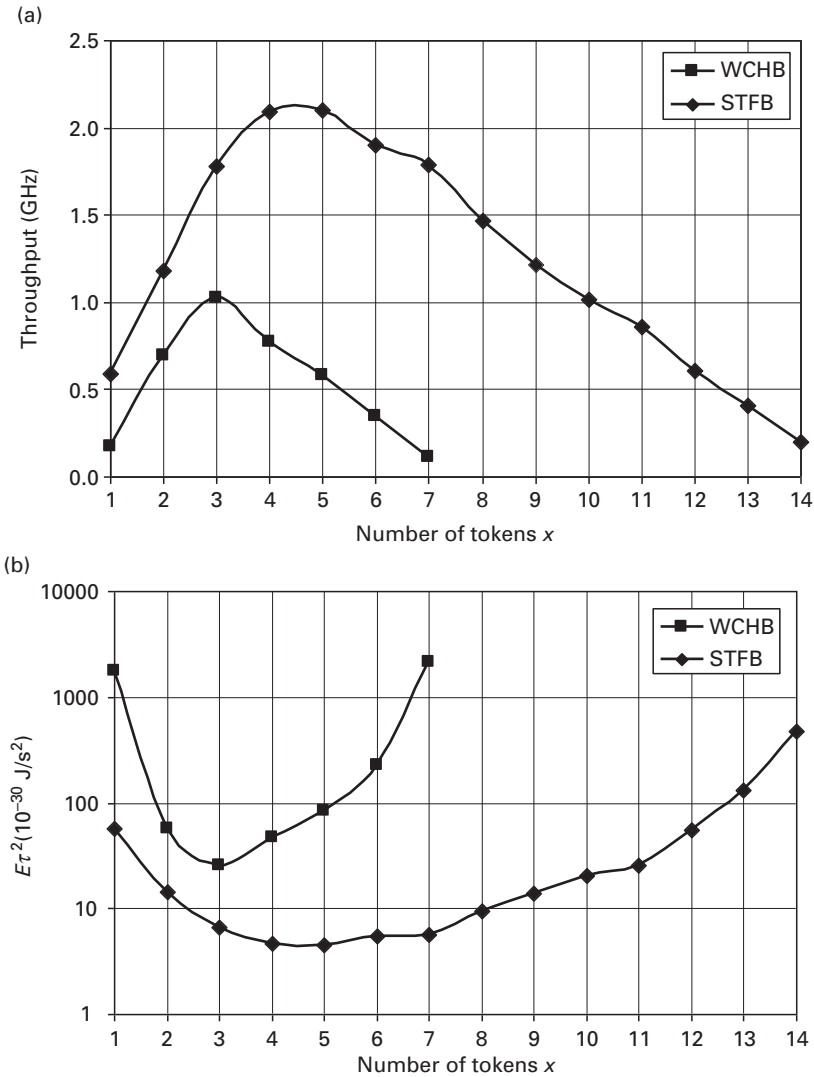

Figure 13.33. (a) Throughput and (b) $E\tau^2$ metric versus pipeline occupancy for two 15-buffer pipelines.

13.6.6 Performance analysis

Figure 13.33 shows simulation results for two 15-buffer pipelines implemented with WCHB and STFB buffer templates. Weak-conditioned half-buffer is the fastest buffer implementation of the QDI templates [16][17] (precharged half-buffer is better for more complex stages). Figure 13.33(a) demonstrates that the smaller cycle time of the single-track full-buffer template offers higher throughput for the same number of stages, or equivalent throughput with fewer stages, since it can handle more tokens per stage than WCHB. Figure 13.33(b) shows the power

metric $E\tau^2$ [25], where the better efficiency of STFB is evident (by a factor 10 at peak throughput). This metric allows us to say that the STFB pipeline could match the WCHB speed (because of the lowering of the power supply voltage) and would require much less energy per token to perform the same job.

13.7 Back-end design flow and library development

The preliminary version of an STFB Asynchronous Standard Cell Library for Cadence tools, which was released through the MOSIS Educational Program [15], contains all common sub-cells for dual and 1-of-3 rail logic cells and specific cells used in the test chip. At the time of writing, work was being done to add Verilog behavioral views of all cells, input capacitance, delays to the library, using the Liberty (.lib) file format [27] allowing delay back-annotation for fast and accurate Verilog simulation [11].

For example, Figure 13.34 shows the layout and Figure 13.35 a schematic for the STFB2_XOR2 cell. This cell is an STFB pipeline stage with two dual-rail input channels and one dual-rail output channel. In the library, this cell has four views: symbol, functional, schematic, and layout. The symbol view is used to instantiate the cell in higher-level schematics. The functional view is the Verilog behavioral description of the cell. The schematic view is a transistor-level schematic of the cell, including the symbols of the sub-cells used to implement this cell. The layout view, similarly to the schematic view, is composed of a cell-specific part and various sub-cells, as shown in Figure 13.34. In this figure, we can see that the STFB2_XOR2 cell includes the eight input transistors that define the XOR function and an STFB2_CORE4I sub-cell. This sub-cell includes four reset transistors, one INV_28_12, one NAND2B_56_24, one NOR2B_14_12OD, which includes the *N*-stack foot transistor, and two STFB_POUT sub-cells (subsection 13.6.2).

Notice that, by rearranging the input transistor connections shown in Figure 13.34(a), we can easily implement other two-input one-output cells such as STFB2_AND2 and STFB2_OR2 [7].

13.8 The evaluation and demonstration chip

A test chip was designed to validate the design flow and the performance of the STFB templates. The central block of the test chip is a 64-bit STFB prefix adder, while the input and output circuitry, both STFB blocks, were designed to feed the adder and sample the results, enabling the checking of its performance and correctness at full throughput.

13.8.1 The prefix adder

The prefix adder architecture, described in detail in [13], allows a high degree of parallelism. Nevertheless, the micro-architecture was defined using equations adapted to 1-of-*N* encoding [7][8]. The 64-bit asynchronous prefix adder is nine

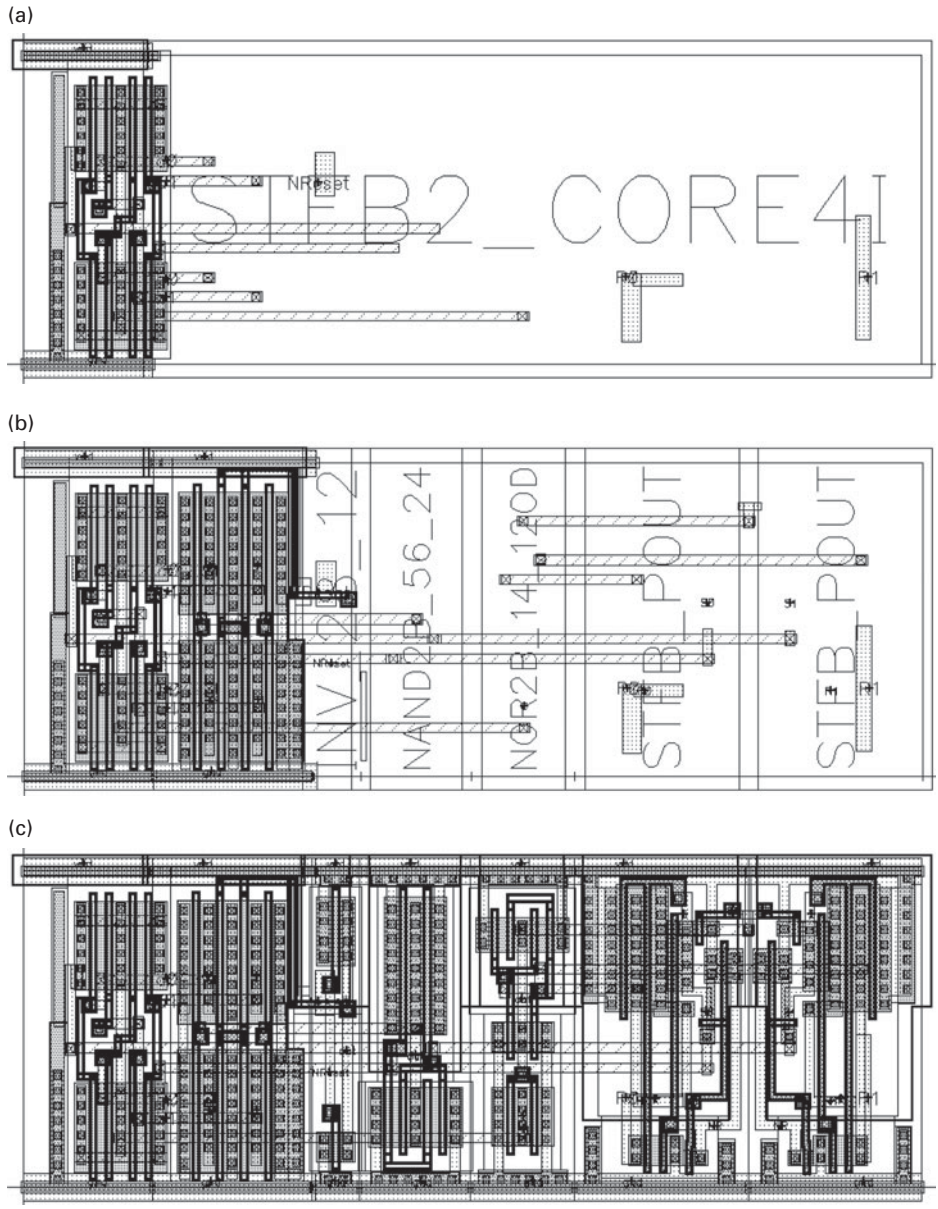

Figure 13.34. STFB2_XOR2 cell: (a) custom layout and STFB2_CORE4I sub-cell, (b) with STFB2_CORE4I sub-cell expanded, and (c) with all sub-cells expanded.

levels deep. This means that, with nine times the forward latency of the STFB stage ($9 \times 2 = 18$ transitions), 64 bits plus carry-out are available. In addition, since the cycle time of the STFB stage is just six transitions, the 64-bit adder can process up to three additions simultaneously (i.e. three tokens in the pipeline) at maximum throughput.

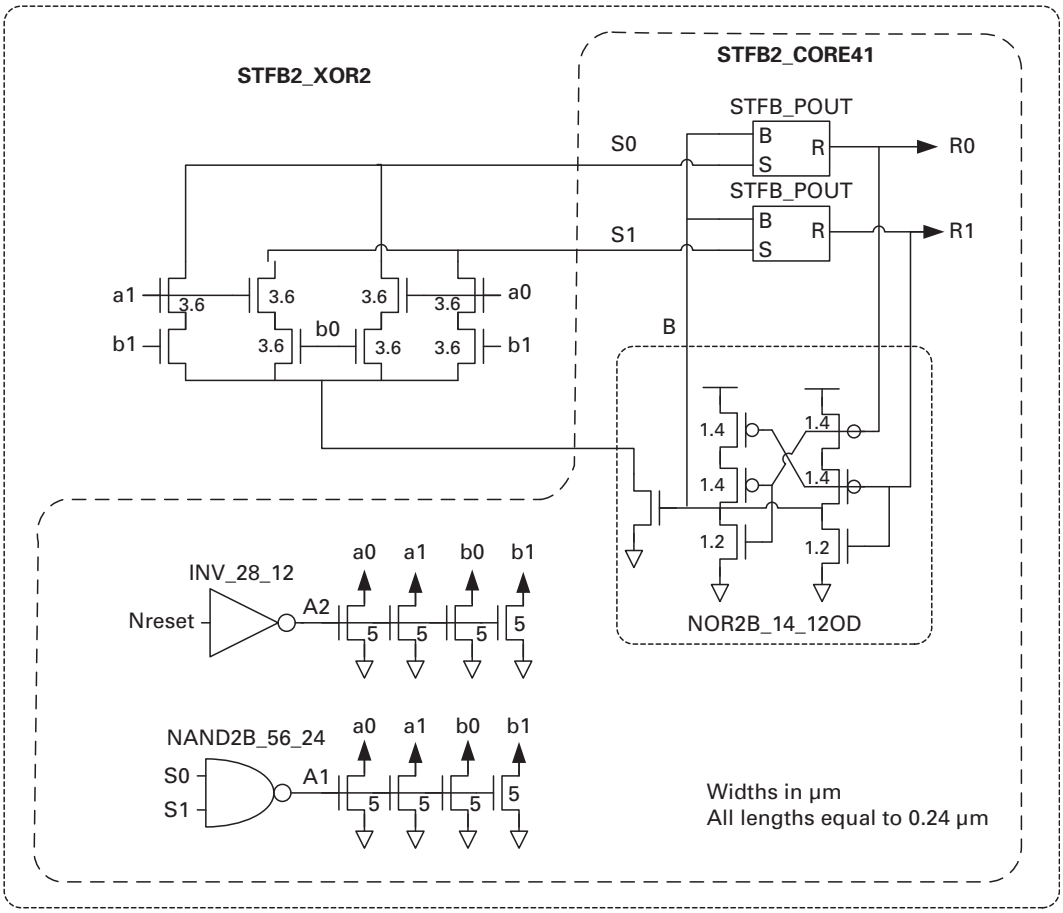

Figure 13.35. STFB2_XOR2 cell schematic.

13.8.2 The input circuitry

Figure 13.36 shows the INPUTGEN129BY9 input generator block, which loads and continuously repeats a test pattern to be fed into the adder. The block is composed of single-rail to single-track converters, split circuits, and 129 nine-stage rings (two 64-bit numbers and carry-in). Figure 13.37 shows the nine-stage ring diagram, where there are seven buffers, one fork cell F, one unconditional merge M, one XOR, and the controlled bit generator BG. After the tokens are loaded into the rings, the BG cell is enabled with the "Go" signal (not shown). Since now the XOR stage has one token in each input, it generates a token that enters the fork stage, where one copy is sent to the adder and another is sent back into the ring.

13.8.3 The output circuitry

In order to test the adder when it was running at full throughput, it was connected to a programmable output circuitry that sampled the 65-bit result

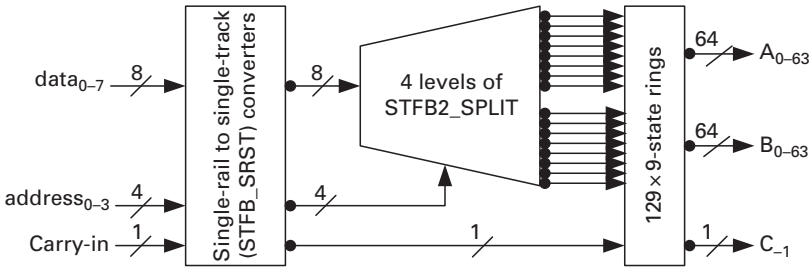

Figure 13.36. INPUTGEN129BY9 block diagram.

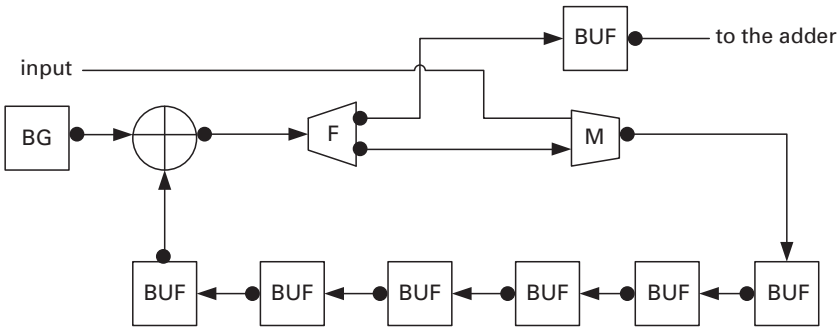

Figure 13.37. Nine-stage ring utilized in the input circuitry.

(64-bit sum and one-bit carry-out), forwarding to the output pins one out of n results ($0 \leq n \leq 7840$). The SAMPLER65BY1000 output circuit is implemented with three 30-stage rings, each connected to a 65-bit split structure. The 30-stage rings are similar to the nine-stage ring in Figure 13.37; they have 27 buffers in the loop instead of just six, and they can be individually loaded with a sequence of tokens.

Figure 13.38 illustrates the sampler circuit where each split stage, controlled by a 30-stage ring, directs an input token either to a bit bucket BB, where the token is destroyed, or to the next split. The 65-bit output of the last split is sent to the output pins. The carry-out is separated, converted to single-rail, and sent to its exclusive pin. The 64-bit sum is sent to an asynchronous mux element (M) that routes to the output one byte at the time, starting with the most significant. The 30-stage rings can run at full throughput if loaded with 10 tokens each. This would also result in a sample rate equal to 1 out of 1000 results. Moreover, loading the rings with different numbers of tokens can change the sample rate. We need to be careful, however, not to slow down the adder if we want to check its performance at full throughput. If the first ring is loaded with 10 tokens, the other two can be loaded with 28 tokens each, yielding a sampling rate up to 1 out of 7840 results without limiting the adder throughput.

Note that if the external test circuit is slow in consuming the output results then the sampler, the adder, and the input circuit will slow down to accommodate it and no sampled data will be lost.

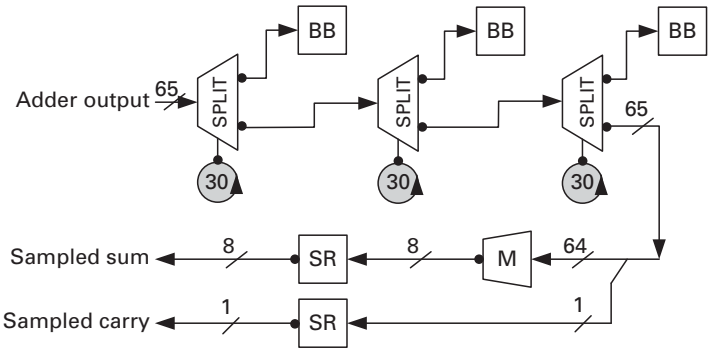

Figure 13.38. Block diagram showing SAMPLER65BY1000, mux 64 to 8, and single-rail converters.

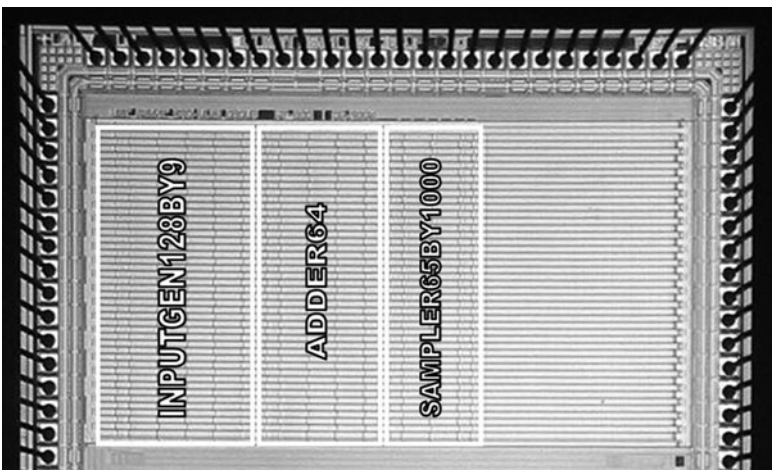

Figure 13.39. Single-track full-buffer blocks in the ASYNC1b chip.

13.8.4 The chip implementation

Figure 13.39 shows the die photo of the demonstration chip (ASYNC1b), where the STFB blocks are placed under a power grid implemented with a metal-5 Layer, which hides the circuits underneath. The place and route for each block was performed separately with area utilization 80%. The blocks have the same height (1.7 mm) and the placements of the adder block pins match that of their neighbors. The total area is 4.1 mm^2.

Notice that performing place and route on separated blocks significantly reduces the probability that a very long wire could compromise the performance and functionality of the design, as discussed in subsection 13.6.1. In fact, a post-layout check guaranteed that no STFB signal wires were longer than 1 mm. In addition, as filler cells, a total of 1.6 nF in bypass capacitors was added to the chip.

Table 13.1. STFB, PCHB, and static CMOS templates

Function	Cell	Latency	Cycle time	Area (μm^2)	Area ratio
Buffer	STFB	189 ps (2)	463 ps (6)	415	4.5
	PCHB	220 ps (2)	1.96 ns (14)	726	7.9
	CMOS	165 ps (2)	—	92	1.0
Two-input AND/OR	STFB	177 ps (2)	471 ps (6)	472	4.6
	PCHB	176 ps (2)	2.05 ns (14)	968	9.3
	CMOS	177 ps (2)	—	104	1.0
Two-input XOR	STFB	177 ps (2)	471 ps (6)	472	2.6
	PCHB	179 ps (2)	2.12 ns (14)	1048	5.7
	CMOS	225/312 ps (2/3)	—	184	1.0

13.8.5 Power distribution and electromigration

Nanosim post-layout typical simulation (transistor model TT, 25 °C, $V_{dd} = 2.5$ V), running at full throughput, achieved 1.4 GHz and required a power supply current of 3.3 A [8]. On the basis of these numbers, the voltage drop and electromigration were taken into account in the design of the power grid [15]; 28 pins were allocated for the power supply [7].

Regarding the single-track channels, although they also conduct unidirectional currents, no electromigration problem was observed (subsection 13.8.8). It may become important for higher current densities (heavier loads, faster transitions, or a smaller process), as discussed below in Section 13.9.

13.8.6 Comparisons

Table 13.1 shows for comparison some STFB pipeline stages with PCHB [16] stages and static standard cell CMOS gates driving another stage through a 1-mm-long line (process TSMC, 0.25 μm, TT, 2.5 V, 25 °C). The numbers in parentheses give the latency and cycle time in terms of the corresponding numbers of transitions. The static CMOS standard-cell gates used in this comparison were designed under the same standard-cell specification as utilized for the STFB and PCHB pipeline stages, and they were implemented using a 2× gate (i.e. a gate having double the minimum strength) followed by an inverter implemented with a 10-μm-wide PMOS and 5-μm-wide NMOS in order to match the driving strengths.

As shown in the table, STFB is smaller, faster, has more capacity, and utilizes fewer wires than PCHB. In addition, the STFB area is closer to that of static standard-cell CMOS gates if we add to them the latch, flip-flop, and clock-tree overheads required for fine-grain synchronous pipeline designs. Moreover, the single track full-buffer capacity, 1-of-N data encoding, and two-phase protocol allow close to full-custom performance in a standard-cell flow, which is not possible with static standard-cell CMOS gates.

Goldovsky *et al.* [13] implemented a 32-bit full-custom static CMOS prefix adder using the Lucent 0.25 μm process. It achieved 1 ns latency, used

0.03 mm^2, and consumed 32 mW at 400 MHz. These numbers do not include the flip-flop, latch, and clock tree necessary for the synchronous design. On the basis of the 64-bit STFB prefix adder, it is possible to estimate values for a 32-bit STFB prefix adder: 1.8 ns latency, 0.43 mm^2 area, and 1.44 W at 1.4 GHz. The static CMOS version was implemented using labor-intensive full-custom transistor sizing and layout, while the STFB version uses standard cells in an automatic place-and-route flow. Although the area and power required by the STFB circuit are significantly higher, the power efficiency metric $E\tau^2$ [22] is about the same for both circuits ($\sim 5 \times 10^{-28}$ J s^2). One may argue that, in some applications, three or four static CMOS circuits could be used in parallel to achieve a similar throughput with somewhat smaller area and power than STFB. However, STFB still allows close to full-custom performance with standard-cell design time.

13.8.7 Demonstration-chip implementation and test

For the ASYNC1b chip, the package utilized is a ceramic 132-pin pin grid array in which the STFB circuits use 56 pins: 28 power supply pins (14 V_{dd} and 14 ground), 12 bi-directional pins, 13 input pins, and three supply pins for the pads (3.3 V, 2.5 V, and V_{ss} (ground)). The remaining 72 pins of the package are taken by the QDI part of the ASYNC1b chip [23][24]. It runs on top of an evaluation board [7] that disables the QDI part of the chip, and it uses a field-programmable gate array (FPGA) to setup and run the STFB part. Once programmed, the FPGA loads the STFB input block with the operands, sets the sample rate in the output block, and runs the ASYNC1b chip by acknowledging all requests as they come out of the chip.

13.8.8 Test results

Figure 13.40 shows the measured waveforms of chip number 3 (all 40 samples were numbered for tracking purposes); channel 1 shows the acknowledge signal produced by the FPGA for every carry-out request from the chip. Its frequency, 313 kHz, indicates that the 64-bit adder is running at 1.25 GHz since the sample rate was set to 1 : 4000. The channel 2 signal shows the acknowledge signal of the result, sent on the output one byte at a time, at a rate of 200 ns each (5 MHz). The sampler rings, as explained in subsection 13.8.3, can be programmed with different numbers of tokens, allowing various sample rates, and with the selecting bits in different positions in order to define the sampling sequence.

Initially, after the rising edge of the reset (NReset) signal, configuration data is loaded into the chip. Notice that by loading ring 0 with 11 tokens, and rings 1 and 2 with 19 tokens we have a sample rate of 1 : 3971. Also, since the input rings are much faster than the adder, without reducing the adder throughput we can load, for example, three carries, three 64-bit operands for A, and four 64-bit operands for B, resulting in 12 combinations.

Figure 13.41 shows the operation of the logic analyzer demonstration chip. After a rising "Go" signal, the input rings start feeding the adder continuously

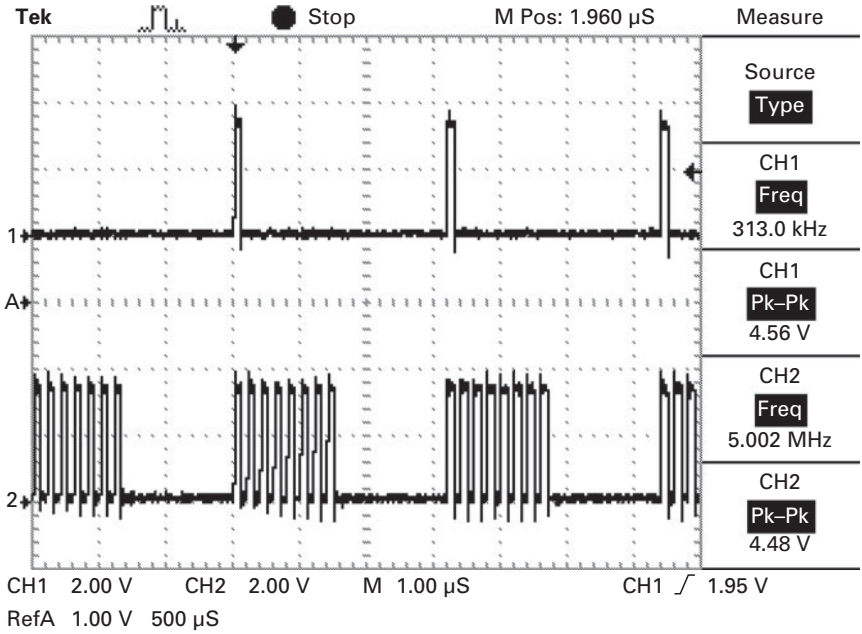

Figure 13.40. Chip number 3 at 1.25 GHz (2.5 V on-chip, 2.26 A, 40 °C package, fan at 1.5 inches).

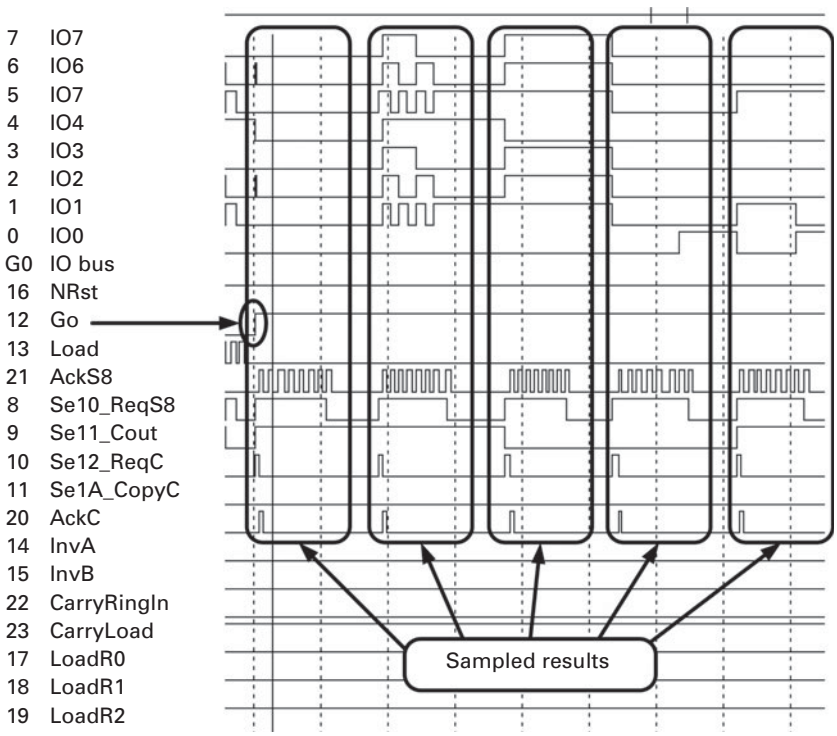

Figure 13.41. Logic-analyzer-captured wave form of the running mode.

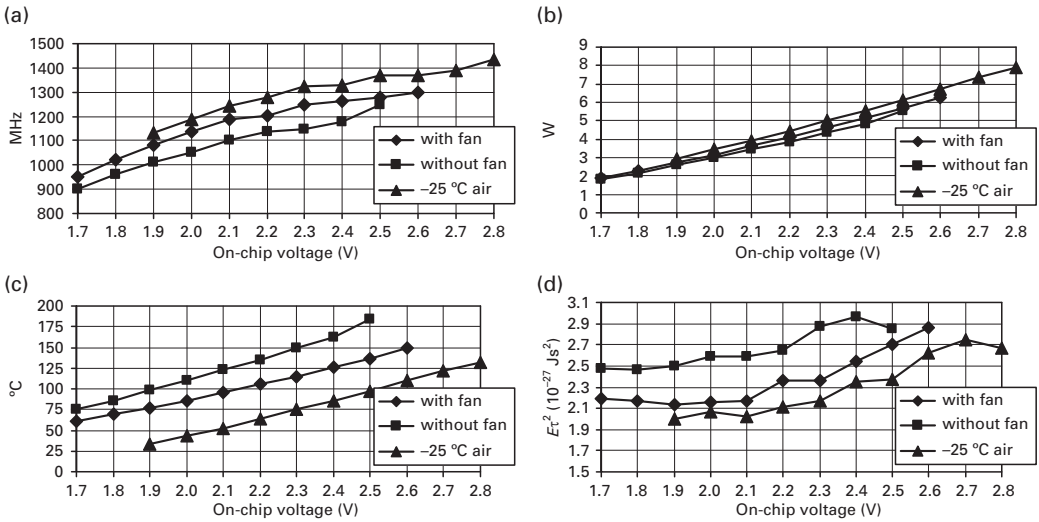

Figure 13.42. Measurements of performance for chips 3 and 4: (a) throughput, (b) power dissipation, (c) estimated junction temperature, (d) power metric. The measurements at −25 °C in air refer to chip 4 only.

while the output rings sample the results, allowing the first result to go out, then the 3972nd, the 7943rd, and so on. The resulting values and sequence are as expected for all test cases and sample rates applied.

Figure 13.42 shows some performance measurements for samples 3 and 4 for different supply voltages, with or without fan or the forcing of cool air. Notice that the voltage drop from the power supply to the voltage inside the chip is significant owing to the high level of current required. The on-chip voltage is measured by two supply pins (one at V_{DD} and the other at ground), which are connected to a voltmeter instead of the power supply. This means that the entire chip current is supplied through 13 pins at V_{DD} and 13 pins at ground, which represents about 170 mA per pin at full throughput (2.5 V, 1.28 GHz). At full throughput, due to the on-chip power grid size, we estimate that the voltage at the top of the 64-bit adder block to be approximately 0.1 V below the on-chip value.

Notice that the cooling operation yields a higher throughput and is more efficient (lower $E\tau^2$), since the power dissipation is about the same. The junction temperature was estimated from the ambient temperature, using the package values of the thermal coefficients, 20 °C/W with the fan (forced air) and 29 °C/W without the fan. Comparing with the simulation results [8], we can see that the top measured performance is close to (just 1.4% below) the TT 2.5 V 25 °C simulation case. However, for a real chip, we have to remember that the die temperature will be much higher and that there will be a voltage drop on the real power grid. Considering these effects, the performance of the real design is as expected. Figure 13.42 also shows results where the cooled chip reaches 1.45 GHz, under a forcing temperature system that blows air at −25 °C. Since the die temperature is cooler, the

performance gets close to the simulated performance. However, the voltage drop in the real power grid remains. The on-chip voltage range of our test was limited by the operation of the chip. Voltages above 2.6 V and below 1.7 V caused the chip to stop running when it was tested with just the fan. The reason for the upper limit is likely to be that on-chip noise induces or kills tokens, which in turn causes a complete halt of the circuit. The lower limit is likely to be due to the assumption that three transitions is sufficient to discharge the line (the charge operation has the staticizer and RCD feed-back to compensate; this is not the case for the SCD). The lower supply voltage would make the reset transistors weak and tokens would be left on the long channels clogging the pipeline and halting the circuit.

The overall performance of the chip is very good and its operation is stable. The tested samples were used continuously for several hours at full throughput without presenting wrong operations or detectable performance variation.

13.9 Conclusions and open questions

Single-track full-buffer templates are proposed for high-speed area-efficient asynchronous non-linear pipeline design. A STFB standard-cell library using TSMC 0.25 μm technology has been generated and is freely available through the MOSIS Educational Program. A complete STFB design with 260 k transistors has been successfully implemented and tested, reaching a measured throughput of 1.45 GHz.

The proposed templates have higher throughput than the fastest known QDI templates and lower latency than the most advanced GasP templates. Consequently, for systems that are latency-critical, STFB templates may yield a significant performance advantage even though GasP bundled data may have a smaller cycle time.

The STFB templates and implementation issues have been presented. The timing constraints and the performance of the STFB templates were compared with QDI templates. The small cycle time of the STFB templates allows the STFB circuits to operate at very high throughputs with small distances and periods of time between consecutive data tokens, resulting in smaller and faster circuits than their QDI alternatives. The power efficiency $E\tau^2$ is also advantageous, as was shown in subsection 13.6.6.

The demonstration design includes the input generating circuit, a 64-bit prefix adder, and a programmable position-and-rate-output sampler circuit. All these circuits were implemented using our STFB standard-cell library in a conventional back-end flow, which resulted in a simple, fast, and efficient design process that should be easily understood by synchronous designers.

The demonstration design chip exploits the advantages of the small STFB cycle time. The input circuit uses 129 nine-stage rings, which are examples of high-speed loops processing multiple data tokens. The 64-bit prefix adder represents a high-complexity design with large STFB stages operating with dual rail and 1-of-3 channels. The sampler circuit uses multiple rings running at different rates. In

addition, all the support logic to load the operands and unload the results is implemented with STFB stages.

As a continuation of this work, changes could be made in the template to improve the noise margins. In particular, if a smaller-feature-size process is targeted, different transistor sizing and/or the use of the static single-track (SST) protocol [7][12], discussed in subsection 13.4.1,where the single-track wires are continuously driven, are areas of future work. The 10-transition STFB template [10], discussed in subsection 13.4.2, may also be used to improve reliability over process variations, because of its self-reset characteristics and larger timing margins.

13.9.1 *N*-stack height limit

One open issue is the height limit of the transistor *N*-stack for the STFB template. Owing to its high performance, low latency, and short cycle time, the state node S needs to be driven and restored very fast. This may impose a limit on the high of the *N*-stack. Successful stages were implemented with four NMOS transistors in series (including the foot transistors). However, QDI templates allow a much higher *N*-stack since they use a left-hand environment completion detector (LCD) and the cycle time can vary significantly to accommodate a long computation time.

13.9.2 Electron migration effect

Differently from the conventional CMOS logic line drive, the single-track protocol charges the wires utilized in the single-track channels from one side and discharges them from the other. This means that most current flowing through the wire is unidirectional (as is the current in the power lines). Therefore, it becomes necessary to observe the process's current density limit in order to avoid the electron migration effect. This limit was not reached in the design presented in Section 13.8, but it may become an issue for deeper processes and may require the use of channel wires wider than the minimum width, adding an extra cost of area and power. This open issue needs to be investigated; furthermore, the electron migration limit for single-track channels may be higher since the current is not constant (it occurs only during transitions).

13.10 Exercises

13.1. In Exercise 11.1, we considered an unconditional cell with 1-of-2 inputs A and B and a 1-of-2 output channel Z, where Z implements Z = NAND(A, B). Draw the transistor-and-gate-level diagram using the STFB template. Compare the area with that of the PCHB template described in Chapter 11, on the basis of the relative numbers of transistors.

13.2. In Exercise 11.4, we considered a leaf cell with conditional output behavior as follows: A is a 1-of-2 input channel, S is a 1-of-2 input channel, and O is a 1-of-2 output channel; if S is a 1 then the token on A is consumed and copy

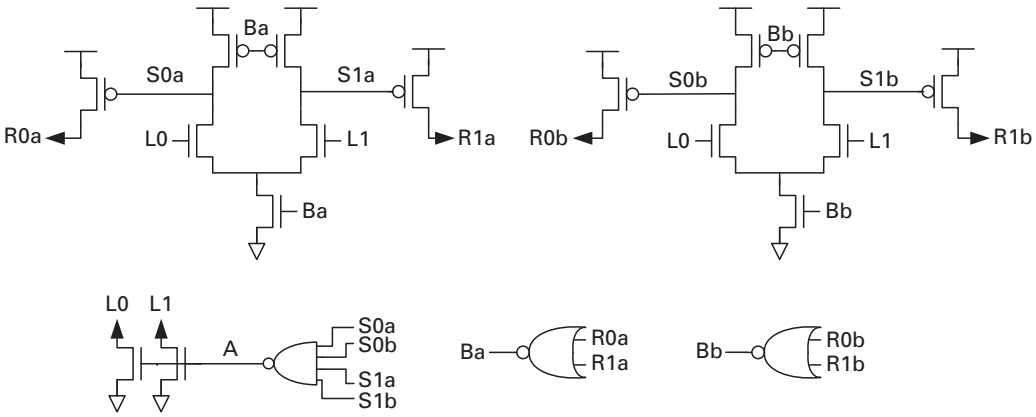

Figure 13.43. Single-track full-buffer circuit for Exercise 13.4.

of the value on O is sent. If S is a 0 then the token on A is consumed but no data is sent on S. Draw the transistor-and-gate-level diagrams for implementations using the STFB template.

13.3. In Exercise 11.5, we considered a leaf cell with conditional input behavior as follows: A is a 1-of-2 input channel, S is a 1-of-2 input channel, and O is a 1-of-2 output channel; if S is a 1 then the token on A is consumed and a copy of the data is sent on O. If S is a 0, a valid token on A is awaited and a token is sent along O but A is not acknowledged (i.e. consumed). In this way, the token on A is conditionally re-used, implementing a type of memory. Draw transistor-and-gate-level diagram for implementations that follow the STFB template.

13.4. A student Fred Flip proposed the following STFB one-bit copy with dual-rail input channel (L0, L1) and two dual-rail output channels (R0a, R1a) and (R0b, R1b) (see Figure 13.43). It has at least one bug. Give a sequence of events that explains why Fred's circuit does not always work.

References

[1] J. M. Rabaey, *Digital Integrated Circuits*, Prentice Hall Electronics and VLSI Series, 1996.

[2] K. van Berkel and A. Bink, "Single-track handshaking signaling with application to micropipelines and handshake circuits," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, March 1996, pp. 122–133.

[3] I. E. Sutherland and S. Fairbanks, "GasP: a minimal FIFO control," in *Proc. ASYNC*, 2001, pp. 46–53.

[4] J. C. Ebergen, J. Gainsley, J. K. Lexau, and I. E. Sutherland, "GasP control for domino circuits," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, April 2005, pp. 12–22.

[5] J. Ebergen, "Squaring the FIFO in GasP," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, March 2001, pp. 194–205.
[6] M. Ferretti and P. Beerel, "Asynchronous 1-of-*n* logic using single-track protocol," Dept of Electrical Engineering – Systems, University of Southern California, CENG Technical Report No. 01–03, 2001.
[7] M. Ferretti, "Single-track asynchronous pipeline template," Ph.D. thesis, University of Southern California, 2004.
[8] M. Ferretti, R. Ozdag, and P. Beerel, "High performance asynchronous ASIC back-end design flow using single-track full-buffer standard cells," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, April 2004, pp. 95–105.
[9] M. Ferretti and P. A. Beerel, "High performance asynchronous design using single-track full-buffer standard cells," *IEEE J. Solid-State Circuits*, vol. 41, no. 6, pp. 1444–1454, June 2006.
[10] M. Ferretti and P. A. Beerel, "Single-track asynchronous pipeline templates using 1-of-*N* encoding," in *Proc. Conf. on Design, Automation and Test in Europe (DATE)*, March 2002, pp. 1008–1015.
[11] P. Golani and P. A. Beerel, "Back annotation in high speed asynchronous design," in *Proc. PATMOS 2005*, September 2005, pp. 256–261.
[12] P. Golani and P. A. Beerel, "High-performance noise-robust standard-cell asynchronous library," in *Proc. IEEE Computer Society Annual Symp. on VLSI*, March 2006, pp. 256–261.
[13] A. Goldovsky, R. Kolagotla, C. J. Nicol, and M. Besz, "A 1.0-nsec 32-bit tree adder in 0.25-mm static CMOS," in *Proc. 42nd IEEE Midwest Symp. on Circuits and Systems*, vol. 2, no. 4, pp. 608–612, 1999.
[14] A. Goldovsky, H. R. Srinivas, R. Kolagotla, and R. Hengst, "A folded 32-bit prefix tree adder in 0.16-mm static CMOS," in *Proc. 43rd IEEE Midwest Symp. on Circuits and Systems*, vol. 2, no. 4, pp. 368–373, August 2000.
[15] USC Asynchronous CAD/VLSI Group, "Asynchronous CMOS single-track full-buffer standard cell library for TSMC 0.25 μm," MOSIS *Educational Program*, http://www.mosis.org/MEP/, Revision 0.3, June 2004.
[16] A. M. Lines, "Pipelined asynchronous circuits," *Master's thesis,* California Institute of Technology, June 1995.
[17] A. Lines, "Nexus: an asynchronous crossbass interconnect for synchronous system-on-chip designs," in *Proc. 11th Symposium on High Performance Interconnects*, August 2003, pp. 2–9.
[18] A. J. Martin, A. Lines, R. Manohar, *et al.*, "The design of an asynchronous MIPS R3000 microprocessor," in *Proc. Advanced Research in VLSI*, September 1997, pp. 164–181.
[19] A. J. Martin, "Towards an energy complexity of computation," *Information Processing Lett.*, vol. 77, pp. 181–187, 2001.
[20] M. Nyström, "Asynchronous pulse logic," Ph.D. thesis, California Institute of Technology, 2001.
[21] M. Nyström and A. Martin, *Asynchronous Pulse Logic*, Kluwer Academic, 2002.
[22] M. Nyström, E. Ou, and A. Martin, "An 8-bit digital divider implemented in asynchronous pulse logic," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, April 2004, pp. 229–239.

[23] R. Ozdag and P. Beerel, "A channel based asynchronous low power high performance standard-cell based sequential decoder implemented with QDI templates," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, April 2004, pp. 187–197.
[24] R. O. Ozdag, "Template-based asynchronous circuit design," Ph.D. thesis, University of Southern California, 2004.
[25] J. Teifel, D. Fang, D. Biermann, C. Kelly, and R. Manohar, "Energy-efficient pipelines," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, April 2002, pp. 23–33.
[26] J. Sparso and S. Furber, *Principles of Asynchronous Circuits Design – A System Perspective*, Kluwer Academic, 2001.
[27] *Synopsys, Liberty User Guide*, *vols. 1 and 2*, version 2003.10, October 2003.
[28] P. Joshi, P. Beerel, J. Gainsley, I. Sutherland, and M. Roncken. Static timing analysis of single track circuits. Design Automation Conference, User Track Poster.
[29] P. Joshi, Static Timing Analysis of GasP. M.S. Thesis, Dec. 2008.

14 Asynchronous crossbar

System-on-chip (SoC) technology gives the ability to place multiple functional "systems" on a single silicon chip, cutting the development cycle while increasing product functionality, performance, and quality. In the SoC design methodology, complex tasks or chips are divided into several independent functional blocks and then each block is realized using standard design methodologies with existing CAD tools. The key for a successful SoC endeavor is implementation of the on-chip infrastructure that connects these functional blocks to form a system.

Globally asynchronous, locally synchronous (GALS) SoC design architectures offer a powerful way to solve the interconnect issues with SOC chips that would otherwise arise in a bus-based system such as a globally clocked design with long data transition times, large timing margins, and performance targets that are usually missed.

A GALS-based SoC links islands of locally clocked synchronous logic using an interconnect that has its own local timing, which is decoupled and independent of the surrounding logic blocks. While many methods can be used for asynchronously interconnecting disparate logic blocks, two of the most popular are shared buses with clock domain conversion from each block to the common bus and asynchronous crossbar switches [1].

The controversial part of GALS, which is based on asynchronous technology, is the great unknown: the asynchronous interconnect [1]. An asynchronous GALS architecture provides the foundation for the massive amount of transistors available in a deep sub-micron (DSM) chip. It is estimated that these devices can support as many as two dozen IP blocks. In a bus-based GALS design, and also in an end-to-end synchronous design, it may take several clock cycles to move data from one end of a chip to another – introducing dozens of nanoseconds of latency just for interblock connectivity.

However, in an asynchronous GALS design the interconnect is transparent, acting much more like a simple wire than an arbitrated bus system where multiple clocked buffers stand between the sending and receiving blocks [1]. Asynchronous GALS also provides a seamless way to bridge the different speeds of each logic block. A clockless interconnect provides speed-independent ports that allow data to enter the interconnect at the native speed of the initiating block and to be switched out of the interconnect at the speed of the receiving logic block, with no attendant metastability or rate-mismatch problems.

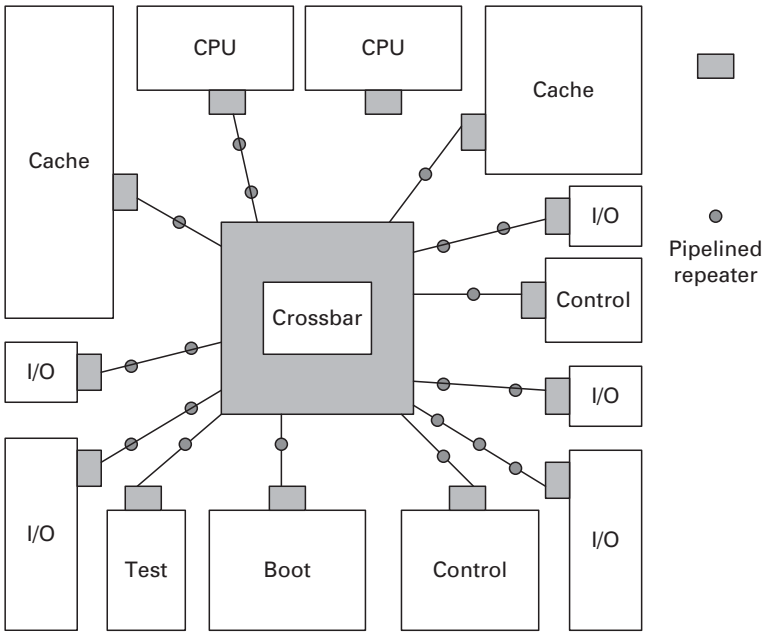

Figure 14.1. Typical system on chip (SoC) using Nexus.

Power is also a significant issue in SoC designs because the increased circuit density drives up heat dissipation. When not processing data, each asynchronous circuit consumes only its leakage current. This results in an average power consumption that rises and falls with ongoing processing. It also means that clock distribution circuitry is not needed.

14.1 Fulcrum's Nexus asynchronous crossbar

Fulcrum's Nexus is a crossbar-based interconnect that connects several locally asynchronous modules to each other on the same chip [16]. Each module is capable of supporting a different clock frequency and converts from the asynchronous domain to the synchronous domain and vice versa with a clock domain converter (CDC). Data enters Nexus from a synchronous interface, is converted to the asynchronous domain via the CDC, propagated to its destination module, and then converted to the synchronous domain again via the destination CDC. The asynchronous channels carry the data across the chip via the central crossbar. The internal asynchronous crossbar handles arbitration and routing and resolves contention on the output ports. All ports are flow-controlled and are designed to be delay-insensitive. A high-level configuration using the Nexus is illustration in Figure 14.1.

Nexus transfers data from a source port to a destination port in bursts. A burst is a variable number of data words, terminated by a tail bit. Each burst is routed

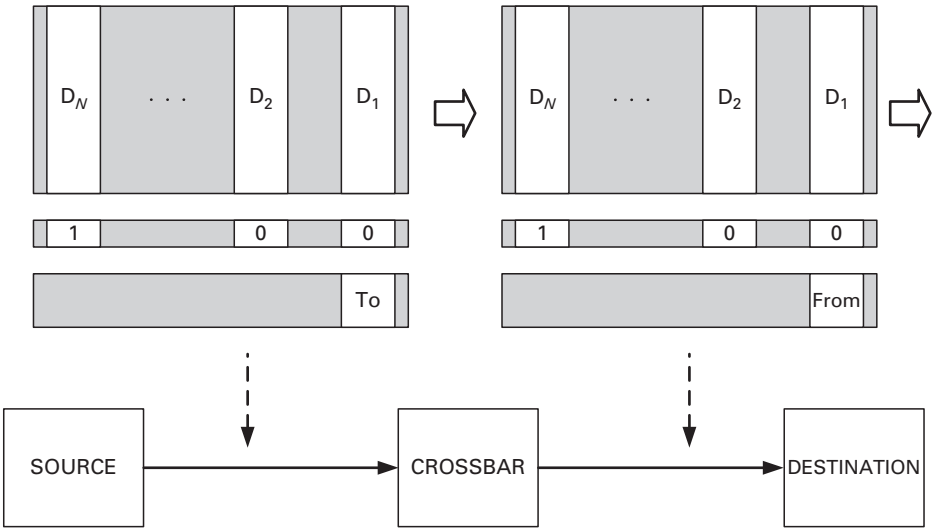

Figure 14.2. The Nexus burst format.

with a sideband "To" channel to indicate its destination. This sideband information is conveyed into a "From" channel when it arrives at the destination port, to indicate the source port sending the data burst. These bursts are routed automatically and cannot be fragmented, interleaved, duplicated, or dropped.

The structure of the burst format is illustrated in Figure 14.2.

When a device connected to one port wants to send a data burst to device connected to another port, it sends the To information alongside the first data word of the burst. If this is not the last word then the associated tail bit will be set to 0. Since more than one input port may want to send a data burst to the same output port, these potential requests are arbitrated at the destination port. The winning input port of the arbitration is granted a direct link from the source port to the destination port, and this link will open and be available only to the source-destination port pair until the data burst is completely transmitted and the tail bit arrives at the destination port.

The existing versions of Nexus support a 16-bit bi-directional port with a 36-bit datapath, plus the tail bit and a four-bit To/From channel.

14.1.1 The crossbar

The asynchronous crossbar component of Nexus, illustrated in Figure 14.3, is implemented by decomposing it into smaller circuits that communicate on channels.

The largest part of the crossbar is the datapath, which is responsible for MUXing all input data channels to all output data channels. A control channel S[i] for each input port i is used to split the incoming data to 16 possible output

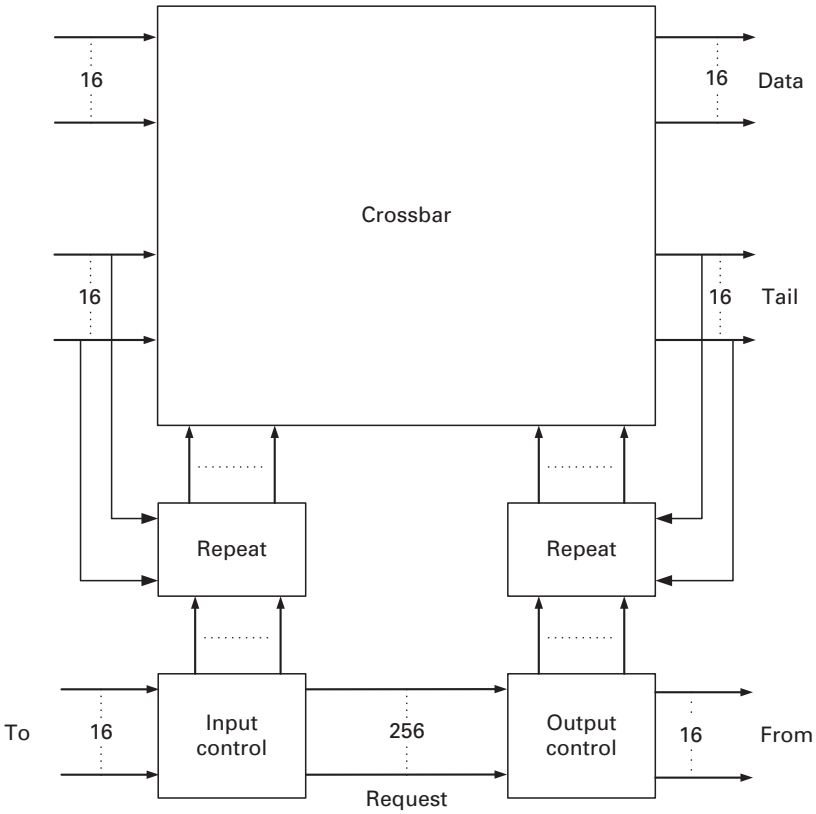

Figure 14.3. The crossbar high-level architecture.

destinations. The data on the control channel comes form the "To" portion of the incoming data burst and is used to direct the incoming data burst to the specified destination. Similarly, a control channel M[*i*] for each output port is used to merge data from 16 possible input ports. The merge control channel comes from the arbiter output on the output side and is also used to generate the "From" portion of the outgoing data burst. In between the input and output control generator circuits and the datapath are repeat circuits, which replicate the same split–merge control data until a tail bit 1 passes through the link, indicating that the data has been completely transmitted and the link between the sender port and the receiver port can be released to be used by a different source or destination port. The micro-architectural organization of these components is illustrated in Figure 14.4.

14.1.2 Input control

Each input port has an input control unit that receives the To data and copies it to the S control to select to which port the data should be sent. It also sends a token on a request channel (implemented with a 1 data wire and a 1 acknowledge wire)

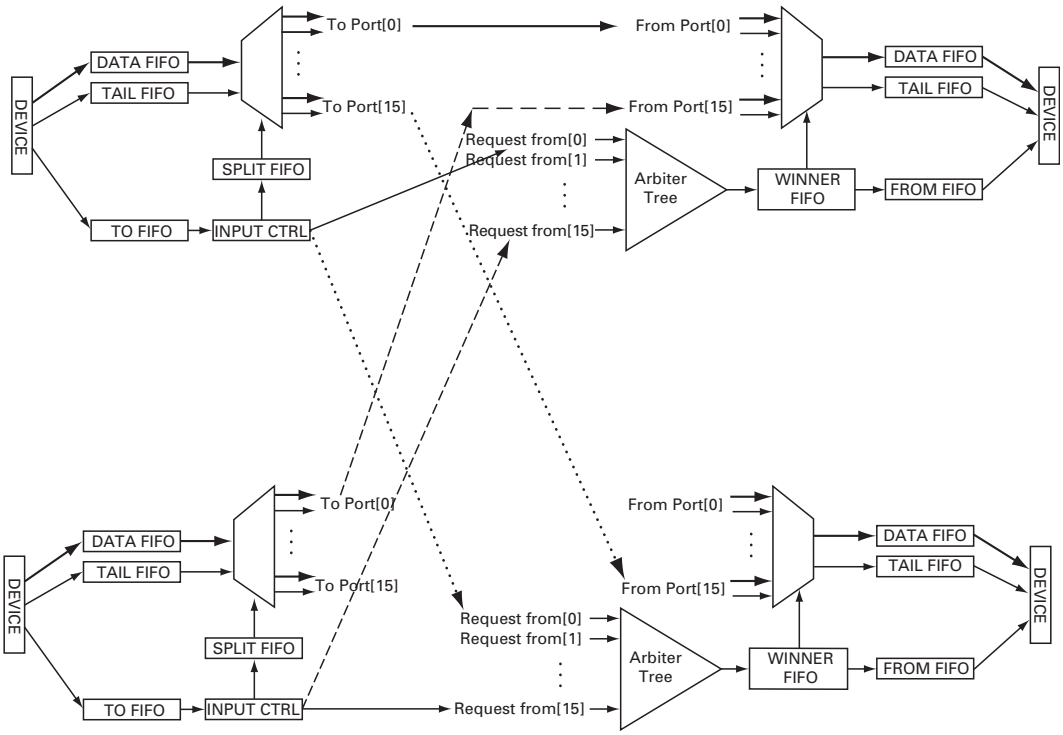

Figure 14.4. The crossbar micro-architecture.

to the selected output control unit. There are 256 channels, connecting all 16 input control units to all 16 output control units.

14.1.3 Output control

The output control unit receives requests from one or more input ports. It arbitrates among the input requests, picks one, and sends its port number on the From channel to the output and on the M channel to merge data from the chosen input port. If multiple requests arrive at exactly the same time, metastability occurs and so a metastability filter is included to enable the system to wait for it to resolve. In a QDI circuit, such metastability introduces only minor uncertainty in the latency of arbitration; it does not introduce the possibility of failure. It is the fixed clock period that makes metastability a source of failure in synchronous designs.

If an input port were able to make two requests to two output ports at once, it might win its second request first. If another input had also requested the same output ports, but in the opposite order, it too could have its second request win first. Thus the two inputs would each have won permission to send to their second destination but would have data waiting on input ports intended for their first destination. This results in deadlock.

There are two ways to solve this problem. Both essentially require that each input have at most one request outstanding at a time. This means that an input must be certain that it has won its first request before starting upon a second request. This can be done by introducing backward-flowing grant channels from the outputs to the inputs. A trickier approach is to remove enough of the integrated pipelining that the first request token blocks the second until it has won the arbitration. This second approach is substantially smaller in scale.

In either approach, the corresponding constraint introduces a performance bottleneck in the control circuitry that causes it to be slower than the rest of the system, by 25% or 100% for the two approaches, respectively. However, since arbitration is also pipelined and occurs once per burst and not once per word, such bottlenecks rarely affect actual performance and never matter for bursts of two words or longer.

14.2 Clock domain converter

The clock domain converter (CDC) consists of two independent circuits: a synchronous-to-asynchronous converter (S2A) for outbound traffic and an asynchronous-to-synchronous converter (A2S) for inbound data. In both directions, full sender and receiver flow control is propagated across the boundary.

14.2.1 Synchronization control circuit

The S2A and A2S converters have essentially the same control circuit for data transfer. The control circuit has an "enable" output signal to indicate that the control unit is ready to accept more data and a "valid" input signal to indicate that new data has arrived, to be transferred to its left-hand side. Similarly, it receives an enable input signal from the right-hand side indicating that the next unit is ready to accept more data and also produces a valid signal to indicate that it is sending new data. On the asynchronous side of the converter the enable and valid signals form an e1-of-1 channel (data rail and an acknowledge signal in the opposite direction). On the synchronous side they are the request and grant channels. The converter control also receives the clock form the synchronous module and produces a "go" signal to latch the data.

14.2.2 Clock domain converter datapath

The S2A datapath uses the control output to latch the synchronous data into a flip-flop and perform an asynchronous handshake. It works on the assumption that the asynchronous handshake will complete within a clock cycle without blocking. The A2S datapath is similar; it takes an asynchronous 1-of-4 code and latches it in flip-flops. It works on the assumption that, once the transfer is started, all asynchronous data is present and will complete the handshake within a clock cycle.

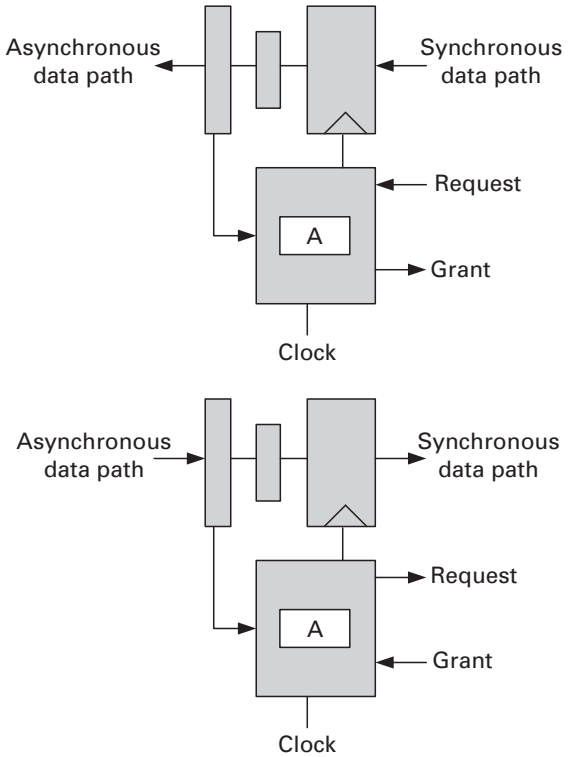

Figure 14.5. S2A and A2S converters (upper and lower diagrams).

To guarantee that the asynchronous datapaths are ready, a completion-detection circuit is needed to check that either all bits of an asynchronous input have arrived or that there is enough space in the asynchronous output channel. Instead of trying to complete all bits in a single C-element tree, a pipeline completion circuit that can sustain a high frequency regardless of datapath width is used. The A2S completion sends a 1-of-1 channel to the control circuit to indicate that there is space in the output FIFO. A few extra tokens are initialized on this channel to match the amount of storage available between the flip-flop and the completion detector. Figure 14.5 illustrates the structure of the S2A and A2S converters.

By checking that all input bits have arrived or that all output bits have buffer space, the converters can safely tolerate an arbitrary amount of relative skew between bits introduced by the asynchronous interconnect.

14.2.3 Latency

The latency is measured from when an input channel becomes valid to when the output channel becomes valid, assuming that the link is initially empty. On the synchronous side, the measurement point is when the request becomes valid on a

rising edge. The S2A has very little latency since the asynchronous token is created directly from the rising clock through a small logic depth. On the A2S, there is a similar amount of asynchronous latency in completing the incoming asynchronous bits and making the control decision. However, the A2S also must wait for metastability resolution and may be forced to wait an extra cycle if it "just misses" the sample window.

14.2.4 Noise analysis

Asynchronous circuits are susceptible to noise glitches for a larger portion of their cycle than a synchronous design with flip-flops and combinational logic. Although this design never glitches, analog noise from charge sharing or capacitive coupling can cause glitches. Dynamic logic susceptible to charge sharing is always contained within a small leaf cell and can be checked locally. Various design guidelines and netlist (connectivity) transformations are used to fix any charge-sharing problems without resorting to precharging the internal nodes. Capacitive coupling tends to affect longer wires, but these are strongly driven by inverters and also have inverters or pipeline repeater spaces at reasonable distances. The frequent use of 1-of-4 encoding means that usually only one of a group of four wires could be an aggressor.

14.2.5 Characterization results

The Nexus interconnect has been fabricated and characterized in numerous processes with minor design variations. It has been found to be functionally correct and to give a high performance in all cases. In TSMC 180 nm G, all Nexus components operate up to 450 MHz at 1.8 V and 25 °C. In TSMC 150 nm G, they operate at 480 MHz at 1.5 V and 25 °C. In the TSMC 130 nm low-K process (where K is the dielectric constant) they operate at 1.35 GHz at 1.2 V and 25 °C.

References

[1] A. Gravel, "Taking GALS to 65-nm designs," *EE-Times*, 10/11/2004.
[2] H. Kapadia and M. Horowitz, "Using partitioning to help convergence in the standard-cell design automation method," in *Proc. Design Automation Conf.*, June 1999, pp. 592–597.
[3] M. T. Bohr, "Silicon trends and limits for advanced microprocessors," *Commun. ACM*, vol. 41, no. 3, pp. 80–87, March 1998.
[4] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–1076, September 2001.
[5] M. Singh and M. Theobald, "Generalized latency-insensitive systems for single-clock and multi-clock architectures," in *Proc. Conf. on Design, Automation and Test in Europe (DATE)*, February 2004.

[6] Fujitsu Ltd, Fujitsu Laboratories Ltd, and Hitachi Ltd, "Component wrapper language," http://www.labs.fujitsu.com/en/techinfo/cwl/index.htm.
[7] D. M. Chapiro, "Globally-asynchronous locally-synchronous systems," Ph.D. dissertation, Stanford University, October 1984.
[8] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Latency insensitive protocols," in *Proc. 11th Int. Conf. on Computer Aided Verification*, 1999, pp. 123–133.
[9] K. Y. Yun and R. P. Donohue, "Pausible clocking: a first step toward heterogeneous systems," in *Proc. Int. Conf. on Computer Design (ICCD)*, October 1996.
[10] A. Chakraborty and M. R. Greenstreet, "Efficient self-timed interfaces for crossing clock domains," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, IEEE Computer Society Press, May 2003, pp. 78–88.
[11] A. Davis and S. M. Nowick, "An introduction to asynchronous circuit design," Dept of Computer Science, University of Utah, Technical Report UUCS-97–013, September 1997.
[12] R. Ginosar, "Fourteen ways to fool your synchronizer," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, IEEE Computer Society Press, May 2003, pp. 89–96.
[13] J. Muttersbach, T. Villiger, and W. Fichtner, "Practical design of globally-asynchronous locally-synchronous systems," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, April 2000, pp. 52–59.
[14] P. Glaskowski, "Pentium 4 (partially) previewed," *Microprocessor Rep.*, vol. 14, no. 8, pp. 10–13, August 2000.
[15] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Latency insensitive protocols," in *Proc. 11th Int. Conf. on Computer-Aided Verification*, N. Halbwachs and D. Peled, eds., vol. 1633, Springer-Verlag, 1999. pp. 123–133.
[16] A. Lines, "Nexus: an asynchronous crossbar interconnect for synchronous system-on-chip designs," in *Proc. Hot Chips Conf.*, Stanford, 2003.

15 Design example: the Fano algorithm

In this chapter we present the Fano algorithm, a convolutional code decoder, and its efficient semi-custom synchronous implementation [5]. The algorithm is used in communication systems to decode the symbols received over a noisy communication channel. We also present an efficient asynchronous counterpart, which we will use to explore the challenges in designing asynchronous chips. Figure 15.1 gives a schematic view of a communication system.

15.1 The Fano algorithm

In communication systems a convolutional code is a type of error correcting code in which an m-bit symbol is decoded into an n-bit symbol, where m/n is the *code rate* ($n \geq m$), and the decoding process is a function of the last k information symbols, where k is the constraint length of the code. The redundancy often allows convolutional codes to improve the performance of communication systems such as digital radios, mobile phones, satellite links, and Bluetooth implementations.

15.1.1 Background of the algorithm

The Fano algorithm [1]–[3] is a tree-search algorithm that achieves good performance with low average complexity at a sufficiently high signal-to-noise ratio (SNR). A tree comprises nodes and branches and associated with each branch is a *branch metric* (or weight, or cost). A path is a sequence of nodes connected by branches and the path metric is obtained as the sum of the corresponding branch metrics. An optimal tree-search algorithm determines the complete path (i.e. from root to leaf) with the minimum path metric, while a good but suboptimal tree-search algorithm finds a path with a metric close to this minimum.

The Fano algorithm searches through a tree sequentially, always moving from one node to a neighboring node until a leaf node is reached. It is a depth-first tree-search algorithm [1], meaning that it searches as few paths as possible to obtain a good path. Thus, the metric of a path being considered is compared against a threshold T. The relation between T and the metric is determined by the statistics of the branch metrics (i.e. the underlying model) and the results of partial path

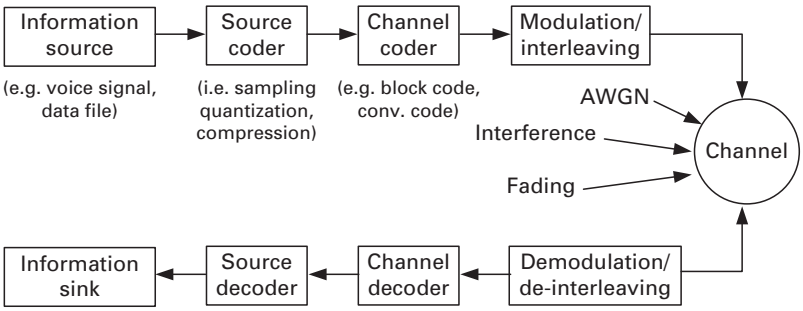

Figure 15.1. Communication system overview.

exploration. The latter is reflected by dynamically adjusting the threshold to minimize the number of paths explored.

The key steps of the algorithm are the decisions about which way to move (i.e. forward, or deeper, into the tree, or backward) and adjustment of the threshold. Intuitively, the algorithm should move forward only when the partial path to a node has a path weight greater than T. If no forward branches satisfy this threshold condition, the algorithm backtracks and searches for other partial paths that satisfy the threshold test. If all such partial paths are exhausted, it will *loosen* the threshold and continue. Morever, if the current partial-path metric is significantly above the threshold then it may *tighten* the threshold. Threshold tightening prevents continual backtracking to the root node (tracebacks) at the potential cost of missing the optimal path. Moreover, a maximum *traceback depth limit* is often imposed to limit the worst-case complexity. The details of the Fano algorithm are illustrated in the flow chart depicted in Figure 15.2; a more detailed explanation can be found in [2] [3].

The decoding of a convolutional code with known channel parameters can be viewed as a tree-search problem whose optimal solution is provided by the Viterbi algorithm [2], a breadth-first fixed-complexity algorithm. The Fano algorithm is known to perform near-optimal decoding of convolutional codes but with significantly lower average complexity than the Viterbi algorithm.

Figure 15.3 illustrates a search path with no traceback, common in systems with a very high SNR, where the data sent is identical to the data received and the sequential decoding algorithm can find an optimal solution very efficiently.

Figure 15.4 illustrates a search path with a minor traceback. The search algorithm goes deep into the search tree for a couple of branches, but eventually backtracks and finds the correct solution.

Figure 15.5 illustrates a search path with major tracebacks. The search algorithm wastes considerable effort by investigating many incorrect paths.

15.1.2 The synchronous design

This subsection describes the efficient normalization scheme used to optimize the algorithm, the architecture at the register transfer level, and the statistics of the chip.

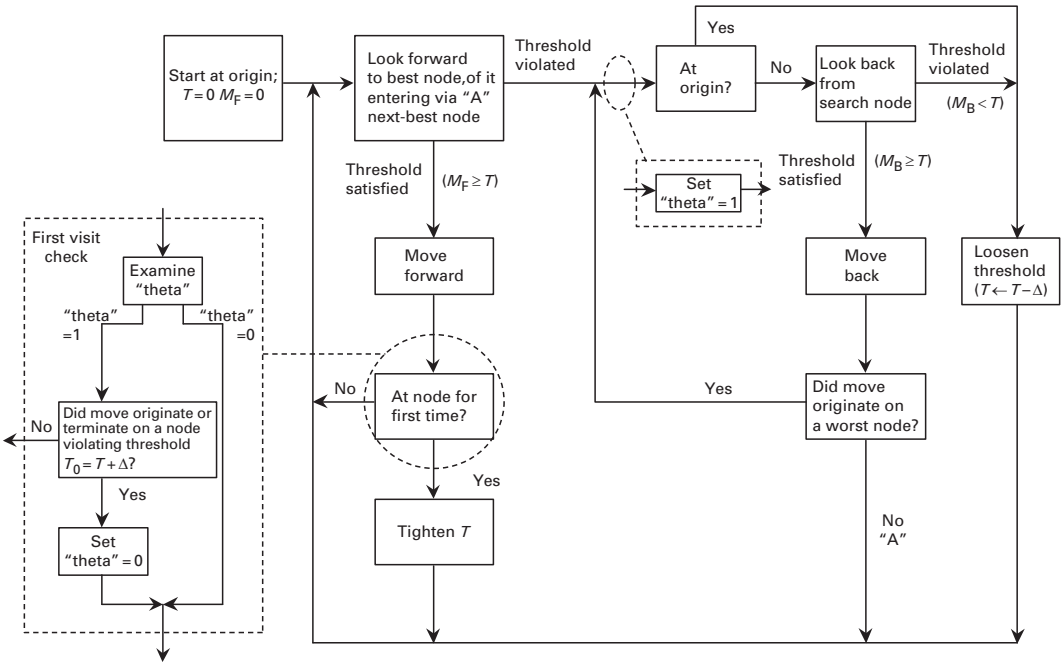

Figure 15.2. Flow-chart of the Fano algorithm: θ is a state variable, A labels the net, $M_$F and $M_$B are the actions "move forward," "move backward."

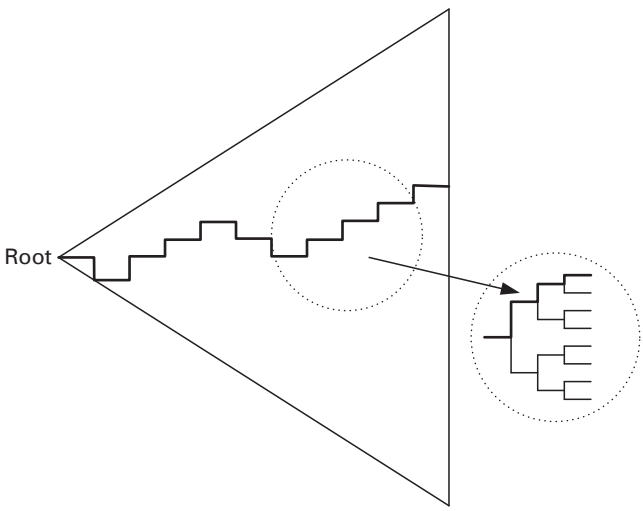

Figure 15.3. A path with no traceback in the search tree (represented by the triangle).

Normalization and its benefits

The basic idea behind normalization is to change the point of reference (e.g. from the origin of the tree to a current node under consideration). Normalization is often necessary to prevent hardware overflow or underflow. It is interesting that in

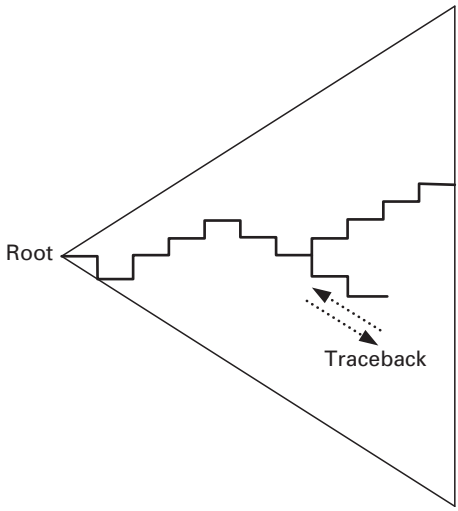

Figure 15.4. A path with a minor traceback.

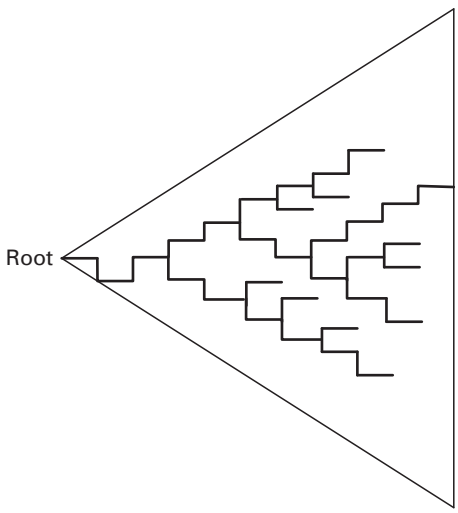

Figure 15.5. A search path with major tracebacks.

traditional communication algorithms such as the Viterbi algorithm, normalization often yields significant performance and area overheads, which hardware designers generally avoid by using slightly larger bit widths and modulo arithmetic [4]. In contrast, normalization in the Fano algorithm can yield a smaller, faster, and more energy efficient design.

In particular, normalization is applied to the variables in such a way as to make the current node's metric always equal to zero. This is equivalent to subtracting it from every variable in the algorithm and does not change the overall behavior of the algorithm. The advantages of this type of normalization in the Fano algorithm

are as follows. (i) Additions involving the current metric (i.e. during the threshold check) are removed and comparisons with the current metric (i.e. during the first visit check and the threshold-tightening steps) are reduced to a one-bit sign check. (ii) The normalization of the next threshold (subtracting the current node's metric from it) can be done by the arithmetic logic unit (ALU) that compares the threshold with the next metric and thus consumes negligible additional energy. (iii) Lastly, normalization enables us to work with numbers with smaller magnitudes, which can be represented with fewer bits.

Register-transfer-level design

The register-transfer-level architecture is illustrated in Figure 15.6(a). The threshold adjust unit (TAU) is shown in more detail in Figure 15.6(b). At each clock cycle, the best and next-best branch metrics are calculated using data stored in memory. (See [3] for more details regarding the branch metric computation.) The threshold adjust unit compares the error metric with the current threshold to determine whether a forward move can be performed and simultaneously calculates speculatively two normalized next thresholds; the first is applicable if now a forward move is taken and the second is applicable if the threshold is to be loosened (the new threshold is found by the subtraction of Δ from T).

On the basis of the above results, either the move will be made and the precomputed threshold will be stored or the threshold T will be loosened, all in one clock cycle. Additional clock cycles are needed to compute a tightening of the threshold if (i) a forward move is made, (ii) the first visit check is passed, and (iii) the precomputed tightened threshold is not in the range Δ. Fortunately, with reasonable choices of Δ, computer simulations suggest that these additional cycles of tightening are rarely needed. Similar speculative execution allows us to perform a look and move back in one clock cycle.

The register-transfer-level architecture shown in Figure 15.6 is controlled by the finite-state machine (FSM) illustrated in Figure 15.7. Three states, S2–S4, make up the main algorithm. In each state the branch metric unit computes the selected branch metric using data stored in the sequence memory. Depending on the control bits from the FSM (not shown), the selected branch metric is associated with the best or worst branch. In either case the corresponding input bit is sent to the decision memory where, if that branch is taken, it is used to update the selected path.

In state S2, the machine looks forward, moves forward if possible, and, if necessary, performs one step of threshold tightening. More specifically, after the selected branch metric is computed, the FSM performs a threshold check to see whether the machine can move forward. That is, in Figure 15.6(b) ALU 3 computes T minus the selected branch metric and the FSM examines the most significant bit. If the sign bit is a 1, the branch metric is no smaller than T and the threshold check passes. Otherwise, the threshold check fails. Meanwhile, ALU 1 and 2 speculatively compute $T + \Delta$ and $T + \Delta$ minus the selected branch metric, respectively. These values, along with a state variable θ, shown in Figure 15.2, allow the FSM to determine whether the first visit check passes. That is, the first

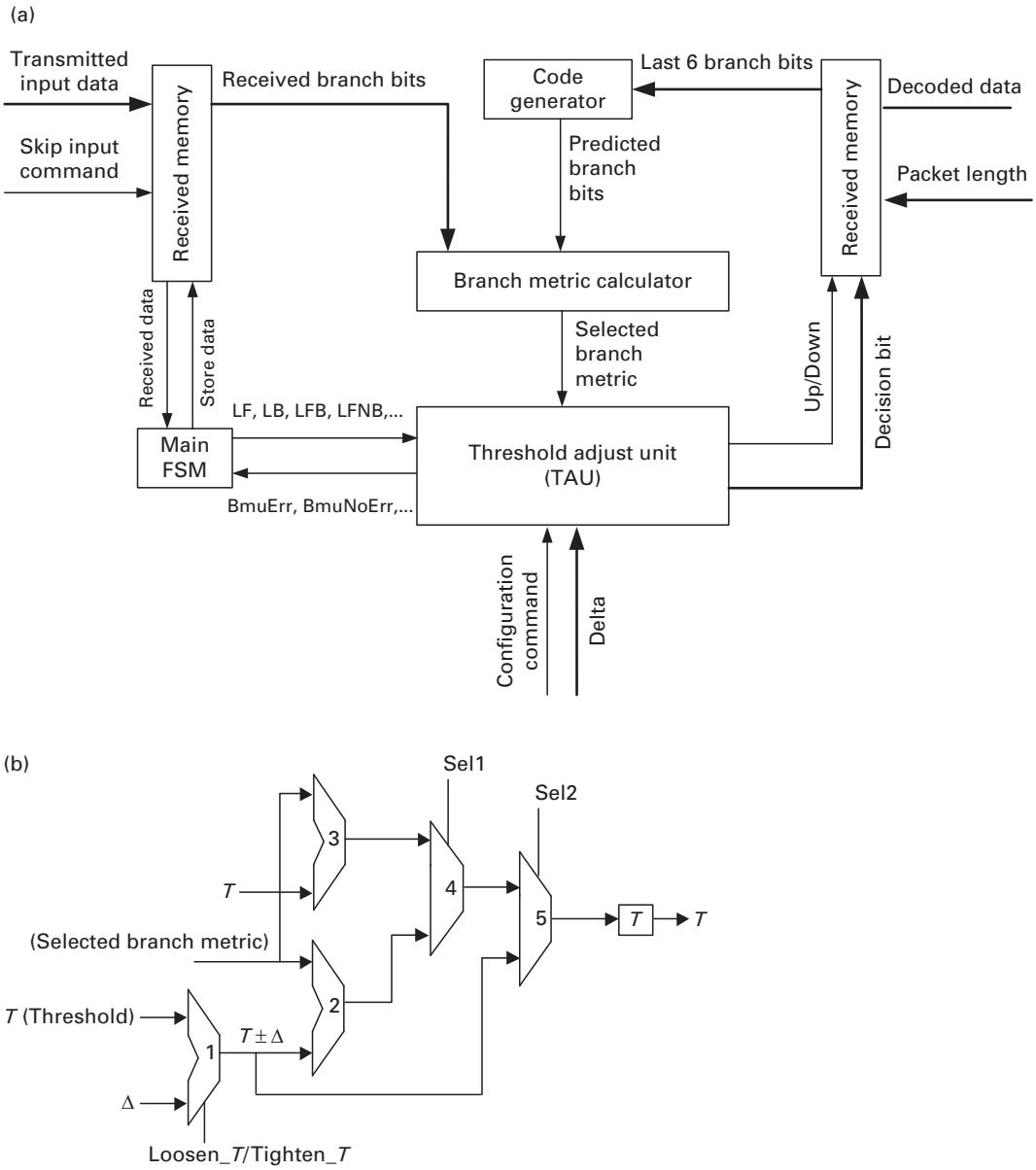

Figure 15.6. (a) Register-transfer-level architecture of the synchronous Fano algorithm. (b) The threshold adjust unit (TAU), which incorporates three arithmetic logic units, ALU 1–3.

visit check passes if and only if $\theta = 0$ or $T+\Delta$ is positive or if $T+\Delta$ minus the selected branch metric is positive.

On the basis of the above results, the FSM acts in one of three ways. (i) The threshold check passes and a forward move is performed but the first visit check fails, so that NextState is set to state S2, in preparation for another look forward. (ii) Both the threshold check and the first visit check pass, in which case the FSM

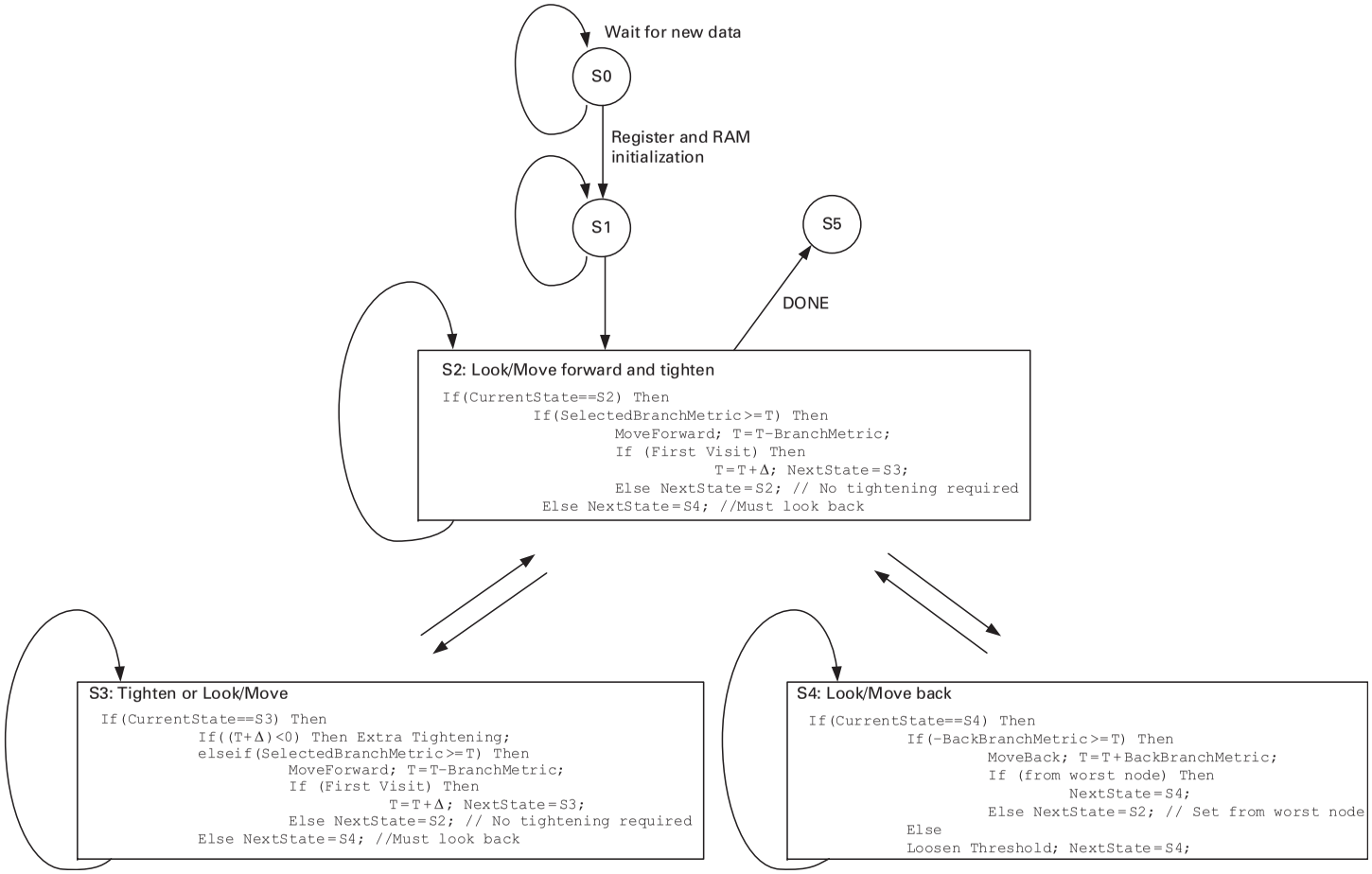

Figure 15.7. Finite-state machine describing the RTL architecture.

moves to state S3. (iii) The threshold check fails and the FSM moves to state S4 in preparation for a look and/or move backward. In case (i) the threshold register is updated with T minus the selected branch metric, as computed by ALU 3. In case (ii), however, the threshold is updated with the tighter threshold $T+\Delta$ computed by module 1. In case (iii) the threshold register remains unchanged.

In state S3, the FSM checks whether a subsequent tightening is needed (by computing and checking the sign of $\Delta+T$). Simultaneously, it speculatively performs a threshold check (by checking that the branch metric is no smaller than T); this is required if the threshold need not be immediately tightened. If tightening is required then NextState is set to state S3. For the case where no immediate tightening is needed, the FSM performs the same move, look forward, tightening, and next-state operations as in state S2.

State S3 is entered when the threshold check fails in either state S2 or state S3. In state S4, a look backward is performed and, if possible, a backward move is made and the threshold is updated with the re-normalized threshold. Both the look backward and the re-normalization are performed through ALU 3 by adding T and the selected (backward) branch metric. Specifically, the look-backward check is satisfied if and only if the negative of the selected branch metric is greater than or equal to the threshold, i.e. if the result of the ALU 3 operation is negative and the re-normalized threshold is precisely the output of ALU 3. If a backward move is performed and it is originated from a worst node, via an additional FSM flag, NextState is set to state S4, in preparation for another look backward. Alternatively, NextState is set to state S2 in preparation for a look forward to the next-best node; this is controlled by a LookNextBest flag not shown in order to simplify Figure 15.7. If the backward look fails, however, the threshold is updated with a loosened threshold value speculatively computed by module 1, and NextState is set to state S2.

The key feature of this speculative control strategy is that each forward move typically takes only one clock cycle, with negligible performance overhead associated with the first visit check or tightening. In particular, with reasonable choices of Δ, computer simulations suggest that additional cycles of tightening are rarely needed.

Synchronous chip implementation

The chip that was designed all the way to gds (the file format created by the designer for the manufacturer) supports a packet length $N = 128$. The depth of the search tree including seven tail bits, is thus 135. It supports a rate $-1/2$ convolutional code (i.e., $n = 2$) with generator polynomials $1+\mathrm{D}+\mathrm{D2}+\mathrm{D5}+\mathrm{D7}$ and $1+\mathrm{D3}+\mathrm{D4}+\mathrm{D5}+\mathrm{D6}+\mathrm{D7}$ (here, for example, D5 means the fifth bit from the last). For this prototype the chip has fixed branch metrics $B(0) = 2$, $B(1) = -7$, and $B(2) = -16$, requiring five bits for their representation. These metrics are ideal for an SNR range $1 < E_b/N_0(\mathrm{dB}) < 3$. In practice, they would be dynamically

adjusted when the estimated channel SNR is outside this range; this may require an extra bit.

Automatic placement and routing tools were used with a combination of synthesized and manually laid-out components in the 0.5μm HP14B CMOS process. The layout has area 1.2 mm × 1.8 mm. Powermill was used to estimate the performance of the design. At 1.5 V power supply the design successfully operated at 15 MHz, and at 3.3 V it successfully operated at 100 MHz.

15.2 The asynchronous Fano algorithm

Deeper analysis of the Fano algorithm shows that the operation of the algorithm can be divided into two regions, the *error-free region* and the *error region*. In the error-free region the algorithm moves forward, while the received bits from the sender are error free and match the expected bits. In this region of operation the un-normalized threshold is incremented with a constant value, namely the value given for an error-free branch of the tree. If the threshold value is known at the time at which the algorithm enters the error-free region then the next value of the threshold can be calculated. The normalized threshold, however, stays in the range $-\Delta \leq T \leq 0$ and rotates through a finite number of values in a predetermined order.

Consequently, instead of calculating the threshold values explicitly, a pointer to a lookup table containing these predetermined values is incremented. When an error is encountered, the design enters the error region; the current value of the threshold is accessed from the lookup table and the full algorithm is applied in order to determine whether to move forward, move backward, or loosen the threshold. The algorithm stays in the error region until a node in the search tree is reached for the first time and the move by which the node was reached was a forward move. At this point the algorithm moves back into the error-free region. The algorithm continues until the end of the tree by alternating between the error-free and the error regions.

For high-SNR applications, most received packets have little or no errors and therefore the decoding process consists mostly of reading the data from the memory, comparing it with the predicted data, and writing the decision to the memory. The process involves few or none of the multi-bit additions, subtractions, or comparisons associated with loosening or tightening the threshold. This fact motivates a two-block architecture that is specifically designed to handle the two different operating regions of the algorithm efficiently.

15.2.1 The asynchronous Fano architecture

The asynchronous architecture shown in Figure 15.8 localizes the error-free region in a small block that is highly optimized. In particular, the branch metric unit (BMU) is partitioned into a skip-ahead unit optimized for the error-free

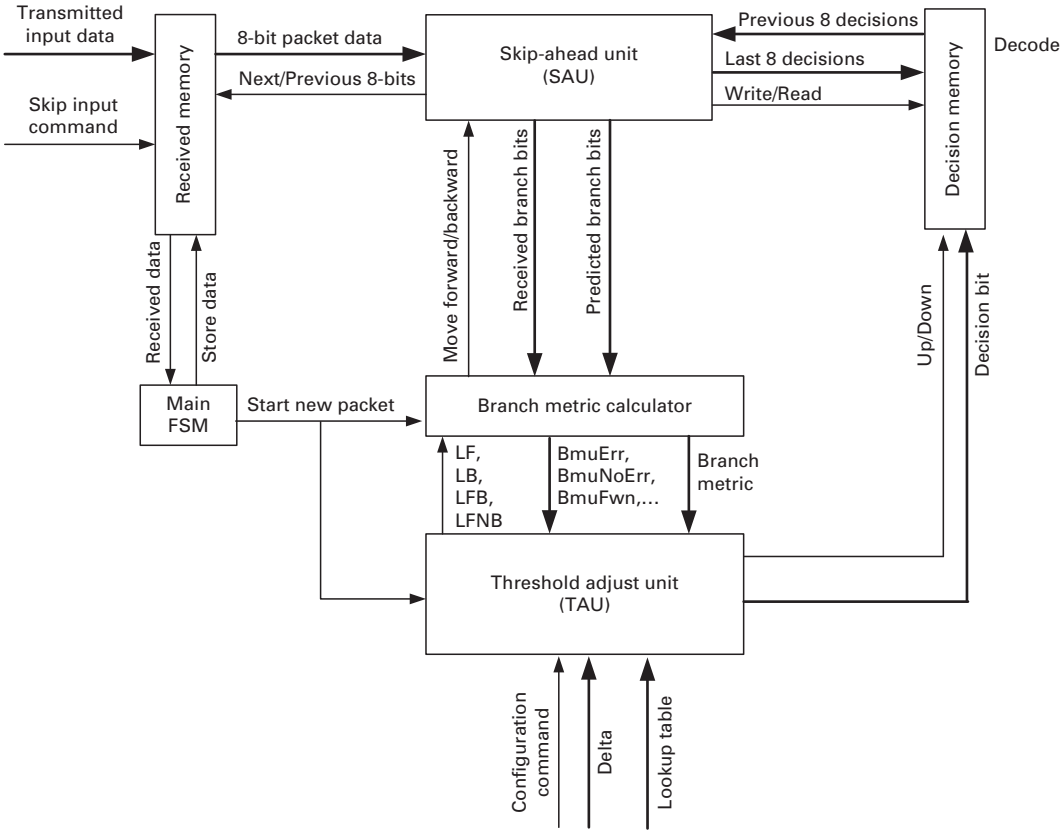

Figure 15.8. RTL architecture of the asynchronous implementation. The SAU, branch metric calculator, and TAU comprise the BMU.

region and a threshold adjust unit and branch metric calculator that are active only in the error region and have implementations analogous to the synchronous version.

The data received at the decoder via the transmitted input data channel is stored in the received memory. The fast skip-ahead unit requests data from the received memory in eight-word chunks via the Next/Previous channel, where each data word is two bits wide for the (7, 1, 2) code. As the skip-ahead unit decodes the code and moves forward in the tree, it stores its decisions locally. Every eight decisions are sent to the decision memory via the Last-eight-decisions channel. When an error is encountered, the skip-ahead unit may need to go backward in the tree to explore different branches, by requesting previous decisions from the decision memory that arrive on the Previous-eight-decisions channel. The data flow between the decision memory and the skip-ahead unit is controlled via the Write/Read channel.

In the error-free region, the received bits are read from the received memory and decoded in the skip-ahead unit. The resulting decisions are then sent to the

decision memory. In this region, the main FSM, branch metric calculator, and TAU are inactive.

When an error is encountered, the SAU informs the branch metric calculator via the error channel and also sends to the branch metric calculator the received branch bits and the predicted branch bits calculated using the previous decisions and the convolutional code. Depending on the move commanded by the threshold adjust unit via the look-forward best (LFB), look-backward (LB), look-forward next-best (LFNB), and look-forward-best-until-error (LFBTE) channels, the branch metric calculator calculates and compares the branches, selects the appropriate one, and sends it to the TAU with additional information notifying whether the move originated from a worse branch and whether the branch had any errors (via the additional BmuErr and BmuFwn channels). Every time the TAU is accessed for the first time after an error has occurred, the TAU reads the normalized threshold from the look-up table and updates the threshold value. The TAU is implemented in an analogous way to the synchronous version and may move forward, move backward, or adjust the T threshold. Upon deciding a move, the relevant information is sent to the SAU and a new command is issued to the branch metric calculator. Finally when a new error-free node is reached for the first time, the TAU stores the normalized threshold, updates the pointer to the look-up table, and resumes operation in the fast SAU. The operation switches back and forth between the SAU and the TAU until all the data is encoded. Upon reaching the end of the tree, the decision memory sends out the decoded data.

The fact that the asynchronous circuit has no global clock allows the asynchronous architecture to be naturally divided into two blocks, each operating at its ideal speed, that communicate, only when and where this is needed, via the interblock asynchronous channels.

15.2.2 The skip-ahead unit

A high-level implementation of the SAU is shown in Figure 15.9. The core of the SAU is the error detector, which compares the predicted branch bits with the received branch bits and stores the decision. To operate at full rate, the memories must keep up with writing and reading one data word per decoding cycle. As the memory capacity increases, this becomes a difficult task and for this reason we opted to use shift registers to act as caches for the bigger memories. In particular, the fast data register stores eight words from the received memory and the fast decision register acts as an eight-word read and write cache for the decision memory. When the received memory sends an eight-word packet to the fast data register, the received memory speculates that the SAU will not encounter any errors and moves forward, thus preparing to send a new set of data. This cache structure allows the larger memory to run at one-eighth the speed of the SAU. The same motivation applies to the use of the fast decision register, with the difference that it is a read and write register. Both registers have an associated controller to request and send data to their respective memories.

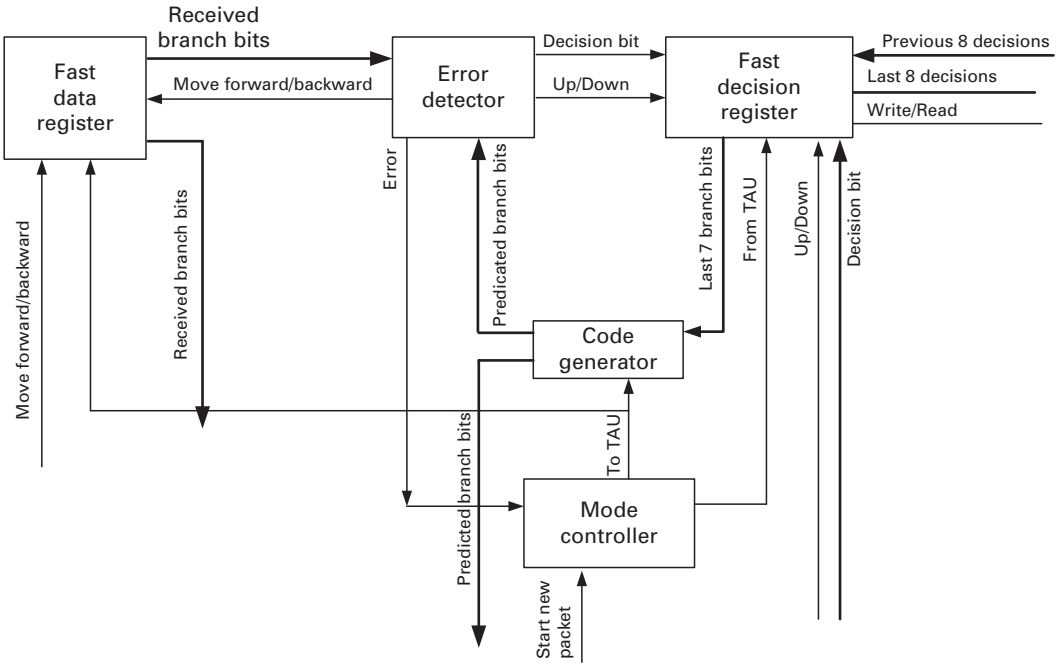

Figure 15.9. Detailed implementation of the skip-ahead unit.

The most recent decisions in the search tree, which always reside in the fast decision register, are sent to the code generator, which predicts the values of the new branch bits. The predicted values of the branch bits are compared with the received values in the error detector. If there is a match, indicating no error, the decision is stored in the fast decision register; then an internal counter and the pointer in the look-up table are incremented and new data is requested from the shift registers via the Move forward/backward and Up/Down channels. If there is no match then an error is encountered. The predicted and received branch bits are sent to the branch metric calculator and the controls of the shift registers are transferred to the TAU.

The critical loop in the error-free region consists of the fast shift register, the error detector, the fast decision register, and the code generator. For high-SNR operation, most of the time the decoder operates in the error-free region; therefore the goal must be to achieve high speed in this region by optimizing the circuit. However, if the circuit encounters an error then it enters the error region, and the critical path consists of the fast shift register and the convolutional code generator serving data to the slow BMU. In the error region the operation is the same as in the synchronous version, consisting of a number of sequential operations. In this region the speed is expected to be comparable with that in the synchronous case.

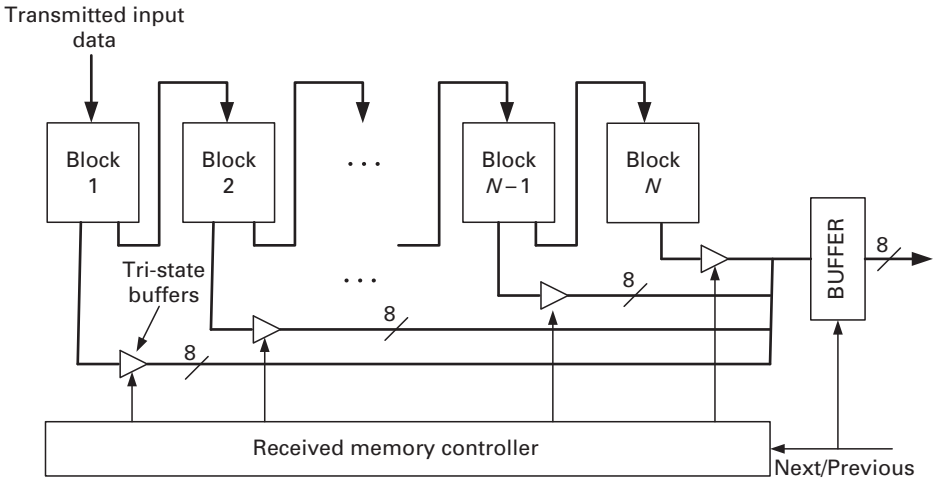

Figure 15.10. Implementation of the received memory.

15.2.3 The memory design

Since the chip supports a packet length of only 135 bits (128 data and seven tail bits), we opted to design the main data memory blocks of the received and decision memories using standard PCHB templates. However, we introduced unacknowledged tri-state buffers on the data bus as an efficient way to allow multiple drivers of the bus. This is typical in synchronous design but does introduce some minor timing assumptions not typical of PCHB-based designs. We also used standard place and route tools for the physical design of the memories for faster design time at the cost of more area and power consumption.

In particular, as depicted in Figure 15.10 the received memory consists of N blocks, where each block can hold eight words. For the (7, 1, 2) convolutional code each word is two bits. The blocks are FIFOs implemented with PCHBs. At any time only one tri-state buffer is enabled, allowing only one block to send its data. The Fano algorithm is a sequential tree-search algorithm, therefore SAU accesses the memory sequentially via the Next/Previous channel. The received memory controller responds to the request by enabling a preceding or succeeding tri-state buffer and sending new data. The buffer captures the new data and sends it to the requesting unit. The timing assumption for correct operation is that the delay from the Next/Previous channel through the received memory controller and the selected tri-state buffer should be less than the delay from the Next/Previous channel to the output buffer. Moreover, the output buffer should only latch its input when the enabled tri-state buffers outputs have changed and stabilized.

The decision memory has a similar structure; however, since it is a read and write memory each block can be accessed individually to read from or to write to.

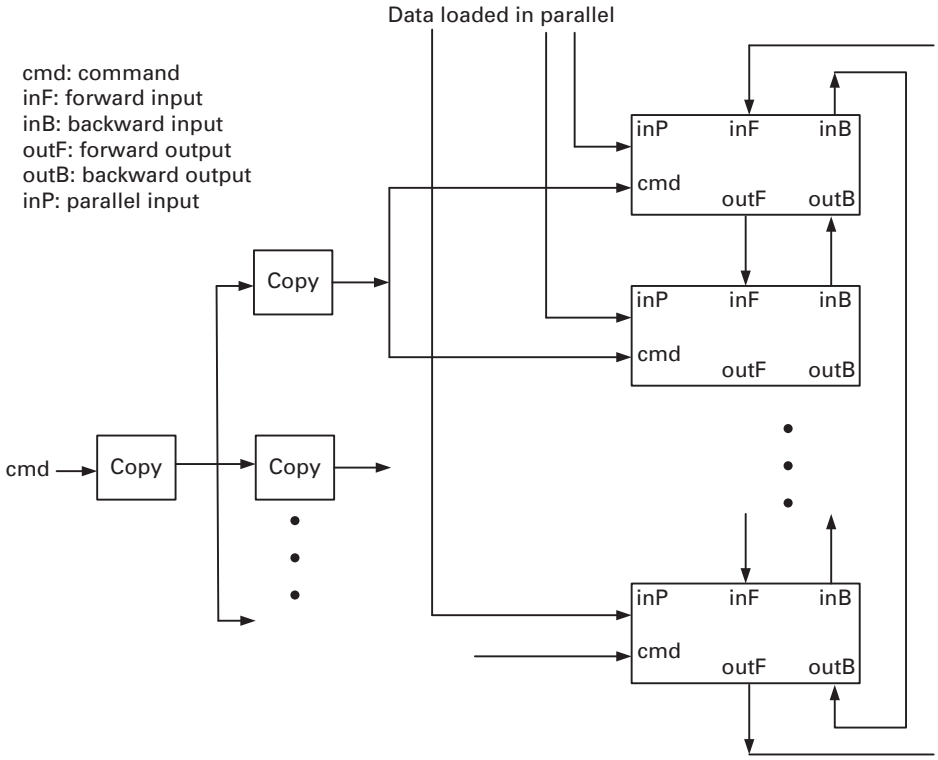

Figure 15.11. Implementation of a one-bit fast shift register.

15.2.4 The fast data and decision registers

The fast data register is implemented using two eight-word one-bit shift registers, as shown in Figure 15.11. The register consists of eight conditional-input conditional-output one-bit memory pipeline stages. Depending on the command (cmd) it either shifts forward by receiving new data from inF and sending the old to outF, shifts backward by receiving data from inB and sending the old to outB, or loads eight words in parallel from the main memory. The parallel-load command overwrites the old data tokens inside each stage. The command channel cmd should go to all the stages; however, to prevent the use of a large C-tree to generate the cmd acknowledge signal the cmd signal is broadcast with a tree of copy buffers. Altough this solution reduces the load on the cmd channel, if it were to be copied to all stages directly then it would increase the critical loop delay of the algorithm.

The fast data register is implemented similarly.

15.2.5 Simulation results and comparison

The core layout of the chip designed in TSMC 0.25μm CMOS technology is illustrated in Figure 15.12. Nanosim simulations on the extracted layout

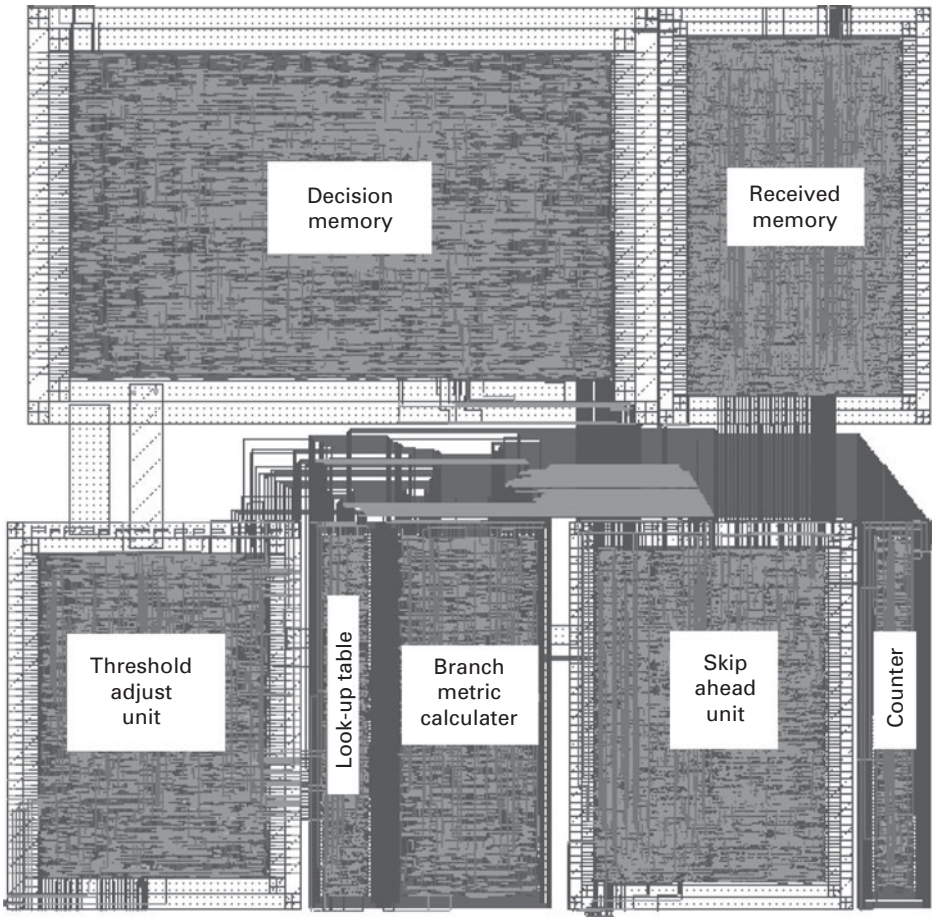

Figure 15.12. Layout of the asynchronous Fano architecture.

show that the circuit runs at 450 MHz, consumes 32 mW at 25 °C, and has area 2600 μm × 2600 μm = 6.76 mm^2. The asynchronous chip runs about 2.15 times faster than its synchronous counterpart. However, it occupies five times the area. This is partially due to the fact that both memories, which occupy half the chip area in the asynchronous chip, are implemented with PCHBs. Lastly, the design consumes one-third of the power of its synchronous counterpart.

Figure 15.13(a) below shows the post-layout simulation results for the circuit operating in the error-free region. Since the fast data and decision registers can only hold eight words, once the data held by the fast data register is consumed then a new set of data is requested from the main received memory. This request and data transfer causes a slight delay, which can be observed in the waveforms as the gaps that occur every eight pulses. Since there are no errors in the error-free

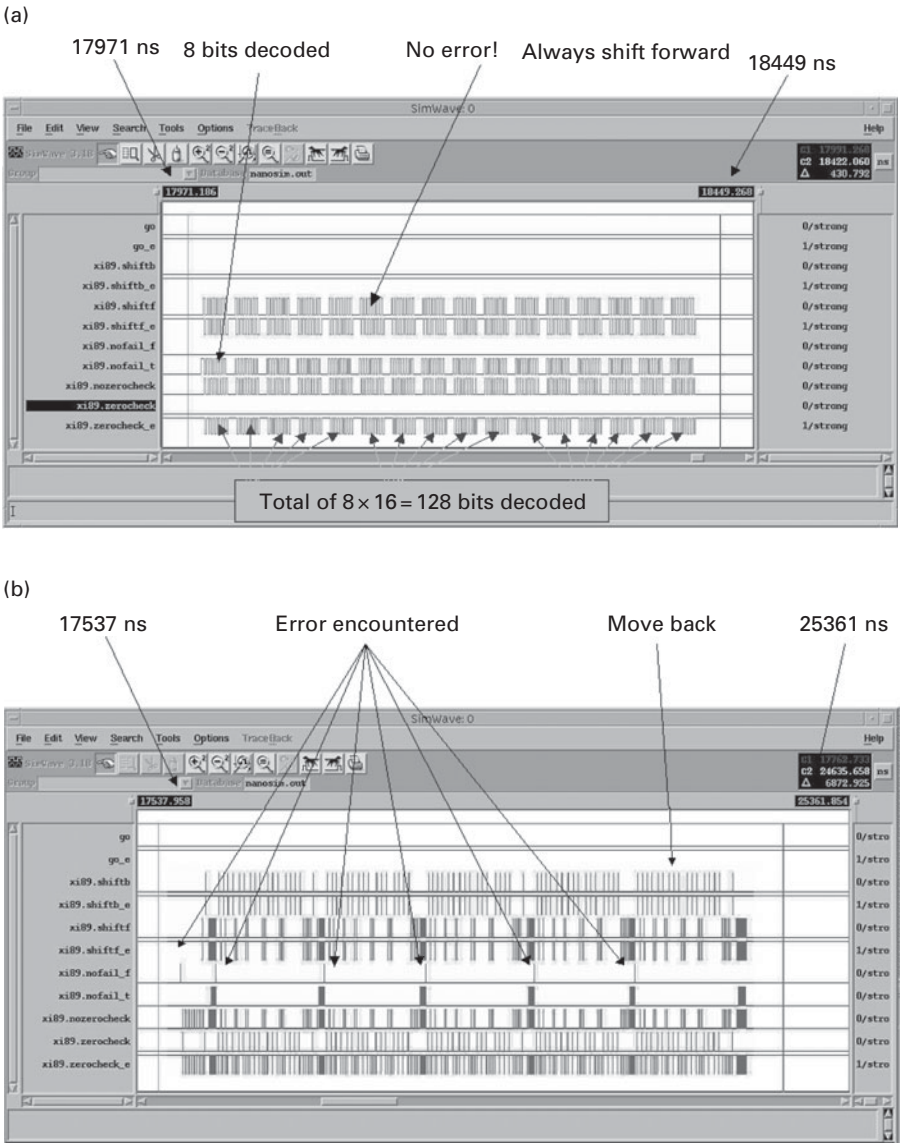

Figure 15.13. (a) error-free-region and (b) error-region operation waveforms.

region, the nofail_f signal used to indicate an error is never asserted by the detection logic but instead the nofail_t signal, which indicates that there are no errors, is asserted.

However, as shown in Figure 15.13(b), in the error region, as errors are encountered the decoder moves back and forth to find the correct path. This can be observed with the assertion of the shiftb (shiftback) and nofail_f signals.

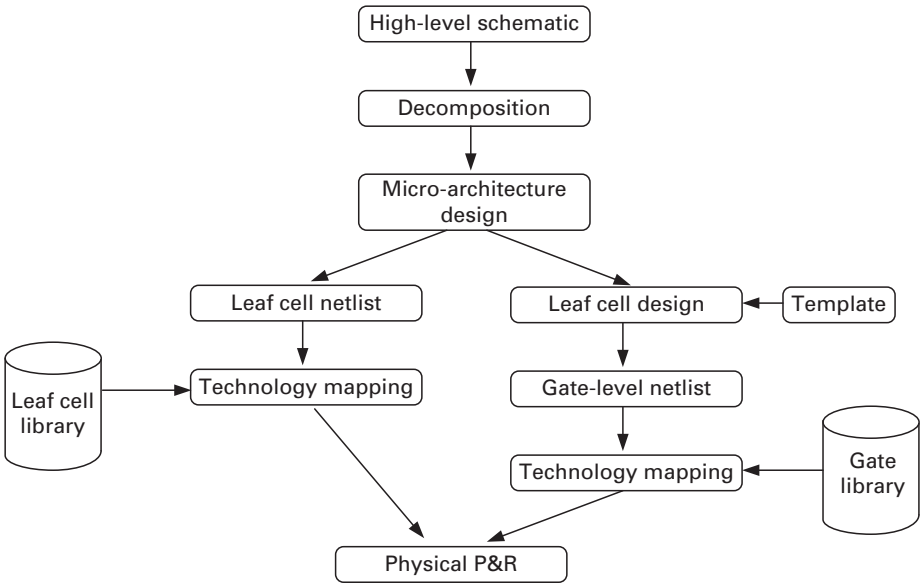

Figure 15.14. Asynchronous circuit design flow. The final step is the physical place and route operation.

15.3 An asynchronous semi-custom physical design flow

This section focuses on the gate-level and physical design aspects of asynchronous semi-custom physical design flow that the USC Asynchronous Design Group has refined and used in the implementation of the asynchronous Fano design.

The basic steps of the template-based semi-custom asynchronous physical design flow are illustrated in Figure 15.14.

A high-level schematic is developed based on the C and Verilog codes used to describe the Fano algorithm. This high-level schematic is hierarchically implemented by decomposing the design to the lowest-level communicating blocks, namely the PCHB leaf cells. In the micro-architecture step the designer can choose to implement the architecture with various methods, ranging from fine-grain pipelines that are template-based and use delay-insensitive cells to components relying on bounded-delay-based cells with no pipelining at all. The asynchronous Fano design has been implemented with fine-grain pipelining using PCHB templates. Slack optimization to improve performance is also completed in this step. At the end of the micro-architecture design there are two possible options.

One option is to pursue the decomposition and generate a leaf cell design. This will depend on the template used (PCHB, RSPCHB, LP3/1, LPHC, or STFB, etc.). The next step is to generate the gate-level netlist of the whole circuit as in synchronous design. The gate library, consisting of static and dynamic gates, will

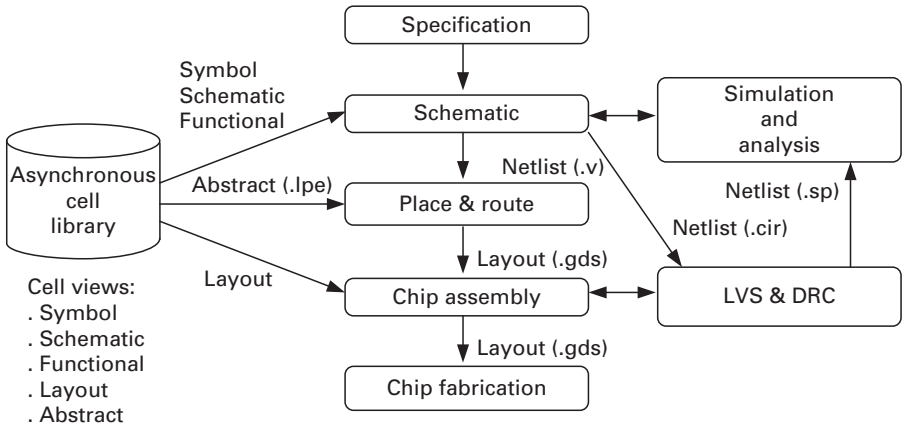

Figure 15.15. Physical design flow using standard CAD tools.

be mapped to the netlist and the design can be laid out using standard place and route tools.

The other option is to generate a leaf cell netlist, rather than going any further, and to use a leaf cell library. The leaf cell library would be mapped to the netlist, and the automatic place and route procedure would be done at the leaf cell level rather than at the gate level, which is lower. This option will probably yield denser circuits with a better performance since the leaf cells would be optimized and laid out using more of a full-custom approach, though even automatic place and route can be applied to generate the leaf cells. Choosing the first option and applying place and route directly to a gate netlist could lead to a number of undesired effects. One such effect is a less dense circuit since, rather than sharing area and optimizing leaf cells, the leaf cells will be implemented with discrete gates. Another issue is that the handshaking circuits might not be as close to the dynamic functional evaluation circuit when place and route is applied to the gate netlist rather than the leaf cell netlist, thereby affecting performance.

15.3.1 Physical design flow using standard CAD tools

One of the biggest obstacles today of designing asynchronous circuits is the lack of CAD tools specifically targeted for the design of such chips. However, it is still possible to complete a fairly complex chip in a reasonable amount of time using the standard CAD tools used for synchronous design. Figure 15.15 illustrates the flow.

There is no difference in the initial specification step of the design for the synchronous and asynchronous cases, since a specification typically describes the expected functionality (Boolean operations) of the designed block, as well as the delay times, the silicon area, and other properties such as power dissipation. Usually the design specifications allow considerable freedom to the circuit

```
module Dynamic_BUFFER_Function (nBUF0, nBUF1, A0, A1, BUFe, en, BUF1, BUF0);
     output nBUF0;
     output nBUF1;
     output BUF0;
     output BUF1;

     input A0;
     input A1;
     input BUFe;
     input en;

     reg nBUF1, nBUF0, BUF1, BUF0, temp;

     initial begin temp=0; end

     parameter D1=10; //unit delay 1
     parameter D2=20;

     always @(BUFe or A0 or A1 or en)
     begin
          if(BUFe==1 && en==1 && temp==0)
          begin
               if(A1==1 && A0==0) begin nBUF1 <= #D1 0; nBUF0 <= #D1 1;
                                        BUF1 <= #D2 1; BUF0 <= #D2 0; temp=1; end
               else if(A1==0 && A0==1) begin nBUF1 <= #D1 1; nBUF0 <= #D1 0;
                                             BUF1 <= #D2 0; BUF0 <= #D2 1; temp=1; end
          end

          else if(BUFe==0 && en==0)
          begin
               nBUF1 <= #D1 1; nBUF0 <= #D1 1; BUF1 <= #D2 0; BUF0 <= #D2 0;
               temp=0;
          end
     end
```

Figure 15.16. The functional description of a dynamic buffer.

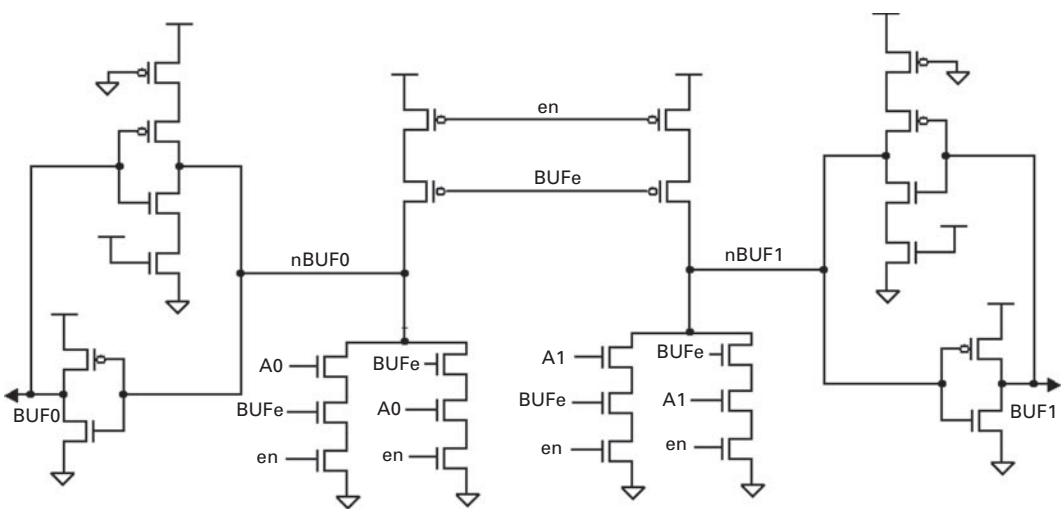

Figure 15.17. The transistor-level view of a dynamic buffer.

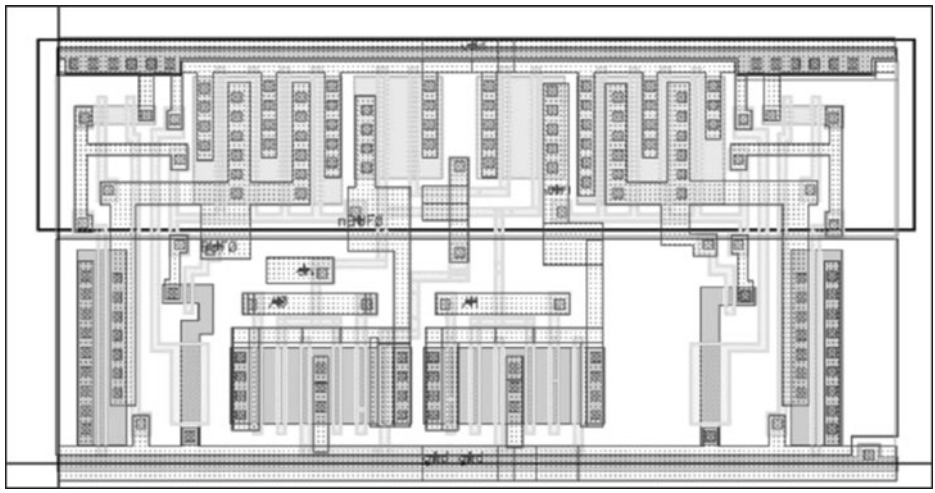

Figure 15.18. The layout of a dynamic buffer.

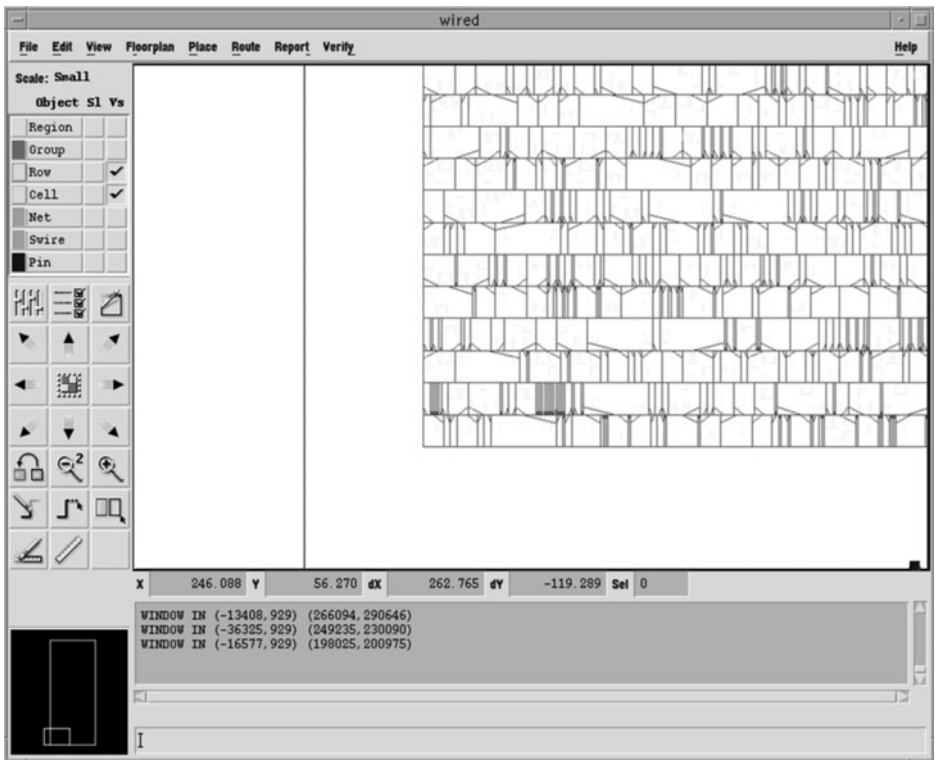

Figure 15.19. Cell placement in Silicon Ensemble.

designer with regard to the choice of a specific circuit topology, the individual placement of the devices, the locations of the input and output pins, and the overall aspect ratio (width-to-height ratio) of the final design.

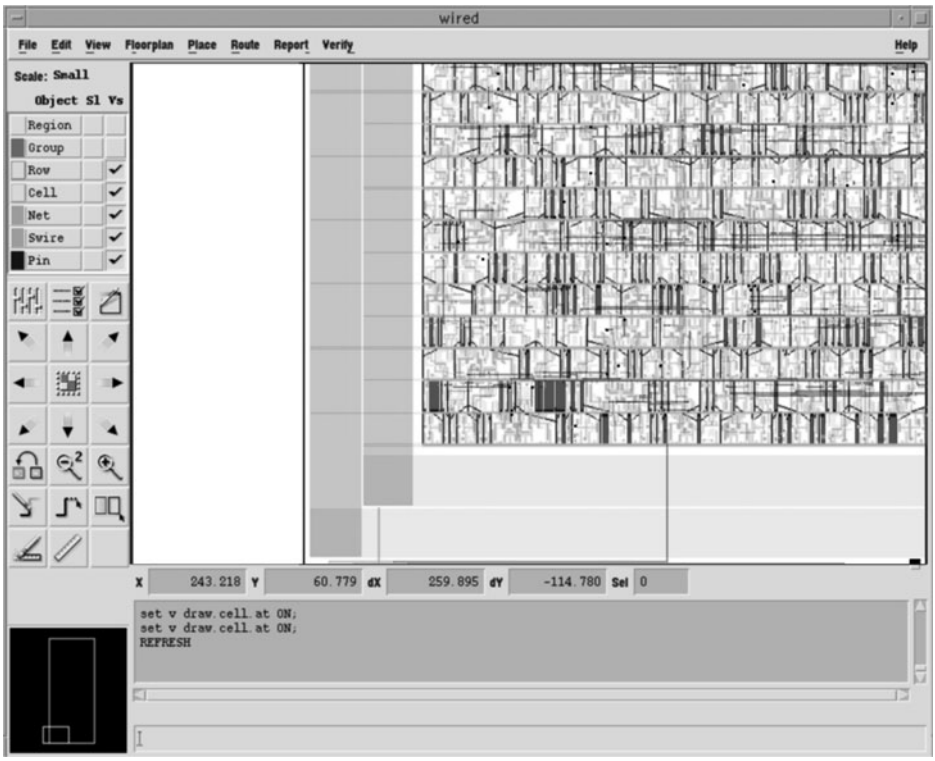

Figure 15.20. Routed counter block with Silicon Ensemble.

The actual implementation of an asynchronous circuit starts at the schematic level. The traditional method for capturing (i.e. describing) a transistor-level gate-level, or block-level design is via the schematic editor. The functional description of the dynamic logic circuit used as a buffer is shown in Figure 15.16. Note that there are a number of ways to describe this behavior.

Once the design is completed and its correctness has been verified at the behavioral level, the schematic (transistor) views of the cells are implemented for HSPICE simulation. A transistor-level view of the dynamic buffer is shown in Figure 15.17.

The layout view for the dynamic buffer is shown in Figure 15.18.

One important aspect of designing cells for dynamic logic is charge sharing and transistor sizing. After a number of test simulations on individual cells we decided to use 8× for the size of the output transistors and 2× for the pull-down transistors. The staticizer inverters were set to approximately one-tenth the strength of the pull-down transistors in order to balance reliability of operation against speed. The other aspect to be taken into account for reliable operation is charge sharing. Unlike in the schematic in Figure 15.17, if the nBUF1 and nBUF0 signals were generated using the A, en, and BUFe signals as a stack of

```
*
* CADENCE/LPE SPICE FILE: SPICE
*                  DATE : 5-JUN-2003
*
******    MOS XTOR     PARAMETERS  FROM : 7MOSXREF
*.GLOBAL    VDD!  GND!
.SUBCKT INC2   DATA  REQ  ACK  NRST4  LO  LI
*
******    CORNER ADJUSTMENT FACTOR = 0.0000000
******
MM2-XI60-XI36 XI36-A NET0432 VDD!  VDD!  PCH  L=0.24U  W=2.80U  AD=1.04P
+          PD=3.54U  AS=1.88P PS=6.94U NRS=0.079 NRD=0.079
MM3-XI60-XI36 XI36-A NR<6>  VDD!  VDD!  PCH  L=0.24U W=2.80U AD=1.04P
+          PD=3.54U  AS=1.88P PS=6.94U NRS=0.079 NRD=0.079
MM7-XI60-XI36  XI36-XI60-NET029  NET0432  XI36-A  GND! NCH  L=0.24U W=1.20U
+          AD=0.24P  PD=1.60U  AS=0.44P  PS=1.94U NRS=0.183  NRD=0.167
MM7-XI60-XI36-1  685  NET0432  GND!  GND!  NCH  L=0.24U  W=1.2OU  AD=0.24P
+          PD=1.60U AS=0.80P  PS=3.74U  NRS=0.183  NRD=0.167
...
MM1-XI59-3 NET72  XI59-NET35  VDD!  VDD!  PCH L=0.24U  W=2.50U  AD=0.93P
+          PD=3.24U AS=1.65P  PS=6.32U  NRS=0.088 NRD=0.088
*
*-----  TOTAL # OF MOS TRANSISTORS FOUND :    2018
*
*
C1    NET77  GND!  8.00421E-15
C2    NET209 GND!  1.06917E-14
C3    NET188 GND!  1.16892E-14
C4    NET121 GND!  1.34065E-14
C5    NET215 GND!  1.02445E-14
...
C583  XI36-XI56-NET016      GND!  6.91710E-17
C584  XI29-XI50-XI50-NET016 GND!  1.85150E-17
C585  XI30-XI50-XI50-NET016 GND!  4.84647E-17
*
*-----  TOTAL # OF CAPS FOUND :    585
*-----               COMMENTED :    2
*
.ENDS
```

Figure 15.21. Extracted netlist of a block.

three transistors in series then there the internal dynamic nodes nBUF1 and nBUF0 could lose their value owing to charge sharing. This scenario could occur if A and en were asserted high, thus turning on their respective transistors, and BUFe were still asserted low. To prevent this problem, we opted to use a widely known symmetrization solution, that of doubling the pull-down logic and cross-coupling it, as illustrated in Figure 15.17.

To reduce the load on the automatic place and route tool and to meet the performance requirements of the circuit, we partitioned the top-level design into a number of blocks, as shown in Figure 15.12. The place and route tool was not

timing-based, the intention being to show that a QDI-based asynchronous circuit will work for any delays, provided that the isochronic fork assumption is met. Figure 15.19 shows the cell placement of the counter block. The picture is a zoom into the lower-left corner of the design for clarity.

Figure 15.20 illustrates the same cell placement, showing the full picture and with routing completed. Each block was streamed back into the layout editor for DRC and (layout versus schematic) (LVS) checks against its transistor-level netlist. The LVS check also generates an extracted netlist of the design for HSPICE simulation. A short sample of the extracted netlist is shown in Figure 15.21. The flattened netlist consists of two parts, the transistor connections and the extracted capacitances.

The layout of the whole design is shown in Figure 15.12. All the blocks were individually placed and routed using the automatic tool. However, the routing between the blocks was done manually.

References

[1] J. B. Anderson and S. Mohan, "Sequential coding algorithms: a survey cost analysis," *IEEE Trans. Commun.*, COM-32, pp. 169–176, February 1984.

[2] S. Lin and D. J. Costello Jr, *Error Control Coding: Fundamentals and Applications*, Prentice Hall, 1983.

[3] J. M. Wozencraft and I. M. Jacobs, *Principles of Communication Engineering*, John Wiley and Sons, 1965.

[4] P. J. Black, "Algorithms and architectures for high-speed Viterbi decoding," Ph.D. thesis, Stanford University, 1993.

[5] R. O. Ozdag and P. A. Beerel, "An asynchronous low-power high performance sequential decoder implemented with QDI templates," *IEEE Trans. on VLSI Systems*, vol. 14, issue 9 pp. 975–985, September 2006.

Index