Asynchronous Circuit Design

A Tutorial

Jens Sparsø

Asynchronous circuit design

A Tutorial

Jens Sparsø
Technical University of Denmark

Foreword

This material in this booklet originally appeared as:

> J. Sparsø. Asynchronous circuit design - a tutorial. Chaptes 1-8 in J. Sparsø and S. Furber (eds.), *Principles of asynchronous circuit design - A systems perspective*. Kluwer Academic Publishers, 2001. 337 pages.

The writing of this textbook was initiated as a dissemination effort within the European Low-Power Initiative for Electronic System Design (ESD-LPD) as explained in the acknowledgements. As this author was neither involved in nor funded by any of the projects under the ESD-LPD cluster, and hence not contractually obliged to contribute to the book, copyright for chapters 1-8 was transferred to Kluwer Academic Publishers for a limited period. For this reason it is now possible for me to make the material available in the public domain (for non-commercial educational use). By doing this it is my hope that many more can benefit from it.

Apart from a few bug fixes everything (including page numbering) is the same. I have included the original preface and epilogue sections as they communicate the aim and perspective of the text. I have also included the original acknowledgements as they still apply.

Since its publication in 2001 the material has been used in courses taught at a number of universities including: The University of Manchester (UK), Technion (Israel), FORTH (Greece). It has also been used at a summer school in Grenoble in 2003 and a winter school at Cambridge University in 2005; both organized by the ACiD-WG network of excellence. At the Technical University of Denmark the author is teaching a 5 ECTS credit point one semester course using the material supplemented a couple of articles and by manuals and tutorials on the Balsa and Petrify synthesis tools. For laboratory exercises and design projects we have used Petrify, Balsa, and VHDL.

If you use the material for regular class teaching, I would be grateful if you drop me an email. Any comments, suggestions for improvements or extensions, bug reports etc. are also welcomed. When citing the material please refer to the original book mentioned above.

Jens Sparsø
Technical University of Denmark
April 2006

Email: jsp@imm.dtu.dk

Contents

Preface

This book was compiled to address a perceived need for an introductory text on asynchronous design. There are several highly technical books on aspects of the subject, but no obvious starting point for a designer who wishes to become acquainted for the first time with asynchronous technology. We hope this book will serve as that starting point.

The reader is assumed to have some background in digital design. We assume that concepts such as logic gates, flip-flops and Boolean logic are familiar. Some of the latter sections also assume familiarity with the higher levels of digital design such as microprocessor architectures and systems-on-chip, but readers unfamiliar with these topics should still find the majority of the book accessible.

The intended audience for the book comprises the following groups:

- Industrial designers with a background in conventional (clocked) digital design who wish to gain an understanding of asynchronous design in order, for example, to establish whether or not it may be advantageous to use asynchronous techniques in their next design task.

- Students in Electronic and/or Computer Engineering who are taking a course that includes aspects of asynchronous design.

The book is structured in three parts. Part I is a tutorial in asynchronous design. It addresses the most important issue for the beginner, which is how to think about asynchronous systems. The first big hurdle to be cleared is that of mindset – asynchronous design requires a different mental approach from that normally employed in clocked design. Attempts to take an existing clocked system, strip out the clock and simply replace it with asynchronous handshakes are doomed to disappoint. Another hurdle is that of circuit design methodology – the existing body of literature presents an apparent plethora of disparate approaches. The aim of the tutorial is to get behind this and to present a single unified and coherent perspective which emphasizes the common ground. In this way the tutorial should enable the reader to begin to understand the characteristics of asynchronous systems in a way that will enable them to

'think outside the box' of conventional clocked design and to create radical new design solutions that fully exploit the potential of clockless systems.

Once the asynchronous design mindset has been mastered, the second hurdle is designer productivity. VLSI designers are used to working in a highly productive environment supported by powerful automatic tools. Asynchronous design lags in its tools environment, but things are improving. Part II of the book gives an introduction to Balsa, a high-level synthesis system for asynchronous circuits. It is written by Doug Edwards (who has managed the Balsa development at the University of Manchester since its inception) and Andrew Bardsley (who has written most of the software). Balsa is not the solution to all asynchronous design problems, but it is capable of synthesizing very complex systems (for example, the 32-channel DMA controller used on the DRACO chip described in Chapter 15) and it is a good way to develop an understanding of asynchronous design 'in the large'.

Knowing how to think about asynchronous design and having access to suitable tools leaves one question: what can be built in this way? In Part III we offer a number of examples of complex asynchronous systems as illustrations of the answer to this question. In each of these examples the designers have been asked to provide descriptions that will provide the reader with insights into the design process. The examples include a commercial smart card chip designed at Philips and a Viterbi decoder designed at the University of Manchester. Part III closes with a discussion of the issues that come up in the design of advanced asynchronous microprocessors, focusing on the Amulet processor series, again developed at the University of Manchester.

Although the book is a compilation of contributions from different authors, each of these has been specifically written with the goals of the book in mind – to provide answers to the sorts of questions that a newcomer to asynchronous design is likely to ask. In order to keep the book accessible and to avoid it becoming an intimidating size, much valuable work has had to be omitted. Our objective in introducing you to asynchronous design is that you might become acquainted with it. If your relationship develops further, perhaps even into the full-blown affair that has smitten a few, included among whose number are the contributors to this book, you will, of course, want to know more. The book includes an extensive bibliography that will provide food enough for even the most insatiable of appetites.

JENS SPARSØ AND STEVE FURBER, SEPTEMBER 2001

Acknowledgments

Many people have helped significantly in the creation of this book. In addition to writing their respective chapters, several of the authors have also read and commented on drafts of other parts of the book, and the quality of the work as a whole has been enhanced as a result.

The editors are also grateful to Alan Williams, Russell Hobson and Steve Temple, for their careful reading of drafts of this book and their constructive suggestions for improvement.

Part I of the book has been used as a course text and the quality and consistency of the content improved by feedback from the students on the spring 2001 course "49425 Design of Asynchronous Circuits" at DTU.

Any remaining errors or omissions are the responsibility of the editors.

The writing of this book was initiated as a dissemination effort within the European Low-Power Initiative for Electronic System Design (ESD-LPD), and this book is part of the book series from this initiative. As will become clear, the book goes far beyond the dissemination of results from projects within in the ESD-LPD cluster, and the editors would like to acknowledge the support of the working group on asynchronous circuit design, ACiD-WG, that has provided a fruitful forum for interaction and the exchange of ideas. The ACiD-WG has been funded by the European Commission since 1992 under several Framework Programmes: FP3 Basic Research (EP7225), FP4 Technologies for Components and Subsystems (EP21949), and FP5 Microelectronics (IST-1999-29119).

I

ASYNCHRONOUS CIRCUIT DESIGN – A TUTORIAL

Author: Jens Sparsø

Technical University of Denmark

jsp@imm.dtu.dk

Abstract Asynchronous circuits have characteristics that differ significantly from those of synchronous circuits and, as will be clear from some of the later chapters in this book, it is possible exploit these characteristics to design circuits with very interesting performance parameters in terms of their power, performance, electromagnetic emissions (EMI), etc.

Asynchronous design is not yet a well-established and widely-used design methodology. There are textbooks that provide comprehensive coverage of the underlying theories, but the field has not yet matured to a point where there is an established currriculum and university tradition for teaching courses on asynchronous circuit design to electrical engineering and computer engineering students.

As this author sees the situation there is a gap between understanding the fundamentals and being able to design useful circuits of some complexity. The aim of Part I of this book is to provide a tutorial on asynchronous circuit design that fills this gap.

More specifically the aims are: (i) to introduce readers with background in synchronous digital circuit design to the fundamentals of asynchronous circuit design such that they are able to read and understand the literature, *and* (ii) to provide readers with an understanding of the "nature" of asynchronous circuits such that they are to design non-trivial circuits with interesting performance parameters.

The material is based on experience from the design of several asynchronous chips, and it has evolved over the last decade from tutorials given at a number of European conferences and from a number of special topics courses taught at the Technical University of Denmark and elsewhere. In May 1999 I gave a one-week intensive course at Delft University of Technology and it was when preparing for this course I felt that the material was shaping up, and I set out to write the following text. Most of the material has recently been used and debugged in a course at the Technical University of Denmark in the spring 2001. Supplemented by a few journal articles and a small design project, the text may be used for a one semester course on asynchronous design.

Keywords: asynchronous circuits, tutorial

Chapter 1

INTRODUCTION

1.1. Why consider asynchronous circuits?

Most digital circuits designed and fabricated today are "synchronous". In essence, they are based on two fundamental assumptions that greatly simplify their design: (1) all signals are binary, and (2) all components share a common and discrete notion of time, as defined by a clock signal distributed throughout the circuit.

Asynchronous circuits are fundamentally different; they also assume binary signals, *but there is no common and discrete time*. Instead the circuits use handshaking between their components in order to perform the necessary synchronization, communication, and sequencing of operations. Expressed in 'synchronous terms' this results in a behaviour that is similar to systematic fine-grain clock gating and local clocks that are not in phase and whose period is determined by actual circuit delays – registers are only clocked where and when needed.

This difference gives asynchronous circuits inherent properties that can be (and have been) exploited to advantage in the areas listed and motivated below. The interested reader may find further introduction to the mechanisms behind the advantages mentioned below in [106].

- Low power consumption, [102, 104, 32, 35, 73, 76]
  . . . due to fine-grain clock gating and zero standby power consumption.
- High operating speed, [119, 120, 63]
  . . . operating speed is determined by actual local latencies rather than global worst-case latency.
- Less emission of electro-magnetic noise, [102, 83]
  . . . the local clocks tend to tick at random points in time.
- Robustness towards variations in supply voltage, temperature, and fabrication process parameters, [62, 72, 74]
  . . . timing is based on matched delays (and can even be insensitive to circuit and wire delays).

- Better composability and modularity, [67, 57, 108, 97, 94]
  . . . because of the simple handshake interfaces and the local timing.

- No clock distribution and clock skew problems,
  . . . there is no global signal that needs to be distributed with minimal phase skew across the circuit.

On the other hand there are also some drawbacks. The asynchronous control logic that implements the handshaking normally represents an overhead in terms of silicon area, circuit speed, and power consumption. It is therefore pertinent to ask whether or not the investment pays off, i.e. whether the use of asynchronous techniques results in a substantial improvement in one or more of the above areas. Other obstacles are a lack of CAD tools and strategies and a lack of tools for testing and test vector generation.

Research in asynchronous design goes back to the mid 1950s [68, 67], but it was not until the late 1990s that projects in academia and industry demonstrated that it is possible to design asynchronous circuits which exhibit significant benefits in nontrivial real-life examples, and that commercialization of the technology began to take place. Recent examples are presented in [80] and in Part III of this book.

1.2. Aims and background

There are already several excellent articles and book chapters that introduce asynchronous design [40, 24, 25, 26, 106, 49, 94] as well as several monographs and textbooks devoted to asynchronous design including [80, 7, 17, 10, 70] – why then write yet another introduction to asynchronous design? There are several reasons:

- My experience from designing several asynchronous chips [93, 77], and from teaching asynchronous design to students and engineers over the past 10 years, is that it takes more than knowledge of the basic principles and theories to design efficient asynchronous circuits. In my experience there is a large gap between the introductory articles and book chapters mentioned above explaining the design methods and theories on the one side, and the papers describing actual designs and current research on the other side. It takes more than knowing the rules of a game to play and win the game. Bridging this gap involves experience and a good understanding of the nature of asynchronous circuits. An experience that I share with many other researchers is that "just going asynchronous" results in larger, slower and more power consuming circuits. *The crux is to use asynchronous techniques to exploit characteristics in the algorithm and architecture of the application in question.* This further implies that

asynchronous techniques may not always be the right solution to the problem.

- Another issue is that asynchronous design is a rather young discipline. Different researchers have proposed different circuit structures and design methods. At a first glance they may seem different – an observation that is supported by different terminologies; but a closer look often reveals that the underlying principles and the resulting circuits are rather similar.
- Finally, most of the above-mentioned introductory articles and book chapters are comprehensive in nature. While being appreciated by those already working in the field, the multitude of different theories and approaches in existence represents an obstacle for the newcomer wishing to get started designing asynchronous circuits.

Compared to the introductory texts mentioned above, the aims of this tutorial are: (1) to provide an introduction to asynchronous design that is more selective, (2) to stress basic principles and similarities between the different approaches, and (3) to take the introduction further towards designing practical and useful circuits.

1.3. Clocking versus handshaking

Figure 1.1(a) shows a synchronous circuit. For simplicity the figure shows a pipeline, but it is intended to represent any synchronous circuit. When designing ASICs using hardware description languages and synthesis tools, designers focus mostly on the data processing and assume the existence of a global clock. For example, a designer would express the fact that data clocked into register $R3$ is a function $CL3$ of the data clocked into $R2$ at the previous clock as the following assignment of variables: $R3 := CL3(R2)$. Figure 1.1(a) represents this high-level view with a universal clock.

When it comes to physical design, reality is different. Todays ASICs use a structure of clock buffers resulting in a large number of (possibly gated) clock signals as shown in figure 1.1(b). It is well known that it takes CAD tools and engineering effort to design the clock gating circuitry and to minimize and control the skew between the many different clock signals. Guaranteeing the two-sided timing constraints – the setup to hold time window around the clock edge – in a world that is dominated by wire delays is not an easy task. The buffer-insertion-and-resynthesis process that is used in current commercial CAD tools may not converge and, even if it does, it relies on delay models that are often of questionable accuracy.

Asynchronous design represents an alternative to this. In an asynchronous circuit the clock signal is replaced by some form of handshaking between neighbouring registers; for example the simple request-acknowledge based hand-

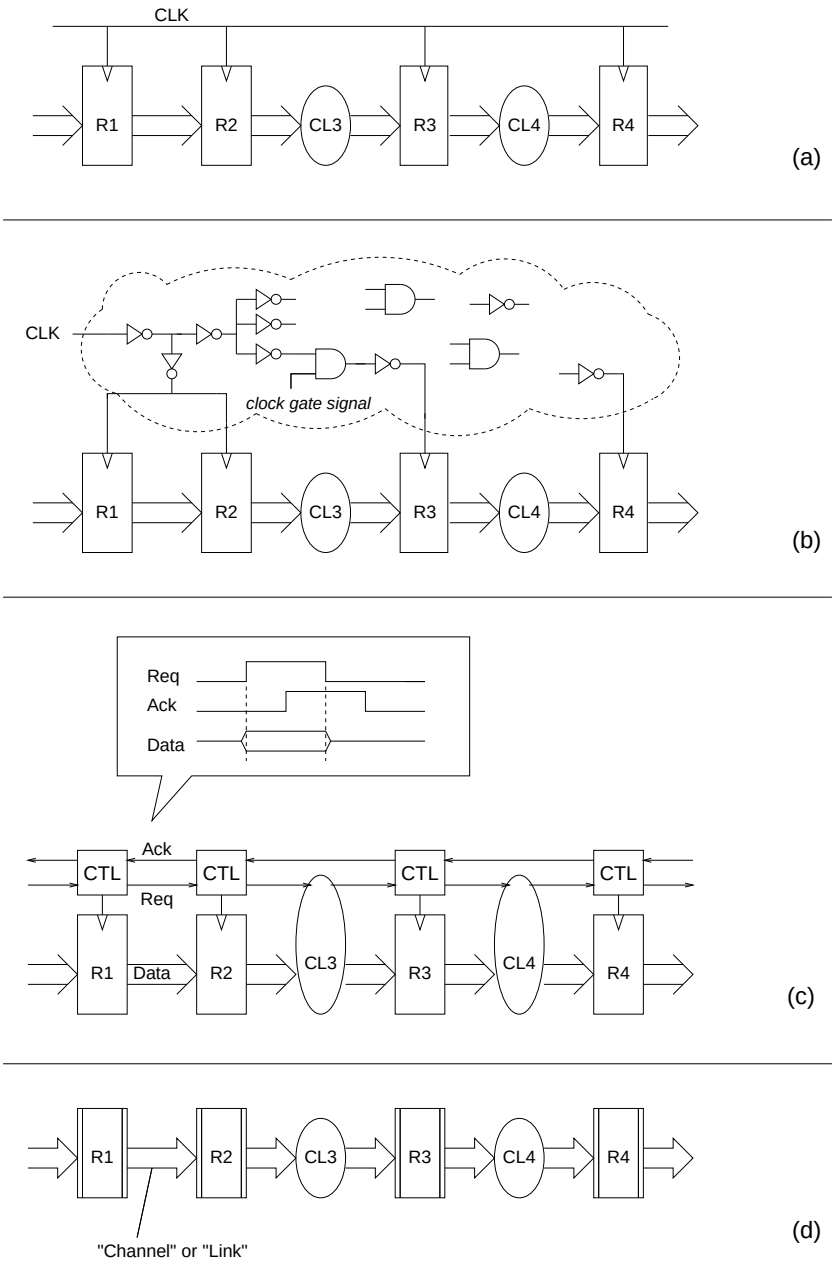


Figure 1.1. (a) A synchronous circuit, (b) a synchronous circuit with clock drivers and clock gating, (c) an equivalent asynchronous circuit, and (d) an abstract data-flow view of the asynchronous circuit. (The figure shows a pipeline, but it is intended to represent any circuit topology).

shake protocol shown in figure 1.1(c). In the following chapter we look at alternative handshake protocols and data encodings, but before departing into these implementation details it is useful to take a more abstract view as illustrated in figure 1.1(d):

- think of the data and handshake signals connecting one register to the next in figure 1.1(c) as a "handshake channel" or "link,"
- think of the data stored in the registers as tokens tagged with data values (that may be changed along the way as tokens flow through combinational circuits), and
- think of the combinational circuits as being transparent to the handshaking between registers; a combinatorial circuit simply absorbs a token on each of its input links, performs its computation, and then emits a token on each of its output links (much like a transition in a Petri net, c.f. section 6.2.1).

Viewed this way, an asynchronous circuit is simply a static data-flow structure [27]. Intuitively, correct operation requires that data tokens flowing in the circuit do not disappear, that one token does not overtake another, and that new tokens do not appear out of nowhere. A simple rule that can ensure this is the following:

> *A register may input and store a new data token from its predecessor if its successor has input and stored the data token that the register was previously holding.* [The states of the predecessor and successor registers are signaled by the incoming request and acknowledge signals respectively.]

Following this rule data is copied from one register to the next along the path through the circuit. In this process subsequent registers will often be holding copies of the same data value but the old duplicate data values will later be overwritten by new data values in a carefully ordered manner, and a handshake cycle on a link will always enclose the transfer of exactly one data-token. Understanding this "token flow game" is crucial to the design of efficient circuits, and we will address these issues later, extending the token-flow view to cover structures other than pipelines. Our aim here is just to give the reader an intuitive feel for the fundamentally different nature of asynchronous circuits.

An important message is that the "handshake-channel and data-token view" represents a very useful abstraction that is equivalent to the register transfer level (RTL) used in the design of synchronous circuits. This *data-flow abstraction*, as we will call it, separates the structure and function of the circuit from the implementation details of its components.

Another important message is that it is the handshaking between the registers that controls the flow of tokens, whereas the combinational circuit blocks must be fully transparent to this handshaking. Ensuring this transparency is not always trivial; it takes more than a traditional combinational circuit, so we will

use the term 'function block' to denote a combinational circuit whose input and output ports are handshake-channels or links.

Finally, some more down-to-earth engineering comments may also be relevant. The synchronous circuit in figure 1.1(b) is "controlled" by clock pulses that are in phase with a periodic clock signal, whereas the asynchronous circuit in figure 1.1(c) is controlled by locally derived clock pulses that can occur at any time; the local handshaking ensures that clock pulses are generated where and when needed. This tends to randomize the clock pulses over time, and is likely to result in less electromagnetic emission and a smoother supply current without the large di/dt spikes that characterize a synchronous circuit.

1.4. Outline of Part I

Chapter 2 presents a number of fundamental concepts and circuits that are important for the understanding of the following material. Read through it but don't get stuck; you may want to revisit relevant parts later.

Chapters 3 and 4 address asynchronous design at the data-flow level: chapter 3 explains the operation of pipelines and rings, introduces a set of handshake components and explains how to design (larger) computing structures, and chapter 4 addresses performance analysis and optimization of such structures, both qualitatively and quantitatively.

Chapter 5 addresses the circuit level implementation of the handshake components introduced in chapter 3, and chapter 6 addresses the design of hazard-free sequential (control) circuits. The latter includes a general introduction to the topics and in-depth coverage of one specific method: the design of speed-independent control circuits from signal transition graph specifications. These techniques are illustrated by control circuits used in the implementation of some of the handshake components introduced in chapter 3.

All of the above chapters 2–6 aim to explain the basic techniques and methods in some depth. The last two chapters are briefer. Chapter 7 introduces more advanced topics related to the implementation of circuits using the 4-phase bundled-data protocol, and chapter 8 addresses hardware description languages and synthesis tools for asynchronous design. Chapter 8 is by no means comprehensive; it focuses on CSP-like languages and syntax-directed compilation, but also describes how asynchronous design can be supported by a standard language, VHDL.

Chapter 2

FUNDAMENTALS

This chapter provides explanations of a number of topics and concepts that are of fundamental importance for understanding the following chapters and for appreciating the similarities between the different asynchronous design styles. The presentation style will be somewhat informal and the aim is to provide the reader with intuition and insight.

2.1. Handshake protocols

The previous chapter showed one particular handshake protocol known as a return-to-zero handshake protocol, figure 1.1(c). In the asynchronous community it is given a more informative name: the 4-phase bundled-data protocol.

2.1.1 Bundled-data protocols

The term *bundled-data* refers to a situation where the data signals use normal Boolean levels to encode information, and where separate request and acknowledge wires are bundled with the data signals, figure 2.1(a). In the *4-phase* protocol illustrated in figure 2.1(b) the request and acknowledge wires also use normal Boolean levels to encode information, and the term 4-phase refers to the number of communication actions: (1) the sender issues data and sets request high, (2) the receiver absorbs the data and sets acknowledge high, (3) the sender responds by taking request low (at which point data is no longer guaranteed to be valid) and (4) the receiver acknowledges this by taking acknowledge low. At this point the sender may initiate the next communication cycle.

The 4-phase bundled data protocol is familiar to most digital designers, but it has a disadvantage in the superfluous return-to-zero transitions that cost unnecessary time and energy. The 2-phase bundled-data protocol shown in figure 2.1(c) avoids this. The information on the request and acknowledge wires is now encoded as signal transitions on the wires and there is no difference between a $0 \rightarrow 1$ and a $1 \rightarrow 0$ transition, they both represent a "signal event". Ideally the 2-phase bundled-data protocol should lead to faster circuits than the 4-phase

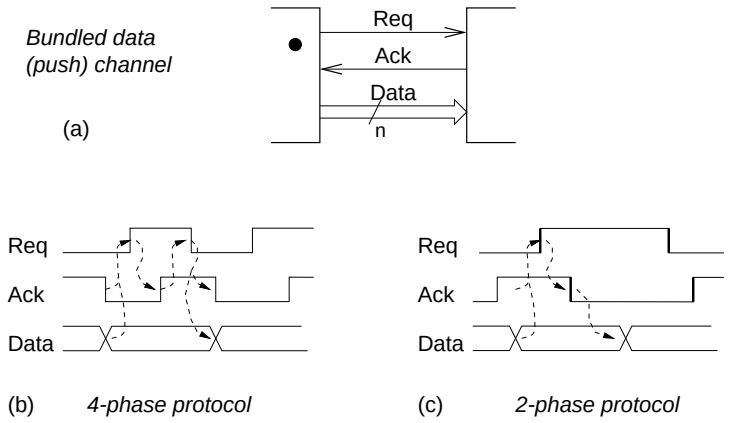


Figure 2.1. (a) A bundled-data channel. (b) A 4-phase bundled-data protocol. (c) A 2-phase bundled-data protocol.

bundled-data protocol, but often the implementation of circuits responding to events is complex and there is no general answer as to which protocol is best.

At this point some discussion of terminology is appropriate. Instead of the term *bundled-data* that is used throughout this text, some texts use the term *single-rail*. The term 'bundled-data' hints at the timing relationship between the data signals and the handshake signals, whereas the term 'single-rail' hints at the use of one wire to carry one bit of data. Also, the term single-rail may be considered consistent with the dual-rail data representation discussed in the next section. Instead of the term *4-phase* handshaking (or signaling) some texts use the terms *return-to-zero (RTZ) signaling* or *level signaling*, and instead of the term *2-phase* handshaking (or signaling) some texts use the terms *non-return-to-zero (NRZ) signaling* or *transition signaling*. Consequently a return-to-zero single-track prococol is the same as a 4-phase bundled-data protocol, etc.

The protocols introduced above all assume that the sender is the active party that initiates the data transfer over the channel. This is known as a *push channel*. The opposite, the receiver asking for new data, is also possible and is called a *pull channel*. In this case the directions of the request and acknowledge signals are reversed, and the validity of data is indicated in the acknowledge signal going from the sender to the receiver. In abstract circuit diagrams showing links/channels as one symbol we will often mark the active end of a channel with a dot, as illustrated in figure 2.1(a).

To complete the picture we mention a number of variations: (1) a channel without data that can be used for synchronization, and (2) a channel where data is transmitted in both directions and where req and ack indicate validity of the data that is exchanged. The latter could be used to interface a read-

only memory: the address would be bundled with req and the data would be bundled with ack. These alternatives are explained later in section 7.1.1. In the following sections we will restrict the discussion to push channels.

All the bundled-data protocols rely on delay matching, such that the order of signal events at the sender's end is preserved at the receiver's end. On a push channel, data is valid before request is set high, expressed formally as $Valid(Data) \prec Req$. This ordering should also be valid at the receiver's end, and it requires some care when physically implementing such circuits. Possible solutions are:

- To control the placement and routing of the wires, possibly by routing all signals in a channel as a bundle. This is trivial in a tile-based datapath structure.
- To have a safety margin at the sender's end.
- To insert and/or resize buffers after layout (much as is done in today's synthesis and layout CAD tools).

An alternative is to use a more sophisticated protocol that is robust to wire delays. In the following sections we introduce a number of such protocols that are completely insensitive to delays.

2.1.2 The 4-phase dual-rail protocol

The 4-phase dual-rail protocol encodes the request signal into the data signals using two wires per bit of information that has to be communicated, figure 2.2. In essence it is a 4-phase protocol using two request wires per bit of information d; one wire $d.t$ is used for signaling a logic 1 (or true), and another wire $d.f$ is used for signaling logic 0 (or false). When observing a 1-bit channel one will see a sequence of 4-phase handshakes where the participating "request" signal in any handshake cycle can be either $d.t$ or $d.f$. This protocol is very

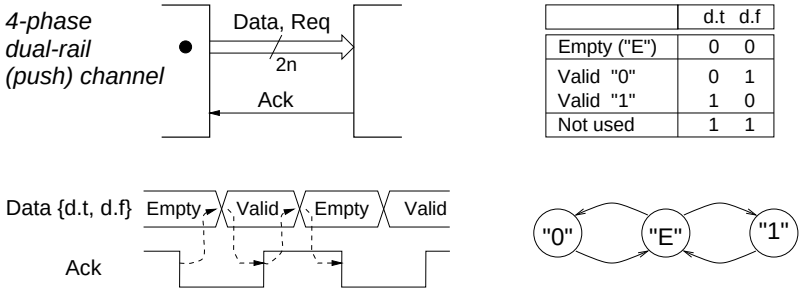


Figure 2.2. A delay-insensitive channel using the 4-phase dual-rail protocol.

robust; two parties can communicate reliably regardless of delays in the wires connecting the two parties – the protocol is *delay-insensitive*.

Viewed together the $\{x.f, x.t\}$ wire pair is a codeword; $\{x.f, x.t\} = \{1, 0\}$ and $\{x.f, x.t\} = \{0, 1\}$ represent "valid data" (logic 0 and logic 1 respectively) and $\{x.f, x.t\} = \{0, 0\}$ represents "no data" (or "spacer" or "empty value" or "NULL"). The codeword $\{x.f, x.t\} = \{1, 1\}$ is not used, and a transition from one valid codeword to another valid codeword is not allowed, as illustrated in figure 2.2.

This leads to a more abstract view of 4-phase handshaking: (1) the sender issues a valid codeword, (2) the receiver absorbs the codeword and sets acknowledge high, (3) the sender responds by issuing the empty codeword, and (4) the receiver acknowledges this by taking acknowledge low. At this point the sender may initiate the next communication cycle. An even more abstract view of what is seen on a channel is a data stream of valid codewords separated by empty codewords.

Let's now extend this approach to bit-parallel channels. An N-bit data channel is formed simply by concatenating N wire pairs, each using the encoding described above. A receiver is always able to detect when all bits are valid (to which it responds by taking acknowledge high), and when all bits are empty (to which it responds by taking acknowledge low). This is intuitive, but there is also some mathematical background – the dual-rail code is a particularly simple member of the family of delay-insensitive codes [112], and it has some nice properties:

- any concatenation of dual-rail codewords is itself a dual-rail codeword.
- for a given N (the number of bits to be communicated), the set of all possible codewords can be *disjointly* divided into 3 sets:
  - the *empty codeword* where all N wire pairs are {0,0}.
  - the *intermediate codewords* where some wire-pairs assume the empty state and some wire pairs assume valid data.
  - the 2^N different *valid codewords.*

Figure 2.3 illustrates the handshaking on an N-bit channel: a receiver will see the empty codeword, a sequence of intermediate codewords (as more and more bits/wire-pairs become valid) and eventually a valid codeword. After receiving and acknowledging the codeword, the receiver will see a sequence of intermediate codewords (as more and more bits become empty), and eventually the empty codeword to which the receiver responds by driving acknowledge low again.

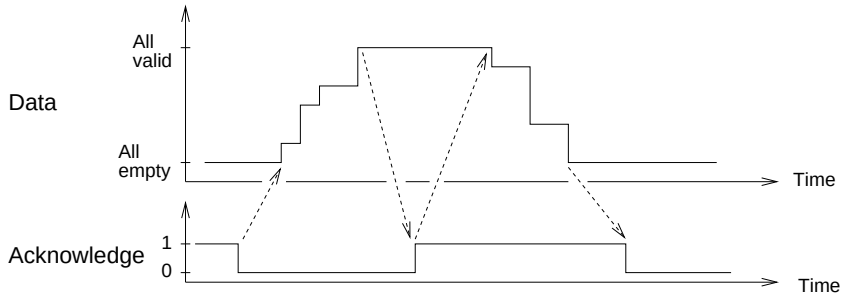


Figure 2.3. Illustration of the handshaking on a 4-phase dual-rail channel.

2.1.3 The 2-phase dual-rail protocol

The 2-phase dual-rail protocol also uses 2 wires $\{d.t, d.f\}$ per bit, but the information is encoded as transitions (events) as explained previously. On an N-bit channel a new codeword is received when exactly one wire in each of the N wire pairs has made a transition. There is no empty value; a valid message is acknowledged and followed by another message that is acknowledged. Figure 2.4 shows the signal waveforms on a 2-bit channel using the 2-phase dual-rail protocol.

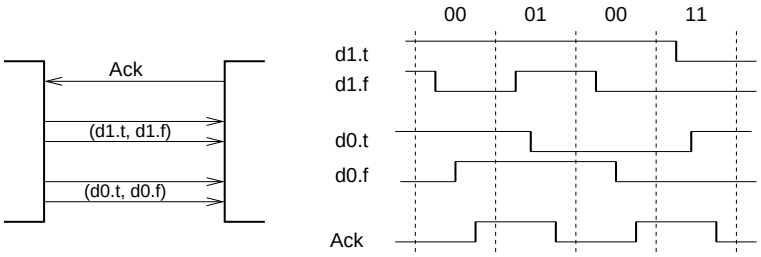


Figure 2.4. Illustration of the handshaking on a 2-phase dual-rail channel.

2.1.4 Other protocols

The previous sections introduced the four most common channel protocols: the 4-phase bundled-data push channel, the 2-phase bundled-data push channel, the 4-phase dual-rail push channel and the 2-phase dual-rail push channel; but there are many other possibilities. The two wires per bit used in the dual-rail protocol can be seen as a one-hot encoding of that bit and often it is useful to extend to 1-of-n encodings in control logic and higher-radix data encodings.

If the focus is on communication rather than computation, m-of-n encodings may be of relevance. The solution space can be expressed as the cross product of a number of options including:

$$\{\text{2-phase}, \text{4-phase}\} \times \{\text{bundled-data}, \text{dual-rail}, \text{1-of-n}, \ldots\} \times \{\text{push}, \text{pull}\}$$

The choice of protocol affects the circuit implementation characteristics (area, speed, power, robustness, etc.). Before continuing with these implementation issues it is necessary to introduce the concept of indication or acknowledgement, as well as a new component, the Muller C-element.

2.2. The Muller C-element and the indication principle

In a synchronous circuit the role of the clock is to define points in time where signals are stable and valid. In between the clock-ticks, the signals may exhibit hazards and may make multiple transitions as the combinational circuits stabilize. This does not matter from a functional point of view. In asynchronous (control) circuits the situation is different. The absence of a clock means that, in many circumstances, signals are required to be valid all the time, that every signal transition has a meaning and, consequently, that hazards and races must be avoided.

Intuitively a circuit is a collection of gates (normally including some feedback loops), and when the output of a gate changes it is seen by other gates that in turn may decide to change their outputs accordingly. As an example figure 2.5 shows one possible implementation of the CTL circuit in figure 1.1(c). The intention here is not to explain its function, just to give an impression of the type of circuit we are discussing. It is obvious that hazards on the Ro, Ai, and Lt signals would be disastrous if the circuit is used in the pipeline of figure 1.1(c).

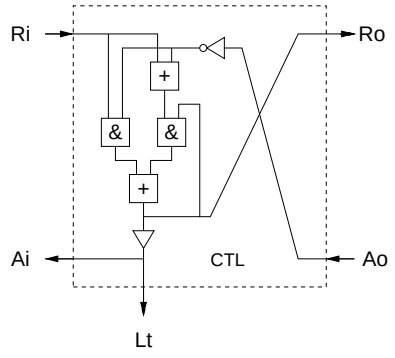
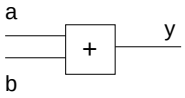
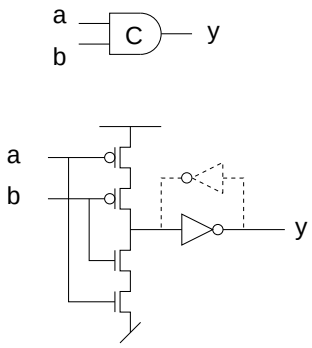


Figure 2.5. An example of an asynchronous control circuit. Lt is a "local" clock that is intended to control a latch.



a	b	y
0	0	0
0	1	1
1	0	1
1	1	1

Figure 2.6. A normal OR gate



Some specifications:

1: if $a = b$ then $y := a$

2: $a = b \mapsto y := a$

3: $y = ab + y(a + b)$

4:

a	b	y
0	0	0
0	1	no change
1	0	no change
1	1	1

Figure 2.7. The Muller C-element: symbol, possible implementation, and some alternative specifications.

The concept of *indication* or *acknowledgement* plays an important role in the design of such circuits. Consider the simple 2-input OR gate in figure 2.6. An observer seeing the output change from 1 to 0 may conclude that *both* inputs are now at 0. However, when seeing the output change from 0 to 1 the observer is not able to make conclusions about *both* inputs. The observer only knows that at least one input is 1, but it does not know which. We say that the OR gate only indicates or acknowledges when both inputs are 0. Through similar arguments it can be seen that an AND gate only indicates when both inputs are 1.

Signal transitions that are not indicated or acknowledged in other signal transitions are the source of hazards and should be avoided. We will address this issue in greater detail later in section 2.5.1 and in chapter 6.

A circuit that is better in this respect is the Muller C-element shown in figure 2.7. It is a state-holding element much like an asynchronous set-reset latch. When both inputs are 0 the output is set to 0, and when both inputs are 1 the output is set to 1. For other input combinations the output does not change. Consequently, an observer seeing the output change from 0 to 1 may conclude that *both* inputs are now at 1; and similarly, an observer seeing the output change from 1 to 0 may conclude that *both* inputs are now 0.

Combining this with the observation that all asynchronous circuits rely on handshaking that involves cyclic transitions between 0 and 1, it should be clear that the Muller C-element is indeed a fundamental component that is extensively used in asynchronous circuits.

2.3. The Muller pipeline

Figure 2.8 shows a circuit that is built from C-elements and inverters. The circuit is known as a Muller pipeline or a Muller distributor. Variations and extensions of this circuit form the (control) backbone of almost all asynchronous circuits. It may not always be obvious at a first glance, but if one strips off the cluttering details, the Muller pipeline is always there as the crux of the matter. The circuit has a beautiful and symmetric behaviour, and once you understand its behaviour, you have a very good basis for understanding most asynchronous circuits.

The Muller pipeline in figure 2.8 is a mechanism that relays handshakes. After all of the C-elements have been initialized to 0 the left environment may start handshaking. To understand what happens let's consider the ith C-element, $C[i]$: It will propagate (i.e. input and store) a 1 from its predecessor, $C[i-1]$, only if its successor, $C[i+1]$, is 0. In a similar way it will propagate (i.e. input and store) a 0 from its predecessor if its successor is 1. It is often useful to think of the signals propagating in an asynchronous circuit as a sequence of waves, as illustrated at the bottom of figure 2.8. Viewed this way, the role of a C-element stage in the pipeline is to propagate crests and troughs of waves in a carefully controlled way that maintains the integrity of each wave.

On any interface between C-element pipeline stages an observer will see correct handshaking, but the timing may differ from the timing of the handshaking on the left hand environment; once a wave has been injected into the Muller pipeline it will propagate with a speed that is determined by actual delays in the circuit.

Eventually the first handshake (request) injected by the left hand environment will reach the right hand environment. If the right hand environment does not respond to the handshake, the pipeline will eventually fill. If this happens the pipeline will stop handshaking with the left hand environment – the Muller pipeline behaves like a ripple through FIFO!

In addition to this elegant behaviour, the pipeline has a number of beautiful symmetries. Firstly, it does not matter if you use 2-phase or 4-phase handshaking. It is the same circuit. The difference is in how you interpret the signals and use the circuit. Secondly, the circuit operates equally well from right to left. You may reverse the definition of signal polarities, reverse the role of the request and acknowledge signals, and operate the circuit from right to left. It is analogous to electrons and holes in a semiconductor; when current flows in

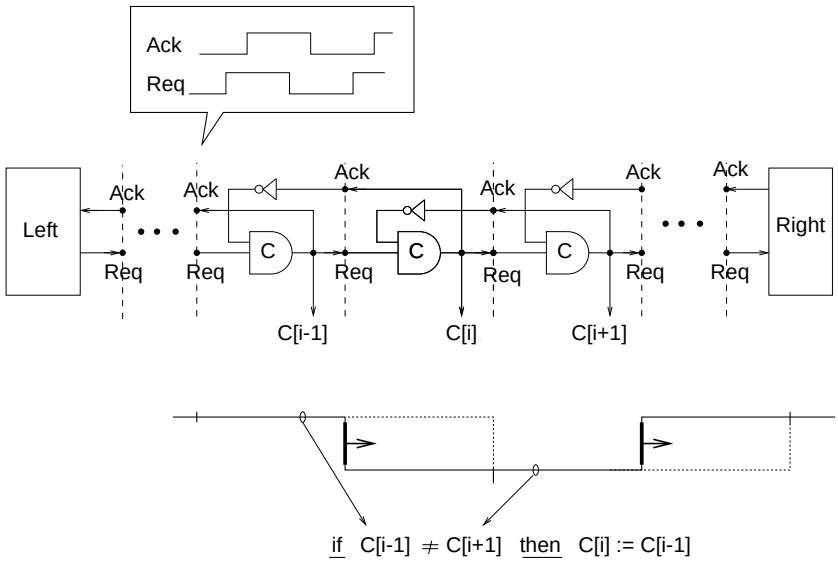


Figure 2.8. The Muller pipeline or Muller distributor.

one direction it may be carried by electrons flowing in one direction or by holes flowing in the opposite direction.

Finally, the circuit has the interesting property that it works correctly regardless of delays in gates and wires – the Muller-pipeline is delay-insensitive.

2.4. Circuit implementation styles

As mentioned previously, the choice of handshake protocol affects the circuit implementation (area, speed, power, robustness, etc.). Most practical circuits use one of the following protocols introduced in section 2.1:

4-phase bundled-data – which most closely resembles the design of synchronous circuits and which normally leads to the most efficient circuits, due to the extensive use of timing assumptions.

2-phase bundled-data – introduced under the name *Micropipelines* by Ivan Sutherland in his 1988 Turing Award lecture.

4-phase dual-rail – the classic approach rooted in David Muller's pioneering work in the 1950s.

Common to all protocols is the fact that the corresponding circuit implementations all use variations of the Muller pipeline for controlling the storage elements. Below we explain the basics of pipelines built using simple transparent

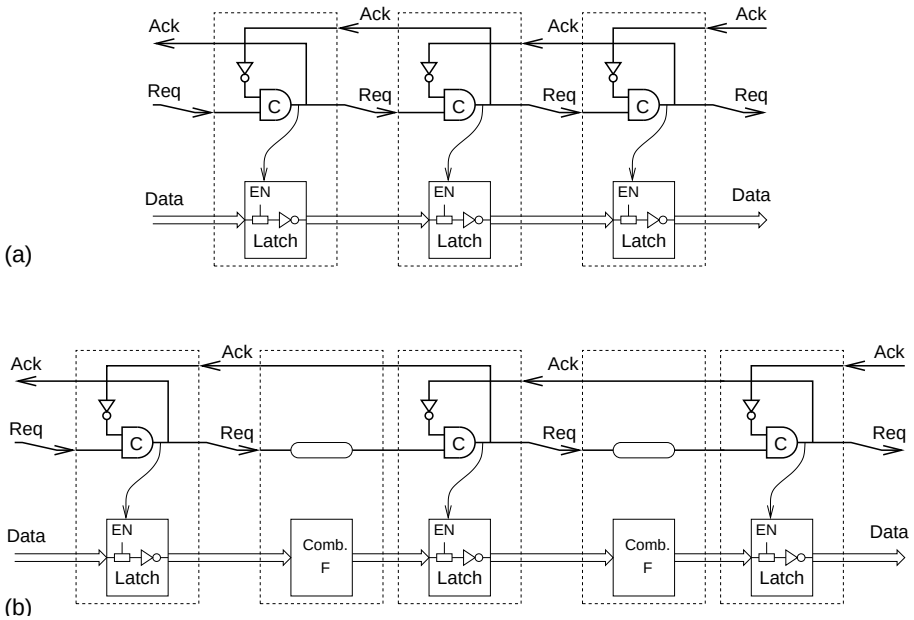


Figure 2.9. A simple 4-phase bundled-data pipeline.

latches as storage elements. More optimized and elaborate circuit implementations and more complex circuit structures are the topics of later chapters.

2.4.1 4-phase bundled-data

A 4-phase bundled-data pipeline is particularly simple. A Muller pipeline is used to generate local clock pulses. The clock pulse generated in one stage overlaps with the pulses generated in the neighbouring stages in a carefully controlled interlocked manner. Figure 2.9(a) shows a FIFO, i.e. a pipeline without data processing, and figure 2.9(b) shows how combinational circuits (also called function blocks) can be added between the latches. To maintain correct behaviour matching delays have to be inserted in the request signal paths.

You may view this circuit as a traditional "synchronous" data-path, consisting of latches and combinational circuits that are clocked by a distributed gated-clock driver, or you may view it as an asynchronous data-flow structure composed of two types of handshake components: latches and function blocks, as indicated by the dashed boxes.

The pipeline implementation shown in figure 2.9 is particularly simple but it has some drawbacks: when it fills the state of the C-elements is (0, 1, 0, 1, etc.), and as a consequence only every other latch is storing data. This is no worse

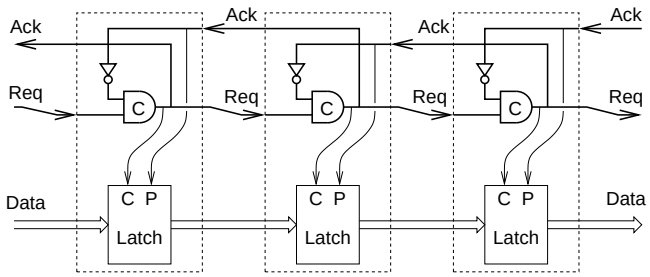


Figure 2.10. A simple 2-phase bundled-data pipeline.

than in a synchronous circuit using master-slave flip-flops, but it is possible to design asynchronous pipelines and FIFOs that are better in this respect. Another disadvantage is speed. The throughput of a pipeline or FIFO depends on the time it takes to complete a handshake cycle and for the above implementation this involves communication with both neighbours. Chapter 7 addresses alternative implementations that are both faster and have better occupancy when full.

2.4.2 2-phase bundled data (Micropipelines)

A 2-phase bundled-data pipeline also uses a Muller pipeline as the backbone control circuit, but the control signals are interpreted as events or transitions, figure 2.10. For this reason special capture-pass latches are needed: events on the C and P inputs alternate, causing the latch to alternate between capture mode and pass mode. This calls for a special latch design as shown in figure 2.11 and explained below. The switch symbol in figure 2.11 is a multiplexer, and the event controlled latch can be understood as two ordinary level sensitive latches (operating in an alternating fashion) followed by a multiplexer and a buffer.

Figure 2.10 shows a pipeline without data processing. Combinational circuits with matching delay elements can be inserted between latches in a similar way to the 4-phase bundled-data approach in figure 2.9.

The 2-phase bundled-data approach was pioneered by Ivan Sutherland in the late 1980s and an excellent introduction is given in his 1988 Turing Award Lecture [97]. The title *Micropipelines* is often used synonymously with the use of the 2-phase bundled-data protocol, but it also refers to the use of a particular set of components that are based on event signalling. In addition to the latch in figure 2.11 these are: AND, OR, Select, Toggle, Call and Arbiter. The above figures 2.10 and 2.11 are similar to figures 15 and 12 in [97], but they emphasise stronger the fact that the control structure is a Muller-pipeline. Some alternative latch designs that are (significantly) smaller and (significantly) slower are also presented in [97].

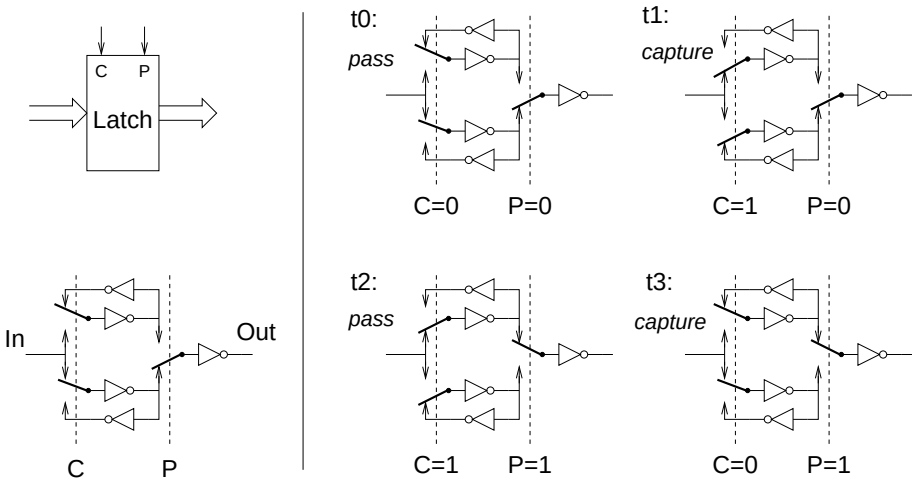


Figure 2.11. Implementation and operation of a capture-pass event controlled latch. At time t_0 the latch is transparent (i.e. in pass mode) and signals C and P are both low. An event on the C input turns the latch into capture mode, etc.

At the conceptual level the 2-phase bundled-data approach is elegant and efficient; compared to the 4-phase bundled-data approach it avoids the power and performance loss that is incurred by the return-to-zero part of the hand-shaking. However, as illustrated by the latch design, the implementation of components that respond to signal transitions is often more complex than the implementation of components that respond to normal level signals. In addition to the storage elements explained above, conditional control logic that responds to signal transitions tends to be complex as well. This has been experienced by this author [93], by the University of Manchester [32, 35] and by many others.

Having said this, the 2-phase bundled-data approach may be the preferred solution in systems with unconditional data-flows and very high speed requirements. But as just mentioned, the higher speed comes at a price: larger silicon area and higher power consumption. In this respect asynchronous design is no different from synchronous design.

2.4.3 4-phase dual-rail

A 4-phase dual-rail pipeline is also based on the Muller pipeline, but in a more elaborate way that has to do with the combined encoding of data and request. Figure 2.12 shows the implementation of a 1-bit wide and three stage deep pipeline without data processing. It can be understood as two Muller pipelines connected in parallel, using a common acknowledge signal per stage to synchronize operation. The pair of C-elements in a pipeline stage can store

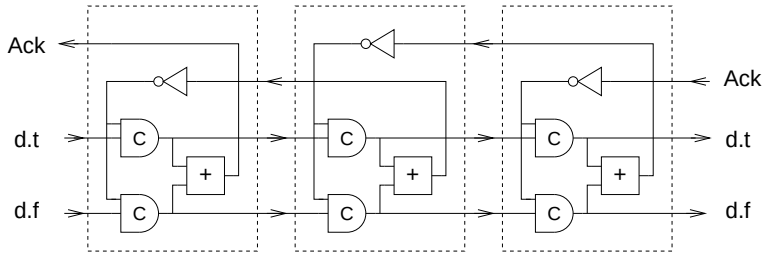


Figure 2.12. A simple 3-stage 1-bit wide 4-phase dual-rail pipeline.

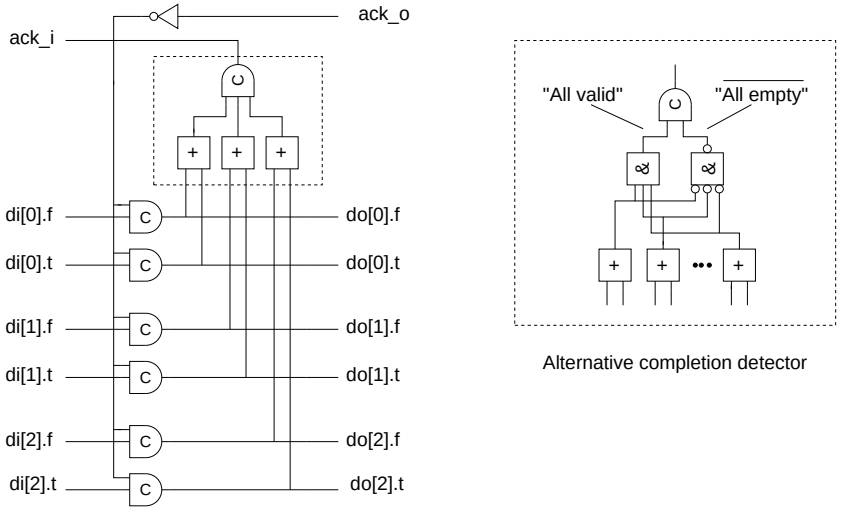
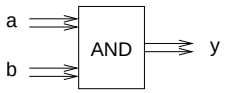


Figure 2.13. An N-bit latch with completion detection.

the empty codeword $\{d.t, d.f\} = \{0, 0\}$, causing the acknowledge signal out of that stage to be 0, or it can store one of the two valid codewords $\{0, 1\}$ and $\{1, 0\}$, causing the acknowledge signal out of that stage to be logic 1. At this point, and referring back to section 2.2, the reader should notice that because the codeword $\{1, 1\}$ is illegal and does not occur, the acknowledge signal generated by the OR gate safely indicates the state of the pipeline stage as being "valid" or "empty."

An N-bit wide pipeline can be implemented by using a number of 1-bit pipelines in parallel. This does not guarantee to a receiver that all bits in a word arrive at the same time, but often the necessary synchronization is done in the function blocks. In [94, 95] we describe a design of this style using the DIMS combinational circuits explained below.



a	b	y.f	y.t
E	E	0	0
		NO CHANGE	
F	F	1	0
F	T	1	0
T	F	1	0
T	T	0	1

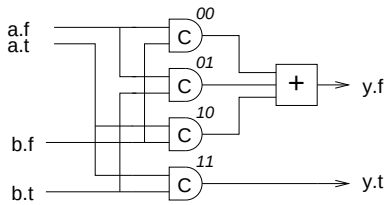


Figure 2.14. A 4-phase dual-rail AND gate: symbol, truth table, and implementation.

If bit-parallel synchronization is needed, the individual acknowledge signals can be combined into one global acknowledge using a C-element. Figure 2.13 shows an N-bit wide latch. The OR gates and the C-element in the dashed box form a *completion detector* that indicates whether the N-bit dual-rail codeword stored in the latch is empty or valid. The figure also shows an implementation of a completion detector using only a 2-input C-element.

Let us now look at how combinational circuits for 4-phase dual-rail circuits are implemented. As mentioned in chapter 1 combinational circuits must be transparent to the handshaking between latches. Therefore, all outputs of a combinational circuit must not become valid until after all inputs have become valid. Otherwise the receiving latch may prematurely set acknowledge high (before all signals from the sending latch have become valid). In a similar way all outputs of a combinational circuit must not become empty until after all inputs have become empty. Otherwise the receiving latch may prematurely set acknowledge low (before all signals from the sending latch have become empty). Consequently a combinational circuit for the 4-phase dual-rail approach involves state holding elements and it exhibits a hysteresis-like behaviour in the empty-to-valid and valid-to-empty transitions.

A particularly simple approach, using only C-elements and OR gates, is illustrated in figure 2.14, which shows the implementation of a dual-rail AND gate.The circuit can be understood as a direct mapping from sum-of-minterms expressions for each of the two output wires into hardware. The circuit waits for all its inputs to become valid. When this happens exactly one of the four C-elements goes high. This again causes the relevant output wire to go high corresponding to the gate producing the desired valid output. When all inputs become empty the C-elements are all set low, and the output of the dual-rail AND gate becomes empty again. Note that the C-elements provide both the necessary 'and' operator and the hysteresis in the empty-to-valid and valid-to-

empty transitions that is required for transparent handshaking. Note also that (again) the OR gate is never exposed to more than one input signal being high.

Other dual-rail gates such as OR and EXOR can be implemented in a similar fashion, and a dual-rail inverter involves just a swap of the true and false wires. The transistor count in these basic dual-rail gates is obviously rather high, and in chapter 5 we explore more efficient circuit implementations. Here our interest is in the fundamental principles.

Given a set of basic dual-rail gates one can construct dual-rail combinational circuits for arbitrary Boolean expressions using normal combinational circuit synthesis techniques. The transparency to handshaking that is a property of the basic gates is preserved when composing gates into larger combinational circuits.

The fundamental ideas explained above all go back to David Muller's work in the late 1950s and early 1960s [68, 67]. While [68] develops the fundamental theory for the design of speed-independent circuits, [67] is a more practical introduction including a design example: a bit-serial multiplier using latches and gates as explained above.

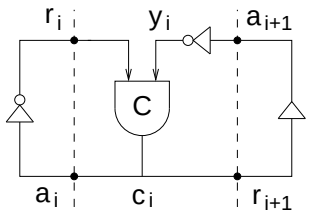
2.5. Theory

Asynchronous circuits can be classified, as we will see below, as being *self-timed*, *speed-independent* or *delay-insensitive* depending on the delay assumptions that are made. In this section we introduce some important theoretical concepts that relate to this classification. The goal is to communicate the basic ideas and provide some intuition on the problems and solutions, and a reader who wishes to dig deeper into the theory is referred to the literature. Some recent starting points are [70, 40, 49, 26, 10].

2.5.1 The basics of speed-independence

We will start by reviewing the basics of David Muller's model of a circuit and the conditions for it being speed-independent [68]. A circuit is modeled along with its (dummy) environment as a closed network of gates, closed meaning that all inputs are connected to outputs and vice versa. Gates are modeled as Boolean operators with arbitrary non-zero delays, and wires are assumed to be ideal. In this context the circuit can be described as a set of concurrent Boolean functions, one for each gate output. The state of the circuit is the set of all gate outputs. Figure 2.15 illustrates this for a stage of a Muller pipeline with an inverter and a buffer mimicing the handshaking behaviour of the left and right hand environments.

A gate whose output is consistent with its inputs is said to be stable; its "next output" is the same as its "current output", $z_i\prime = z_i$. A gate whose inputs have



$$
\begin{aligned}
r_i\prime &= not(c_i) \\
c_i\prime &= r_i y_i + (r_i + y_i)c_i \\
y_i\prime &= not(a_{i+1}) \\
a_{i+1}\prime &= c_i
\end{aligned}
$$

Figure 2.15. Muller model of a Muller pipeline stage with "dummy" gates modeling the environment behaviour.

changed in such a way that an output change is called for is said to be excited; its "next output" is different from its "current output", i.e. $z_i\prime \neq z_i$. After an arbitrary delay an excited gate may spontaneously change its output and become stable. We say that the gate fires, and as excited gates fire and become stable with new output values, other gates in turn become excited, etc.

To illustrate this, suppose that the circuit in figure 2.15 is in state $(r_i, y_i, c_i, a_{i+1}) = (0, 1, 0, 0)$. In this state (the inverter) r_i is excited corresponding to the left environment being about to take request high. After the firing of $r_i \uparrow$ the circuit reaches state $(r_i, y_i, c_i, a_{i+1}) = (1, 1, 0, 0)$ and c_i now becomes excited. For synthesis and analysis purposes one can construct the complete state graph representing all possible sequences of gate firings. This is addressed in detail in chapter 6. Here we will restrict the discussion to an explanation of the fundamental ideas.

In the general case it is possible that several gates are excited at the same time (i.e. in a given state). If one of these gates, say z_i, fires the interesting thing is what happens to the other excited gates which may have z_i as one of their inputs: they may remain excited, or they may find themselves with a different set of input signals that no longer calls for an output change. *A circuit is* speed-independent *if the latter never happens.* The practical implication of an excited gate becoming stable without firing is a potential hazard. Since delays are unknown the gate may or may not have changed its output, or it may be in the middle of doing so when the 'counter-order' comes calling for the gate output to remain unchanged.

Since the model involves a Boolean state variable for each gate (and for each wire segment in the case of delay-insensitive circuits) the state space becomes very large even for very simple circuits. In chapter 6 we introduce signal transition graphs as a more abstract representation from which circuits can be synthesized.

Now that we have a model for describing and reasoning about the behaviour of gate-level circuits let's address the classification of asynchronous circuits.

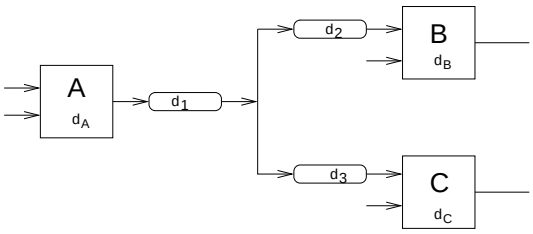


Figure 2.16. A circuit fragment with gate and wire delays. The output of gate A forks to inputs of gates B and C.

2.5.2 Classification of asynchronous circuits

At the gate level, asynchronous circuits can be classified as being self-timed, speed-independent or delay-insensitive depending on the delay assumptions that are made. Figure 2.16 serves to illustrate the following discussion. The figure shows three gates: A, B, and C, where the output signal from gate A is connected to inputs on gates B and C.

A *speed-independent* (SI) circuit as introduced above is a circuit that operates "correctly" assuming positive, bounded but unknown delays in gates and ideal zero-delay wires. Referring to figure 2.16 this means arbitrary d_A, d_B, and d_C, but $d_1 = d_2 = d_3 = 0$. Assuming ideal zero-delay wires is not very realistic in today's semiconductor processes. By allowing arbitrary d_1 and d_2 and by requiring $d_2 = d_3$ the wire delays can be lumped into the gates, and from a theoretical point of view the circuit is still speed-independent.

A circuit that operates "correctly" with positive, bounded but unknown delays in wires as well as in gates is *delay-insensitive* (DI). Referring to figure 2.16 this means arbitrary d_A, d_B, d_C, d_1, d_2, and d_3. Such circuits are obviously extremely robust. One way to show that a circuit is delay-insensitive is to use a Muller model of the circuit where wire segments (after forks) are modeled as buffer components. If this equivalent circuit model is speed-independent, then the circuit is delay-insensitive.

Unfortunately the class of delay-insensitive circuits is rather small. Only circuits composed of C-elements and inverters can be delay-insensitive [59], and the Muller pipeline in figures 2.5, 2.8, and 2.15 is one important example. Circuits that are delay-insensitive with the exception of some carefully identified wire forks where $d_2 = d_3$ are called *quasi-delay-insensitive* (QDI). Such wire forks, where signal transitions occur at the same time at all end-points, are called isochronic (and discussed in more detail in the next section). Typically these isochronic forks are found in gate-level implementations of basic building blocks where the designer can control the wire delays. At the higher levels of abstraction the composition of building blocks would typically be delay-insensitive. After these comments it is obvious that a distinction between DI, QDI and SI makes good sense.

Because the class of delay-insensitive circuits is so small, basically excluding all circuits that compute, most circuits that are referred to in the literature as delay-insensitive are only quasi-delay-insensitive.

Finally a word about *self-timed* circuits: speed-independence and delay-insensitivity as introduced above are (mathematically) well defined properties under the unbounded gate and wire delay model. Circuits whose correct operation relies on more elaborate and/or engineering timing assumptions are simply called self-timed.

2.5.3 Isochronic forks

From the above it is clear that the distinction between speed-independent circuits and delay-insensitive circuits relates to wire forks and, more specifically, to whether the delays to all end-points of a forking wire are identical or not. If the delays are identical, the wire-fork is called *isochronic*.

The need for isochronic forks is related to the concept of indication introduced in section 2.2. Consider a situation in figure 2.16 where gate A has changed its output. Eventually this change is observed on the inputs of gates B and C, and after some time gates B and C may respond to the new input by producing a new output. If this happens we say that the output change on gate A is indicated by output changes on gates B and C. If, on the other hand, only gate B responds to the new input, it is not possible to establish whether gate C has seen the input change as well. In this case it is necessary to strengthen the assumptions to $d_2 = d_3$ (i.e. that the fork is isochronic) and conclude that since the input signal change was indicated by the output of B, gate C has also seen the change.

2.5.4 Relation to circuits

In the 2-phase and 4-phase bundled-data approaches the control circuits are normally speed-independent (or in some cases even delay-insensitive), but the data-path circuits with their matched delays are self-timed. Circuits designed following the 4-phase dual-rail approach are generally quasi-delay-insensitive. In the circuits shown in figures 2.12 and 2.14 the forks that connect to the inputs of several C-elements must be isochronic, whereas the forks that connect to the inputs of several OR gates are delay-insensitive.

The different circuit classes, DI, QDI, SI and self-timed, are not mutually-exclusive ways to build complete systems, but useful abstractions that can be used at different levels of design. In most practical designs they are mixed. For example, in the Amulet processors [34, 33, 37] SI design is used for local asynchronous controllers, bundled-data for local data processing, and DI is used for high-level composition. Another example is the hearing-aid filter bank design presented in [77]. It uses the DI dual-rail 4-phase protocol inside RAM-modules and arithmetic circuits to provide robust completion indication,

and 4-phase bundled-data with SI control at the top levels of design, i.e. somewhat different from the Amulet designs. This emphasizes that the choice of handshake protocol and circuit implementation style is among the factors to consider when optimizing an asynchronous digital system.

It is important to stress that speed-independence and delay-insensitivity are mathematical properties that can be verified for a given implementation. If an abstract component – such as a C-element or a complex And-Or-Invert gate – is replaced by its implementation using simple gates and possibly some wire-forks, then the circuit may no longer be speed-independent or delay-insensitive. As an illustrative example we mention that the simple Muller pipeline stage in figures 2.8 and 2.15 is no longer delay-insensitive if the C-element is replaced by the gate-level implementation shown in figure 2.5 that uses simple AND and OR gates. Furthermore, even simple gates are abstractions; in CMOS the primitives are N and P transistors, and even the simplest gates include forks.

In chapter 6 we will explore the design of SI control circuits in great detail (because theory and synthesis tools are well developed). As SI circuits ignore wire delays completely some care is needed when physically implementing these circuits. In general one might think that the zero wire-delay assumption is trivially satisfied in small circuits involving 10-20 gates, but this need not be the case: a normal place and route CAD tool might spread the gates of a small controller all over the chip. Even if the gates are placed next to each other they may have different logic thresholds on their inputs which in combination with slowly rising or falling signals can cause (and have caused!) circuits to malfunction. For static CMOS and for circuits operating with low supply voltages (e.g. $V_{DD} \sim Vt_N + |Vt_P|$) this is less of a problem, but for dynamic circuits using a larger V_{DD} (e.g. 3.3 V or 5.0 V) the logic thresholds can be very different. This often overlooked problem is addressed in detail in [100].

2.6. Test

When it comes to the commercial exploitation of asynchronous circuits the problem of test comes to the fore. Test is a major topic in its own right, and it is beyond the scope of this tutorial to do anything more than mention a few issues and challenges. Although the following text is brief it assumes some knowledge of testing. The material does not constitute a foundation for the following chapters and it may be skipped.

The previous discussion about Muller circuits (excited gates and the firing of gates), the principle of indication, and the discussion of isochronoic forks ties in nicely with a discussion of testing for stuck at faults. In the stuck-at fault model defects are modeled at the gate level as (individual) inputs and outputs being stuck-at-1 or stuck-at-0. The principle of indication says that all input signal transitions on a gate must be indicated by an output signal transition on the gate. Furthermore, asynchronous circuits make extensive use of handshaking and this

causes signals to exhibit cyclic transitions between 0 and 1. In this scenario, the presence of a stuck-at fault is likely to cause the circuit to halt; if one component stops handshaking the stall tends to "propagate" to neighbouring components, and eventually the entire circuit halts. Consequently, the development of a set of test patterns that exhaustively tests for all stuck-at faults is simply a matter of developing a set of test patterns that toggle all nodes, and this is generally a comparatively simple task.

Since isochronic forks are forks where a signal transition in one or more branches is not indicated in the gates that take these signals as inputs, it follows that isochronic forks imply untestable stuck-at faults.

Testing asynchronous circuits incurs additional problems. As we will see in the following chapters, asynchronous circuits tend to implement registers using latches rather than flip-flops. In combination with the absence of a global clock, this makes it less straightforward to connect registers into scan-paths. Another consequence of the distributed self-timed control (i.e. the lack of a global clock) is that it is less straightforward to single-step the circuit through a sequence of well-defined states. This makes it less straightforward to steer the circuit into particular quiescent states, which is necessary for I_{DDQ} testing, – the technique that is used to test for shorts and opens which are faults that are typical in today's CMOS processes.

The extensive use of state-holding elements (such as the Muller C-element), together with the self-timed behaviour, makes it difficult to test the feed-back circuitry that implements the state holding behaviour. Delay-fault testing represents yet another challenge.

The above discussion may leave the impression that the problem of testing asynchronous circuits is largely unsolved. This is not correct. The truth is rather that the techniques for testing synchronous circuits are not directly applicable. The situation is quite similar to the design of asynchronous circuits that we will address in detail in the following chapters. Here a mix of new and well-known techniques are also needed. A good starting point for reading about the testing of asynchronous circuits is [90]. Finally, we mention that testing is also touched upon in chapters 13 and 15.

2.7. Summary

This chapter introduced a number of fundamental concepts. We will now return to the main track of designing circuits. The reader will probably want to revisit some of the material in this chapter again while reading the following chapters.

Chapter 3

STATIC DATA-FLOW STRUCTURES

In this chapter we will develop a high-level view of asynchronous design that is equivalent to RTL (register transfer level) in synchronous design. At this level the circuits may be viewed as static data-flow structures. The aim is to focus on the behaviour of the circuits, and to abstract away the details of the handshake signaling which can be considered an orthogonal implementation issue.

3.1. Introduction

The various handshake protocols and the associated circuit implementation styles presented in the previous chapters are rather different. However, when looking at the circuits at a more abstract level – the data-flow handshake-channel level introduced in chapter 1 – these differences diminish, and it makes good sense to view the choice of handshake protocol and circuit implementation style as low level implementation decisions that can be made largely independently from the more abstract design decisions that establish the overall structure and operation of the circuit.

Throughout this chapter we will assume a 4-phase protocol since this is most common. From a data-flow point of view this means that the we will be dealing with data streams composed of alternating valid and empty values – in a two-phase protocol we would see only a sequence of valid values, but apart from that everything else would be the same. Furthermore we will be dealing with simple latches as storage elements. The latches are controlled according to the simple rule stated in chapter 1:

> *A latch may input and store a new token (valid or empty) from its predecessor if its successor latch has input and stored the token that it was previously holding.*

Latches are the only components that initiate and take an active part in handshaking; all other components are "transparent" to the handshaking. To ease the distinction between latches and combinational circuits and to emphasize the token flow in circuit diagrams, we will use a box symbol with double vertical lines to represent latches throughout the rest of this tutorial (see figure 3.1).

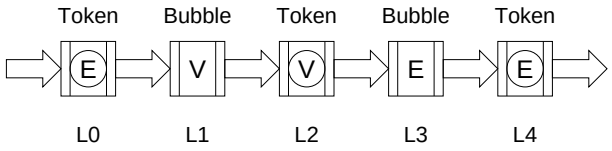


Figure 3.1. A possible state of a five stage pipeline.

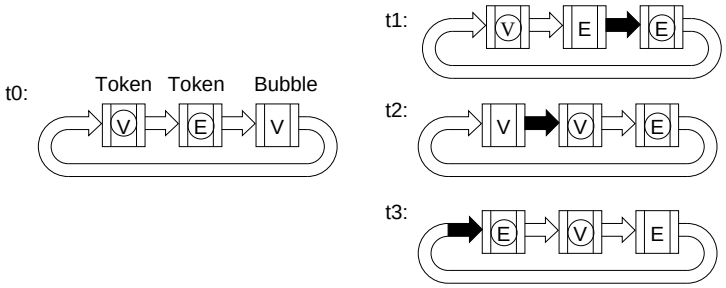


Figure 3.2. Ring: (a) a possible state; and (b) a sequence of data transfers.

3.2. Pipelines and rings

Figure 3.1 shows a snapshot of a pipeline composed of five latches. The "box arrows" represent channels or links consisting of request, acknowledge and data signals (as explained on page 5). The valid value in L1 has just been copied into L2 and the empty value in L3 has just been copied into L4. This means that L1 and L3 are now holding old duplicates of the values now stored in L2 and L4. Such old duplicates are called "bubbles", and the newest/rightmost valid and empty values are called "tokens". To distinguish tokens from bubbles, tokens are represented with a circle around the value. In this way a latch may hold a valid token, an empty token or a bubble. Bubbles can be viewed as catalysts: a bubble allows a token to move forward, and in supporting this the bubble moves backwards one step.

Any circuit should have one or more bubbles, otherwise it will be in a deadlock state. This is a matter of initializing the circuit properly, and we will elaborate on this shortly. Furthermore, as we will see later, the number of bubbles also has a significant impact on performance.

In a pipeline with at least three latches, it is possible to connect the output of the last stage to the input of the first, forming a ring in which data tokens can circulate autonomously. Assuming the ring is initialized as shown in figure 3.2(a) at time t_0 with a valid token, an empty token and a bubble, the first steps of the circulation process are shown in figure 3.2(b), at times t_1, t_2 and

t_3. Rings are the backbone structures of circuits that perform iterative computations. The cycle time of the ring in figure 3.2 is 6 "steps" (the state at t_6 will be identical to the state at t_0). Both the valid token and the empty token have to make one round trip. A round trip involves 3 "steps" and as there is only one bubble to support this the cycle time is 6 "steps". It is interesting to note that a 4-stage ring initialized to hold a valid token, an empty token and two bubbles can iterate in 4 "steps". It is also interesting to note that the addition of one more latch does not re-time the circuit or alter its function (as would be the case in a synchronous circuit); it is still a ring in which a single data token is circulating.

3.3. Building blocks

Figure 3.3 shows a minimum set of components that is sufficient to implement asynchronous circuits (static data-flow structures with deterministic behaviour, i.e. without arbiters). The components can be grouped in four categories as explained below. In the next section we will see examples of the token-flow behaviour in structures composed of these components. Components for mutual exclusion and arbitration are covered in section 5.8.

Latches provide storage for variables and implement the handshaking that supports the token flow. In addition to the normal latch a number of degenerate latches are often needed: a latch with only an output channel is a source that produces tokens (with the same constant value), and a latch with only an input channel is a sink that consumes tokens. Figure 2.9 shows the implementation of a 4-phase bundled-data latch, figure 2.11 shows the implementation of a 2-phase bundled-data latch, and figures 2.12 – 2.13 show the implementation of a 4-phase dual-rail latch.

Function blocks are the asynchronous equivalent of combinatorial circuits. They are transparent/passive from a handshaking point of view. A function block will: (1) wait for tokens on its inputs (an implicit join), (2) perform the required combinatorial function, and (3) issue tokens on its outputs. Both empty and valid tokens are handled in this way. Some implementations assume that the inputs have been synchronized. In this case it may be necessary to use an explicit join component. The implementation of function blocks is addressed in detail in chapter 5.

Unconditional flow control: Fork and join components are used to handle parallel threads of computation. In engineering terms, forks are used when the output from one component is input to more components, and joins are used when data from several independent channels needs to be synchronized – typically because they are (independent) inputs to a circuit. In the following we will often omit joins and forks from circuit

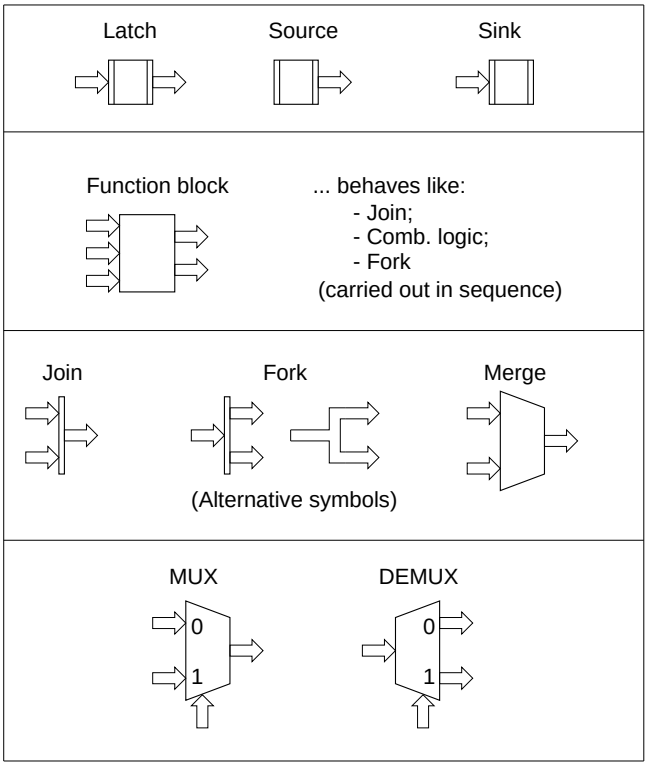


Figure 3.3. A minimum and, for most cases, sufficient set of asynchronous components.

diagrams: the fan-out of a channel implies a fork, and the fan-in of several channels implies a join.

A merge component has two or more input channels and one output channel. Handshakes on the input channels are assumed to be mutually exclusive and the merge relays input tokens/handshakes to the output.

Conditional flow control: MUX and DEMUX components perform the usual functions of selecting among several inputs or steering the input to one of several outputs. The control input is a channel just like the data inputs and outputs. A MUX will synchronize the control channel and the relevant input channel and send the input data to the data output. The other input channel is ignored. Similarly a DEMUX will synchronize the control and data input channels and steer the input to the selected output channel.

As mentioned before the latches implement the handshaking and thereby the token flow in a circuit. All other components must be transparent to the hand-

shaking. This has significant implications for the implementation of these components!

3.4. A simple example

Figure 3.4 shows an example of a circuit composed of latches, forks and joins that we will use to illustrate the token-flow behaviour of an asynchronous circuit. The structure can be described as pipeline segments and a ring connected into a larger structure using fork and join components.

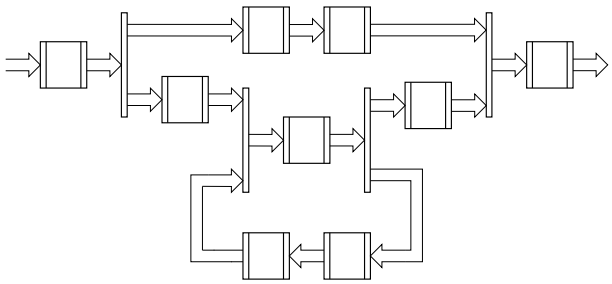


Figure 3.4. An example asynchronous circuit composed of latches, forks and joins.

Assume that the circuit is initialized as shown in figure 3.5 at time t_0: all latches are initialized to the empty value except for the bottom two latches in the ring that are initialized to contain a valid value and an empty value. Values enclosed in circles are tokens and the rest are bubbles. Assume further that the left and right hand environments (not shown) take part in the handshakes that the circuit is prepared to perform. Under these conditions the operation of the circuit (i.e. the flow of tokens) is as illustrated in the snapshots labeled $t_0 - t_{11}$. The left hand environment performs one handshake cycle inputting a valid value followed by an empty value. In a similar way the right environment takes part in one handshake cycle and consumes a valid value and an empty value.

Because the flow of tokens is controlled by local handshaking the circuit could exhibit many other behaviours. For example, at time t_5 the circuit is ready to accept a new valid value from its left environment. Notice also that if the initial state had no tokens in the ring, then the circuit would deadlock after a few steps. It is highly recommended that the reader tries to play the token-bubble data-flow game; perhaps using the same circuit but with different initial states.

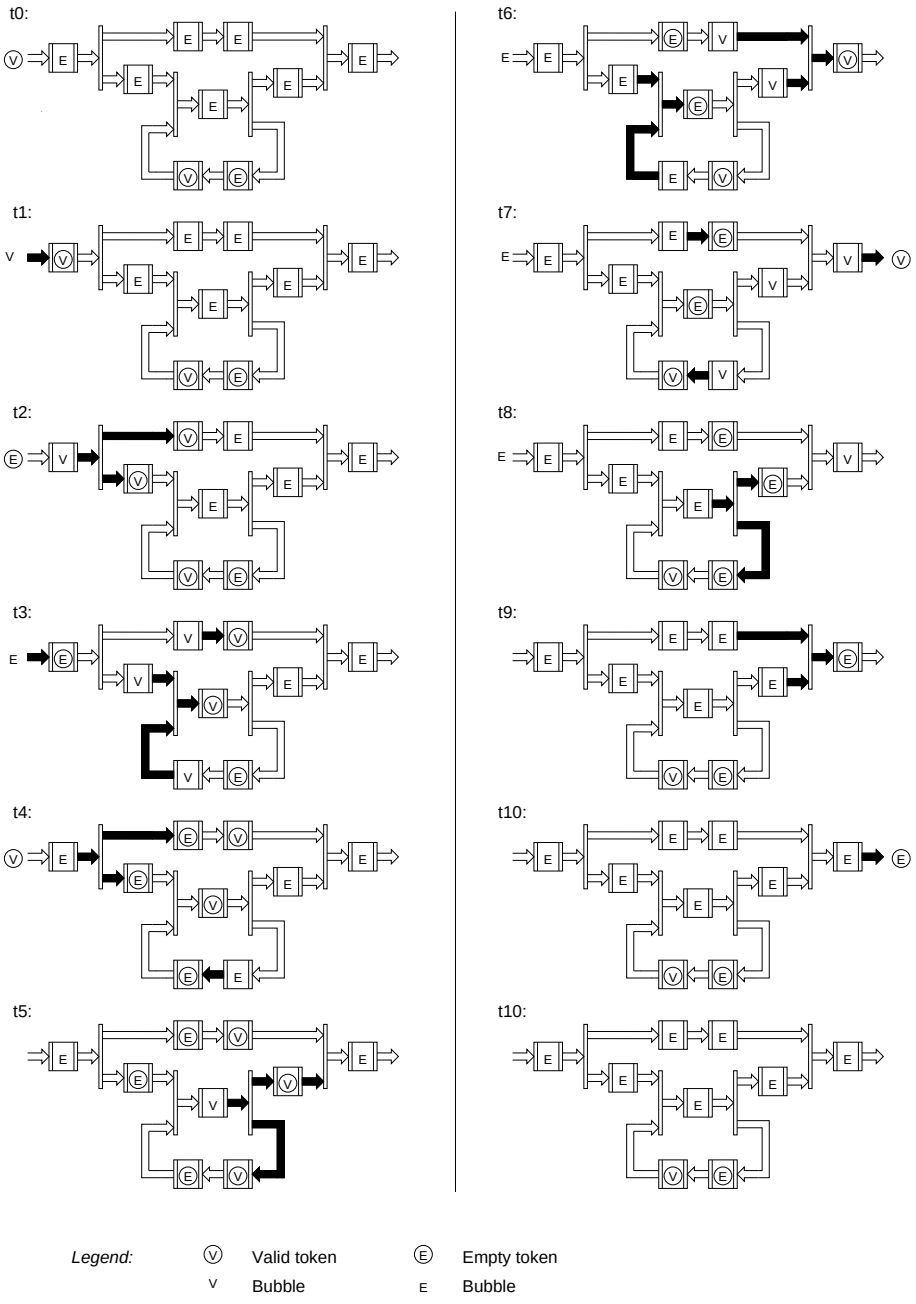


Figure 3.5. A possible operation sequence of the example circuit from figure 3.4.

3.5. Simple applications of rings

This section presents a few simple and obvious circuits based on a single ring.

3.5.1 Sequential circuits

Figure 3.6 shows a straightforward implementation of a finite state machine. Its structure is similar to a synchronous finite state machine; it consists of a function block and a ring that holds the current state. The machine accepts an "input token" that is joined with the "current state token". Then the function block computes the output and the next state, and finally the fork splits these into an "output token" and a "next state token."

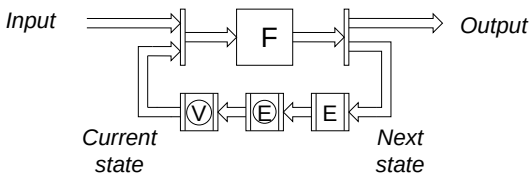


Figure 3.6. Implementation of an asynchronous finite state machine using a ring.

3.5.2 Iterative computations

A ring can also be used to build circuits that implement iterative computations. Figure 3.7 shows a template circuit. The idea is that the circuit will: (1) accept an operand, (2) sequence through the same operation a number of times until the computation terminates and (3) output the result. The necessary control is not shown. The figure shows one particular implementation. Possi-

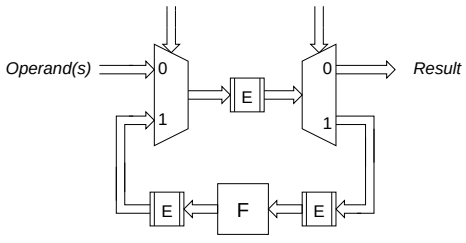


Figure 3.7. Implementation of an iterative computation using a ring.

ble variations involve locating the latches and the function block differently in the ring as well as decomposing the function block and putting these (simpler) function blocks between more latches. In [119] Ted Williams presents a circuit that performs division using a self-timed 5-stage ring. This design was later used in a floating point coprocessor in a commercial microprocessor [120].

3.6. FOR, IF, and WHILE constructs

Very often the desired function of a circuit is expressed using a programming language (C, C++, VHDL, Verilog, etc.). In this section we will show implementation templates for a number of typical conditional structures and loop structures. A reader who is familiar with control-data-flow graphs, perhaps from high-level synthesis, will recognize the great similarities between asynchronous circuits and control-data-flow graphs [27, 96].

if <cond> then <body1> else <body2> An asynchronous circuit template for implementing an *if* statement is shown in figure 3.8(a). The data-type of the input and output channels to the *if* circuit is a record containing all variables in the `<cond>` expression and the variables manipulated by `<body1>` and `<body2>`. The data-type of the output channel from the cond block is a Boolean that controls the DEMUX and MUX components. The FORK associated with this channel is not shown.

Since the execution of `<body1>` and `<body2>` is mutually exclusive it is possible to replace the controlled MUX in the bottom of the circuit with a simpler MERGE as shown in figure 3.8(b). The circuit in figure 3.8 contains no feedback loops and no latches – it can be considered a large function block. The circuit can be pipelined for improved performance by inserting latches.

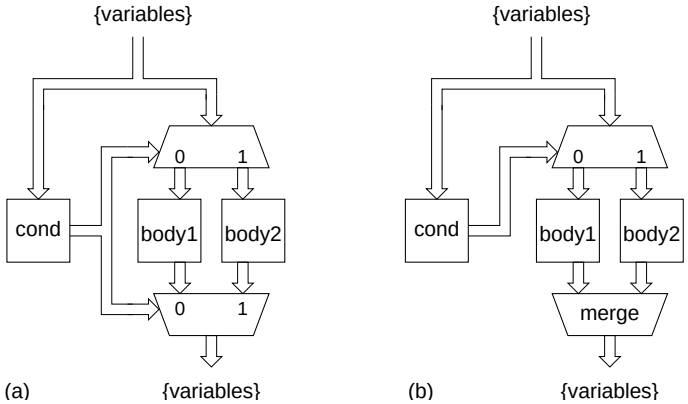


Figure 3.8. A template for implementing *if* statements.

for <count> do <body> An asynchronous circuit template for implementing a *for* statement is shown in figure 3.9. The data-type of the input channel to the *for* circuit is a record containing all variables manipulated in the `<body>` and the loop count, `<count>`, that is assumed to be a non-negative integer. The data-type of the output channel is a record containing all variables manipulated in the `<body>`.

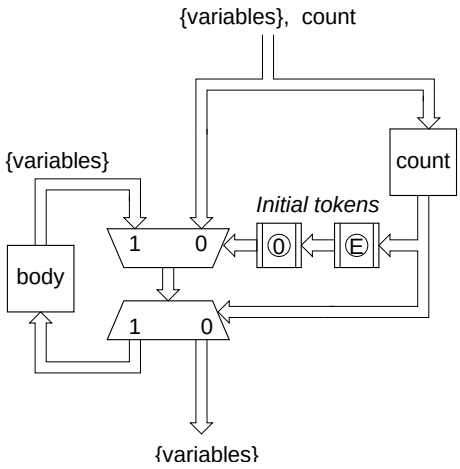


Figure 3.9. A template for implementing *for* statements.

The data-type of the output channel from the count block is a Boolean, and one handshake on the input channel of the count block encloses `<count>` handshakes on the output channel: `<count> - 1` handshakes providing the Boolean value "1" and one (final) handshake providing the Boolean value "0". Notice the two latches on the control input to the MUX. They must be initialized to contain a data token with the value "0" and an empty token in order to enable the *for* circuit to read the variables into the loop.

After executing the *for* statement once, the last handshake of the count block will steer the variables in the loop onto the output channel and put a "0" token and an empty token into the two latches, thereby preparing the *for* circuit for a subsequent activation. The FORK in the input and the FORK on the output of the count block are not shown. Similarly a number of latches are omitted. Remember: (1) all rings must contain at least 3 latches and (2) for each latch initialized to hold a data token there must also be a latch initialized to hold an empty token (when using 4-phase handshaking).

while <cond> do <body> An asynchronous circuit template for implementing a *while* statement is shown in figure 3.10. Inputs to (and outputs from) the circuit are the variables in the `<cond>` expression and the variables manipulated by `<body>`. As before in the *for* circuit, it is necessary to put two latches initialized to contain a data token with the value "0" and an empty token on the control input of the MUX. And as before a number of latches are omitted in the two rings that constitute the *while* circuit. When the *while* circuit terminates (after zero or more iterations) data is steered out of the loop and this also causes the latches on the MUX control input to become initialized properly for the subsequent activation of the circuit.

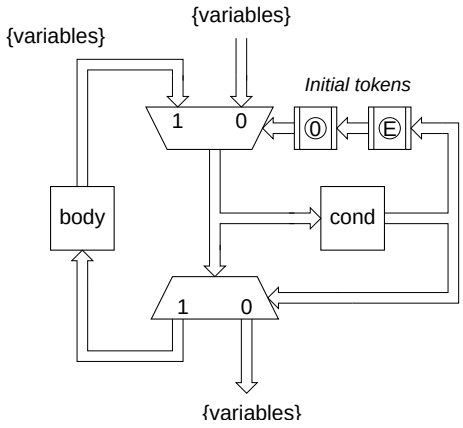


Figure 3.10. A template for implementing *while* statements.

3.7. A more complex example: GCD

Using the templates just introduced we will now design a small example circuit, GCD, that computes the greatest common divisor of two integers. GCD is often used as an introductory example, and figure 3.11 shows a programming language specification of the algorithm.

In addition to its role as a design example in the current context, GCD can also serve to illustrate the similarities and differences between different design techniques. In chapter 8 we will use the same example to illustrate the Tangram language and the associated syntax-directed compilation process (section 8.3.3 on pages 127–128).

The implementation of GCD is shown in figure 3.12. It consists of a *while* template whose body is an *if* template. Figure 3.12 shows the circuit including all the necessary latches (with their initial states). The implementation makes no attempt at sharing resources – it is a direct mapping following the implementation templates presented in the previous section.

```
input (a,b);
while a ≠ b do
    if a > b   then a ← a − b;
               else b ← b − a;
output (a);
```

Figure 3.11. A programming language specification of GCD.

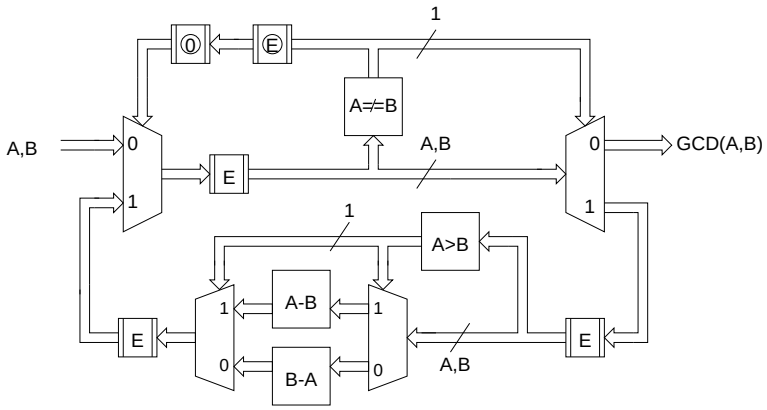


Figure 3.12. An asynchronous circuit implementation of GCD.

3.8. Pointers to additional examples

3.8.1 A low-power filter bank

In [77] we reported on the design of a low-power IFIR filter bank for a digital hearing aid. It is a circuit that was designed following the approach presented in this chapter. The paper also provides some insight into the design of low power circuits as well as the circuit level implementation of memory structures and datapath units.

3.8.2 An asynchronous microprocessor

In [15] we reported on the design of a MIPS microprocessor, called ARISC. Although there are many details to be understood in a large-scale design like a microprocessor, the basic architecture shown in figure 3.13 can be understood as a simple data-flow structure. The solid-black rectangles represent latches, the box-arrows represent channels, and the text-boxes represents function blocks (combinatorial circuits).

The processor is a simple pipelined design with instructions retiring in program order. It consists of a fetch-decode-issue ring with a fixed number of

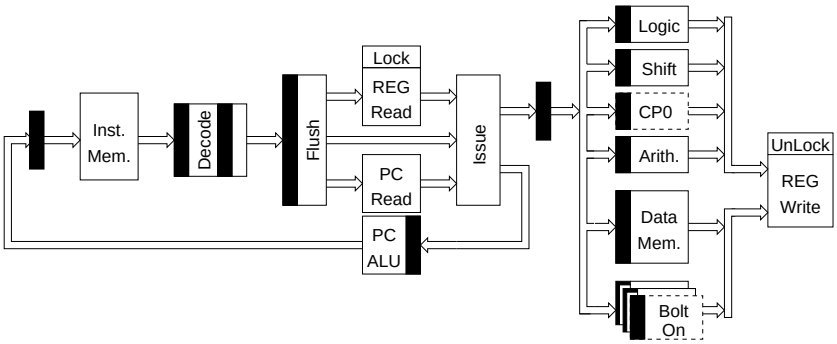


Figure 3.13. Architecture of the ARISC microprocessor.

tokens. This ensures a fixed instruction prefetch depth. The issue stage forks decoded instructions into the execute pipeline and initiates the fetch of one more instruction. Register forwarding is avoided by a locking mechanism: when an instruction is issued for execution the destination register is locked until the write-back has taken place. If a subsequent instruction has a read-after-write data hazard this instruction is stalled until the register is unlocked. The tokens flowing in the design contain all operands and control signals related to the execution of an instruction, i.e. similar to what is stored in a pipeline stage in a synchronous processor. For further information the interested reader is referred to [15]. Other asynchronous microprocessors are based on similar principles.

3.8.3 A fine-grain pipelined vector multiplier

The GCD circuit and the ARISC presented in the preceding sections use bit-parallel communication channels. An example of a static data-flow structure that uses 1-bit channels and fine grain pipelining is the serial-parallel vector multiplier design reported in [94, 95]. Here all necessary word-level synchronization is performed implicitly by the function blocks. The large number of interacting rings and pipeline segments in the static data-flow representation of the design makes it rather complex. After reading the next chapter on performance analysis the interested reader may want to look at this design; it contains several interesting optimizations.

3.9. Summary

This chapter developed a high-level view of asynchronous design that is equivalent to RTL (register transfer level) in synchronous design – static data flow structures. The next chapter address performance analysis at this level of abstraction.

Chapter 4

PERFORMANCE

In this chapter we will address the performance analysis and optimization of asynchronous circuits. The material extends and builds upon the "static data-flow structures view" introduced in the previous chapter.

4.1. Introduction

In a synchronous circuit, performance analysis and optimization is a matter of finding the longest latency signal path between two registers; this determines the period of the clock signal. The global clock partitions the circuit into many combinatorial circuits that can be analyzed individually. This is known as static timing analysis and it is a rather simple task, even for a large circuit.

For an asynchronous circuit, performance analysis and optimization is a global and therefore much more complex problem. The use of handshaking makes the timing in one component dependent on the timing of its neighbours, which again depends on the timing of their neighbours, etc. Furthermore, the performance of a circuit does not depend only on its structure, but also on how it is initialized and used by its environment. The performance of an asynchronous circuit can even exhibit transients and oscillations.

We will first develop a qualitative understanding of the dynamics of the token flow in asynchronous circuits. A good understanding of this is essential for designing circuits with good performance. We will then introduce some quantitative performance parameters that characterize individual pipeline stages and pipelines and rings composed of identical pipeline stages. Using these parameters one can make first-level design decisions. Finally we will address how more complex and irregular structures can be analyzed.

The following text represents a major revision of material from [94] and it is based on original work by Ted Williams [116, 117, 118]. If consulting these references the reader should be aware of the exact definition of a token. Throughout this book a token is defined as a valid data value *or* an empty data value, whereas in the cited references (that deal exclusively with 4-phase handshaking) a token is a valid-empty data pair. The definition used here accentuates the similarity between a token in an asynchronous circuit and the

token in a Petri net. Furthermore it provides some unification between 4-phase handshaking and 2-phase handshaking – 2-phase handshaking is the same game, but without empty-tokens.

In the following we will assume 4-phase handshaking, and the examples we provide all use bundled-data circuits. It is left as an exercise for the reader to make the simple adaptations that are necessary for dealing with 2-phase handshaking.

4.2. A qualitative view of performance

4.2.1 Example 1: A FIFO used as a shift register

The fundamental concepts can be illustrated by a simple example: a FIFO composed of a number of latches in which there are N valid tokens separated by N empty tokens, and whose environment alternates between reading a token from the FIFO and writing a token into the FIFO (see figure 4.1(a)). In this way the nomber of tokens in the FIFO is invariant. This example is relevant because many designs use FIFOs in this way, and because it models the behaviour of shift registers as well as rings – structures in which the number of tokens is also invariant.

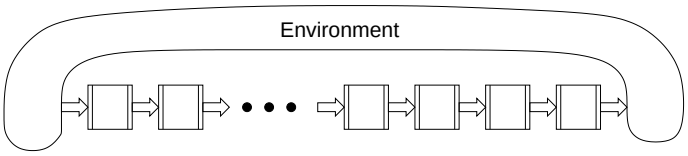
A relevant performance figure is the throughput, which is the rate at which tokens are input to or output from the shift register. This figure is proportional to the time it takes to shift the contents of the chain of latches one position to the right.

Figure 4.1(b) illustrates the behaviour of an implementation in which there are $2N$ latches per valid token and figure 4.1(c) illustrates the behaviour of an implementation in which there are $3N$ latches per valid token. In both examples the number of valid tokens in the FIFO is $N = 3$, and the only difference between the two situations in figure 4.1(b) and 4.1(c) is the number of bubbles.

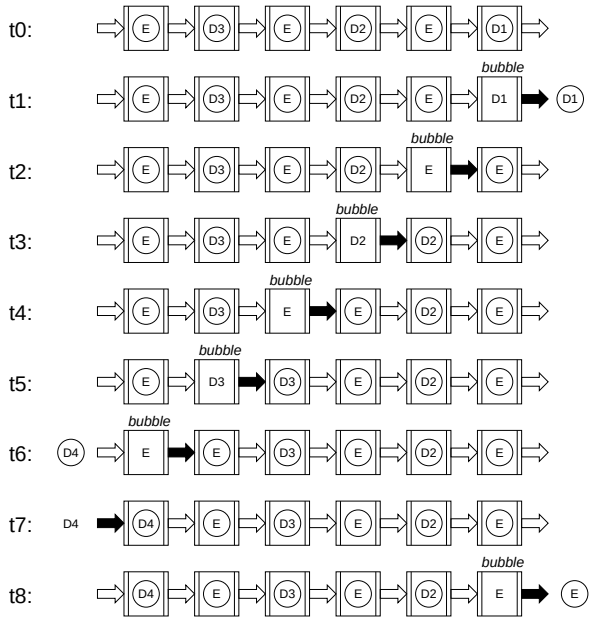
In figure 4.1(b) at time t_1 the environment reads the valid token, $D1$, as indicated by the solid channel symbol. This introduces a bubble that enables data transfers to take place one at a time ($t_2 - t_5$). At time t_6 the environment inputs a valid token, $D4$, and at this point all elements have been shifted one position to the right. Hence, the time used to move all elements one place to the right is proportional to the number of tokens, in this case $2N = 6$ time steps.

Adding more latches increases the number of bubbles, which again increases the number of data transfers that can take place simultaneously, thereby improving the performance. In figure 4.1(c) the shift register has $3N$ stages and therefore one bubble per valid-empty token-pair. The effect of this is that N data transfers can occur simultaneously and the time used to move all elements one place to the right is constant; 2 time steps.

(a) A FIFO and its environment:



(b) N data tokens and N empty tokens in 2N stages:



(c) N data tokens and N empty tokens in 3N stages:

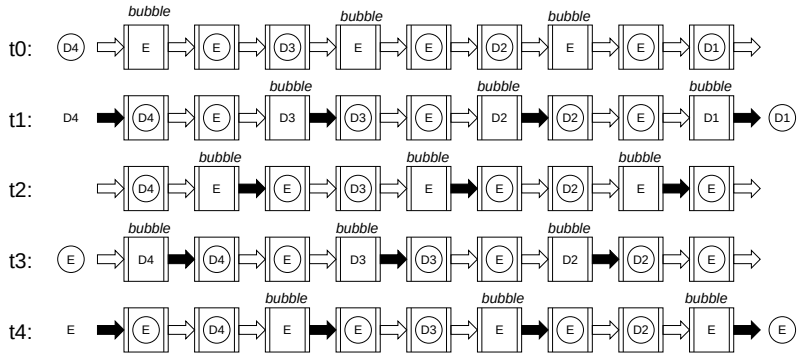


Figure 4.1. A FIFO and its environment. The environment alternates between reading a token from the FIFO and writing at token into the FIFO.

If the number of latches was increased to $4N$ there would be one token per bubble, and the time to move all tokens one step to the right would be only 1 time step. In this situation the pipeline is half full and the latches holding bubbles act as slave latches (relative to the latches holding tokens). Increasing the number of bubbles further would not increase the performance further. Finally, it is interesting to notice that the addition of just one more latch holding a bubble to figure 4.1(b) would double the performance. The asynchronous designer has great freedom in trading more latches for performance.

As the number of bubbles in a design depends on the number of latches per token, the above analysis illustrates that performance optimization of a given circuit is primarily a task of structural modification – circuit level optimization like transistor sizing is of secondary importance.

4.2.2 Example 2: A shift register with parallel load

In order to illustrate another point – that the distribution of tokens and bubbles in a circuit can vary over time, depending on the dynamics of the circuit and its environment – we offer another example: a shift register with parallel load. Figure 4.2 shows an initial design of a 4-bit shift register. The circuit has a bit-parallel input channel, din[3:0], connecting it to a data producing environment. It also has a 1-bit data channel, do, and a 1-bit control channel, ctl, connecting it to a data consuming environment. Operation is controlled by the data consuming environment which may request the circuit to: ($ctl = 0$) perform a parallel load *and* to provide the least significant bit from the bit-parallel channel on the do channel, or ($ctl = 1$) to perform a right shift and provide the next bit on the do channel. In this way the data consuming environment always inputs a control token (valid or empty) to which the circuit always responds by outputting a data token (valid or empty). During a parallel load, the previous content of the shift register is steered into the "dead end" sink-latches. During a right shift the constant 0 is shifted into the most significant position – corresponding to a logical right shift. The data consuming environment is not required to read all the input data bits, and it may continue reading zeros beyond the most significant input data bit.

The initial design shown in figure 4.2 suffers from two performance limiting inexpediencies: firstly, it has the same problem as the shift register in figure 4.1(b) – there are too few bubbles, and the peak data rate on the bit-serial output reduces linearly with the length of the shift register. Secondly, the control signal is forked to all of the MUXes and DEMUXes in the design. This implies a high fan-out of the request and data signals (which requires a couple of buffers) and synchronization of all the individual acknowledge signals (which requires a C-element with many inputs, possibly implemented as a tree of C-elements). The first problem can be avoided by adding a 3rd latch

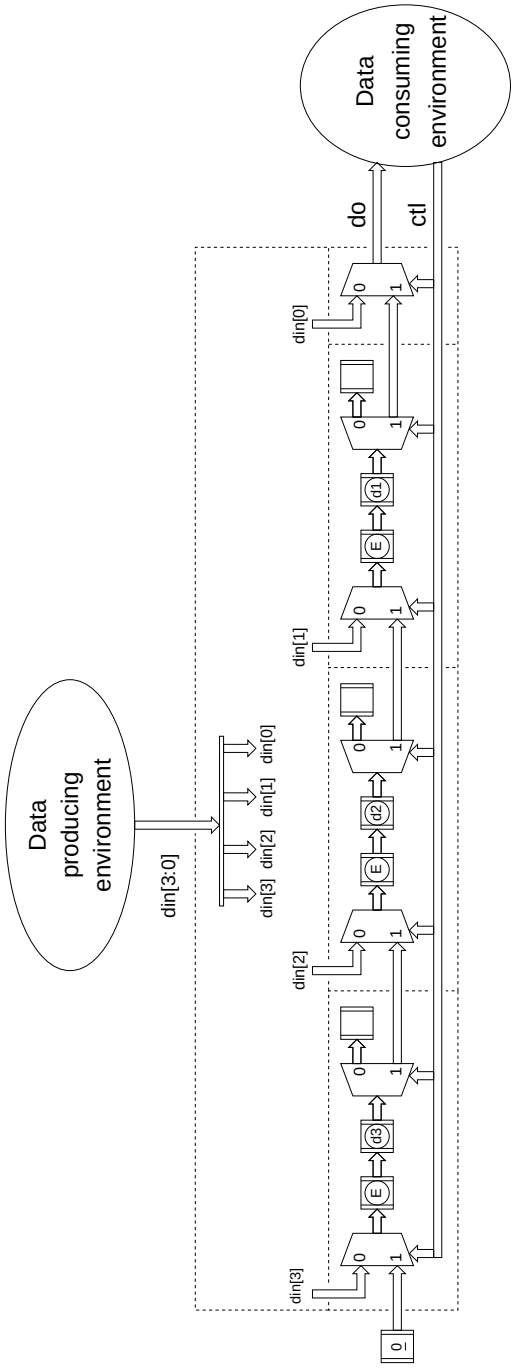


Figure 4.2. Initial design of the shift register with parallel load.

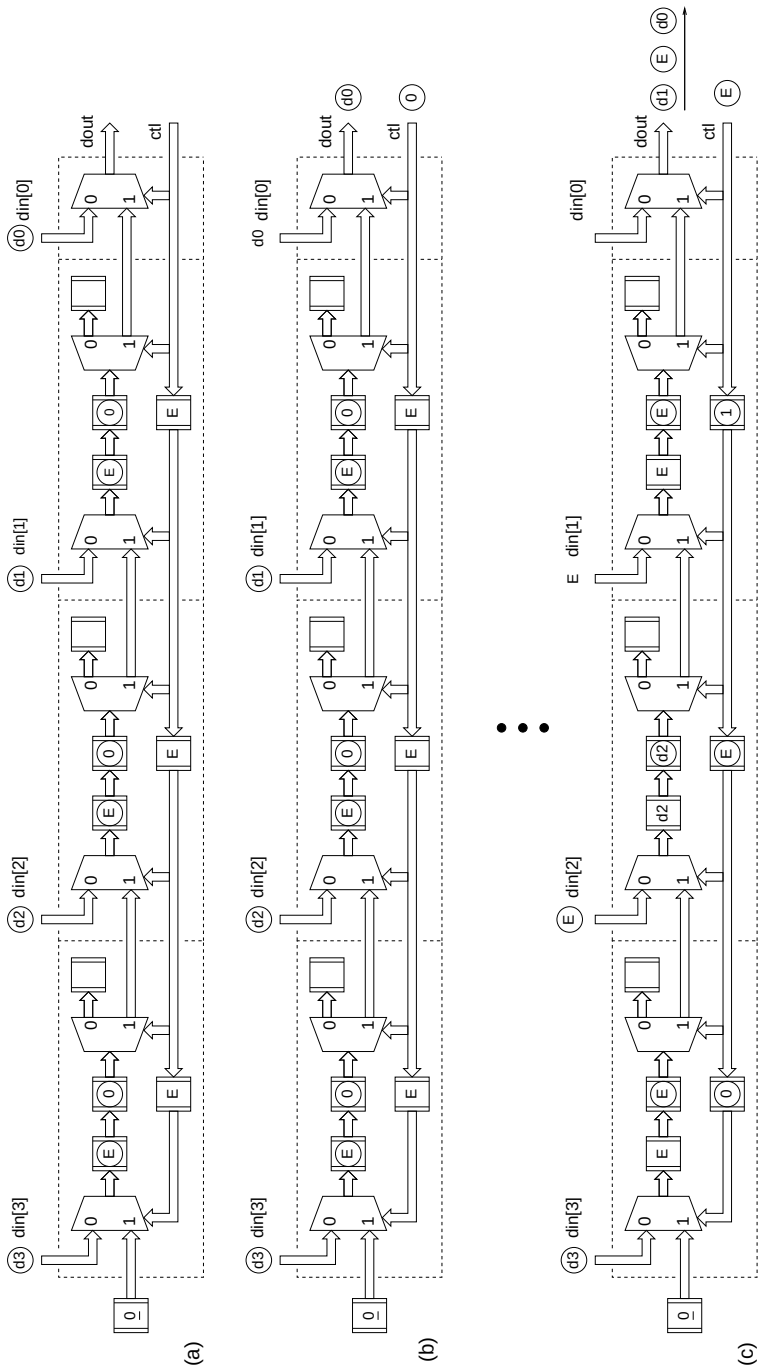


Figure 4.3. Improved design of the shift register with parallel load.

to the datapath in each stage of the circuit corresponding to the situation in figure 4.1(c), but if the extra latches are added to the control path instead, as shown in figure 4.3(a) on page 46, they will solve both problems.

This improved design exhibits an interesting and illustrative dynamic behaviour: initially, the data latches are densely packed with tokens and all the control latches contain bubbles, figure 4.3(a). The first step of the parallel load cycle is shown in figure 4.3(b), and figure 4.3(c) shows a possible state after the data consuming environment has read a couple of bits. The most-significant stage is just about to perform its "parallel load" and the bubbles are now in the chain of data latches. If at this point the data consuming environment paused, the tokens in the control path would gradually disappear while tokens in the datapath would pack again. Note that at any time the total number of tokens in the circuit is constant!

4.3. Quantifying performance

4.3.1 Latency, throughput and wavelength

When the overall structure of a design is being decided, it is important to determine the optimal number of latches or pipeline stages in the rings and pipeline fragments from which the design is composed. In order to establish a basis for first order design decisions, this section will introduce some quantitative performance parameters. We will restrict the discussion to 4-phase handshaking and bundled-data circuit implementations and we will consider rings with only a single valid token. Subsection 4.3.4, which concludes this section on performance parameters, will comment on adapting to other protocols and implementation styles.

The performance of a pipeline is usually characterized by two parameters: *latency* and *throughput* (or its inverse called *period* or *cycle time*). For an asynchronous pipeline a third parameter, the *dynamic wavelength*, is important as well. With reference to figure 4.4 and following [116, 117, 118] these parameters are defined as follows:

Latency: The latency is the delay from the input of a data item until the corresponding output data item is produced. When data flows in the forward direction, acknowledge signals propagate in the reverse direction. Consequently two parameters are defined:

- The *forward latency,* L_f, is the delay from new data on the input of a stage ($Data[i-1]$ or $Req[i-1]$) to the production of the corresponding output ($Data[i]$ or $Req[i]$) provided that the acknowledge signals are in place when data arrives. $L_{f.V}$ and $L_{f.E}$ denote the latencies for propagating a valid token and an empty token respectively. It is assumed that these latencies are constants, i.e. that they

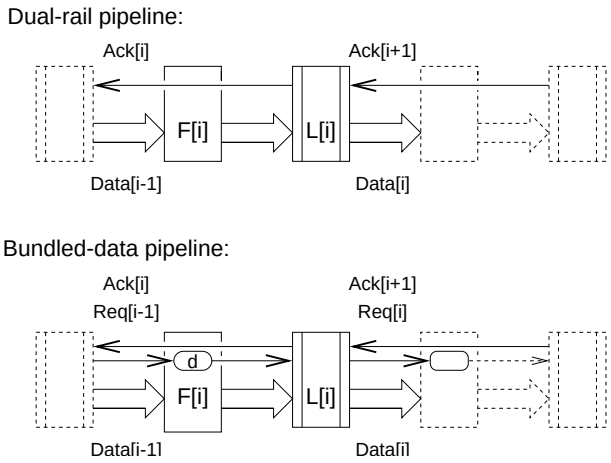


Figure 4.4. Generic pipelines for definition of performance parameters.

are independent of the value of the data. [As forward propagation of an empty token does not "compute" it may be desirable to minimize $L_{f.E}$. In the 4-phase bundled-data approach this can be achieved through the use of an asymmetric delay element.]

- The *reverse latency*, L_r, is the delay from receiving an acknowledge from the succeeding stage (Ack[i+1]) until the corresponding acknowledge is produced to the preceding stage (Ack[i]) provided that the request is in place when the acknowledge arrives. $L_r\downarrow$ and $L_r\uparrow$ denote the latencies of propagating $Ack\downarrow$ and $Ack\uparrow$ respectively.

Period: The period, P, is the delay between the input of a valid token (followed by its succeeding empty token) and the input of the next valid token, i.e. a complete handshake cycle. For a 4-phase protocol this involves: (1) forward propagation of a valid data value, (2) reverse propagation of acknowledge, (3) forward propagation of the empty data value, and (4) reverse propagation of acknowledge. Therefore a lower bound on the period is:

$$P = L_{f.V} + L_r\uparrow + L_{f.E} + L_r\downarrow \tag{4.1}$$

Many of the circuits we consider in this book are symmetric, i.e. $L_{f.V} = L_{f.E}$ and $L_r\uparrow = L_r\downarrow$, and for these circuits the period is simply:

$$P = 2L_f + 2L_r \tag{4.2}$$

We will also consider circuits where $L_{f.V} > L_{f.E}$ and, as we will see in section 4.4.1 and again in section 7.3, the actual implementation of

the latches may lead to a period that is larger than the minimum possible given by equation 4.1. In section 4.4.1 we analyze a pipeline whose period is:

$$P = 2L_r + 2L_{f.V} \tag{4.3}$$

Throughput: The *throughput,* T, is the number of valid tokens that flow through a pipeline stage per unit time: $T = 1/P$

Dynamic wavelength: The dynamic wavelength, W_d, of a pipeline is the number of pipeline stages that a forward-propagating token passes through during P:

$$W_d = \frac{P}{L_f} \tag{4.4}$$

Explained differently: W_d is the distance – measured in pipeline stages – between successive valid or empty tokens, when they flow unimpeded down a pipeline. Think of a valid token as the crest of a wave and its associated empty token as the trough of the wave. If $L_{f.V} \neq L_{f.E}$ the average forward latency $L_f = \frac{1}{2}(L_{f.V} + L_{f.E})$ should be used in the above equation.

Static spread: The static spread, S, is the distance – measured in pipeline stages – between successive valid (or empty) tokens in a pipeline that is full (i.e. contains no bubbles). Sometimes the term *occupancy* is used; this is the inverse of S.

4.3.2 Cycle time of a ring

The parameters defined above are local performance parameters that characterize the implementation of individual pipeline stages. When a number of pipeline stages are connected to form a ring, the following parameter is relevant:

Cycle time: The cycle time of a ring, T_{Cycle}, is the time it takes for a token (valid or empty) to make one round trip through all of the pipeline stages in the ring. To achieve maximum performance (i.e. minimum cycle time), the number of pipeline stages per valid token must match the dynamic wavelength, in which case $T_{Cycle} = P$. If the number of pipeline stages is smaller, the cycle time will be limited by the lack of bubbles, and if there are more pipeline stages the cycle time will be limited by the forward latency through the pipeline stages. In [116, 117, 118] these two modes of operation are called *bubble limited* and *data limited*, respectively.

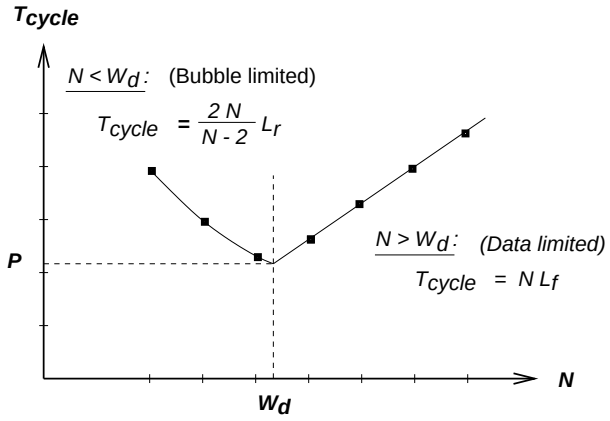


Figure 4.5. Cycle time of a ring as a function of the number of pipeline stages in it.

The cycle time of an N-stage ring in which there is one valid token, one empty token and $N-2$ bubbles can be computed from one of the following two equations (illustrated in figure 4.5):

- When $N \geq W_d$ the cycle time is limited by the forward latency through the N stages:

$$T_{Cycle}(DataLimited) = N \times L_f \tag{4.5}$$

  If $L_{f.V} \neq L_{f.E}$ use $L_f = max\{L_{f.V}; L_{f.E}\}$.

- When $N \leq W_d$ the cycle time is limited by the reverse latency. With N pipeline stages, one valid token and one empty token, the ring contains $N-2$ bubbles, and as a cycle involves $2N$ data transfers (N valid and N empty), the cycle time becomes:

$$T_{Cycle}(BubbleLimited) = \frac{2N}{N-2} L_r \tag{4.6}$$

  If $L_r\uparrow \neq L_r\downarrow$ use $L_r = \frac{1}{2}(L_r\uparrow + L_r\downarrow)$

For the sake of completeness it should be mentioned that a third possible mode of operation called *control limited* exists for some circuit configurations [116, 117, 118]. This is, however, not relevant to the circuit implementation configurations presented in this book.

The topic of performance analysis and optimization has been addressed in some more recent papers [22, 65, 66, 28] and in some of these the term "slack matching" is used (referring to the process of balancing the timing of forward flowing tokens and backward flowing bubbles).

4.3.3 Example 3: Performance of a 3-stage ring

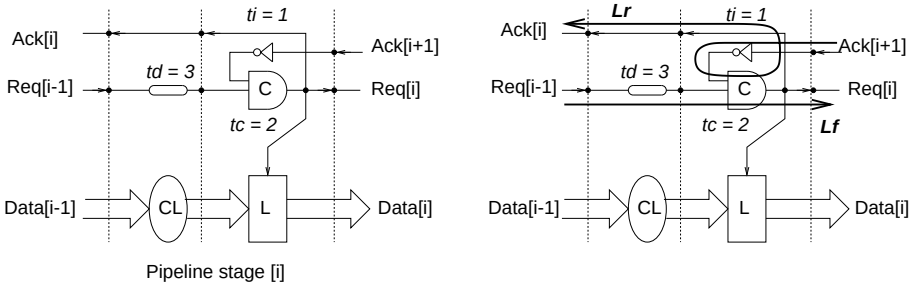


Figure 4.6. A simple 4-phase bundled-data pipeline stage, and an illustration of its forward and reverse latency signal paths.

Let us illustrate the above by a small example: a 3-stage ring composed of identical 4-phase bundled-data pipeline stages that are implemented as illustrated in figure 4.6(a). The data path is composed of a latch and a combinatorial circuit, CL. The control part is composed of a C-element and an inverter that controls the latch and a delay element that matches the delay in the combinatorial circuit. Without the combinatorial circuit and the delay element we have a simple FIFO stage. For illustrative purposes the components in the control part are assigned the following latencies: C-element: $t_c = 2$ ns, inverter: $t_i = 1$ ns, and delay element: $t_d = 3$ ns.

Figure 4.6(b) shows the signal paths corresponding to the forward and reverse latencies, and table 4.1 lists the expressions and the values of these parameters. From these figures the period and the dynamic wavelength for the two circuit configurations are calculated. For the FIFO, $W_d = 5.0$ stages, and for the pipeline, $W_d = 3.2$. A ring can only contain an integer number of stages and if W_d is not integer it is necessary to analyze rings with $\lfloor W_d \rfloor$ and $\lceil W_d \rceil$ stages

Table 4.1. Performance of different simple ring configurations.

	FIFO		Pipeline	
Parameter	*Expression*	*Value*	*Expression*	*Value*
L_r	$t_c + t_i$	3 ns	$t_c + t_i$	3 ns
L_f	t_c	2 ns	$t_c + t_d$	5 ns
$P = 2L_f + 2L_r$	$4t_c + 2t_i$	10 ns	$4t_c + 2t_i + 2t_d$	16 ns
W_d		5 stages		3.2 stages
T_{Cycle} (3 stages)	$6\ L_r$	18 ns	$6\ L_r$	18 ns
T_{Cycle} (4 stages)	$4\ L_r$	12 ns	$4\ L_f$	20 ns
T_{Cycle} (5 stages)	$3.3\ L_r = 5\ L_f$	10 ns	$5\ L_f$	25 ns
T_{Cycle} (6 stages)	$6\ L_f$	12 ns	$6\ L_f$	30 ns

and determine which yields the smallest cycle time. Table 4.1 shows the results of the analysis including cycle times for rings with 3 to 6 stages.

4.3.4 Final remarks

The above presentation made a number of simplifying assumptions: (1) only rings and pipelines composed of identical pipeline stages were considered, (2) it assumed function blocks with symmetric delays (i.e. circuits where $L_{f.V} = L_{f.E}$), (3) it assumed function blocks with constant latencies (i.e. ignoring the important issue of data-dependent latencies and average-case performance), (4) it considered rings with only a single valid token, and (5) the analysis considered only 4-phase handshaking and bundled-data circuits.

For 4-phase dual-rail implementations (where request is embedded in the data encoding) the performance parameter equations defined in the previous section apply without modification. For designs using a 2-phase protocol, some straightforward modifications are necessary: there are no empty tokens and hence there is only one value for the forward latency L_f and one value for the reverse latency L_r. It is also a simple matter to state expressions for the cycle time of rings with more tokens.

It is more difficult to deal with data-dependent latencies in the function blocks and to deal with non-identical pipeline stages. Despite these deficiencies the performance parameters introduced in the previous sections are very useful as a basis for first-order design decisions.

4.4. Dependency graph analysis

When the pipeline stages incorporate different function blocks, or function blocks with asymmetric delays, it is a more complex task to determine the critical path. It is necessary to construct a graph that represents the dependencies between signal transitions in the circuit, and to analyze this graph and identify the critical path cycle [11, 116, 117, 118]. This can be done in a systematic or even mechanical way but the amount of detail makes it a complex task.

The nodes in such a *dependency graph* represent rising or falling signal transitions, and the edges represent dependencies between the signal transitions. Formally, a dependency is a marked graph [20]. Let us look at a couple of examples.

4.4.1 Example 4: Dependency graph for a pipeline

As a first example let us consider a (very long) pipeline composed of identical stages using a function block with asymmetric delays causing $L_{f.E} < L_{f.V}$. Figure 4.7(a) shows a 3-stage section of this pipeline. Each pipeline stage has

the following latency parameters:

$$\begin{aligned} L_{f.V} &= t_{d(0\rightarrow 1)} + t_c = 5 \text{ ns} + 2 \text{ ns} = 7 \text{ ns} \\ L_{f.E} &= t_{d(1\rightarrow 0)} + t_c = 1 \text{ ns} + 2 \text{ ns} = 3 \text{ ns} \\ L_r\uparrow = L_r\downarrow &= t_i + t_c = 3 \text{ ns} \end{aligned}$$

There is a close relationship between the circuit diagram and the dependency graph. As signals alternate between rising transitions (↑) and falling transitions (↓) – or between valid and empty data values – the graph has two nodes per circuit element. Similarly the graph has two edges per wire in the circuit. Figure 4.7(b) shows the two graph fragments that correspond to a pipeline stage, and figure 4.7(c) shows the dependency graph that corresponds to the 3 pipeline stages in figure 4.7(a).

A label outside a node denotes the circuit delay associated with the signal transition. We use a particular style for the graphs that we find illustrative: the nodes corresponding to the forward flow of valid and empty data values are organized as two horizontal rows, and nodes representing the reverse flowing acknowledge signals appear as diagonal segments connecting the rows.

The cycle time or period of the pipeline is the time from a signal transition until the same signal transition occurs again. The cycle time can therefore be determined by finding the longest simple cycle in the graph, i.e. the cycle with the largest accumulated circuit delay which does not contain a sub-cycle. The dotted cycle in figure 4.7(c) is the longest simple cycle. Starting at point A the corresponding period is:

$$\begin{aligned} P &= \underbrace{t_{D(0\rightarrow 1)} + t_C}_{L_{f.V}} + \underbrace{t_I + t_C}_{L_r\downarrow} + \underbrace{t_{D(0\rightarrow 1)} + t_C}_{L_{f.V}} + \underbrace{t_I + t_C}_{L_r\uparrow} \\ &= 2L_r + 2L_{f.V} = 20 \text{ ns} \end{aligned}$$

Note that this is the period given by equation 4.3 on page 49. An alternative cycle time candidate is the following:

$$\underbrace{R_{[i]}\uparrow; Req_{[i]}\uparrow}_{L_{f.V}}; \underbrace{A_{[i-1]}\downarrow; Req_{[i-1]}\downarrow}_{L_r\downarrow}; \underbrace{R_{[i]}\downarrow; Req_{[i]}\downarrow}_{L_{f.E}}; \underbrace{A_{[i-1]}\uparrow; Req_{[i-1]}\uparrow}_{L_r\uparrow};$$

and the corresponding period is:

$$P = 2L_r + L_{f.V} + L_{f.E} = 16 \text{ ns}$$

Note that this is the minimum possible period given by equation 4.1 on page 48. The period is determined by the longest cycle which is 20 ns. Thus, this example illustrates that for some (simple) latch implementations it may not be possible to reduce the cycle time by using function blocks with asymmetric delays ($L_{f.E} < L_{f.V}$).

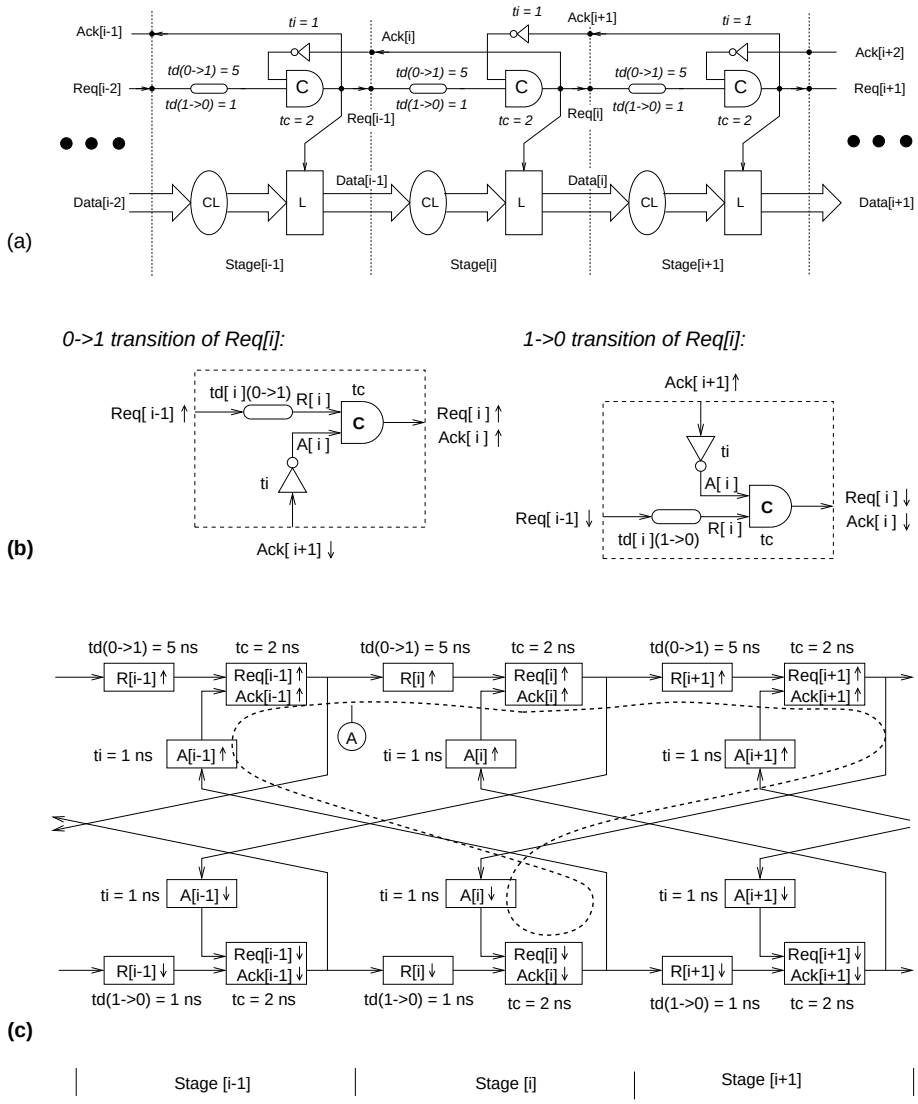


Figure 4.7. Data dependency graph for a 3-stage section of a pipeline: (a) the circuit diagram, (b) the two graph fragments corresponding to a pipeline stage, and (c) the resulting data-dependency graph.

4.4.2 Example 5: Dependency graph for a 3-stage ring

As another example of dependency graph analysis let us consider a three stage 4-phase bundled-data ring composed of different pipeline stages, figure 4.8(a): stage 1 with a combinatorial circuit that is matched by a symmetric delay el-

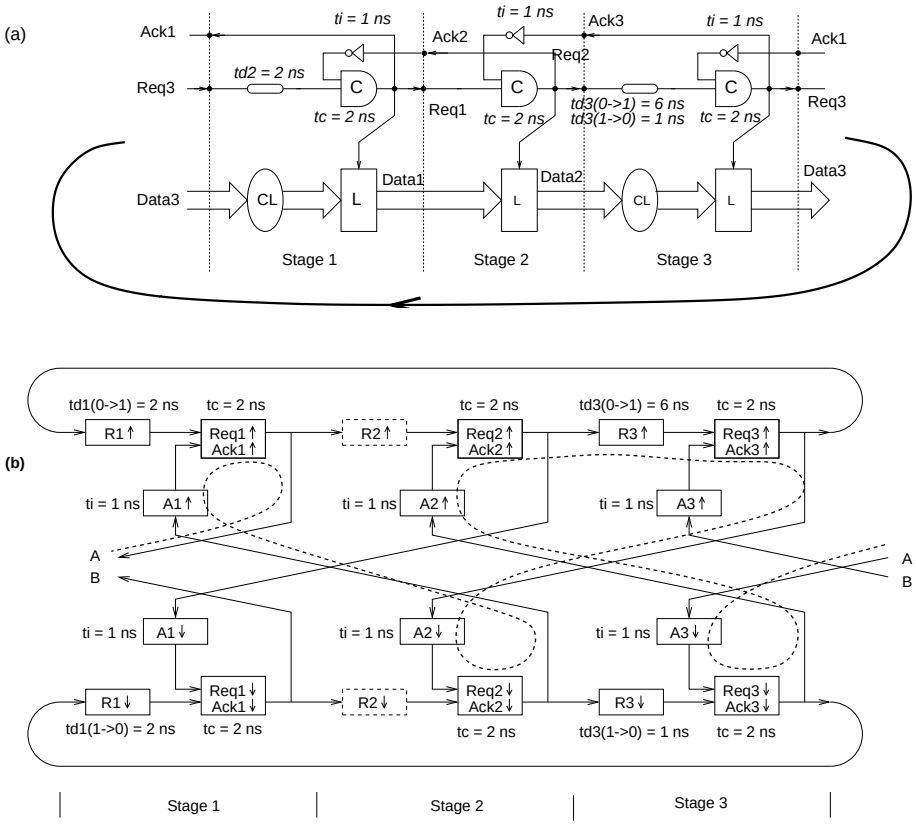


Figure 4.8. Data dependency graph for an example 3-stage ring: (a) the circuit diagram for the ring and (b) the resulting data-dependency graph.

ement, stage 2 without combinatorial logic, and stage 3 with a combinatorial circuit that is matched by an asymmetric delay element.

The dependency graph is similar to the dependency graph for the 3-stage pipeline from the previous section. The only difference is that the output port of stage 3 is connected to the input port of stage 1, forming a closed graph. There are several "longest simple cycle" candidates:

1 A cycle corresponding to the forward flow of valid-tokens:

(R1↑; Req1↑; R2↑; Req2↑; R3↑; Req3↑)

For this cycle, the cycle time is $T_{Cycle} = 14$ ns.

2 A cycle corresponding to the forward flow of empty-tokens:

(R1↓; Req1↓; R2↓; Req2↓; R3↓; Req3 ↓)

For this cycle, the cycle time is $T_{Cycle} = 9$ ns.

3 A cycle corresponding to the backward flowing bubble:

(A1↑; Req1↑; A3↓; Req3↓; A2↑; Req2↑; A1↓; Req1↓; A3↑; Req3↑; A2↓; Req2↓)

For this cycle, the cycle time is $T_{Cycle} = 6L_r = 18$ ns.

The 3-stage ring contains one valid-token, one empty-token and one bubble, and it is interesting to note that the single bubble is involved in six data transfers, and therefore makes two reverse round trips for each forward round trip of the valid-token.

4 There is, however, another cycle with a slightly longer cycle time, as illustrated in figure 4.8(b). It is the cycle corresponding to the backward-flowing bubble where the sequence:

(A1↓; Req1↓; A3↑) is replaced by (R3↑)

For this cycle the cycle time is $T_{Cycle} = 6L_r = 20$ ns.

A dependency graph analysis of a 4-stage ring is very similar. The only difference is that there are two bubbles in the ring. In the dependency graph this corresponds to the existence of two "bubble cycles" that do not interfere with each other.

The dependency graph approach presented above assumes a closed circuit that results in a closed dependency graph. If a component such as a pipeline fragment is to be analyzed it is necessary to include a (dummy) model of its environment as well – typically in the form of independent and eager token producers and token consumers, i.e. dummy circuits that simply respond to handshakes. Figure 2.15 on page 24 illustrated this for a single pipeline stage control circuit.

Note that a dependency graph as introduced above is similar to a signal transition graph (STG) which we will introduce more carefully in chapter 6.

4.5. Summary

This chapter addressed the performance analysis of asynchronous circuits at several levels: firstly, by providing a qualitative understanding of performance based on the dynamics of tokens flowing in a circuit; secondly, by introducing quantitative performance parameters that characterize pipelines and rings composed of identical pipeline stages and, thirdly, by introducing dependency graphs that enable the analysis of pipelines and rings composed of non-identical stages.

At this point we have covered the design and performance analysis of asynchronous circuits at the "static data-flow structures" level, and it is time to address low-level circuit design principles and techniques. This will be the topic of the next two chapters.

Chapter 5

HANDSHAKE CIRCUIT IMPLEMENTATIONS

In this chapter we will address the implementation of handshake components. First, we will consider the basic set of components introduced in section 3.3 on page 32: (1) the latch, (2) the unconditional data-flow control elements join, fork and merge, (3) function blocks, and (4) the conditional flow control elements MUX and DEMUX. In addition to these basic components we will also consider the implementation of mutual exclusion elements and arbiters and touch upon the (unavoidable) problem of metastability. The major part of the chapter (sections 5.3–5.6) is devoted to the implementation of function blocks and the material includes a number of fundamental concepts and circuit implementation styles.

5.1. The latch

As mentioned previously, the role of latches is: (1) to provide storage for valid and empty tokens, and (2) to support the flow of tokens via handshaking with neighbouring latches. Possible implementations of the handshake latch were shown in chapter 2: Figure 2.9 on page 18 shows how a 4-phase bundled-data handshake latch can be implemented using a conventional latch and a control circuit (the figure shows several such examples assembled into pipelines). In a similar way figure 2.11 on page 20 shows the implementation of a 2-phase bundled-data latch, and figures 2.12-2.13 on page 21 show the implementation of a 4-phase dual-rail latch.

A handshake latch can be characterized in terms of the *throughput*, the *dynamic wavelength* and the *static spread* of a FIFO that is composed of identical latches. Common to the two 4-phase latch designs mentioned above is that a FIFO will fill with every other latch holding a valid token and every other latch holding an empty token (as illustrated in figure 4.1(b) on page 43). Thus, the static spread for these FIFOs is $S = 2$.

A 2-phase implementation does not involve empty tokens and consequently it may be possible to design a latch whose static spread is $S = 1$. Note, however, that the implementation of the 2-phase bundled-data handshake latch

in figure 2.11 on page 20 involves several level-sensitive latches; the utilization of the level sensitive latches is no better.

Ideally, one would want to pack a valid token into every level-sensitive latch, and in chapter 7 we will address the design of 4-phase bundled-data handshake latches that have a smaller static spread.

5.2. Fork, join, and merge

Possible 4-phase bundled-data and 4-phase dual-rail implementations of the fork, join, and merge components are shown in figure 5.1. For simplicity the figure shows a fork with two output channels only, and join and merge components with two input channels only. Furthermore, all channels are assumed to be 1-bit channels. It is, of course, possible to generalize to three or more inputs and outputs respectively, and to extend to n-bit channels. Based on the explanation given below this should be straightforward, and it is left as an exercise for the reader.

4-phase fork and join A fork involves a C-element to combine the acknowledge signals on the output channels into a single acknowledge signal on the input channel. Similarly a 4-phase bundled-data join involves a C-element to combine the request signals on the input channels into a single request signal on the output channel. The 4-phase dual-rail join does not involve any active components as the request signal is encoded into the data.

The particular fork in figure 5.1 duplicates the input data, and the join concatenates the input data. This happens to be the way joins and forks are mostly used in our static data-flow structures, but there are many alternatives: for example, the fork could split the input data which would make it more symmetric to the join in figure 5.1. In any case the difference is only in how the input data is transferred to the output. From a control point of view the different alternatives are identical: a join synchronizes several input channels and a fork synchronizes several output channels.

4-phase merge The implementation of the merge is a little more elaborate. Handshakes on the input channels are mutually exclusive, and the merge simply relays the active input handshake to the output channel.

Let us consider the implementation of the 4-phase bundled-data merge first. It consists of an asynchronous control circuit and a multiplexer that is controlled by the input request. The control circuit is explained below.

The request signals on the input channels are mutually exclusive and may simply be ORed together to produce the request signal on the output channel.

For each input channel, a C-element produces an acknowledge signal in response to an acknowledge on the output channel provided that the input channel has valid data. For example, the C-element driving the x_{ack} signal is set high

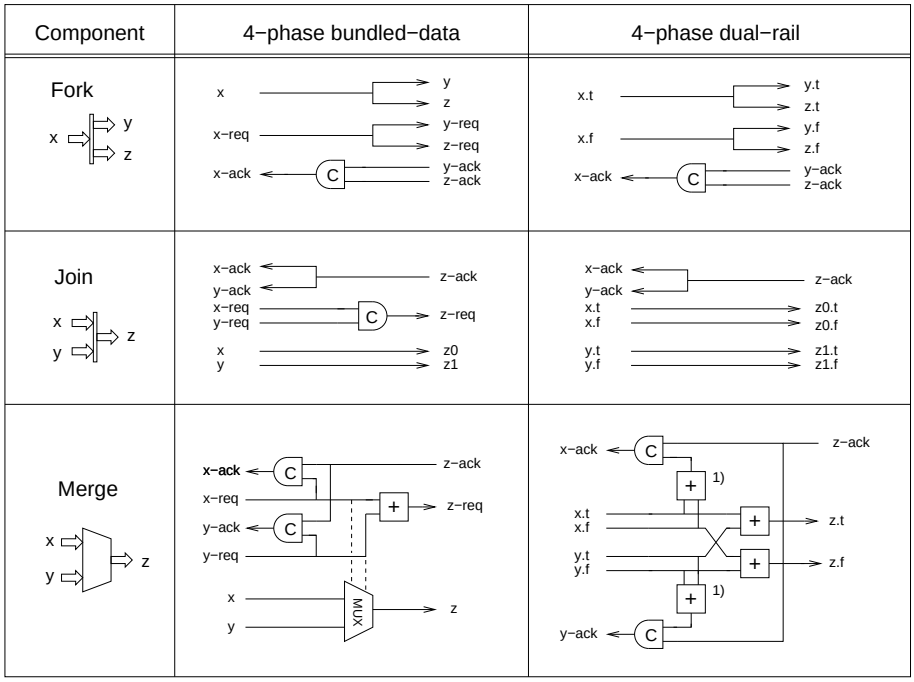


Figure 5.1. 4-phase bundled-data and 4-phase dual-rail implementations of the fork, join and merge components.

when x_{req} and z_{ack} have both gone high, and it is reset when both signals have gone low again. As z_{ack} goes low in response to x_{req} going low, it will suffice to reset the C-element in response to z_{ack} going low. This optimization is possible if asymmetric C-elements are available, figure 5.2. Similar arguments applies for the C-element that drives the y_{ack} signal. A more detailed introduction to generalized C-elements and related state-holding devices is given in chapter 6, sections 6.4.1 and 6.4.5.

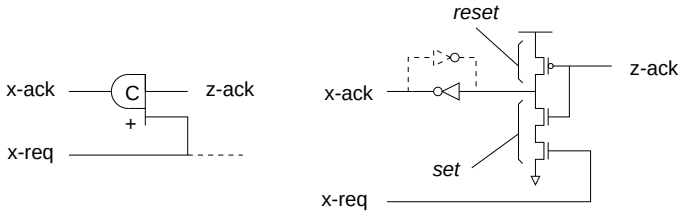


Figure 5.2. A possible implementation of the upper asymmetric C-element in the 4-phase bundled-data merge in figure 5.1.

The implementation of the 4-phase dual-rail merge is fairly similar. As request is encoded into the data signals an OR gate is used for each of the two output signals $z.t$ and $z.f$. Acknowledge on an input channel is produced in response to an acknowledge on the output channel provided that the input channel has valid data. Since the example assumes 1-bit wide channels, the latter is established using an OR gate (marked "1"), but for N-bit wide channels a completion detector (as shown in figure 2.13 on page 21) would be required.

2-phase fork, join and merge Finally a word about 2-phase bundled-data implementations of the fork, join and merge components: the implementation of 2-phase bundled-data fork and join components is identical to the implementation of the corresponding 4-phase bundled-data components (assuming that all signals are initially low).

The implementation of a 2-phase bundled-data merge, on the other hand, is complex and rather different, and it provides a good illustration of why the implementation of some 2-phase bundled-data components is complex. When observing an individual request or acknowledge signal the transitions will obviously alternate between rising and falling, but since nothing is known about the sequence of handshakes on the input channels there is no relationship between the polarity of a request signal transition on an input channel and the polarity of the corresponding request signal transition on the output channel. Similarly there is no relationship between the polarity of an acknowledge signal transition on the output channel and the polarity of the corresponding acknowledge signal transition on the input channel channel. This calls for some kind of storage element on each request and acknowledge signal produced by the circuit. This brings complexity, as does the associated control logic.

5.3. Function blocks – The basics

This section will introduce the fundamental principles of function block design, and subsequent sections will illustrate function block implementations for different handshake protocols. The running example will be an N-bit ripple carry adder.

5.3.1 Introduction

A function block is the asynchronous equivalent of a combinatorial circuit: it computes one or more output signals from a set of input signals. The term "function block" is used to stress the fact that we are dealing with circuits with a purely functional behaviour.

However, in addition to computing the desired function(s) of the input signals, a function block must also be transparent to the handshaking that is implemented by its neighbouring latches. This transparency to handshaking is what

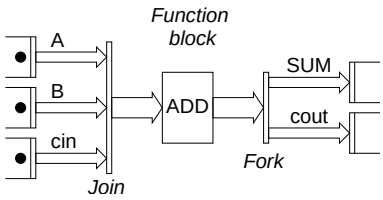


Figure 5.3. A function block whose operands and results are provided on separate channels requires a join of the inputs and a fork on the output.

makes function blocks different from combinatorial circuits and, as we will see, there are greater depths to this than is indicated by the word "transparent" – in particular for function blocks that implicitly indicate completion (which is the case for circuits using dual-rail signals).

The most general scenario is where a function block receives its operands on separate channels and produces its results on separate channels, figure 5.3. The use of several independent input and output channels implies a join on the input side and a fork on the output side, as illustrated in the figure. These can be implemented separately, as explained in the previous section, or they can be integrated into the function block circuitry. In what follows we will restrict the discussion to a scenario where all operands are provided on a single channel and where all results are provided on a single channel.

We will first address the issue of handshake transparency and then review the fundamentals of ripple carry addition, in order to provide the necessary background for discussing the different implementation examples that follow. A good paper on the design of function blocks is [71].

5.3.2 Transparency to handshaking

The general concepts are best illustrated by considering a 4-phase dual-rail scenario – function blocks for bundled data protocols can be understood as a special case. Figure 5.4(a) shows two handshake latches connected directly and figure 5.4(b) shows the same situation with a function block added between the two latches. The function block must be transparent to the handshaking. Informally this means that if observing the signals on the ports of the latches, one should see the same sequence of handshake signal transitions; the only difference should be some slow-down caused by the latency of the function block.

A function block is obviously not allowed to produce a request on its output before receiving a request on its input; put the other way round, a request on the output of the function block should indicate that all of the inputs are valid and that all (relevant) internal signals and all output signals have been computed.

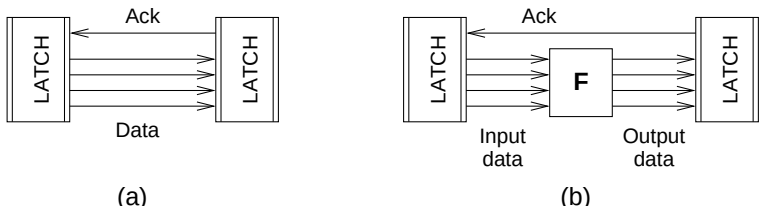
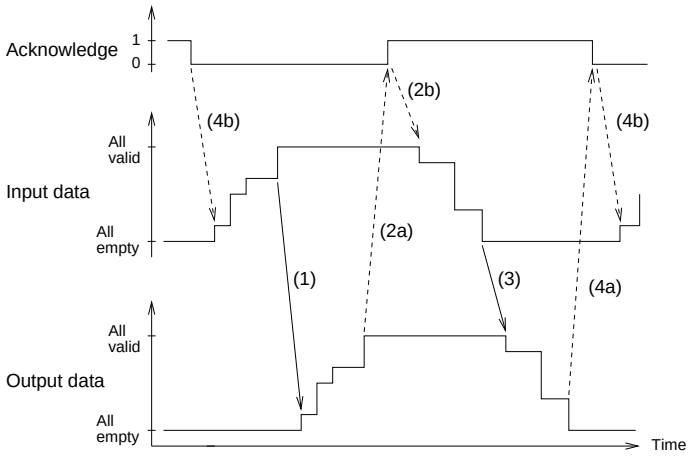


Figure 5.4. (a) Two latches connected directly by a handshake channel and (b) the same situation with a function block added between the latches. The handshaking as seen by the latches in the two situations should be the same, i.e. the function block must be designed such that it is transparent to the handshaking.

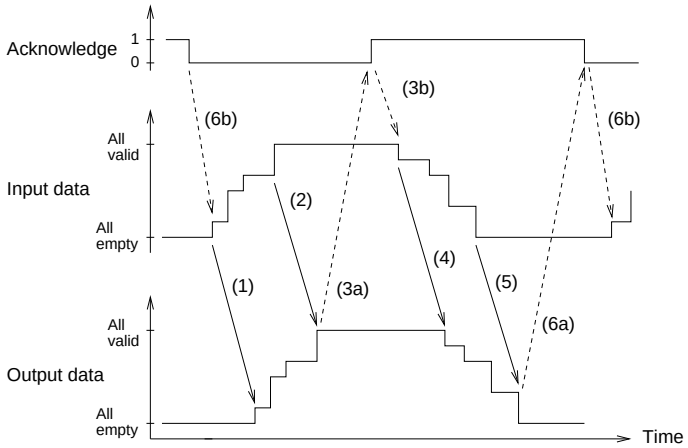
(Here we are touching upon the principle of indication once again.) In 4-phase protocols a symmetric set of requirements apply for the return-to-zero part of the handshaking.

Function blocks can be characterized as either *strongly indicating* or *weakly indicating* depending on how they behave with respect to this handshake transparency. The signalling that can be observed on the channel between the two



(1)	"All inputs become defined"	⇒	"Some outputs become defined"
(2)	"All outputs become defined"	⇒	"Some inputs become undefined"
(3)	"All inputs become undefined"	⇒	"Some outputs become undefined"
(4)	"All outputs become undefined"	⇒	"Some inputs become defined"

Figure 5.5. Signal traces and event orderings for a strongly indicating function block.



(1)	"Some inputs become defined"	$\prec$	"Some outputs become defined"
(2)	"All inputs become defined"	$\prec$	"All outputs become defined"
(3)	"All outputs become defined"	$\prec$	"Some inputs become undefined"
(4)	"Some inputs become undefined"	$\prec$	"Some outputs become undefined"
(5)	"All inputs become undefined"	$\prec$	"All outputs become undefined"
(6)	"All outputs become undefined"	$\prec$	"Some inputs become defined"

Figure 5.6. Signal traces and event orderings for a weakly indicating function block.

latches in figure 5.4(a) was illustrated in figure 2.3 on page 13. We can illustrate the handshaking for the situation in figure 5.4(b) in a similar way.

- A function block is *strongly indicating*, as illustrated in figure 5.5, if (1) it waits for all of its inputs to become valid before it starts to compute and produce valid outputs, and if (2) it waits for all of its inputs to become empty before it starts to produce empty outputs.
- A function block is *weakly indicating*, as illustrated in figure 5.6, if (1) it starts to compute and produce valid outputs as soon as possible, i.e. when some but not all input signals have become valid, and if (2) it starts to produce empty outputs as soon as possible, i.e. when some but not all input signals have become empty.

For a weakly indication function block to behave correctly, it is necessary to require that it never produces all valid outputs until after all inputs have become valid, and that it never produces all empty outputs until after all inputs have become empty. This behaviour is identical to Seitz's weak conditions in [91]. In [91] Seitz further explains that it can be proved that if the individual components satisfy the weak conditions then any "valid combinatorial circuit structure" of

function blocks also satisfies the weak conditions, i.e. that function blocks may be combined to form larger function blocks. By "valid combinatorial circuit structure" we mean a structure where no components have inputs or outputs left unconnected and where there are no feed-back signal paths. Strongly indicating function blocks have the same property – a "valid combinatorial circuit structure" of strongly indicating function blocks is itself a strongly indicating function block.

Notice that both weakly and strongly indicating function blocks exhibit a hysteresis-like behaviour in the valid-to-empty and empty-to-valid transitions: (1) some/all outputs must remain valid until after some/all inputs have become empty, and (2) some/all outputs must remain empty until after some/all inputs have become valid. It is this hysteresis that ensures handshake transparency, and the implementation consequence is that one or more state holding circuits (normally in the form of C-elements) are needed.

Finally, a word about the 4-phase bundled-data protocol. Since $Req\uparrow$ is equivalent to "all data signals are valid" and since $Req\downarrow$ is equivalent to "all data signals are empty," a 4-phase bundled-data function block can be categorized as strongly indicating.

As we will see in the following, strongly indicating function blocks have worst-case latency. To obtain actual case latency weakly indicating function blocks must be used. Before addressing possible function block implementation styles for the different handshake protocols it is useful to review the basics of binary ripple-carry addition, the running example in the following sections.

5.3.3 Review of ripple-carry addition

Figure 5.7 illustrates the implementation principle of a ripple-carry adder. A 1-bit full adder stage implements:

$$s = a \oplus b \oplus c \tag{5.1}$$

$$d = ab + ac + bc \tag{5.2}$$

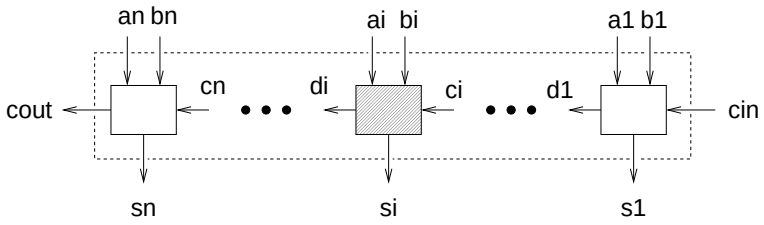


Figure 5.7. A ripple-carry adder. The carry output of one stage d_i is connected to the carry input of the next stage c_{i+1}.

In many implementations inputs a and b are recoded as:

$$p = a \oplus b \quad \text{(“propagate” carry)} \tag{5.3}$$
$$g = ab \quad \text{(“generate” carry)} \tag{5.4}$$
$$k = \overline{a}\overline{b} \quad \text{(“kill” carry)} \tag{5.5}$$

. . . and the output signals are computed as follows:

$$s = p \oplus c \tag{5.6}$$
$$d = g + pc \quad \text{or alternatively} \tag{5.7a}$$
$$\overline{d} = k + p\overline{c} \tag{5.7b}$$

For a ripple-carry adder, the worst case critical path is a carry rippling across the entire adder. If the latency of a 1-bit full adder is t_{add} the worst case latency of an N-bit adder is $N \cdot t_{add}$. This is a very rare situation and in general the longest carry ripple during a computation is much shorter. Assuming random and uncorrelated operands the average latency is $log(N) \cdot t_{add}$ and, if numerically small operands occur more frequently, the average latency is even less. Using normal Boolean signals (as in the bundled-data protocols) there is no way to know when the computation has finished and the resulting performance is thus worst-case.

By using dual-rail carry signals $(d.t, d.f)$ it is possible to design circuits that indicate completion as part of the computation and thus achieve actual case latency. The crux is that a dual-rail carry signal, d, conveys one of the following 3 messages:

$(d.t, d.f)$ = (0,0) = Empty	“The carry has not been computed yet” (possibly because it depends on c)
$(d.t, d.f)$ = (1,0) = True	“The carry is 1”
$(d.t, d.f)$ = (0,1) = False	“The carry is 0”

Consequently it is possible for a 1-bit adder to output a valid carry without waiting for the incoming carry if its inputs make this possible ($a = b = 0$ or $a = b = 1$). This idea was first put forward in 1955 in a paper by Gilchrist [38]. The same idea is explained in [46, pp. 75-78] and in [91].

5.4. Bundled-data function blocks

5.4.1 Using matched delays

A bundled-data implementation of the adder in figure 5.7 is shown in figure 5.8. It is composed of a traditional combinatorial circuit adder and a matching delay element. The delay element provides a constant delay that matches the *worst case* latency of the combinatorial adder. This includes the worst case

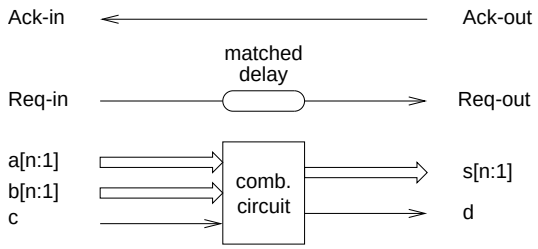


Figure 5.8. A 4-phase bundled data implementation of the $N-$bit handshake adder from figure 5.7.

critical path in the circuit – a carry rippling across the entire adder – as well as the worst case operating conditions. For reliable operation some safety margin is needed.

In addition to the combinatorial circuit itself, the delay element represents a design challenge for the following reasons: to a first order the delay element will track delay variations that are due to the fabrication process spread as well as variations in temperature and supply voltage. On the other hand, wire delays can be significant and they are often beyond the designer's control. Some design policy for matched delays is obviously needed. In a full custom design environment one may use a dummy circuit with identical layout but with weaker transistors. In a standard cell automatic place and route environment one will have to accept a fairly large safety margin or do post-layout timing analysis and trimming of the delays. The latter sounds tedious but it is similar to the procedure used in synchronous design where setup and hold times are checked and delays trimmed after layout.

In a 4-phase bundled-data design an asymmetric delay element may be preferable from a performance point of view, in order to perform the return-to-zero part of the handshaking as quickly as possible. Another issue is the power consumption of the delay element. In the ARISC processor design reported in [15] the delay elements consumed 10 % of the total power.

5.4.2 Delay selection

In [79] Nowick proposed a scheme called "speculative completion". The basic principle is illustrated in figure 5.9. In addition to the desired function some additional circuitry is added that selects among several matched delays. The estimate must be conservative, i.e. on the safe side. The estimation can be based on the input signals and/or on some internal signals in the circuit that implements the desired function.

For an N-bit ripple-carry adder the propagate signals (c.f. equation 5.3) that form the individual 1-bit full adders (c.f. figure 5.7) may be used for the estimation. As an example of the idea consider a 16-bit adder. If $p_8 = 0$ the

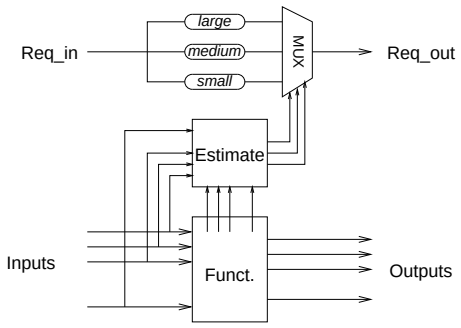


Figure 5.9. The basic idea of "speculative completion".

longest carry ripple can be no longer than 8 stages, and if $p_{12} \wedge p_8 \wedge p_4 = 0$ the longest carry ripple can be no longer than 4 stages. Based on such simple estimates a sufficiently large matched delay is selected. Again, if a 4-phase protocol is used, asymmetric delay elements are preferable from a performance point of view.

To the designer the trade-off is between an aggressive estimate with a large circuit overhead (area and power) or a less aggressive estimate with less overhead. For more details on the implementation and the attainable performance gains the reader is is referred to [79, 81].

5.5. Dual-rail function blocks

5.5.1 Delay insensitive minterm synthesis (DIMS)

In chapter 2 (page 22 and figure 2.14) we explained the implementation of an AND gate for dual-rail signals. Using the same basic topology it is possible to implement other simple gates such as OR, EXOR, etc. An inverter involves no active circuitry as it is just a swap of the two wires.

Arbitrary functions can be implemented by combining gates in exactly the same way as when one designs combinatorial circuits for a synchronous circuit. The handshaking is implicitly taken care of and can be ignored when composing gates and implementing Boolean functions. This has the important implication that existing logic synthesis techniques and tools may be used, the only difference is that the basic gates are implemented differently.

The dual-rail AND gate in figure 2.14 is obviously rather inefficient: 4 C-elements and 1 OR gate totaling approximately 30 transistors – a factor five greater than a normal AND gate whose implementation requires only 6 transistors. By implementing larger functions the overhead can be reduced. To illustrate this figure 5.10(b)-(c) shows the implementation of a 1-bit full adder. We will discuss the circuit in figure 5.10(d) shortly.

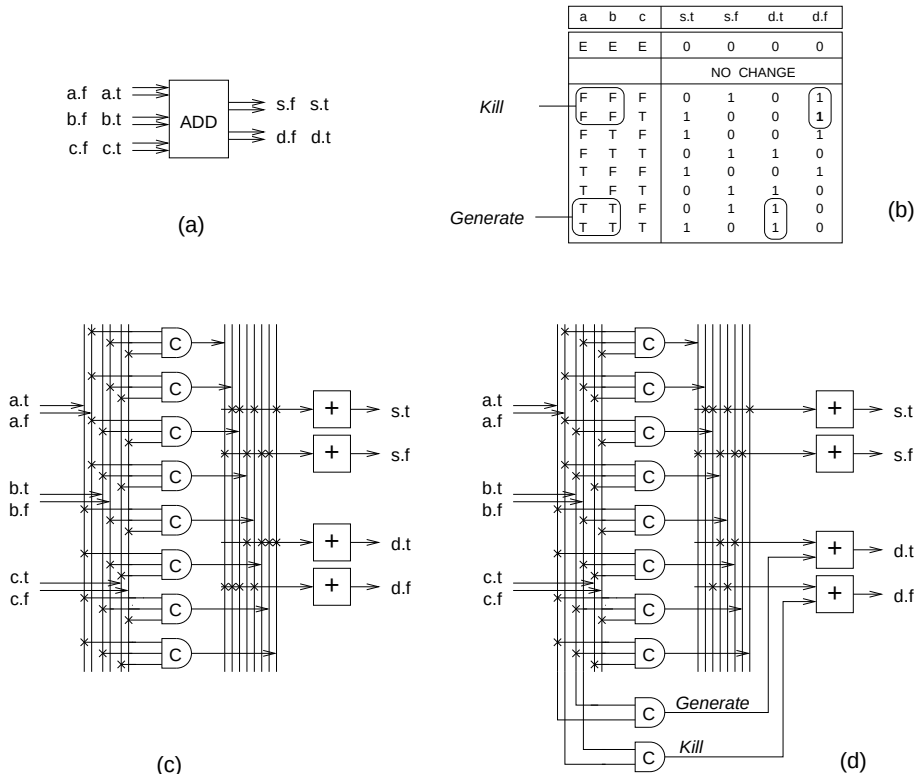


Figure 5.10. A 4-phase dual-rail full-adder: (a) Symbol, (b) truth table, (c) DIMS implementation and (d) an optimization that makes the full adder weakly indicating.

The PLA-like structure of the circuit in figure 5.10(c) illustrates a general principle for implementing arbitrary Boolean functions. In [94] we called this approach DIMS – Delay-Insensitive Minterm Synthesis – because the circuits are delay-insensitive and because the C-elements in the circuits generate all minterms of the input variables. The truth tables have 3 groups of rows specifying the output when the input is: (1) the empty codeword to which the circuit responds by setting the output empty, (2) an intermediate codeword which does not affect the output, or (3) a valid codeword to which the circuit responds by setting the output to the proper valid value.

The fundamental ideas explained above all go back to David Muller's work in the late 1950s and early 1960s [68, 67]. While [68] develops the fundamental theorem for the design of speed-independent circuits, [67] is a more practical introduction including a design example: a bit-serial multiplier using latches and gates as explained above.

Referring to section 5.3.2, the DIMS circuits as explained here can be categorized as strongly indicating, and hence they exhibit worst case latency. In

an N-bit ripple-carry adder the empty-to-valid and valid-to-empty transitions will ripple in strict sequence from the least significant full adder to the most significant one.

If we change the full-adder design slightly as illustrated in figure 5.10(d) a valid d may be produced before the c input is valid ("kill" or "generate"), and an N-bit ripple-carry adder built from such full adders will exhibit actual-case latency – the circuits are weakly indicating function blocks.

The designs in figure 5.10(c) and 5.10(d), and ripple-carry adders built from these full adders, are all symmetric in the sense that the latency of propagating an empty value is the same as the latency of propagating the preceding valid value. This may be undesirable. Later in section 5.5.4 we will introduce an elegant design that propagates empty values in constant time (with the latency of 2 full adder cells).

5.5.2 Null Convention Logic

The C-elements and OR gates from the previous sections can be seen as n-of-n and 1-of-n threshold gates with hysteresis, figure 5.11. By using arbitrary m-of-n threshold gates with hysteresis – an idea proposed by Theseus Logic, Inc., [29] – it is possible to reduce the implementation complexity. An m-of-n threshold gate with hysteresis will set its output high when any m inputs have gone high and it will set its output low when all its inputs are low. This elegant circuit implementation idea is the key element in Theseus Logic's Null Convention Logic. At the higher levels of design NCL is no different from the data-flow view presented in chapter 3 and NCL has great similarities to the circuit design styles presented in [67, 92, 94, 71]. Figure 5.11 shows that OR

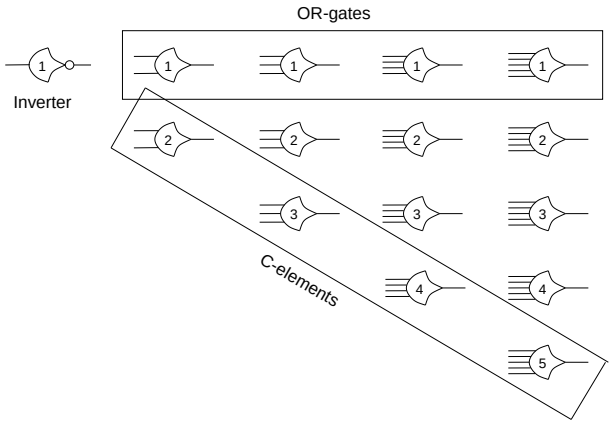


Figure 5.11. NCL gates: $m-$of$-n$ threshold gates with hysteresis ($1 \leq m \leq n$).

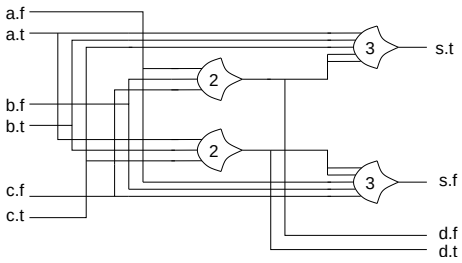


Figure 5.12. A full adder using NCL gates.

gates and C-elements can be seen as special cases in the world of threshold gates. The digit inside a gate symbol is the threshold of the gate. Figure 5.12 shows the implementation of a dual-rail full adder using NCL threshold gates. The circuit is weakly indicating.

5.5.3 Transistor-level CMOS implementations

The last two adder designs we will introduce are based on CMOS transistor-level implementations using dual-rail signals. Dual-rail signals are essentially what are produced by precharged differential logic circuits that are used in memory structures and in logic families like DCVSL, figure 5.13 [115, 41].

In a bundled-data design the precharge signal can be the request signal on the input channel to the function block. In a dual-rail design the precharge p-type transistors may be replaced by transistor networks that detect when all inputs

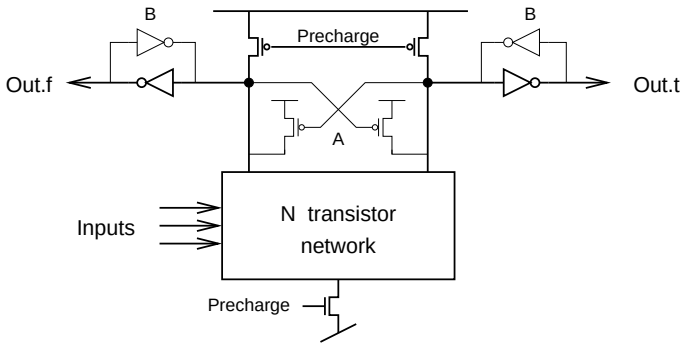


Figure 5.13. A precharged differential CMOS combinatorial circuit. By adding the cross-coupled p-type transistors labeled “A” or the (weak) feedback-inverters labeled “B” the circuit becomes (pseudo)static.

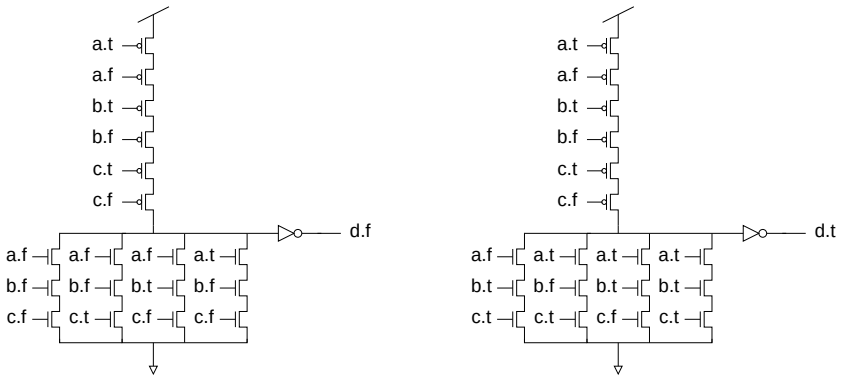


Figure 5.14. Transistor-level implementation of the carry signal for the strongly indicating full adder from figure 5.10(c).

are empty. Similarly the pull down n-type transistor signal paths should only conduct when the required input signals are valid.

Transistor implementations of the DIMS and NCL gates introduced above are thus straightforward. Figure 5.14 shows a transistor-level implementation of a carry circuit for a strongly-indicating full adder. In the pull-down circuit each column of transistors corresponds to a minterm. In general when implementing DCVSL gates it is possible to share transistors in the two pull-down networks, but in this particular case it has not been done in order to illustrate better the relationship between the transistor implementation and the gate implementation in figure 5.10(c).

The high stacks of p-type transistors are obviously undesirable. They may be replaced by a single transistor controlled by an "all empty" signal generated elsewhere. Finally, we mention that the weakly-indicating full adder design presented in the next section includes optimizations that minimize the p-type transistor stacks.

5.5.4 Martin's adder

In [61] Martin addresses the design of dual-rail function blocks in general and he illustrates the ideas using a very elegant dual-rail ripple-carry adder. The adder has a small transistor count, it exhibits actual case latency when adding valid data, and it propagates empty values in constant time – the adder represents the ultimate in the design of weakly indicating function blocks.

Looking at the weakly-indicating transistor-level carry circuit in figure 5.14 we see that d remains valid until a, b, and c are all empty. If we designed a similar sum circuit its output s would also remain valid until a, b, and c are all empty. The weak conditions in figure 5.6 only require that one output remains

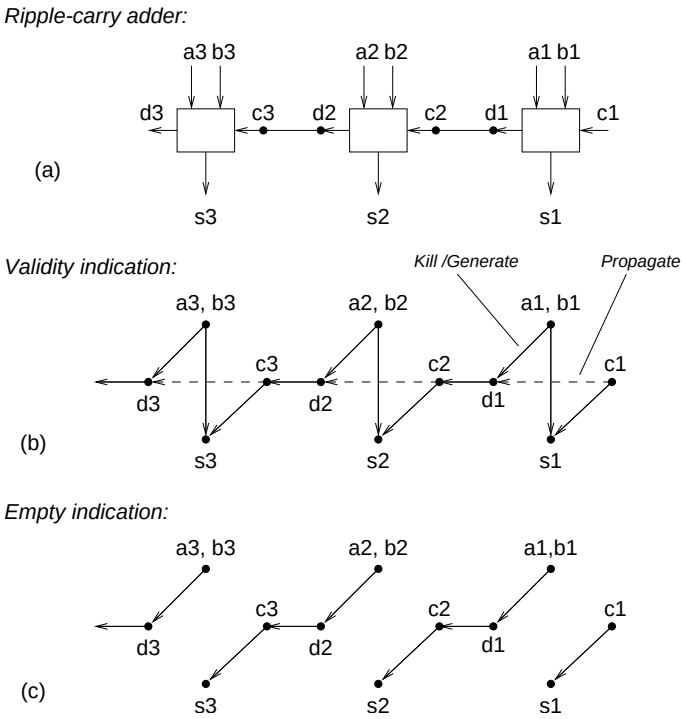


Figure 5.15. (a) A 3-stage ripple-carry adder and graphs illustrating how valid data (b) and empty data (c) propagate through the circuit (Martin [61]).

valid until all inputs have become invalid. Hence it is allowed to split the indication of a, b and c being empty among the carry and the sum circuits.

In [61] Martin uses some very illustrative directed graphs to express how the output signals indicate when input signals and internal signals are valid or empty. The nodes in the graphs are the signals in the circuit and the directed edges represent indication dependencies. Solid edges represent *guaranteed* dependencies and dashed edges represent *possible* dependencies. Figure 5.15(a) shows three full adder stages of a ripple-carry adder, and figures 5.15(b) and 5.15(c) show how valid and empty inputs respectively propagate through the circuit.

The propagation and indication of valid values is similar to what we discussed above in the other adder designs, but the propagation and indication of empty values is different and exhibits constant latency. When the outputs $d3$, $s3$, $s2$, and $s1$ are all valid this indicates that all input signals and all internal carry signals are valid. Similarly when the outputs $d3$, $s3$, $s2$, and $s1$ are all empty this indicates that all input signals and all internal carry signals are empty – the ripple-carry adder satisfies the weak conditions.

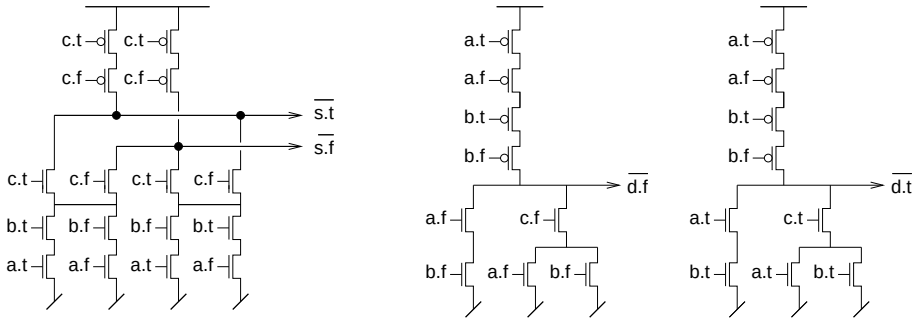


Figure 5.16. The CMOS transistor implementation of Martin's adder [61, Fig. 3].

The corresponding transistor implementation of a full adder is shown in figure 5.16. It uses 34 transistors, which is comparable to a traditional combinatorial circuit implementation.

The principles explained above apply to the design of function blocks in general. "Valid/empty indication (or acknowledgement), dependency graphs" as shown in figure 5.15 are a very useful technique for understanding and designing circuits with low latency and the weakest possible indication.

5.6. Hybrid function blocks

The final adder we will present has 4-phase bundled-data input and output channels and a dual-rail carry chain. The design exhibits characteristics similar to Martin's dual-rail adder presented in the previous section: actual case latency when propagating valid data, constant latency when propagating empty data, and a moderate transistor count. The basic structure of this hybrid adder is shown in figure 5.17. Each full adder is composed of a carry circuit and a sum circuit. Figure 5.18(a)-(b) shows precharged CMOS implementations of the two circuits. The idea is that the circuits precharge when $Req_{in} = 0$, evaluate when $Req_{in} = 1$, detect when all carry signals are valid and use this information to indicate completion, i.e. $Req_{out}\uparrow$. If the latency of the completion detector does not exceed the latency in the sum circuit in a full adder then a matched delay element is needed as indicated in figure 5.17.

The size and latency of the completion detector in figure 5.17 grows with the size of the adder, and in wide adders the latency of the completion detector may significantly exceed the latency of the sum circuit. An interesting optimization that reduces the completion detector overhead – possibly at the expense of a small increase in overall latency ($Req_{in}\uparrow$ to $Req_{out}\uparrow$) – is to use a mix of strongly and weakly indicating function blocks [75]. Following the naming convention established in figure 5.7 on page 64 we could make, for example,

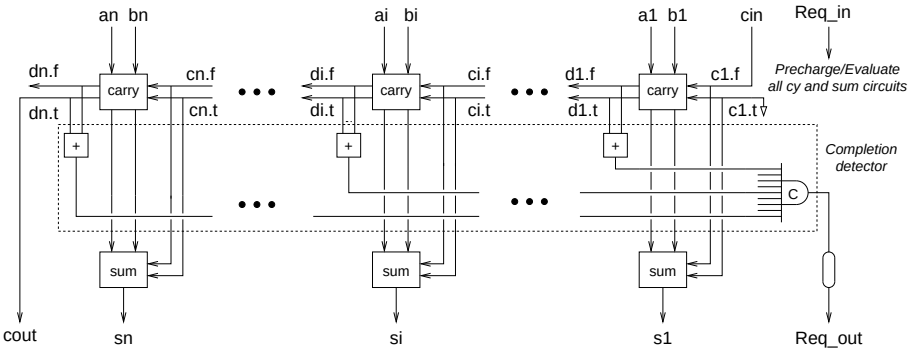


Figure 5.17. Block diagram of a hybrid adder with 4-phase bundled-data input and output channels and with an internal dual-rail carry chain.

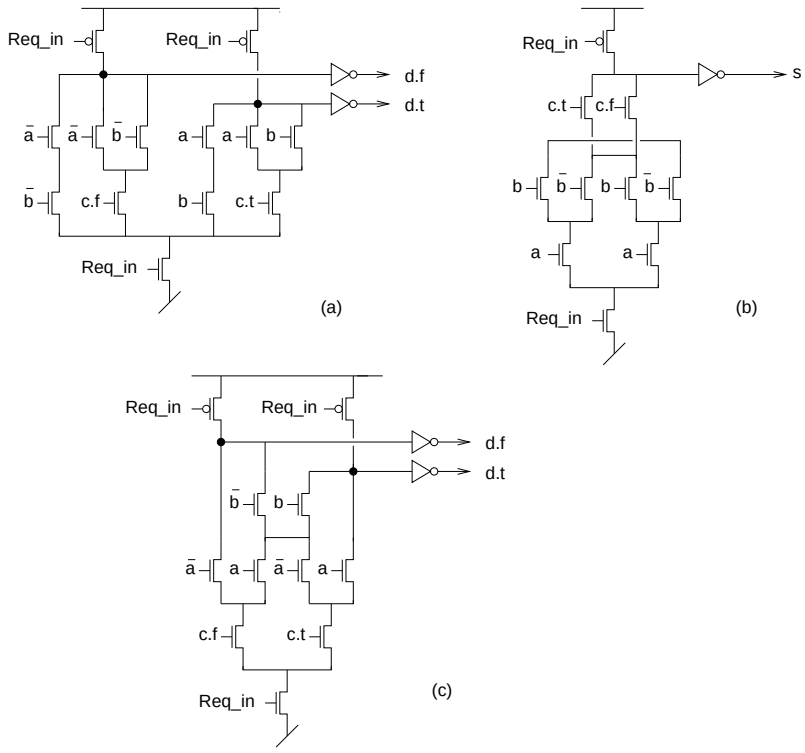


Figure 5.18. The CMOS transistor implementation of a full adder for the hybrid adder in figure 5.17: (a) a weakly indicating carry circuit, (b) the sum circuit and (c) a strongly indicating carry circuit.

adders 1, 4, 7, ... weakly indicating and all other adders strongly indicating. In this case only the carry signals out of stages 3, 6, 9, ... need to be checked to detect completion. For $i = 3, 6, 9, \ldots$ d_i indicates the completion of d_{i-1} and d_{i-2} as well. Many other schemes for mixing strongly and weakly indicating full adders are possible. The particular scheme presented in [75] exploited the fact that typical-case operands (sampled audio) are numerically small values, and the design detects completion from a single carry signal.

Summary – function block design

The previous sections have explained the basics of how to implement function blocks and have illustrated this using a variety of ripple-carry adders. The main points were "transparency to handshaking" and "actual case latency" through the use of weakly-indicating components.

Finally, a word of warning to put things into the right perspective: to some extent the ripple-carry adders explained above over-sell the advantages of average-case performance. It is easy to get carried away with elegant circuit designs but it may not be particularly relevant at the system level:

- In many systems the worst-case latency of a ripple-carry adder may simply not be acceptable.
- In a system with many concurrently active components that synchronize and exchange data at high rates, the slowest component at any given time tends to dominate system performance; the average-case performance of a system may not be nearly as good as the average-case latency of its individual components.
- In many cases addition is only one part of a more complex compound arithmetic operation. For example, the final design of the asynchronous filter bank presented in [77] did not use the ideas presented above. Instead we used entirely strongly-indicating full adders because this allowed an efficient two-dimensional precharged compound add-multiply-accumulate unit to be implemented.

5.7. MUX and DEMUX

Now that the principles of function block design have been covered we are ready to address the implementation of the MUX and DEMUX components, c.f. figure 3.3 on page 32. Let's recapitulate their function: a MUX will synchronize the control channel and relay the data and the handshaking of the selected input channel to the output data channel. The other input channel is ignored (and may have a request pending). Similarly a DEMUX will synchronize the control and the data input channels and steer the input to the selected output channel. The other output channel is passive and in the idle state.

If we consider only the "active" channels then the MUX and the DEMUX can be understood and designed as function blocks – they must be transparent to the handshaking in the same way as function blocks. The control channel and the (selected) input data channel are first joined and then an output is produced. Since no data transfer can take place without the control channel and the (selected) input data channel both being active, the implementations become strongly indicating function blocks.

Let's consider implementations using 4-phase protocols. The simplest and most intuitive designs use a dual-rail control channel. Figure 5.19 shows the implementation of the MUX and the DEMUX using the 4-phase bundled-data

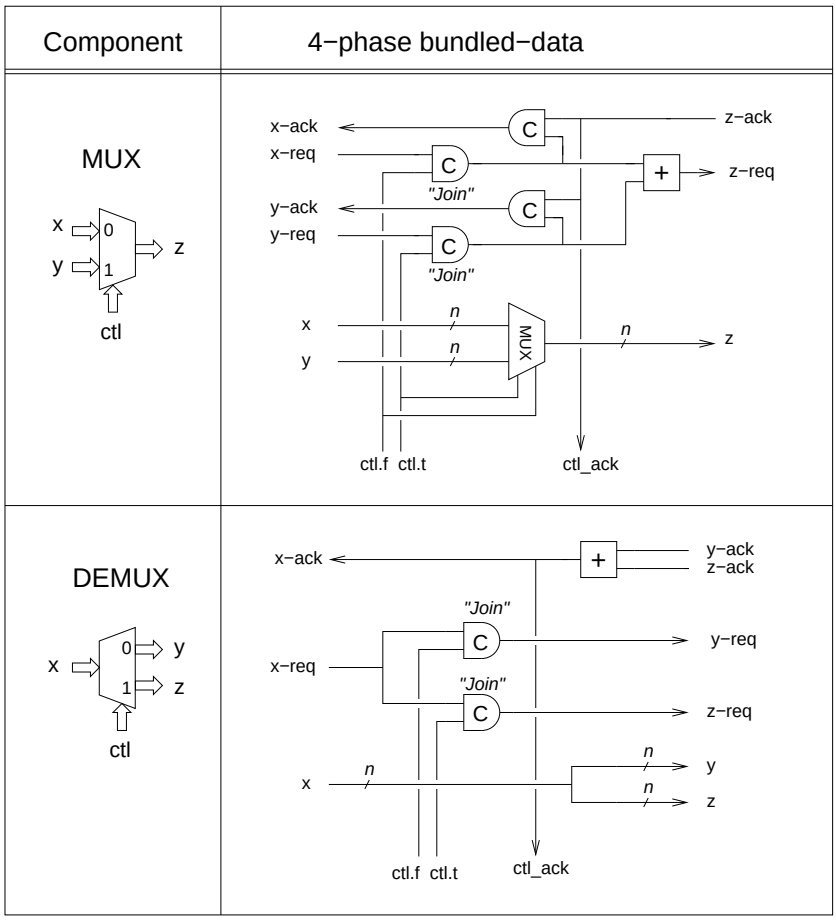


Figure 5.19. Implementation of MUX and DEMUX. The input and output data channels x, y, and z use the 4-phase bundled-data protocol and the control channel ctl uses the 4-phase dual-rail protocol (in order to simplify the design).

protocol on the input and output data channels and the 4-phase dual-rail protocol on the control channel. In both circuits $ctl.t$ and $ctl.f$ can be understood as two mutually exclusive requests that select between the two alternative input-to-output data transfers, and in both cases $ctl.t$ and $ctl.f$ are joined with the relevant input requests (at the C-elements marked "Join"). The rest of the MUX implementation is then similar to the 4-phase bundled-data MERGE in figure 5.1 on page 59. The rest of the DEMUX should be self explanatory; the handshaking on the two output ports are mutually exclusive and the acknowledge signals y_{ack} and z_{ack} are ORed to form $x_{ack} = ctl_{ack}$.

All 4-phase dual-rail implementations of the MUX and DEMUX components are rather similar, and all 4-phase bundled-data implementations may be obtained by adding 4-phase bundled-data to 4-phase dual-rail protocol conversion circuits on the control input. At the end of chapter 6, an all 4-phase bundled-data MUX will be one of the examples we use to illustrate the design of speed-independent control circuits.

5.8. Mutual exclusion, arbitration and metastability

5.8.1 Mutual exclusion

Some handshake components (including MERGE) require that the communication along several (input) channels is mutually exclusive. For the simple static data-flow circuit structures we have considered so far this has been the case, but in general one may encounter situations where a resource is shared between several independent parties/processes.

The basic circuit needed to deal with such situations is a mutual exclusion element (MUTEX), figure 5.20 (we will explain the implementation shortly). The input signals $R1$ and $R2$ are two requests that originate from two independent sources, and the task of the MUTEX is to pass these inputs to the corresponding outputs $G1$ and $G2$ in such a way that at most one output is active at any given time. If only one input request arrives the operation is trivial. If one input request arrives well before the other, the latter request is blocked until the first request is de-asserted. The problem arises when both input signals are asserted

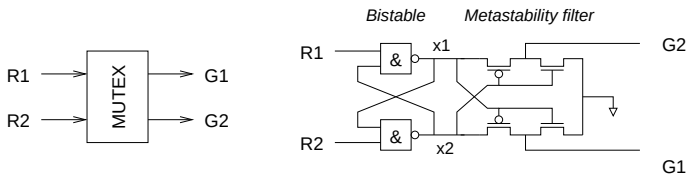


Figure 5.20. The mutual exclusion element: symbol and possible implementation.

at the same time. Then the MUTEX is required to make an arbitrary decision, and this is where metastability enters the picture.

The problem is exactly the same as when a synchronous circuit is exposed to an asynchronous input signal (one that does not satisfy set-up and hold time requirements). For a clocked flip-flop that is used to *synchronize* an asynchronous input signal, the question is whether the data signal made its transition before or after the active edge of the clock. As with the MUTEX the question is again which signal transition occured first, and as with the MUTEX a random decision is needed if the transition of the data signal coincides with the active edge of the clock signal.

The fundamental problem in a MUTEX and in a synchronizer flip-flop is that we are dealing with a bi-stable circuit that receives requests to enter each of its two stable states at the same time. This will cause the circuit to enter a metastable state in which it may stay for an unbounded length of time before randomly settling in one of its stable states. The problem of synchronization is covered in most textbooks on digital design and VLSI, and the analysis of metastability that is presented in these textbooks applies to our MUTEX component as well. A selection of references is: [70, sect. 9.4] [39, sect. 5.4 and 6.5] [115, sect. 5.5.7] [88, sect. 6.2.2 and 9.4-5] [114, sect. 8.9].

For the synchronous designer the problem is that metastability may persist beyond the time interval that has been allocated to recover from potential metastability. It is simply not possible to obtain a decision within a bounded length of time. The asynchronous designer, on the other hand, will eventually obtain a decision, but there is no upper limit on the time he will have to wait for the answer. In [14] the terms "time safe" and "value safe" are introduced to denote and classify these two situations.

A possible implementation of the MUTEX, as shown in figure 5.20, involves a pair of cross coupled NAND gates and a metastability filter. The cross coupled NAND gates enable one input to block the other. If both inputs are asserted at the same time, the circuit becomes metastable with both signals $x1$ and $x2$ halfway between supply and ground. The metastability filter prevents these undefined values from propagating to the outputs; $G1$ and $G2$ are both kept low until signals $x1$ and $x2$ differ by more than a transistor threshold voltage.

The metastability filter in figure 5.20 is a CMOS transistor-level implementation from [60]. An NMOS predecessor of this circuit appeared in [91]. Gate-level implementations are also possible: the metastability filter can be implemented using two buffers whose logic thresholds have been made particularly high (or low) by "trimming" the strengths of the pull-up and pull-down transistor paths ([115, section 2.3]). For example, a 4-input NAND gate with all its inputs tied together implements a buffer with a particularly high logic threshold. The use of this idea in the implementation of mutual exclusion elements is described in [2, 105].

5.8.2 Arbitration

The MUTEX can be used to build a handshake arbiter that can be used to control access to a resource that is shared between several autonomous independent parties. One possible implementation is shown in figure 5.21.

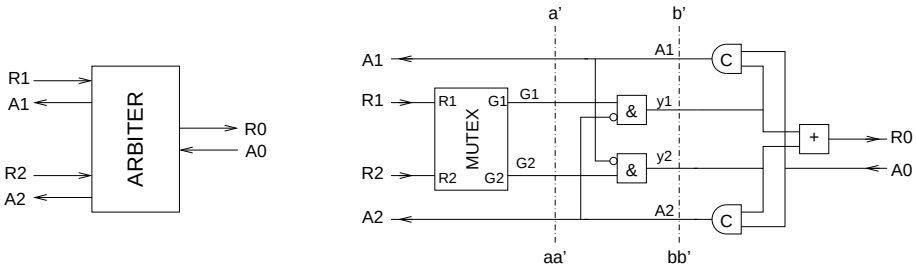


Figure 5.21. The handshake arbiter: symbol and possible implementation.

The MUTEX ensures that signals $G1$ and $G2$ at the a'–aa' interface are mutually exclusive. Following the MUTEX are two AND gates whose purpose it is to ensure that handshakes on the $(y1, A1)$ and $(y2, A2)$ channels at the b'–bb' interface are mutually exclusive: $y2$ can only go high if $A1$ is low and $y1$ can only go high if signal $A2$ is low. In this way, if handshaking is in progress along one channel, it blocks handshaking on the other channel. As handshaking along channels $(y1, A1)$ and $(y2, A2)$ are mutually exclusive the rest of the arbiter is simply a MERGE, c.f., figure 5.1 on page 59. If data needs to be passed to the shared resource a multiplexer is needed in exactly the same way as in the MERGE. The multiplexer may be controlled by signals $y1$ and/or $y2$.

5.8.3 Probability of metastability

Let us finally take a quantitative look at metastability: if $P(met_t)$ denotes the probability of the MUTEX being metastable for a period of time of t or longer (within an observation interval of one second), and if this situation is considered a failure, then we may calculate the mean time between failure as:

$$MTBF = \frac{1}{P(met_t)} \tag{5.8}$$

The probability $P(met_t)$ may be calculated as:

$$P(met_t) = P(met_t | met_{t=0}) \cdot P(met_{t=0}) \tag{5.9}$$

where:

- $P(met_t|met_{t=0})$ is the probability that the MUTEX is still metastable at time t given that it was metastable at time $t = 0$.
- $P(met_{t=0})$ is the probability that the MUTEX will enter metastability within a given observation interval.

The probability $P(met_{t=0})$ can be calculated as follows: the MUTEX will go metastable if its inputs $R1$ and $R2$ are exposed to transitions that occur almost simultaneously, i.e. within some small time window Δ. If we assume that the two input signals are uncorrelated and that they have average switching frequencies f_{R1} and f_{R2} respectively, then:

$$P(met_{t=0}) = \frac{1}{\Delta \cdot f_{R1} \cdot f_{R2}} \tag{5.10}$$

which can be understood as follows: within an observation interval of one second the input signal $R2$ makes $1/f_{R2}$ attempts at hitting one of the $1/f_{R1}$ time intervals of duration Δ where the MUTEX is vulnerable to metastability.

The probability $P(met_t|met_{t=0})$ is determined as:

$$P(met_t|met_{t=0}) = e^{-t/\tau} \tag{5.11}$$

where τ expresses the ability of the MUTEX to exit the metastable state spontaneously. This equation can be explained in two different ways and experimental results have confirmed its correctness. One explanation is that the cross coupled NAND gates have no memory of how long they have been metastable, and that the only probability distribution that is "memoryless" is an exponential distribution. Another explanation is that a small-signal model of the cross-coupled NAND gates at the metastable point has a single dominating pole.

Combining equations 5.8–5.11 we obtain:

$$MTBF = \frac{e^{\,t/\tau}}{\Delta \cdot f_{R1} \cdot f_{R2}} \tag{5.12}$$

Experiments and simulations have shown that this equation is reasonably accurate provided that t is not very small, and experiments or simulations may be used to determine the two parameters Δ and τ. Representative values for good circuit designs implemented in a 0.25 μm CMOS process are $\Delta = 30ps$ and $\tau = 25ps$.

5.9. Summary

This chapter addressed the implementation of the various handshake components: latch, fork, join, merge, function blocks, mux, demux, mutex and arbiter). A significant part of the material addressed principles and techniques for implementing function blocks.

Chapter 6

SPEED-INDEPENDENT CONTROL CIRCUITS

This chapter provides an introduction to the design of asynchronous sequential circuits and explains in detail one well-developed specification and synthesis method: the synthesis of speed-independent control circuits from signal transition graph specifications.

6.1. Introduction

Over time many different formalisms and theories have been proposed for the design of asynchronous control circuits (e.g. sequential circuits or state machines). The multitude of approaches arises from the combination of: (a) different specification formalisms, (b) different assumptions about delay models for gates and wires, and (c) different assumptions about the interaction between the circuit and its environment. Full coverage of the topic is far beyond the scope of this book. Instead we will first present some of the basic assumptions and characteristics of the various design methods and give pointers to relevant literature and then we will explain in detail one method: the design of speed-independent circuits from signal transition graphs – a method that is supported by a well-developed public domain tool, Petrify.

A good starting point for further reading is a book by Myers [70]. It provides in-depth coverage of the various formalisms, methods, and theories for the design of asynchronous sequential circuits and it provides a comprehensive list of references.

6.1.1 Asynchronous sequential circuits

To start the discussion figure 6.1 shows a generic synchronous sequential circuit and two alternative asynchronous control circuits: a Huffman style fundamental mode circuit with buffers (delay elements) in the feedback signals, and a Muller style input-output mode circuit with wires in the feedback path.

The synchronous circuit is composed of a set of registers holding the current state and a combinational logic circuit that computes the output signals and the next state signals. When the clock ticks the next state signals are copied into the registers thus becoming the current state. Reliable operation only requires that

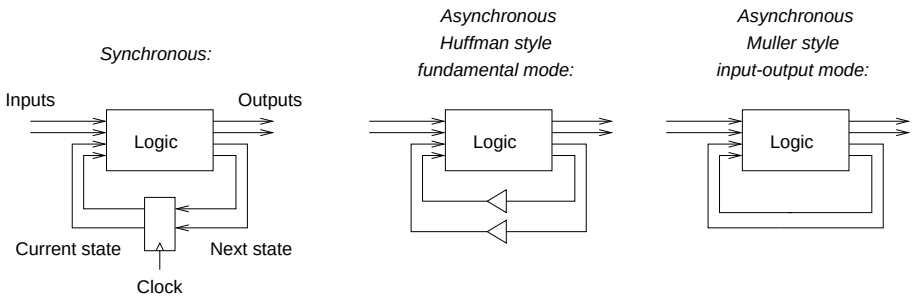


Figure 6.1. (a) A synchronous sequential circuit. (b) A Huffman style asynchronous sequential circuit with buffers in the feedback path, and (c) a Muller style asynchronous sequential circuit with wires in the feedback path.

the next state output signals from the combinational logic circuit are stable in a time window around the rising edge of the clock, an interval that is defined by the setup and hold time parameters of the register. Between two clock ticks the combinational logic circuit is allowed to produce signals that exhibit hazards. The only thing that matters is that the signals are ready and stable when the clock ticks.

In an asynchronous circuit there is no clock and all signals have to be valid at all times. This implies that at least the output signals that are seen by the environment must be free from all hazards. To achieve this, it is sometimes necessary to avoid hazards on internal signals as well. This is why the synthesis of asynchronous sequential circuits is difficult. Because it is difficult researchers have proposed different methods that are based on different (simplifying) assumptions.

6.1.2 Hazards

For the circuit designer a hazard is an unwanted glitch on a signal. Figure 6.2 shows four possible hazards that may be observed. A circuit that is in a stable state does not spontaneously produce a hazard – hazards are related to the dynamic operation of a circuit. This again relates to the dynamics of the input signals as well as the delays in the gates and wires in the circuit. A discussion of hazards is therefore not possible without stating precisely which delay model is being used and what assumptions are made about the interaction between the circuit and its environment. There are greater theoretical depths in this area than one might think at a first glance.

Gates are normally assumed to have delays. In section 2.5.3 we also discussed wire delays, and in particular the implications of having different delays in different branches of a forking wire. In addition to gate and wire delays it is also necessary to specify which delay model is being used.

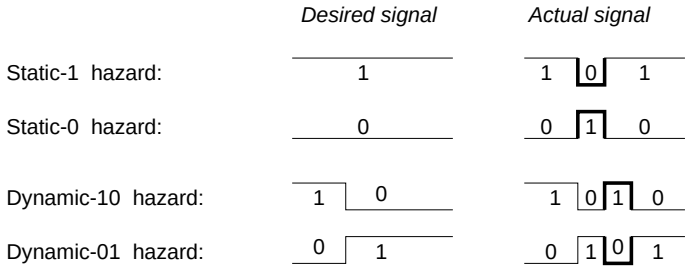


Figure 6.2. Possible hazards that may be observed on a signal.

6.1.3 Delay models

A *pure delay* that simply shifts any signal waveform later in time is perhaps what first comes to mind. In the hardware description language VHDL this is called a *transport delay*. It is, however, not a very realistic model as it implies that the gates and wires have infinitely high bandwidth. A more realistic delay model is the *inertial delay* model. In addition to the time shifting of a signal waveform, an inertial delay suppresses short pulses. In the inertial delay model used in VHDL two parameters are specified, the *delay time* and the *reject time*, and pulses shorter than the reject time are filtered out. The inertial delay model is the default delay model used in VHDL.

These two fundamental delay models come in several flavours depending on how the delay time parameter is specified. The simplest is a *fixed delay* where the delay is a constant. An alternative is a *min-max delay* where the delay is unknown but within a lower and upper bound: $t_{min} \leq t_{delay} \leq t_{max}$. A more pessimistic model is the *unbounded delay* where delays are positive (i.e. not zero), unknown and unbounded from above: $0 < t_{delay} < \infty$. This is the delay model that is used for gates in speed-independent circuits.

It is intuitive that the inertial delay model and the min-max delay model both have properties that help filter out some potential hazards.

6.1.4 Fundamental mode and input-output mode

In addition to the delays in the gates and wires, it is also necessary to formalize the interaction between the circuit being designed and its environment. Again, strong assumptions may simplify the design of the circuit. The design methods that have been proposed over time all have their roots in one of the following assumptions:

Fundamental mode: The circuit is assumed to be in a state where all input signals, internal signals, and output signals are stable. In such a stable state the environment is allowed to change one input signal. After that,

the environment is not allowed to change the input signals again until the entire circuit has stabilized. Since internal signals such as state variables are unknown to the environment, this implies that the longest delay in the circuit must be calculated and the environment is required to keep the input signals stable for at least this amount of time. For this to make sense, the delays in gates and wires in the circuit have to be bounded from above. The limitation on the environment is formulated as an absolute time requirement.

The design of asynchronous sequential circuits based on fundamental mode operation was pioneered by David Huffman in the 1950s [44, 45].

Input-output mode: Again the circuit is assumed to be in a stable state. Here the environment is allowed to change the inputs. When the circuit has produced the corresponding output (and it is allowable that there are no output changes), the environment is allowed to change the inputs again. There are no assumptions about the internal signals and it is therefore possible that the next input change occurs before the circuit has stabilized in response to the previous input signal change.

The restrictions on the environment are formulated as causal relations between input signal transitions and output signal transitions. For this reason the circuits are often specified using trace based methods where the designer specifies all possible sequences of input and output signal transitions that can be observed on the interface of the circuit. Signal transition graphs, introduced later, are such a trace-based specification technique.

The design of asynchronous sequential circuits based on the input-output mode of operation was pioneered by David Muller in the 1950s [68, 67]. As mentioned in section 2.5.1, these circuits are speed-independent.

6.1.5 Synthesis of fundamental mode circuits

In the classic work by Huffman the environment was only allowed to change one input signal at a time. In response to such an input signal change, the combinational logic will produce new outputs, of which some are fed back, figure 6.1(b). In the original work it was further required that only one feedback signal changes (at a time) and that the delay in the feedback buffer is large enough to ensure that the entire combinational circuit has stabilized before it sees the change of the feedback signal. This change may, in turn, cause the combinational logic to produce new outputs, etc. Eventually through a sequence of single signal transitions the circuit will reach a stable state where the environment is again allowed to make a single input change. Another way of expressing this behaviour is to say that the circuit starts out in a stable state

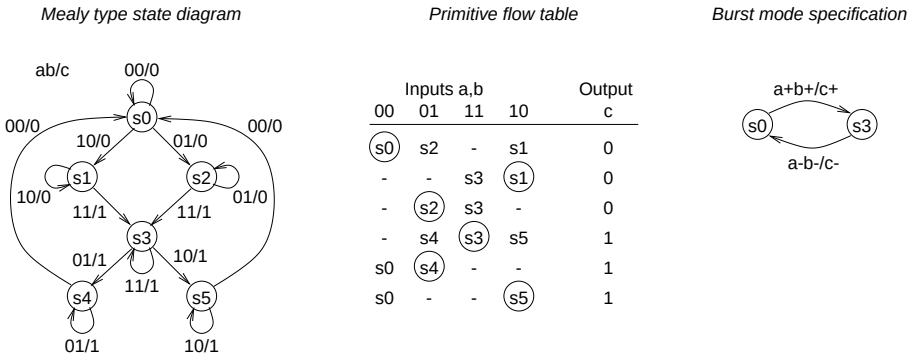


Figure 6.3. Some alternative specifications of a Muller C-element: a Mealy state diagram, a primitive flow table, and a burst-mode state diagram.

(which is defined to be a state that will persist until an input signal changes). In response to an input signal change the circuit will step through a sequence of transient, unstable states, until it eventually settles in a new stable state. This sequence of states is such that from one state to the next only one variable changes.

The interested reader is encouraged to consult [52], [99] or [70] and to specify and synthesize a C-element. The following gives a flavour of the design process and the steps involved:

- The design *may* start with a state graph specification that is very similar to the specification of a synchronous sequential circuit. This is optional. Figure 6.3 shows a Mealy type state graph specification of the C-element.

The classic design process involves the following steps:

- The intended sequential circuit is specified in the form of a primitive flow table (a state table with one row per stable state). Figure 6.3 shows the primitive flow table specification of a C-element.
- A minimum-row reduced flow table is obtained by merging compatible states in the primitive flow table.
- The states are encoded.
- Boolean equations for output variables and state variables are derived.

Later work has generalized the fundamental mode approach by allowing a restricted form of multiple-input and multiple-output changes. This approach is called *burst mode* [23, 19]. When in a stable state, a burst-mode circuit will wait for a set of input signals to change (in arbitrary order). After such

an input burst has completed the machine computes a burst of output signals and new values of the internal variables. The environment is not allowed to produce a new input burst until the circuit has completely reacted to the previous burst – fundamental mode is still assumed, but only between bursts of input changes. For comparison, figure 6.3 also shows a burst-mode specification of a C-element. Burst-mode circuits are specified using state graphs that are very similar to those used in the design of synchronous circuits. Several mature tools for synthesizing burst-mode controllers have been developed in academia [30, 122]. These tools are available in the public domain.

6.2. Signal transition graphs

The rest of this chapter will be devoted to the specification and synthesis of speed-independent control circuits. These circuits operate in input-output mode and they are naturally specified using signal transition graphs, (STGs). An STG is a petri net and it can be seen as a formalization of a timing diagram. The synthesis procedure that we will explain in the following consists of: (1) Capturing the behaviour of the intended circuit and its environment in an STG. (2) Generating the corresponding state graph, and adding state variables if needed. (3) Deriving Boolean equations for the state variables and outputs.

6.2.1 Petri nets and STGs

Briefly, a Petri net [1, 87, 69] is a graph composed of directed arcs and two types of nodes: transitions and places. Depending on the interpretation that is assigned to places, transitions and arcs, Petri nets can be used to model and analyze many different (concurrent) systems. Some places can be marked with tokens and the Petri net model can be "executed" by firing transitions. A transition is enabled to fire if there are tokens on all of its input places, and an enabled transition must eventually fire. When a transition fires, a token is removed from each input place and a token is added to each output place. We will show an example shortly. Petri nets offer a convenient way of expressing choice and concurrency.

It is important to stress that there are many variations of and extensions to Petri nets – Petri nets are a family of related models and not a single, unique and well defined model. Often certain restrictions are imposed in order to make the analysis for certain properties practical. The STGs we will consider in the following belong to such a restricted subclass: an STG is a 1-bounded Petri net in which only simple forms of input choice are allowed. The exact meaning of "1-bounded" and "simple forms of input choice" will be defined at the end of this section.

In an STG the transitions are interpreted as signal transitions and the places and arcs capture the causal relations between the signal transitions. Figure 6.4

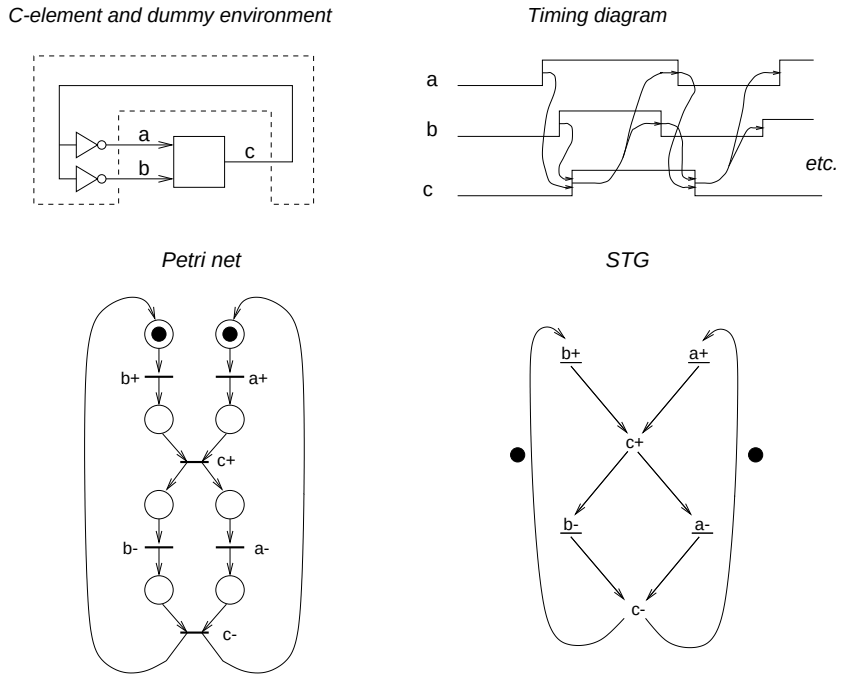


Figure 6.4. A C-element and its 'well behaved' dummy environment, its specification in the form of a timing diagram, a Petri net, and an STG formalization of the timing diagram.

shows a C-element and a 'well behaved' dummy environment that maintains the input signals until the C-element has changed its outputs. The intended behaviour of the C-element could be expressed in the form of a timing diagram as shown in the figure. Figure 6.4 also shows the corresponding Petri net specification. The Petri net is marked with tokens on the input places to the $a+$ and $b+$ transitions, corresponding to state $(a, b, c) = (0, 0, 0)$. The $a+$ and $b+$ transitions may fire in any order, and when they have both fired the $c+$ transition becomes enabled to fire, etc. Often STGs are drawn in a simpler form where most places have been omitted. Every arc that connects two transitions is then thought of as containing a place. Figure 6.4 shows the STG specification of the C-element.

A given marking of a Petri net corresponds to a possible state of the system being modeled, and by executing the Petri net and identifying all possible markings it is possible to derive the corresponding state graph of the system. The state graph is generally much more complex than the corresponding Petri net.

An STG describing a meaningful circuit enjoys certain properties, and for the synthesis algorithms used in tools like Petrify to work, additional properties and restrictions may be required. An STG is a Petri net with the following characteristics:

1 **Input free choice:** The selection among alternatives must only be controlled by mutually exclusive inputs.

2 **1-bounded:** There must never be more than one token in a place.

3 **Liveness:** The STG must be free from deadlocks.

An STG describing a meaningful speed-independent circuit has the following characteristics:

4 **Consistent state assignment:** The transitions of a signal must strictly alternate between $+$ and $-$ in any execution of the STG.

5 **Persistency:** If a signal transition is enabled it must take place, i.e. it must not be disabled by another signal transition. The STG specification of the circuit must guarantee persistency of internal signals (state variables) and output signals, whereas it is up to the environment to guarantee persistency of the input signals.

In order to be able to synthesize a circuit implementation the following characteristic is required:

6 **Complete state coding (CSC):** Two or more different markings of the STG must not have the same signal values (i.e. correspond to the same state). If this is not the case, it is necessary to introduce extra state variables such that different markings correspond to different states. The synthesis tool Petrify will do this automatically.

6.2.2 Some frequently used STG fragments

For the newcomer it may take a little practice to become familiar with specifying and designing circuits using STGs. This section explains some of the most frequently used templates from which one can construct complete specifications.

The basic constructs are: *fork, join, choice* and *merge,* see figure 6.5. The choice is restricted to what is called *input free choice*: the transitions following the choice place must represent mutually exclusive input signal transitions. This requirement is quite natural; we will only specify and design deterministic circuits. Figure 6.6 shows an example Petri net that illustrates the use of fork, join, free choice and merge. The example system will either perform transitions

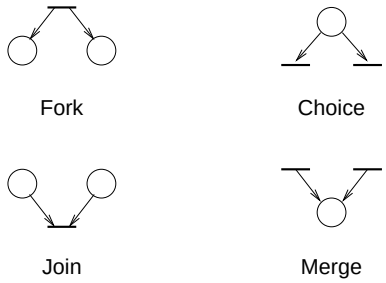


Figure 6.5. Petri net fragments for fork, join, free choice and merge constructs.

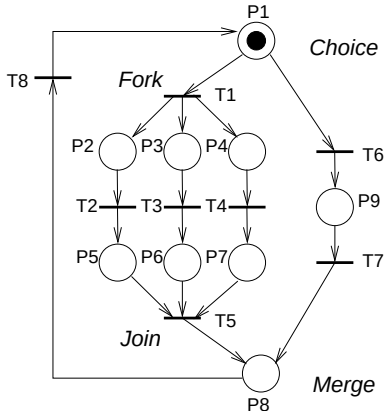


Figure 6.6. An example Petri net that illustrates the use of fork, join, free choice and merge.

$T6$ and $T7$ in sequence, or it will perform $T1$ followed by the concurrent execution of transitions $T2$, $T3$ and $T4$ (which may occur in any order), followed by $T5$.

Towards the end of this chapter we will design a 4-phase bundled-data version of the MUX component from figure 3.3 on page 32. For this we will need some additional constructs: a *controlled choice* and a Petri net fragment for the input end of a bundled-data channel.

Figure 6.7 shows a Petri net fragment where place $P1$ and transitions $T3$ and $T4$ represent a controlled choice: a token in place $P1$ will engage in either transition $T3$ or transition $T4$. The choice is controlled by the presence of a token in either $P2$ or $P3$. It is crucial that there can never be a token in both these places at the same time, and in the example this is ensured by the mutually exclusive input signal transitions $T1$ and $T2$.

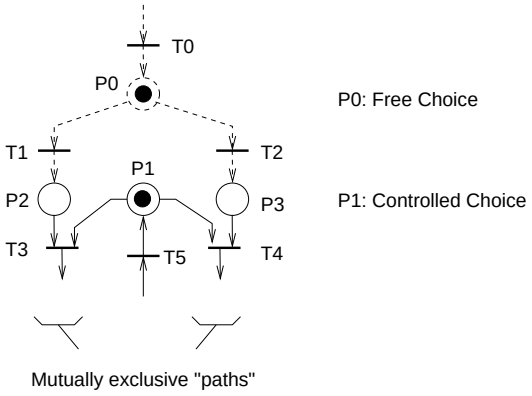


Figure 6.7. A Petri net fragment including a controlled choice.

Figure 6.8 shows a Petri net fragment for a component with a one-bit input channel using a 4-phase bundled-data protocol. It could be the control channel used in the MUX and DEMUX components introduced in figure 3.3 on page 32. The two transitions $dummy1$ and $dummy2$ do not represent transitions on the three signals in the channel, they are dummy transitions that facilitate expressing the specification. These dummy transitions represent an extension to the basic class of STGs.

Note also that the four arcs connecting:
place $P5$ and transition $Ctl+$
place $P5$ and transition $Ctl-$
place $P6$ and transition $dummy2$
place $P7$ and transition $dummy1$
have arrows at both ends. This is a shorthand notation for an arc in each direction. Note also that there are several instances where a place is both an input place and a output place for a transition. Place $P5$ and transition $Ctl+$ is an example of this.

The overall structure of the Petri net fragment can be understood as follows: at the top is a sequence of transitions and places that capture the handshaking on the Req and Ack signals. At the bottom is a loop composed of places $P6$ and $P7$ and transitions $Ctl+$ and $Ctl-$ that captures the control signal changing between high and low. The absence of a token in place $P5$ when Req is high expresses the fact that Ctl is stable in this period. When Req is low and a token is present in place $P5$, Ctl is allowed to make as many transitions as it desires. When $Req+$ fires, a token is put in place $P4$ (which is a controlled choice place). The Ctl signal is now stable, and depending on its value one of the two transitions $dummy1$ or $dummy2$ will become enabled and eventually

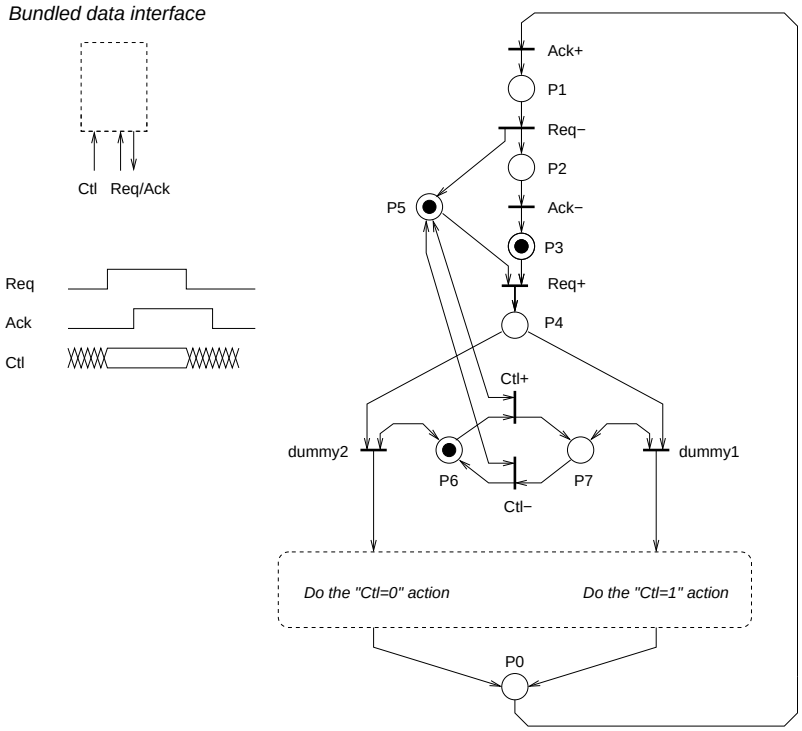


Figure 6.8. A Petri net fragment for a component with a one-bit input channel using a 4-phase bundled-data protocol.

fire. At this point the intended input-to-output operation that is not included in this example may take place, and finally the handshaking on the control port finishes ($Ack+$; $Req-$; $Ack-$).

6.3. The basic synthesis procedure

The starting point for the synthesis process is an STG that satisfies the requirements listed on page 88. From this STG the corresponding state graph is derived by identifying all of the possible markings of the STG that are reachable given its initial marking. The last step of the synthesis process is to derive Boolean equations for the state variables and output variables.

We will go through a number of examples by hand in order to illustrate the techniques used. Since the state of a circuit includes the values of all of the signals in the circuit, the computational complexity of the synthesis process can be large, even for small circuits. In practice one would always use one of the CAD tools that has been developed – for example Petrify that we will introduce later.

6.3.1 Example 1: a C-element

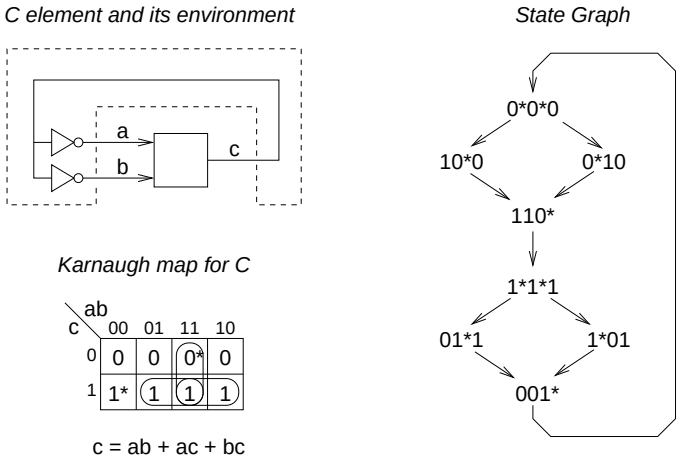


Figure 6.9. State graph and Boolean equation for the C-element STG from figure 6.4.

Figure 6.9 shows the state graph corresponding to the STG specification in figure 6.4 on page 87. Variables that are excited in a given state are marked with an asterisk. Also shown in figure 6.9 is the Karnaugh map for output signal c. The Boolean expression for c must cover states in which $c = 1$ and states where it is excited, $c = 0^*$ (changing to 1). In order to better distinguish excited variables from stable ones in the Karnaugh maps, we will use R (rising) instead of 0^* and F (falling) instead of 1^* throughout the rest of this book.

It is comforting to see that we can successfully derive the implementation of a known circuit, but the C-element is really too simple to illustrate all aspects of the design process.

6.3.2 Example 2: a circuit with choice

The following example provides a better illustration of the synthesis procedure, and in a subsequent section we will come back to this example and explain more efficient implementations. The example is simple – the circuit has only 2 inputs and 2 outputs – and yet it brings forward all relevant issues. The example is due to Chris Myers of the University of Utah who presented it in his 1996 course EE 587 "Asynchronous VLSI System Design." The example has roots in the papers [5, 6].

Figure 6.10 shows a specification of the circuit. The circuit has two inputs a and b and two outputs c and d, and the circuit has two alternative behaviours as illustrated in the timing diagram. The corresponding STG specification is shown in figure 6.11 along with the state graph for the circuit. The STG includes only

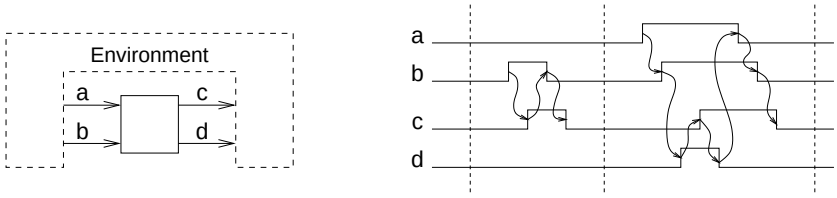


Figure 6.10. The example circuit from [5, 6].

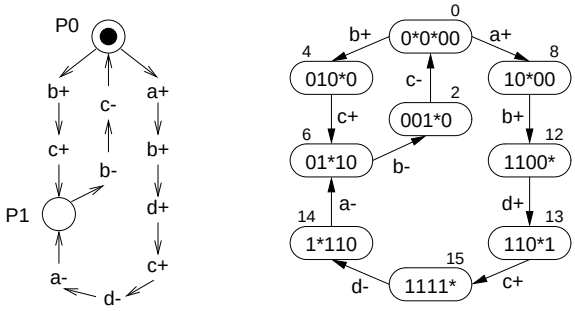


Figure 6.11. The STG specification and the corresponding state graph.

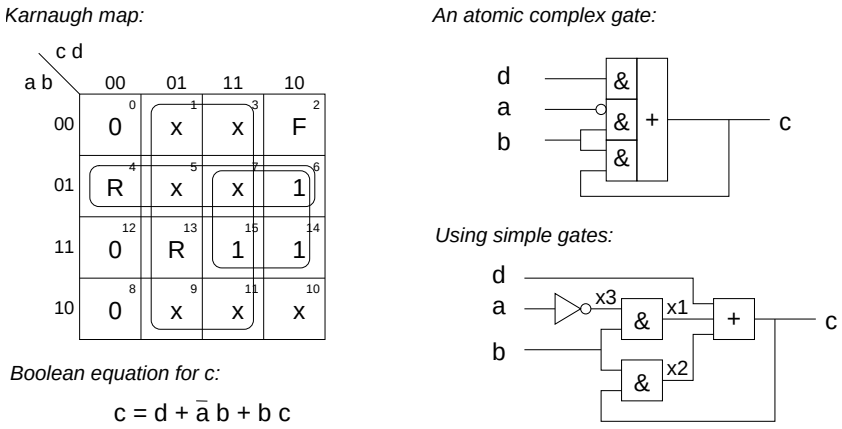


Figure 6.12. The Karnaugh map, the Boolean equation, and two alternative gate-level implementations of output signal c.

the free choice place $P0$ and the merge place $P1$. All arcs that directly connect two transitions are assumed to include a place. The states in the state diagram have been labeled with decimal numbers to ease filling out the Karnaugh maps.

The STG satisfies all of the properties 1-6 listed on page 88 and it is thus possible to proceed and derive Boolean equations for output signals c and d. [Note: In state 0 both inputs are marked to be excited, $(a, b) = (0^*, 0^*)$, and in states 4 and 8 one of the signals is still 0 but no longer excited. This is a problem of notation only. In reality only one of the two variables is excited in state 0, but we don't know which one. Furthermore, the STG is only required to be persistent with respect to the internal signals and the output signals. Persistency of the input signals must be guaranteed by the environment].

For output c, figure 6.12 shows the Karnaugh map, the Boolean equation and two alternative gate implementations: one using a single atomic And-Or-Invert gate, and one using simple AND and OR gates. Note that there are states that are not reachable by the circuit. In the Karnaugh map these states correspond to don't cares. The implementation of output signal d is left as an exercise for the reader ($d = ab\bar{c}$).

6.3.3 Example 2: Hazards in the simple gate implementation

The STG in figure 6.10 satisfies all of the implementation conditions 1-6 (including persistency), and consequently an implementation where each output signal is implemented by a single atomic complex gate is hazard free. In the case of c we need a complex And-Or gate with inversion of input signal a. In general such an atomic implementation is not feasible and it is necessary to decompose the implementation into a structure of simpler gates. Unfortunately this will introduce extra variables, and these extra variables may not satisfy the persistency requirement that an excited signal transition must eventually fire. Speed-independence preserving logic decomposition is therefore a very interesting and relevant topic [12, 53].

The implementation of c using simple gates that is shown in figure 6.12 is not speed-independent; it may exhibit both static and dynamic hazards, and it provides a good illustration of the mechanisms behind hazards. The problem is that the signals $x1$, $x2$ and $x3$ are not included in the original STG and state graph. A detailed analysis that includes these signals would *not* satisfy the persistency requirement. Below we explain possible failure sequences that may cause a static-1 hazard and a dynamic-10 hazard on output signal c. Figure 6.13 illustrates the discussion.

A static-1 hazard may occur when the circuit goes through the following sequence of states: 12, 13, 15, 14. The transition from state 12 to state 13 corresponds to d going high and the transition from state 15 to state 14

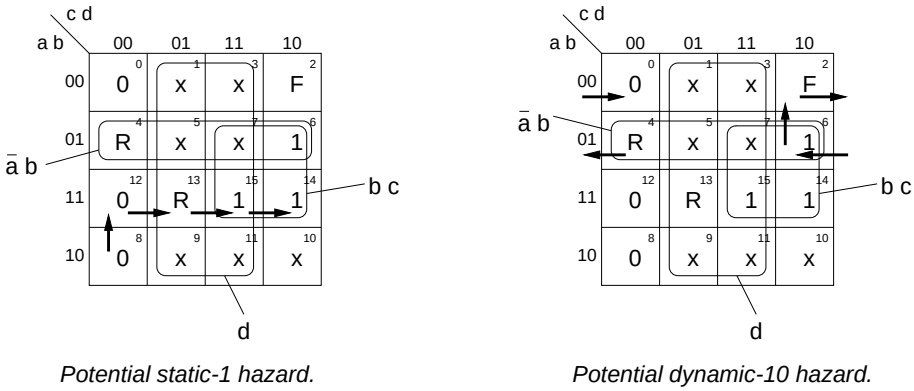


Figure 6.13. The Karnaugh maps for output signal c showing state sequences that may lead to hazards.

corresponds to d going low again. In state 13 c is excited (R) and it is supposed to remain high throughout states 13, 15, 14, and 6. States 13 and 15 are covered by the cube d, and state 14 is covered by cube bc that is supposed to "take over" and maintain $c = 1$ after d has gone low. If the AND gate with output signal $x2$ that corresponds to cube bc is slow we have the problem - the static-1 hazard.

A dynamic-10 hazard may occur when the circuit goes through the following sequence of states: 4, 6, 2, 0. This situation corresponds to the upper AND gate (with output signal $x1$) and the OR gate relaying $b+$ into $c+$ and $b-$ into $c-$. However, after the $c+$ transition the lower AND gate, $x2$, becomes excited (R) as well, but the firing of this gate is not indicated by any other signal transition – the OR gate already has one input high. If the lower AND gate ($x2$) fires, it will later become excited (F) in response to $c-$. The net effect of this is that the lower AND gate ($x2$) may superimpose a 0-1-0 pulse onto the c output after the intended $c-$ transition has occured.

In the above we did not consider the inverter with input signal a and output signal $x3$. Since a is not an input to any other gate, this decomposition is SI.

In summary both types of hazard are related to the circuit going through a sequence of states that are covered by several cubes that are supposed to maintain the signal at the same (stable) level. The cube that "takes over" represents a signal that may not be indicated by any other signal. In essence it is the same problem that we touched upon in section 2.2 on page 14 and in section 2.4.3 on page 20 – an OR gate can only indicate when the first input goes high.

6.4. Implementations using state-holding gates

6.4.1 Introduction

During operation each variable in the circuit will go through a sequence of states where it is (stable) 0, followed by one or more states where it is excited (R), followed by a sequence of states where it is (stable) 1, followed by one or more states where it is excited (F), etc. In the above implementation we were covering all states where a variable, z, was high or excited to go high ($z = 1$ and $z = R = 0*$).

An alternative is to use a state-holding device such as a set-reset latch. The Boolean equations for the set and reset signals need only cover the $z = R = 0*$ states and the $z = F = 1*$ states respectively. This will lead to simpler equations and potentially simpler decompositions. Figure 6.14 shows the implementation template using a standard set-reset latch and an alternative solution based on a standard C-element. In the latter case the reset signal must be inverted. Later, in section 6.4.5, we will discuss alternative and more elaborate implementations, but for the following discussion the basic topologies in figure 6.14 will suffice.

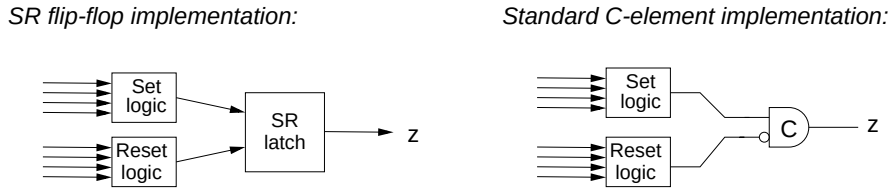


Figure 6.14. Possible implementation templates using (simple) state holding elements.

At this point it is relevant to mention that the equations for when to set and reset the state-holding element for signal z can be found by rewriting the original equation (that covers states in which $z = R$ and $z = 1$) into the following form:

$$z = \text{“Set”} + z \cdot \overline{\text{“Reset”}} \tag{6.1}$$

For signal c in the above example (figure 6.12 on page 93) we would get the following *set* and *reset functions:* $c_{set} = d + \overline{a}b$ and $c_{reset} = \overline{b}$ (which is identical to the result in figure 6.15 in section 6.4.3). Furthermore it is obvious that for all reachable states (only) the set and reset functions for a signal z must never be active at the same time:

$$\text{“Set”} \wedge \text{“Reset”} \equiv 0$$

The following sections will develop the idea of using state-holding elements and we will illustrate the techniques by re-implementing example 2 from the previous section.

6.4.2 Excitation regions and quiescent regions

The above idea of using a state-holding device for each variable can be formalized as follows:

An **excitation region**, ER, for a variable z is a maximally-connected set of states in which the variable is excited:

- ER($z+$) denotes a region of states where $z = R = 0$*
- ER($z-$) denotes a region of states where $z = F = 1$*

A **quiescent region**, QR, for a variable z is a maximally-connected set of states in which the variable is not excited:

- QR($z+$) denotes a region of states where $z = 1$
- QR($z-$) denotes a region of states where $z = 0$

For a given circuit the state space can be disjointly divided into one or more regions of each type.

The **set function** (cover) for a variable z:

- must contain all states in the ER($z+$) regions
- may contain states from the QR($z+$) regions
- may contain states not reachable by the circuit

The **reset function** (cover) for a variable z:

- must contain all states in the ER($z-$) regions
- may contain states from the QR($z-$) regions
- may contain states not reachable by the circuit

In section 6.4.4 below we will add what is known as the **monotonic cover constraint** or the **unique entry constraint** in order to avoid hazards:

- A cube (product term) in the set or reset function of a variable must only be entered through a state where the variable is excited.

Having mentioned this last constraint, we have above a complete recipe for the design of speed-independent circuits where each non-input signal is implemented by a state holding device. Let us continue with example 2.

6.4.3 Example 2: Using state-holding elements

Figure 6.15 illustrates the above procedure for example 2 from sections 6.3.2 and 6.3.3. As before, the Boolean equations (for the set and reset functions) may need to be implemented using atomic complex gates in order to ensure that the resulting circuit is speed-independent.

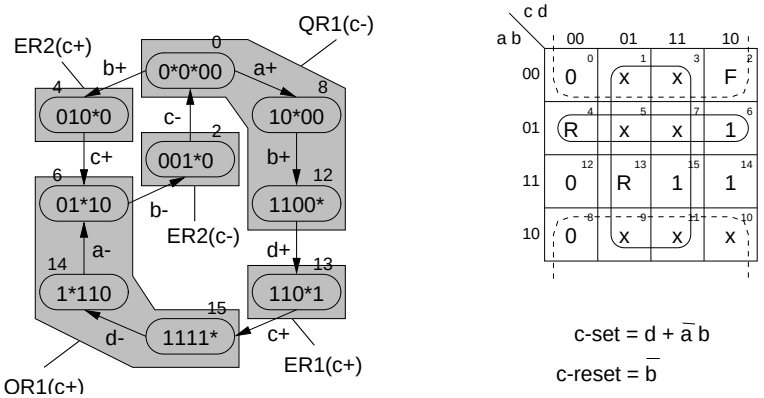


Figure 6.15. Excitation and quiescent regions in the state diagram for signal c in the example circuit from figure 6.10, and the corresponding derivation of equations for the set and reset functions.

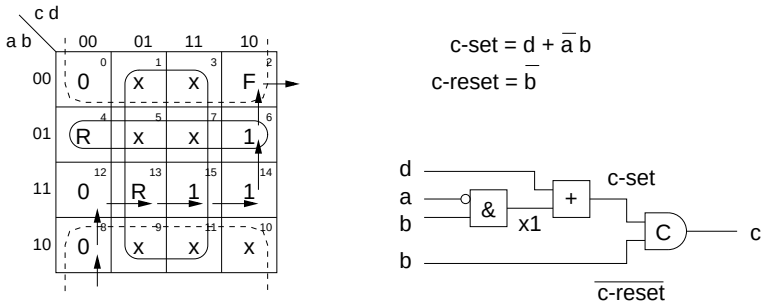


Figure 6.16. Implementation of c using a standard C-element and simple gates, along with the Karnaugh map from which the set and reset functions were derived.

6.4.4 The monotonic cover constraint

A standard C-element based implementation of signal c from above, with the set and reset functions implemented using simple gates, is shown in figure 6.16 along with the Karnaugh map from which the set and reset functions are derived. The set function involves two cubes d and $\overline{a}b$ that are input signals to an OR gate. This implementation may exhibit a dynamic-10 hazard on the c_{set}-signal

in a similar way to that discussed previously. The Karnaugh map in figure 6.16 shows the sequence of states that may lead to a malfunction: (8, 12, 13, 15, 14, 6, 0). Signal d is low in state 12, high in states 13 and 15, and low again in state 14. This sequence of states corresponds to a pulse on d. Through the OR gate this will create a pulse on the c_{set} signal that will cause c to go high. Later in state 2, c will go low again. This is the desired behaviour. The problem is that the internal signal $x1$ that corresponds to the other cube in the expression for c_{set} becomes excited ($x1 = R$) in state 6. If this AND gate is slow this may produce an unintended pulse on the c_{set} signal after c has been reset again.

If the cube $\overline{a}b$ (that covers states 4, 5, 7, and 6) is reduced to include only states 4 and 5 corresponding to $c_{set} = d + \overline{a}b\overline{c}$ we would avoid the problem. The effect of this modification is that the OR gate is never exposed to more than one input signal being high, and when this is the case we do not have problems with the principle of indication (c.f. the discussion of indication and dual-rail circuits in chapter 2). Another way of expressing this is that a cover cube must only be entered through states belonging to an excitation region. This requirement is known as:

- the **monotonic cover constraint:** only one product term in a sum-of-products implementation is allowed to be high at any given time. Obviously the requirement need only be satisfied in the states that are reachable by the circuit, or alternatively

- the **unique entry constraint:** cover cubes may only be entered through excitation region states.

6.4.5 Circuit topologies using state-holding elements

In addition to the set-reset flip-flop and the standard C-element based templates presented above, there are a number of alternative solutions for implementing variables using a state-holding device.

A popular approach is the *generalized C-element* that is available to the CMOS transistor-level designer. Here the state-holding mechanism and the set and reset functions are implemented in one (atomic) compound structure of n- and p-type transistors. Figure 6.17 shows a gate-level symbol for a circuit where $z_{set} = ab$ and $z_{reset} = \overline{b}\overline{c}$ along with dynamic and static CMOS implementations.

An alternative implementation that may be attractive to a designer using a standard cell library that includes (complex) And-Or-Invert gates is shown in figure 6.18. The circuit has the interesting property that it produces both the desired signal z and its complement $\overline{z}$ and during transitions it *never* produces $(z, \overline{z}) = (1, 1)$. Again, the example is a circuit where $z_{set} = ab$ and $z_{reset} = \overline{b}\overline{c}$.

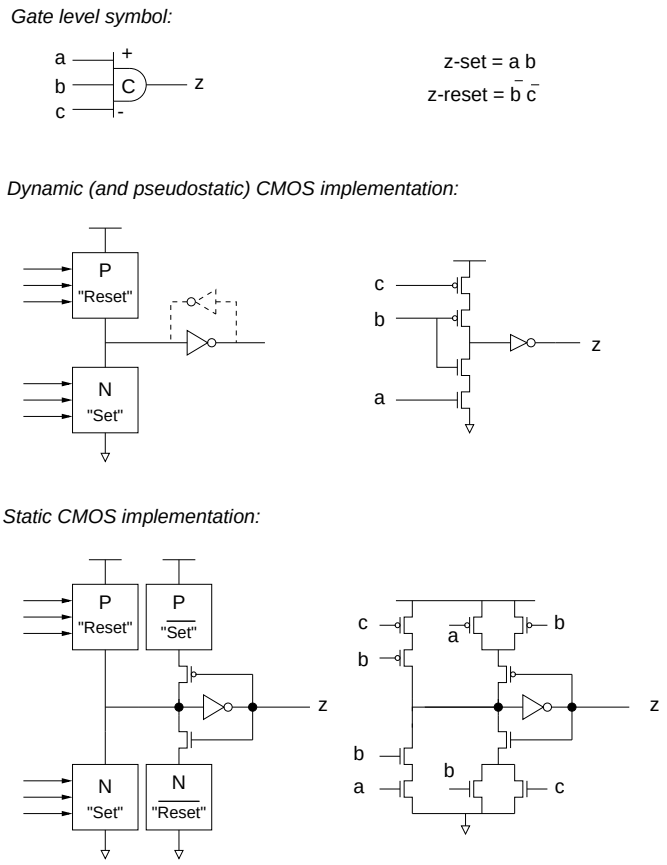


Figure 6.17. A generalized C-element: gate-level symbol, and some CMOS transistor implementations.

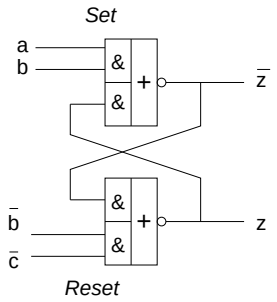


Figure 6.18. An SR implementation based on two complex And-Or-Invert gates.

6.5. Initialization

Initialization is an important aspect of practical circuit design, and unfortunately it has not been addressed in the above. The synthesis process *assumes* an initial state that corresponds to the initial marking of the STG, and the resulting synthesized circuit is a correct speed-independent implementation of the specification *provided that the circuit starts out in the same initial state.* Since the synthesized circuits generally use state-holding elements or circuitry with feedback loops it is necessary to actively force the circuit into the intended initial state.

Consequently, the designer has to do a manual post-synthesis hack and extend the circuit with an extra signal which, when active, sets all state-holding constructs into the desired state. Normally the circuits will not be speed-independent with respect to this initialization signal; it is assumed to be asserted for long enough to cause the desired actions before it is de-asserted.

For circuit implementations using state-holding elements such as set-reset latches and standard C-elements, initialization is trivial provided that these components have special clear/preset signals in addition to their normal inputs. In all other cases the designer has to add an initialization signal to the relevant Boolean equations explicitly. If the synthesis process is targeting a given cell library, the modified logic equations may need further logic decomposition, and as we have seen this may compromise speed-independence.

The fact that initialization is not included in the synthesis process is obviously a drawback, but normally one would implement a library of control circuits and use these as building blocks when designing circuits at the more abstract "static data-flow structures" level as introduced in chapter 3.

Initializing all control circuits as outlined above is a simple and robust approach. However, initialization of asynchronous circuits based on handshake components may also be achieved by an implicit approach that that exploits the function of the circuit to "propagate" initial signal values into the circuit. In Tangram (section 8.3, and chapter 13 in part III) this this is called self-initialization, [101].

6.6. Summary of the synthesis process

The previous sections have covered the basic theory for synthesizing SI control circuits from STG specifications. The style of presentation has deliberately been chosen to be an informal one with emphasis on examples and the intuition behind the theory and the synthesis procedure.

The theory has roots in work done by the following Universities and groups: University of Illinois [68], MIT [18, 16], Stanford [6], IMEC [110, 121], St. Petersburg Electrical Engineering Institute [111], and the multinational group of researchers who have developed the Petrify tool [21] that we will introduce

in the next section. This author has attended several discussions from which it is clear that in some cases the concepts and theories have been developed independently by several groups, and I will refrain from attempting a precise history of the evolution. The reader who is interested in digging deeper into the subject is encouraged to consult the literature; in particular the book by Myers [70].

In summary the synthesis process outlined in the previous sections involves the following steps:

1 Specify the desired behaviour of the circuit and its (dummy) environment using an STG.

2 Check that the STG satisfies properties 1-5 on page 88: 1-bounded, consistent state assignment, liveness, only input free choice and controlled choice and persistency. An STG satisfying these conditions is a valid specification of an SI circuit.

3 Check that the specification satisfies property 6 on page 88: complete state coding (CSC). If the specification does not satisfy CSC it is necessary to add one or more state variables or to change the specification (which is often possible in 4-phase control circuits where the down-going signal transitions can be shuffled around). Some tools (including Petrify) can insert state variables automatically, whereas re-shuffling of signals – which represents a modification of the specification – is a task for the designer.

4 Select an implementation template and derive the Boolean equations for the variables themselves, or for the set and reset functions when state holding devices are used. Also decide if these equations can be implemented in atomic gates (typically complex AOI-gates) or if they are to be implemented by structures of simpler gates. These decisions may be set by switches in the synthesis tools.

5 Derive the Boolean equations for the desired implementation template.

6 Manually modify the implementation such that the circuit can be forced into the desired initial state by an explicit reset or initialization signal.

7 Enter the design into a CAD tool and perform simulation and layout of the circuit (or the system in which the circuit is used as a component).

6.7. Petrify: A tool for synthesizing SI circuits from STGs

Petrify is a public domain tool for manipulating Petri nets and for synthesizing SI control circuits from STG specifications. It is available from `http://www.lsi.upc.es/~jordic/petrify/petrify.html`.

Petrify is a typical UNIX program with many options and switches. As a circuit designer one would probably prefer a push-button synthesis tool that accepts a specification and produces a circuit. Petrify can be used this way but it is more than this. If you know how to play the game it is an interactive tool for specifying, checking, and manipulating Petri nets, STGs and state graphs. In the following section we will show some examples of how to design speed-independent control circuits.

Input to Petrify is an STG described in a simple textual format. Using the program draw_astg that is part of the Petrify distribution (and that is based on the graph visualization package 'dot' developed at AT&T) it is possible to produce a drawing of the STGs and state graphs. The graphs are "nice" but the topological organization may be very different from how the designer thinks about the problem. Even the simple task of checking that an STG entered in textual form is indeed the intended STG may be difficult.

To help ease this situation a graphical STG entry and simulation tool called VSTGL (Visual STG Lab) has been developed at the Technical University of Denmark. To help the designer obtain a correct specification VSTGL includes an interactive simulator that allows the designer to add tokens and to fire transitions. It also carries out certain simple checks on the STG.

VSTGL is available from `http://vstgl.sourceforge.net/` and it is the result of a small student project done by two 4th year students. VSTGL is stable and reliable, though naming of signal transitions may seem a bit awkward.

Petrify can solve CSC violations by inserting state variables, and it can be controlled to target the implementation templates introduced in section 6.4:

- The **-cg** option will produce a complex-gate circuit (one where each non-input signal is implemented in a single complex gate).
- The **-gc** option will produce a generalized C-element circuit. The outputs from Petrify are the Boolean equations for the set and reset functions for each non-input signal.
- The **-gcm** option will produce a generalized C-element solution where the set and reset functions satisfy the monotonic cover requirement. Consequently the solution can also be mapped onto a standard C-element implementation where the set and $reset$ functions are implemented using simple AND and OR gates.
- The **-tm** option will cause Petrify to perform technology mapping onto a gate library that can be specified by the user. Technology mapping can obviously not be combined with the `-cg` and `-gc` options.

Petrify comes with a manual and some examples. In the following section we will go through some examples drawn from the previous chapters of the book.

6.8. Design examples using Petrify

In the following we will illustrate the use of Petrify by specifying and synthesizing: (a) example 2 – the circuit with choice, (b) a control circuit for the 4-phase bundled-data implementation of the latch from figure 3.3 on page 32 and (c) a control circuit for the 4-phase bundled-data implementation of the MUX from figure 3.3 on page 32. For all of the examples we will assume push channels only.

6.8.1 Example 2 revisited

As a first example, we will synthesize the different versions of example 2 that we have already designed manually. Figure 6.19 shows the STG as it is entered into VSTGL. The corresponding textual input to Petrify (the ex2.g file) and the STG as it may be visualized by Petrify are shown in figure 6.20. Note in figure 6.20 that an index is added when a signal transition appears more than once in order to facilitate the textual input.

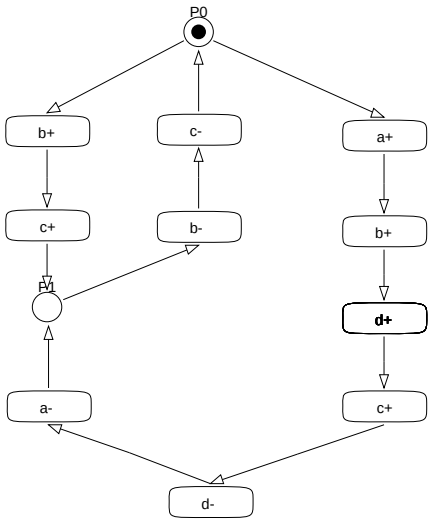


Figure 6.19. The STG of example 2 as it is entered into VSTGL.

Using complex gates

```
> petrify ex2.g -cg -eqn ex2-cg.eqn

The STG has CSC.
# File generated by petrify 4.0 (compiled 22-Dec-98 at 6:58 PM)
# from <ex2.g> on 6-Mar-01 at 8:30 AM
     ....
```

```
.model ex2
.inputs a b
.outputs c d
.graph
P0 a+ b+
c+ P1
b+ c+
P1 b-
c- P0
b- c-
a- P1
d- a-
c+/1 d-
a+ b+/1
b+/1 d+
d+ c+/1
.marking { P0 }
.end
```

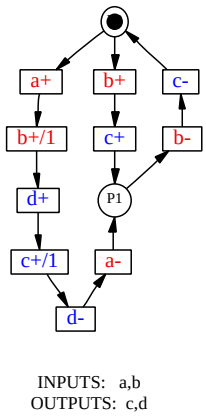


Figure 6.20. The textual description of the STG for example 2 and the drawing of the STG that is produced by Petrify.

```
# The original TS had (before/after minimization) 9/9 states
# Original STG:   2 places,  10 transitions,  13 arcs  ...
# Current STG:    4 places,   9 transitions,  18 arcs  ...
# It is a Petri net with 1 self-loop places
      ...
```

```
> more ex2-cg.eqn
```

```
# EQN file for model ex2
# Generated by petrify 4.0 (compiled 22-Dec-98 at 6:58 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 7.00

INORDER = a b c d;
OUTORDER = [c] [d];
[c] = b (c + a') + d;
[d] = a b c';
```

Using generalized C-elements:

```
> petrify ex2.g -gc -eqn ex2-gc.eqn
   ...
> more ex2-gc.eqn
```

```
# EQN file for model ex2
# Generated by petrify 4.0 (compiled 22-Dec-98 at 6:58 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 12.00
```

```
INORDER = a b c d;
OUTORDER = [c] [d];
[0] = a' b + d;
[1] = a b c';
[d] = d c' + [1];          # mappable onto gC
[c] = c b + [0];           # mappable onto gC
```

The equations for the generalized C-elements should be "interpreted" according to equation 6.1 on page 96

Using standard C-elements and set/reset functions that satisfy the **monotonic cover constraint**:

```
> petrify ex2.g -gcm -eqn ex2-gcm.eqn
   ...
> more ex2-gcm.eqn

# EQN file for model ex2
# Generated by petrify 4.0 (compiled 22-Dec-98 at 6:58 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 10.00

INORDER = a b c d;
OUTORDER = [c] [d];
[0] = a' b c' + d;
[d] = a b c';
[c] = c b + [0];           # mappable onto gC
```

Again, the equations for the generalized C-element should be "interpreted" according to equation 6.1 on page 96.

6.8.2 Control circuit for a 4-phase bundled-data latch

Figure 6.21 shows an asynchronous handshake latch with a dummy environment on its left and right side. The latch can be implemented using a normal N-bit wide transparent latch and the control circuit we are about to design. A driver may be needed for the latch control signal Lt. In order to make the latch controller robust and independent of the delay in this driver, we may feed the buffered signal (Lt) back such that the controller knows when the signal has been presented to the latch. Figure 6.21 also shows fragments of the STG specification – the handshaking of the left and right hand environments and ideas about the behaviour of the latch controller. Initially Lt is low and the latch is transparent, and when new input data arrives they will flow through the latch. In response to $Rin+$, and provided that the right hand environment is ready for another handshake ($Aout = 0$), the controller may generate $Rout+$ right away.

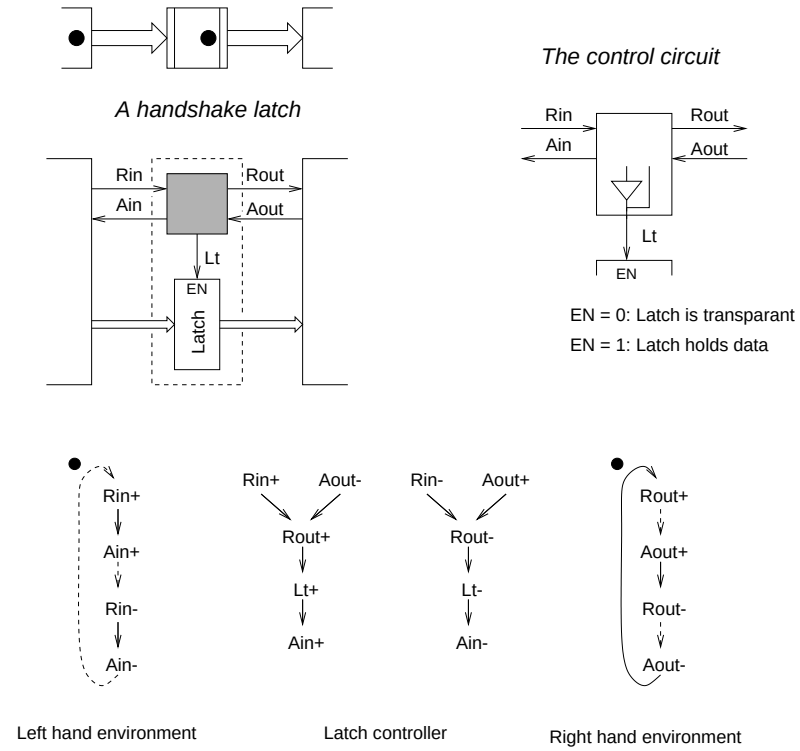


Figure 6.21. A 4-phase bundled-data handshake latch and some STG fragments that capture ideas about its behaviour.

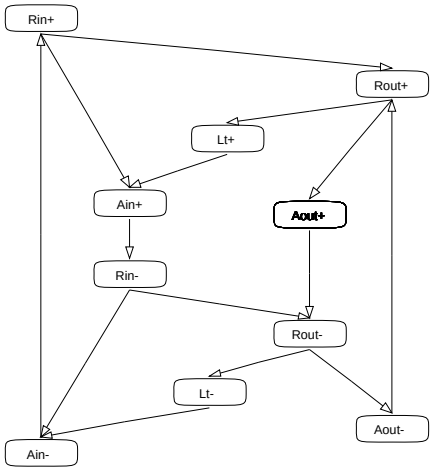


Figure 6.22. The resulting STG for the latch controller (as input to VSTGL).

Furthermore the data should be latched, $Lt+$, and an acknowledge sent to the left hand environment, $Ain+$. A symmetric scenario is possible in response to $Rin-$ when the latch is switched back into the transparent mode. Combining these STG fragments results in the STG shown in figure 6.22.
Running Petrify yields the following:

```
> petrify lctl.g -cg -eqn lctl-cg.eqn

The STG has CSC.
# File generated by petrify 4.0 (compiled 22-Dec-98 at 6:58 PM)
# from <lctl.g> on 6-Mar-01 at 11:18 AM
...
# The original TS had (before/after minimization) 16/16 states
# Original STG:   0 places,  10 transitions,  14 arcs (  0 pt + ...
# Current STG:    0 places,  10 transitions,  12 arcs (  0 pt + ...
# It is a Marked Graph
.model lctl
.inputs  Aout Rin
.outputs  Lt Rout Ain
.graph
Rout+ Aout+ Lt+
Lt+ Ain+
Aout+ Rout-
Rin+ Rout+
Ain+ Rin-
Rin- Rout-
Ain- Rin+
Rout- Lt- Aout-
Aout- Rout+
Lt- Ain-
.marking { <Aout-,Rout+> <Ain-,Rin+> }
.end

> more lctl-cg.eqn

# EQN file for model lctl
# Generated by petrify 4.0 (compiled 22-Dec-98 at 6:58 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 7.00

INORDER = Aout Rin Lt Rout Ain;
OUTORDER = [Lt] [Rout] [Ain];
[Lt] = Rout;
[Rout] = Rin (Rout + Aout') + Aout' Rout;
[Ain] = Lt;
```

The equation for `[Rout]` may be rewritten as:

```
[Rout] = Rin Aout' + Rout (Rin + Aout')
```

which can be recognized to be a C-element with inputs `Rin` and `Aout'`.

6.8.3 Control circuit for a 4-phase bundled-data MUX

After the above two examples, where we have worked out already well-known circuit implementations, let us now consider a more complex example that cannot (easily) be done by hand. Figure 6.23 shows the handshake multiplexer from figure 3.3 on page 32. It also shows how the handshake MUX can be implemented by a "regular" combinational circuit multiplexer and a control circuit. Below we will design a speed-independent control circuit for a 4-phase bundled-data MUX.

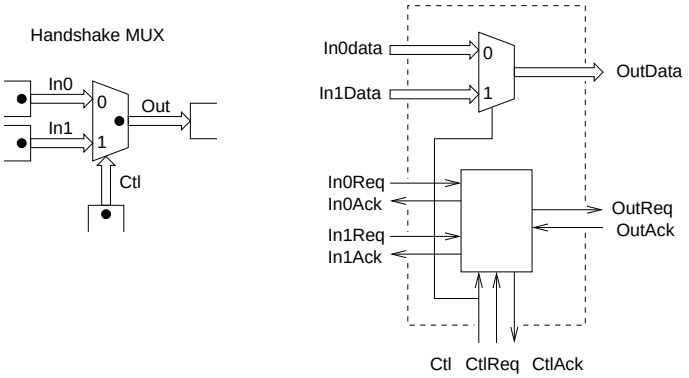


Figure 6.23. The handshake MUX and the structure of a 4-phase bundled-data implementation.

The MUX has three input channels and we *must* assume they are connected to three *independent* dummy environments. The dots remind us that the channels are push channels. When specifying the behaviour of the MUX control circuit and its (dummy) environment it is important to keep this in mind. A typical error when drawing STGs is to specify an environment with a more limited behaviour than the real environment. For each of the three input channels the STG has cycles involving ($Req+$; $Ack+$; $Req-$; $Ack-$; etc.), and each of these cycles is initialized to contain a token.

As mentioned previously, it is sometimes easier to deal with control channels using dual-rail (or in general $1-$of$-N$) data encodings since this implies dealing with one-hot (decoded) control signals. As a first step towards the STG for a MUX using entirely 4-phase bundled-data channels, figure 6.24 shows an STG for a MUX where the control channel uses dual-rail signals ($Ctl.t$, $Ctl.f$ and $CtlAck$). This STG can then be combined with the STG-fragment for a 4-phase bundled-data channel from figure 6.8 on page 91, resulting in the STG in figure 6.25. The "intermediate" STG in figure 6.24 emphasizes the fact that the MUX can be seen as a controlled join – the two mutually exclusive and structurally identical halves are basically the STGs of a join.

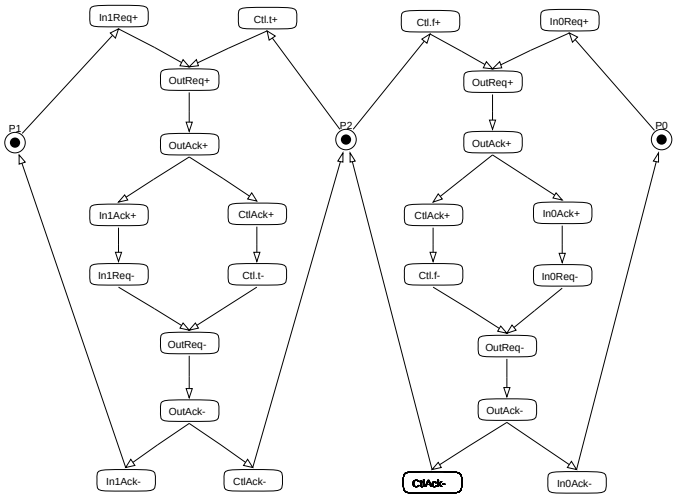


Figure 6.24. The STG specification of the control circuit for a 4-phase bundled-data MUX using a 4-phase dual-rail control channel. Combined with the STG fragment for a bundled-data (control) channel the resulting STG for an all 4-phase dual-rail MUX is obtained (figure 6.25).

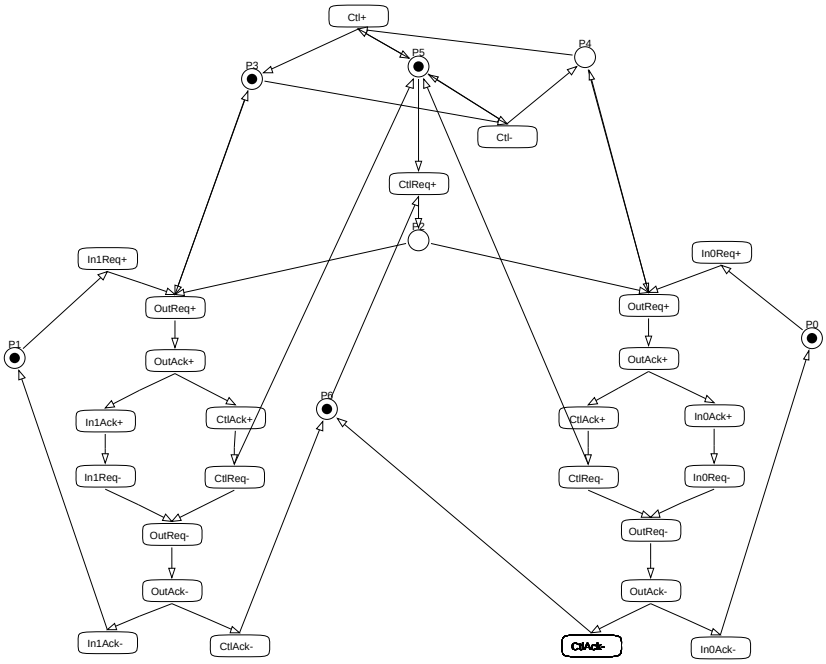


Figure 6.25. The final STG specification of the control circuit for the 4-phase bundled-data MUX. All channels, including the control channel, are 4-phase bundled-data.

Below is the result of running Petrify, this time with the `-o` option that writes the resulting STG (possibly with state signals added) in a file rather than to stdout.

```
>petrify MUX4p.g -o MUX4p-csc.g -gcm -eqn MUX4p-gcm.eqn

  State coding conflicts for signal In1Ack
  State coding conflicts for signal In0Ack
  State coding conflicts for signal OutReq
The STG has no CSC.
Adding state signal: csc0
The STG has CSC.

> more MUX4p-gcm.eqn

# EQN file for model MUX4p
# Generated by petrify 4.0 (compiled 22-Dec-98 at 6:58 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 29.00

INORDER = In0Req OutAck In1Req Ctl CtlReq In1Ack In0Ack OutReq
          CtlAck csc0;
OUTORDER = [In1Ack] [In0Ack] [OutReq] [CtlAck] [csc0];
[In1Ack] = OutAck csc0';
[In0Ack] = OutAck csc0;
[2] = CtlReq (In1Req csc0' + In0Req Ctl');
[3] = CtlReq' (In1Req' csc0' + In0Req' csc0);
[OutReq] = OutReq [3]' + [2];      # mappable onto gC
[5] = OutAck' csc0;
[CtlAck] = CtlAck [5]' + OutAck;   # mappable onto gC
[7] = OutAck' CtlReq';
[8] = CtlReq Ctl;
[csc0] = csc0 [8]' + [7];          # mappable onto gC
```

As can be seen, the STG does not satisfy CSC (complete state coding) as several markings correspond to the same state vector, so Petrify adds an internal state-signal $csc0$. The intuition is that after $CtlReq-$ the Boolean signal Ctl is no longer valid but the MUX control circuit has not yet finished its job. If the circuit can't see what to continue doing from its input signals it needs an internal state variable in which to keep this information. The signal $csc0$ is an active-low signal: it is set low if $Ctl = 0$ when $CtlReq+$ and it is set back to high when $OutAck$ and $CtlReq$ are both low. The fact that the signal $csc0$ is high when all channels are idle (all handshake signals are low) should be kept in mind when dealing with reset, c.f. section 6.5.

The exact details of how the state variable is added can be seen from the STG that includes $csc0$ which is produced by Petrify before it synthesizes the logic expressions for the circuit.

It is sometimes possible to avoid adding a state variable by re-shuffling signal transitions. It is not always obvious what yields the best solution. In principle more concurrency should improve performance, but it also results in a larger state-space for the circuit and this often tends to result in larger and slower circuits. A discussion of performance also involves the interaction with the environment. There is plenty of room for exploring alternative solutions.

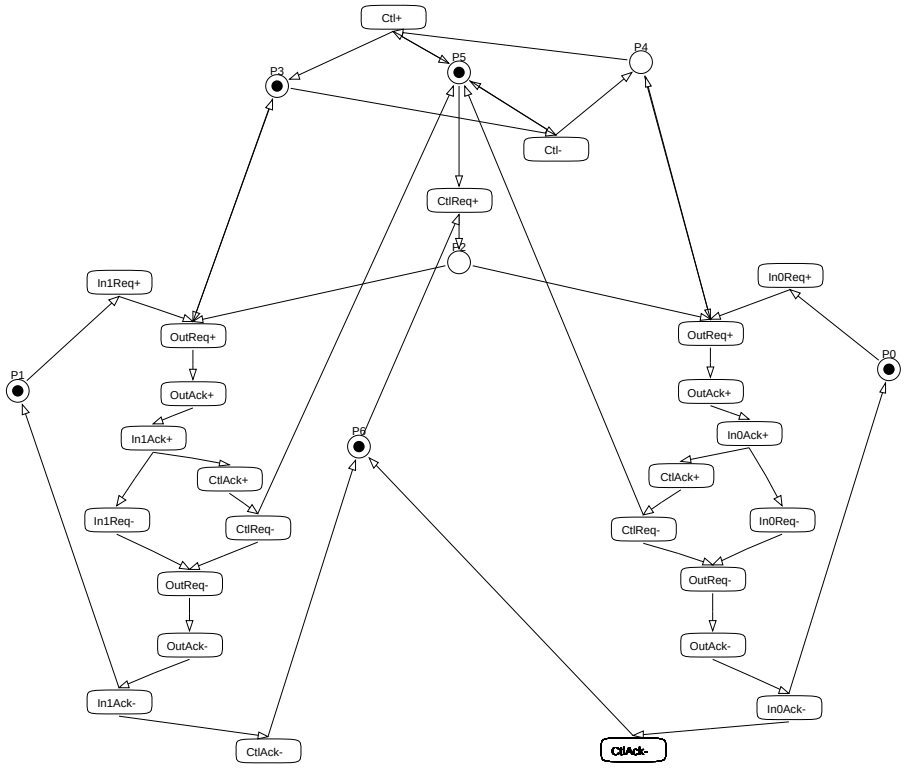


Figure 6.26. The modified STG specification of the 4-phase bundled-data MUX control circuit.

In figure 6.26 we have removed some concurrency from the MUX STG by ordering the transitions on $In0Ack/In1Ack$ and $CtlAck$ ($In0Ack+ \prec CtlAck+$, $In1Ack+ \prec CtlAck+$ etc.). This STG satisfies CSC and the resulting circuit is marginally smaller:

```
> more MUX4p-gcm.eqn
# EQN file for model MUX4pB
# Generated by petrify 4.0 (compiled 22-Dec-98 at 6:58 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 27.00
```

```
INORDER = In0Req OutAck In1Req Ctl CtlReq In1Ack In0Ack OutReq CtlAck;
OUTORDER = [In1Ack] [In0Ack] [OutReq] [CtlAck];
[0] = Ctl CtlReq OutAck;
[1] = Ctl' CtlReq OutAck;
[2] = CtlReq (Ctl' In0Req + Ctl In1Req);
[3] = CtlReq' (In0Ack' In1Req' + In0Req' In0Ack);
[OutReq] = OutReq [3]' + [2];              # mappable onto gC
[CtlAck] = In1Ack + In0Ack;
[In1Ack] = In1Ack OutAck + [0];            # mappable onto gC
[In0Ack] = In0Ack OutAck + [1];            # mappable onto gC
```

6.9. Summary

This chapter has provided an introduction to the design of asynchronous sequential (control) circuits with the main focus on speed-independent circuits and specifications using STGs. The material was presented from a practical view in order to enable the reader to go ahead and design his or her own speed-independent control circuits. This, rather than comprehensiveness, has been our goal, and as mentioned in the introduction we have largely ignored important alternative approaches including burst-mode and fundamental-mode circuits.

Chapter 7

ADVANCED 4-PHASE BUNDLED-DATA PROTOCOLS AND CIRCUITS

The previous chapters have explained the basics of asynchronous circuit design. In this chapter we will address 4-phase bundled-data protocols and circuits in more detail. This will include: (1) a variety of channel types, (2) protocols with different data-validity schemes, and (3) a number of more sophisticated latch control circuits. These latch controllers are interesting for two reasons: they are very useful in optimizing the circuits for area, power and speed, and they are nice examples of the types of control circuits that can be specified and synthesized using the STG-based techniques from the previous chapter.

7.1. Channels and protocols

7.1.1 Channel types

So far we have considered only *push channels* where the sender is the active party that initiates the communication of data, and where the receiver is the passive party. The opposite situation, where the receiver is the active party that initiates the communication of data, is also possible, and such a channel is called a *pull channel*. A channel that carries no data is called a *nonput channel* and is used for synchronization purposes. Finally, it is also possible to communicate data from a receiver to a sender along with the acknowledge signal. Such a channel is called a *biput channel*. In a 4-phase bundled-data implementation data from the receiver is bundled with the acknowledge, and in a 4-phase dual-rail protocol the passive party will acknowledge the reception of a codeword by returning a codeword rather than just an an acknowledge signal. Figure 7.1 illustrates these four channel types (nonput, push, pull, and biput) assuming a bundled-data protocol. Each channel type may, of course, use any of the handshake protocols (2-phase or 4-phase) and data encodings (bundled-data, dual-rail, $m-$of$-n$, etc.) introduced previously.

7.1.2 Data-validity schemes

For the bundled-data protocols it is also relevant to define the time interval in which data is valid, and figure 7.2 illustrates the different possibilities.

For a push channel the request signal carries the message "here is new data for you" and the acknowledge signal carries the information "thank you, I have absorbed the data, and you may release the data wires." Similarly, for a pull channel the request signal carries the message "please send new data" and the acknowledge signal carries the message "here is the data that you requested." It is the signal transitions on the request and acknowledge wires that are interpreted in this way. A 4-phase handshake involves two transitions on each wire and, depending on whether it is the rising or the falling transitions on the request and acknowledge signals that are interpreted in this way, several data-validity schemes emerge: early, broad, late and extended early.

Since 2-phase handshaking does not involve any redundant signal transitions there is only one data-validity scheme for each channel type (push or pull), as illustrated in figure 7.2.

It is common to all of the data-validity schemes that the data is valid some time before the event that indicates the start of the interval, and that it remains stable until some time after the event that indicates the end of the interval. Furthermore, all of the data-validity schemes express the requirements of the party that receives the data. The fact that a receiver signals "thank you, I have absorbed the data, and you may go ahead and release the data wires," does not mean that this actually happens – the sender may prolong the data-validity interval, and the receiver may even rely on this.

A typical example of this is the extended-early data-validity schemes in figure 7.2. On a push channel the data-validity interval begins some time before $Req \uparrow$ and ends some time after $Req \downarrow$.

7.1.3 Discussion

The above classification of channel types and handshake protocols stems mostly from Peeters' Ph.D. thesis [86]. The choice of channel type, handshake protocol and data-validity scheme obviously affects the implementation of the handshake components in terms of area, speed, and power. Just as a design may use a mix of different bundled-data and dual-rail protocols, it may also use a mix of channel types and data-validity schemes.

For example, a 4-phase bundled-data push channel using a broad or an extended-early data-validity scheme is a very convenient input to a function block that is implemented using precharged CMOS circuitry: the request signal may directly control the precharge and evaluate transistors because the broad and the extended-early data-validity schemes guarantee that the input data is stable during the evaluate phase.

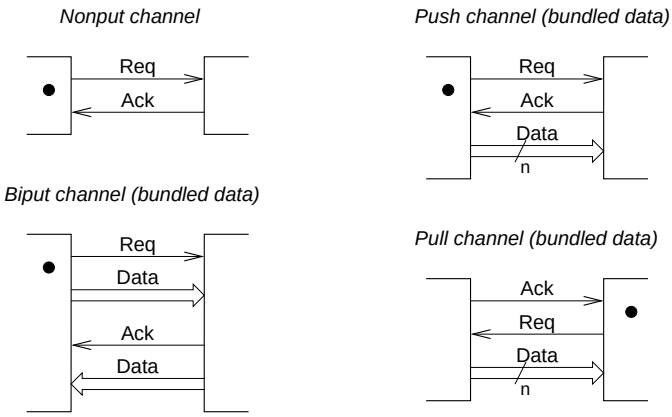


Figure 7.1. The four fundamental channel types: nonput, push, biput, and pull.

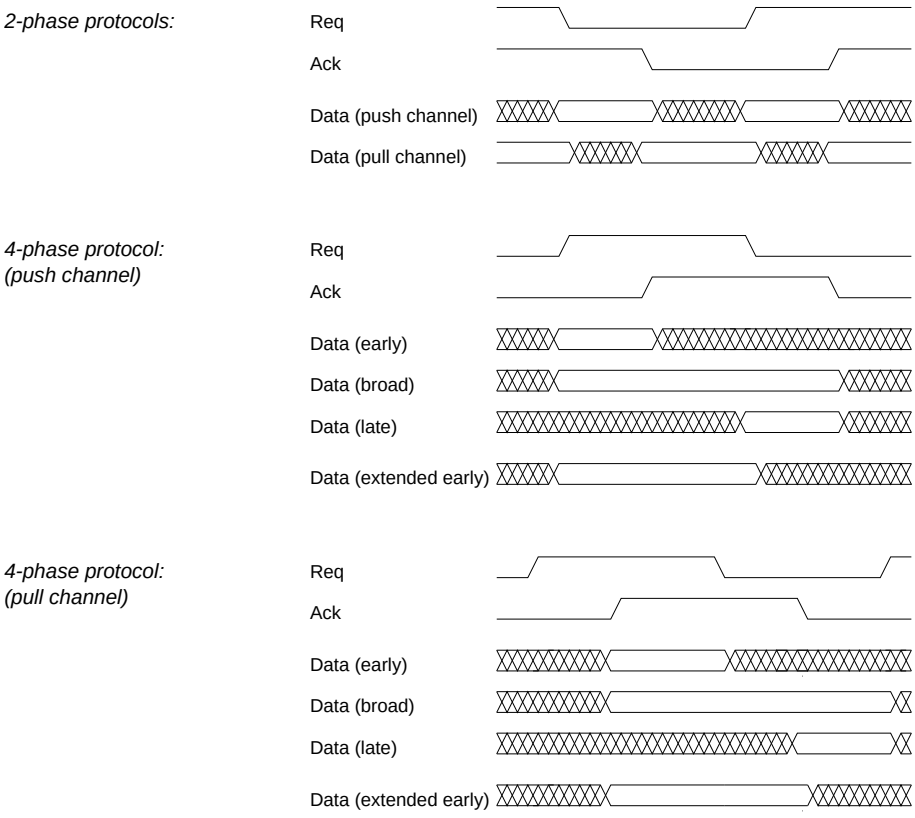


Figure 7.2. Data-validity schemes for 2-phase and 4-phase bundled data.

Another interesting option in a 4-phase bundled-data design is to use function blocks that assume a broad data validity scheme on the input channel and that produce a late data validity scheme on the output channel. Under these assumptions it is possible to use a *symmetric* delay element that matches only half of the latency of the combinatorial circuit. The idea is that the *sum* of the delay of $Req\uparrow$ and $Req\downarrow$ matches the latency of the combinatorial circuit, and that $Req\downarrow$ indicates valid output data. In [86, p.46] this is referred to as *true single phase* because the return-to-zero part of the handshaking is no longer redundant. This approach also has implications for the implementation of the components that connect to the function block.

It is beyond the scope of this text to enter into a discussion of where and when to use the different options. The interested reader is referred to [86, 54] for more details.

7.2. Static type checking

When designing circuits it is useful to think of the combination of channel type and data-validity scheme as being similar to a data type in a programming language, and to do some static type checking of the circuit being designed by asking questions like: "what types are allowed on the input ports of this handshake component?" and "what types are produced on the output ports of this handshake component?". The latter may depend on the type that was provided on the input port. A similar form of type checking for synchronous circuits using two-phase non-overlapping clocks has been proposed in [78] and used in the Genesil silicon compiler [47].

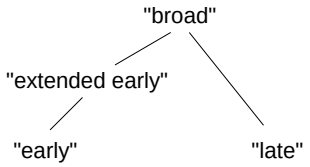


Figure 7.3. Hierarchical ordering of the four data-validity schemes for the 4-phase bundled-data protocol.

Figure 7.3 shows a hierarchical ordering of the four possible types (data validity schemes) for a 4-phase bundled-data push channel: "broad" is the strongest type and it can be used as input to circuits that require any of the weaker types. Similarly "extended early" may be used where only "early" is required. Circuits that are transparent to handshaking (function blocks, join, fork, merge, mux, demux) produce outputs whose type is at most as strong as their (weakest) input type. In general the input and output types are the same but there are examples where this is not the case. The only circuit that can

produce outputs whose type is stronger than the input type is a latch. Let us look at some examples:

- A join that concatenates two inputs of type "extended early" produces outputs that are only "early.'
- From the STG fragments in figure 6.21 on page 107 it may be seen that the simple 4-phase bundled-data latch controller from the previous chapters (figure 2.9 on page 18) assumes "early" inputs and that it produces "extended-early" outputs.
- The 4-phase bundled-data MUX design in section 6.8.3 assumes "extended early" on its control input (the STG in figure 6.25 on page 110 specifies stable input from $CtlReq+$ to $CtlReq-$).

The reader is encouraged to continue this exercise and perhaps draw the associated timing diagrams from which the types of the outputs may be deduced. The type checking explained here is a very useful technique for debugging circuits that exhibit erronous behaviour.

7.3. More advanced latch control circuits

In previous chapters we have only considered 4-phase bundled-data handshake latches using a latch controller consisting of a C-element and an inverter (figure 2.9 on page 18). In [31] this circuit is called a *simple* latch controller, and in [54] it is called an *un-decoupled* latch controller.

When a pipeline or FIFO that uses the simple latch controller fills, every second handshake latch will be holding a valid token and every other handshake

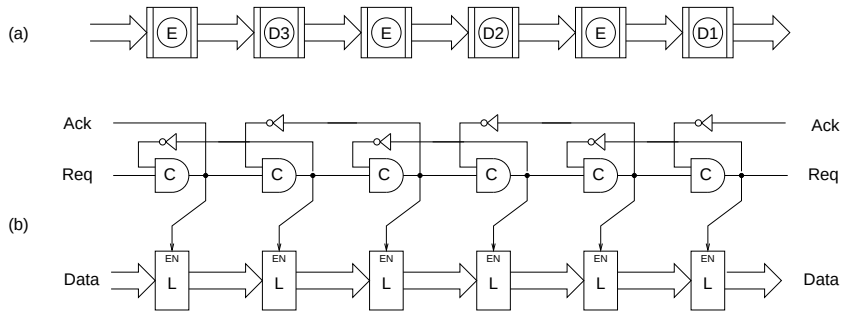


Figure 7.4. (a) A FIFO based on handshake latches, and (b) its implementation using simple latch controllers and level-sensitive latches. The FIFO fills with valid data in every other latch. A latch is transparent when $EN = 0$ and it is opaque (holding data) when $EN = 1$.

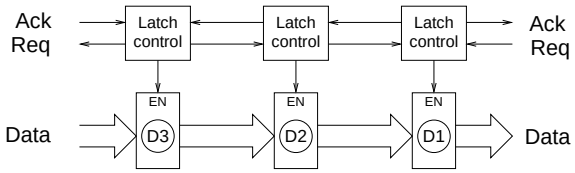


Figure 7.5. A FIFO where every level-sensitive latch holds valid data when the FIFO is full. The semi-decoupled and fully-decoupled latch controllers from [31] allow this behaviour.

latch will be holding an empty token as illustrated in figure 7.4(a) – the static spread of the pipeline is $S = 2$.

This token picture is a bit misleading. The empty tokens correspond to the return-to-zero part of the handshaking and in reality the latches are not "holding empty tokens" – they are transparent, and this represents a waste of hardware resource.

Ideally one would want to store a valid token in every level-sensitive latch as illustrated in figure 7.5 and just "add" the empty tokens to the data stream on the interfaces as part of the handshaking. In [31] Furber and Day explain the design of two such improved 4-phase bundled-data latch control circuits: a *semi-decoupled* and a *fully-decoupled* latch controller. In addition to these specific circuits the paper also provides a nice illustration of the use of STGs for designing control circuits following the approach explained in chapter 6. The three latch controllers have the following characteristics:

- The *simple or un-decoupled* latch controller has the problem that new input data can only be latched when the previous handshake on the output channel has completed, i.e., after $A_{out}\downarrow$. Furthermore, the handshakes on the input and output channels interact tightly: $R_{out}\uparrow \preceq A_{in}\uparrow$ and $R_{out}\downarrow \preceq A_{in}\downarrow$.
- The *semi-decoupled* latch controller relaxes these requirements somewhat: new inputs may be latched after $R_{out}\downarrow$, and the controller may produce $A_{in}\uparrow$ independently of the handshaking on the output channel – the interaction between the input and output channels has been relaxed to: $A_{out}\uparrow \preceq A_{in}\uparrow$.
- The *fully-decoupled* latch controller further relaxes these requirements: new inputs may be latched after $A_{out}\uparrow$ (i.e. as soon as the downstream latch has indicated that it has latched the current data), and the handshaking on the input channel may complete without any interaction with the output channel.

Another potential drawback of the simple latch controller is that it is unable to take advantage of function blocks with asymmetric delays as explained in

Latch controller	Static spread, S	Period, P
"Simple"	2	$2L_r + 2L_{f.V}$
"Semi-decoupled"	1	$2L_r + 2L_{f.V}$
"Fully-decoupled"	1	$2L_r + L_{f.V} + L_{f.E}$

Table 7.1. Summary of the characteristics of the latch controllers in [31].

section 4.4.1 on page 52. The fully-decoupled latch controller presented in [31] does not have this problem. Due to the decoupling of the input and output channels the dependency graph critical cycle that determines the period, P, only visits nodes related to two neighbouring pipeline stages and the period becomes minimum (c.f. section 4.4.1). Table 7.1 summarizes the characteristics of the simple, semi-decoupled and fully-decoupled latch controllers.

All of the above-mentioned latch controllers are "normally transparent" and this may lead to excessive power consumption because inputs that make multiple transitions before settling will propagate through several consecutive pipeline stages. By using "normally opaque" latch controllers every latch will act as a barrier. If a handshake latch that is holding a bubble is exposed to a token on its input, the latch controller switches the latch into the transparent mode, and when the input data have propagated safely into the latch, it will switch the latch back into the opaque mode in which it will hold the data. In the design of the asynchronous MIPS processor reported in [15] we experienced approximately a 50 % power reduction when using normally opaque latch controllers instead of normally transparent latch controllers.

Figure 7.6 shows the STG specification and the circuit implementation of the normally opaque latch controller used in [15]. As seen from the STG there is quite a strong interaction between the input and output channels, but the dependency graph critical cycle that determines the period only visits nodes related to two neighbouring pipeline stages and the period is minimum. It may be necessary to add some delay into the $Lt+$ to $Rout+$ path in order to ensure that input signals have propagated through the latch before $Rout+$. Furthermore the duration of the $Lt = 0$ pulse that causes the latch to be transparent is determined by gate delays in the latch controller itself, and the pulse must be long enough to ensure safe latching of the data. The latch controller assumes a broad data-validity scheme on its input channel and it provides a broad data-validity scheme on its output channel.

7.4. Summary

This chapter introduced a selection of channel types, data-validity schemes and a selection of latch controllers. The presentation was rather brief; the aim was just to present the basics and to introduce some of the many options and

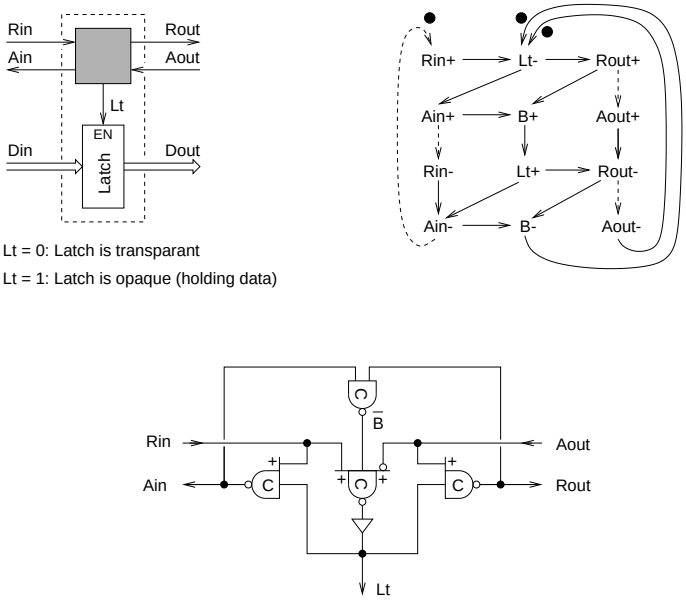


Figure 7.6. The STG specification and the circuit implementation of the normally opaque fully-decoupled latch controller from [15].

possibilities for optimizing the circuits. The interested reader is referred to the literature for more details.

Finally a warning: the static data-flow view of asynchronous circuits presented in chapter 3 (i.e. that valid and empty tokens are copied forward controlled by the handshaking between latch controllers) and the performance analysis presented in chapter 4 assume that all handshake latches use the simple normally transparent latch controller. When using semi-decoupled or fully-decoupled latch controllers, it is necessary to modify the token flow view, and to rework the performance analysis. To a first order one might substitute each semi-decoupled or fully-decoupled latch controller with a pair of simple latch controllers. Furthermore a ring need only include two handshake latches if semi-decoupled or fully-decoupled latch controllers are used.

Chapter 8

HIGH-LEVEL LANGUAGES AND TOOLS

This chapter addresses languages and CAD tools for the high-level modeling and synthesis of asynchronous circuits. The aim is briefly to introduce some basic concepts and a few representative and influential design methods. The interested reader will find more details elsewhere in this book (in Part II and chapter 13) as well as in the original papers that are cited in the text. In the last section we address the use of VHDL for the design of asynchronous circuits.

8.1. Introduction

Almost all work on the high-level modeling and synthesis of asynchronous circuits is based on the use of a language that belongs to the CSP family of languages, rather than one of the two industry-standard hardware description languages, VHDL and Verilog. Asynchronous circuits are highly *concurrent* and communication between modules is based on *handshake channels*. Consequently a hardware description language for asynchronous circuit design should provide efficient primitives supporting these two characteristics. The CSP language proposed by Hoare [42, 43] meets these requirements. CSP stands for "Communicating Sequential Processes" and its key characteristics are:

- Concurrent processes.
- Sequential and concurrent composition of statements within a process.
- Synchronous message passing over point-to-point channels (supported by the primitives send, receive and – possibly – probe).

CSP is a member of a large family of languages for programming concurrent systems in general: OCCAM [48], LOTOS [82, 8], and CCS [64], as well as languages defined specifically for designing asynchronous circuits: Tangram [108, 101], CHP [58], and Balsa [3, 4]. Further details are presented elsewhere in this book on Tangram (in Part III, chapter 13) and Balsa (in Part II).

In this chapter we first take a closer look at the CSP language constructs supporting communication and concurrency. This will include a few sample programs to give a flavour of this type of language. Following this we briefly

explain two rather different design methods that both take a CSP-like program as the starting point for the design:

- At Philips Research Laboratories, van Berkel, Peeters, Kessels et al. have developed a proprietary language, Tangram, and an associated silicon compiler [108, 107, 101, 86]. Using a process called syntax-directed compilation, the synthesis tool maps a Tangram program into a structure of handshake components. Using these tools several significant asynchronous chips have been designed within Philips [103, 104, 109, 50, 51]. The last of these is a smart-card circuit that is described in chapter 13 on page 221.

- At Caltech Martin has developed a language CHP – Communicating Hardware Processes – and a set of tools that supports a partly manual, partly automated design flow that targets highly optimized transistor-level implementations of QDI 4-phase dual-rail circuits [57, 60].

CHP has a syntax that is similar to CSP (using various special symbols) whereas Tangram has a syntax that is more like a traditional programming language (using keywords); but in essence they are both very similar to CSP.

In the last section of this chapter we will introduce a VHDL-package that provides CSP-like message passing and explain an associated VHDL-based design flow that supports a manual step-wise refinement design process.

8.2. Concurrency and message passing in CSP

The "sequential processes" part of the CSP acronym denotes that each process is described by a program whose statements are executed in sequence one by one. A semicolon is used to separate statements (as in many other programming languages). The semicolon can be seen as an operator that combines statements into programs. In this respect a process in CSP is very similar to a process in VHDL. However, CSP also allows the parallel composition of statements within a process. The symbol "||" denotes parallel composition. This feature is not found in VHDL, whereas the fork-join construct in Verilog does allow statement-level concurrency within a process.

The "communicating" part of the CSP acronym refers to synchronous message passing using point-to-point channels as illustrated in figure 8.1, where two processes $P1$ and $P2$ are connected by a channel named C. Using a send statement, $C!x$, process $P1$ sends (denoted by the '!' symbol) the value of its variable x on channel C, and using a receive statement, $C?y$, process $P2$ receives (denoted by the '?' symbol) from channel C a value that is assigned to its variable y. The channel is memoryless and the transfer of the value of variable x in $P1$ into variable y in $P2$ is an atomic action. This has the effect of synchronizing processes $P1$ and $P2$. Whichever comes first will wait for

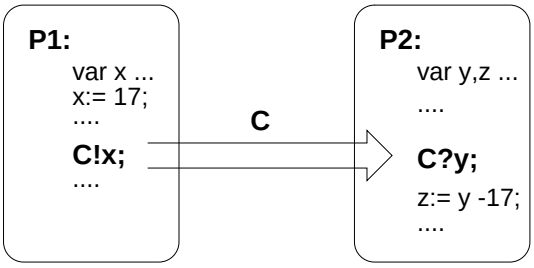


Figure 8.1. Two processes $P1$ and $P2$ connected by a channel C. Process $P1$ sends the value of its variable x to the channel C, and process $P2$ receives the value and assigns it to its variable y.

the other party, and the send and receive statements complete at the same time. The term *rendezvous* is sometimes used for this type of synchronization.

When a process executes a send (or receive) statement, it commits to the communication and suspends until the process at the other end of the channel performs its receive (or send) statement. This may not always be desirable, and Martin has extended CSP with a probe construct [56] which allows the process at the passive end of a channel to probe whether or not a communication is pending on the channel, without committing to any communication. The probe is a function which takes a channel name as its argument and returns a Boolean. The syntax for probing channel C is $\overline{C}$.

As an aside we mention that some languages for programming concurrent systems assume channels with (possibly unbounded) buffering capability. The implication of this is that the channel acts as a FIFO, and the communicating processes do not synchronize when they communicate. Consequently this form of communication is called asynchronous message passing.

Going back to our synchronous message passing, it is obvious that the physical implementation of a memoryless channel is simply a set of wires together with a protocol for synchronizing the communicating processes. It is also obvious that any of the protocols that we have considered in the previous chapters may be used. Synchronous message passing is thus a very useful language construct that supports the high-level modeling of asynchronous circuits by abstracting away the exact details of the data encoding and handshake protocol used on the channel.

Unfortunately both VHDL and Verilog lack such primitives. It is possible to write low-level code that implements the handshaking, but it is highly undesirable to mix such low-level details into code whose purpose is to capture the high-level behaviour of the circuit.

In the following section we will provide some small program examples to give a flavour of this type of language. The examples will be written in Tangram as they also serve the purpose of illustrating syntax-directed compilation in a

subsequent section. The source code, handshake circuit figures, and fragments of the text have been kindly provided by Ad Peeters from Philips.

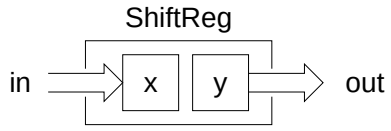
Manchester University has recently developed a similar language and synthesis tool that is available in the public domain [4], and is introduced in Part II of this book. Other examples of related work are presented in [9] and [13].

8.3. Tangram: program examples

This section provides a few simple Tangram program examples: a 2-place shift register, a 2-place ripple FIFO, and a greatest common divisor function.

8.3.1 A 2-place shift register

Figure 8.2 shows the code for a 2-place shift register named `ShiftReg`. It is a process with an input channel `In` and an output channel `Out`, both carrying variables of type `[0..255]`. There are two local variables `x` and `y` that are initialized to `0`. The process performs an unbounded repetition of a sequence of three statements: `out!y`; `y:=x`; `in?x`.

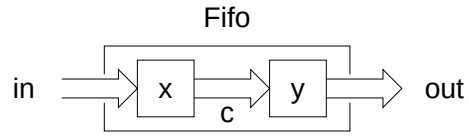



```
  T = type [0..255]
& ShiftReg : main proc(in? chan T & out! chan T).
  begin
  & var x,y: var T := 0
    |
    forever do
      out!y ; y:=x ; in?x
    od
  end
```

Figure 8.2. A Tangram program for a 2-place shift register.

8.3.2 A 2-place (ripple) FIFO

Figure 8.3 shows the Tangram program for a 2-place first-in first-out buffer named `Fifo`. It can be understood as two 1-place buffers that are operating in parallel and that are connected by a channel `c`. At first sight it appears very similar to the 2-place shift register presented above, but a closer examination will show that it is more flexible and exhibits greater concurrency.




```
  T = type [0..255]
& Fifo : main proc(in? chan T & out! chan T).
  begin
  & x,y: var T
  & c : chan T
  |
       forever do in?x ; c!x   od
    || forever do c?y  ; out!y od
  end
```

Figure 8.3. A Tangram program for a 2-place (ripple) FIFO.

8.3.3 GCD using while and if statements

Figure 8.4 shows the code for a module that computes the greatest common divisor, the example from section 3.7. The "`do x<>y then ... od`" is a while statement and, apart from the syntactical differences, the code in figure 8.4 is identical to the code in figure 3.11 on page 39.

The module has an input channel from which it receives the two operands, and an output channel on which it sends the result.

```
  int = type [0..255]
& gcd_if : main proc (in?chan <<int,int>> & out!chan int).
  begin x,y:var int ff
  | forever do
      in?<<x,y>>
    ; do x<>y then
        if x<y then y:=y-x
               else x:=x-y
        fi
      od
    ; out!x
    od
  end
```

Figure 8.4. A Tangram for GCD using while and if statements.

8.3.4 GCD using guarded commands

Figure 8.5 shows an alternative version of GCD. This time the module has separate input channels for the two operands and its body is based on the repetition of a guarded command. The guarded repetition can be seen as a generalization of the while statement. The statement repeats until all guards are false. When at least one of the guards is true, exactly one command corresponding to such a true guard is selected (either deterministically or non-deterministically) and executed.

```
  int = type [0..255]
& gcd_gc : main proc (in1,in2?chan int & out!chan int).
  begin x,y:var int ff
  | forever do
      in1?x || in2?y
    ; do x<y then y:=y-x
      or y<x then x:=x-y
      od
    ; out!x
    od
  end
```

Figure 8.5. A Tangram program for GCD using guarded repetition.

8.4. Tangram: syntax-directed compilation

Let us now address the synthesis process. The design flow uses an intermediate format based on handshake circuits. The front-end design activity is called VLSI programming and, using syntax-directed compilation, a Tangram program is mapped into a structure of handshake components. There is a one-to-one correspondence between the Tangram program and the handshake circuit as will be clear from the following examples. The compilation process is thus fully transparent to the designer, who works entirely at the Tangram program level.

The back-end of the design flow involves a library of handshake circuits that the compiler targets as well as some tools for post-synthesis peephole optimization of the handshake circuits (i.e. replacing common structures of handshake components by more efficient equivalent ones). A number of handshake circuit libraries exist, allowing implementations using different handshake protocols (4-phase dual-rail, 4-phase bundled-data, etc.), and different implementation technologies (CMOS standard cells, FPGAs, etc.). The handshake components can be specified and designed: (i) manually, or (ii) using STGs and Petrify as explained in chapter 6, or (iii) using the lower steps in Martin's transformation-based method that is presented in the next section.

It is beyond the scope of this text to explain the details of the compilation process. We will restrict ourselves to providing a flavour of "syntax-directed compilation" by showing handshake circuits corresponding to the example Tangram programs from the previous section.

8.4.1 The 2-place shift register

As a first example of syntax-directed compilation figure 8.6 shows the handshake circuit corresponding to the Tangram program in figure 8.2.

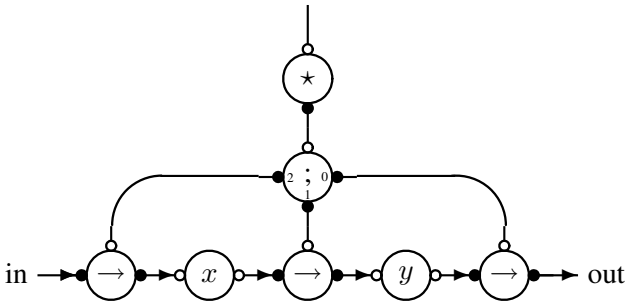


Figure 8.6. The compiled handshake circuit for the 2-place shift register.

Handshake components are represented by circular symbols, and the channels that connect the components are represented by arcs. The small dots on the component symbols represent ports. An open dot denotes a passive port and a solid dot denotes an active port. The arrowhead represents the direction of the data transfer. A nonput channel does not involve the transfer of data and consequently it has no direction and no arrowhead. As can be seen in figure 8.6 a handshake circuit uses a mix of push and pull channels.

The structure of the program is a forever-do statement whose body consists of three statements that are executed sequentially (because they are separated by semicolons). Each of the three statements is a kind of assignment statement: the value of variable `y` is "assigned" to output channel `out`, the value of variable `x` is assigned to variable `y`, and the value received on input chanel `in` is assigned to variable `x`. The structure of the handshake circuit is exactly the same:

- At the top is a *repeater* that implements the forever-do statement. A repeater waits for a request on its passive input port and then it performs an unbounded repetition of handshakes on its active output channel. The handshake on the input channel never completes.
- Below is a 3-way *sequencer* that implements the semicolons in the program text. The sequencer waits for a request on its passive input channel, then it performs in sequence a full handshake on each of its active out-

put channels (in the order indicated by the numbers in the symbol) and finally it completes the handshaking on the passive input channel. In this way the sequencer activates in turn the handshake circuit constructs that correspond to the individual statements in the body of the forever-do statement.

- The bottom row of handshake components includes two *variables*, `x` and `y`, and three *transferers*, denoted by '→'. Note that variables have passive read and write ports. The transferers implement the three statements (`out!y; y:=x; in?x`) that form the body of the forever-do statement, each a form of assignment. A transferer waits for a request on its passive nonput channel and then initiates a handshake on its pull input channel. The handshake on the pull input channel is relayed to the push output channel. In this way the transferer pulls data from its input channel and pushes it onto its output channel. Finally, it completes the handshaking on the passive nonput channel.

8.4.2 The 2-place FIFO

Figure 8.7 shows the handshake circuit corresponding to the Tangram program in figure 8.3. The component labeled 'psv' in the handshake circuit of figure 8.7 is a so-called *passivator*. It relates to the internal channel `c` of the `Fifo` and implements the synchronization and communication between the active sender (`c!x`) and the active receiver (`c?y`).

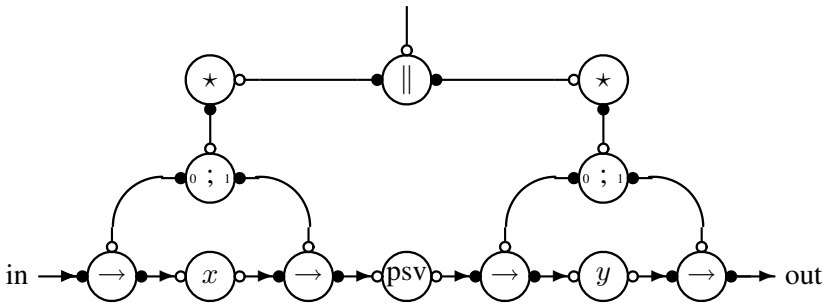


Figure 8.7. Compiled handshake circuit for the FIFO program.

An optimization of the handshake circuit for `Fifo` is shown in figure 8.8. The synchronization in the datapath using a passivator has been replaced by a synchronization in the control using a 'join' component. One may observe that the datapath of this handshake circuit for the FIFO design is the same as that of the shift register, shown in figure 8.2. The only difference is in the control part of the circuits.

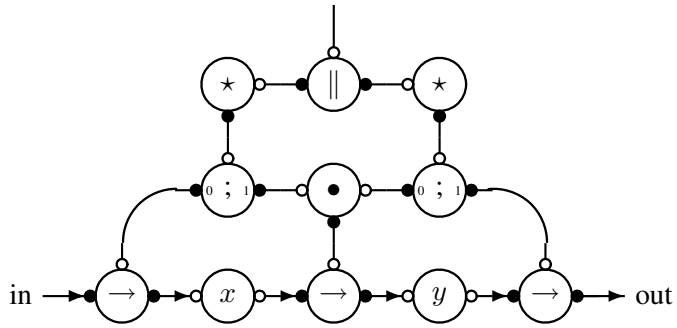


Figure 8.8. Optimized handshake circuit for the FIFO program.

8.4.3 GCD using guarded repetition

As a more complex example of syntax-directed compilation figure 8.9 shows the handshake circuit compiled from the Tangram program in figure 8.5. Compared with the previous handshake circuits, the handshake circuit for the GCD program introduces two new classes of components that are treated in more detail below.

Firstly, the circuit contains a 'bar' and a 'do' component, both of which are data-dependent control components. Secondly, the handshake circuit contains components that do not directly correspond to language constructs, but rather implement sharing: the multiplexer (denoted by 'mux'), the demultiplexer (denoted by 'dmx'), and the fork component (denoted by '•').

Warning: the Tangram fork is identical to the fork in figure 3.3 but the Tangram multiplexer and demultiplexer components are different. The Tangram multiplexer is identical to the merge in figure 3.3 and the Tangram demultiplexer is a kind of "inverse merge." Its output ports are passive and it requires the handshakes on the two outputs to be mutually exclusive.

The 'bar' and the 'do' components: The do and bar component together implement the guarded command construct with two guards, in which the do component implements the iteration part (the `do od` part, including the evaluation of the disjunction of the two guards), and the bar component implements the choice part (the `then or then` part of the command).

The do component, when activated through its passive port, first collects the disjunction of the value of all guards through a handshake on its active data port. When the value thus collected is true, it activates its active nonput port (to activate the selected command), and after completion starts a new evaluation cycle. When the value collected is false, the do component completes its operation by completing the handshake on the passive port.

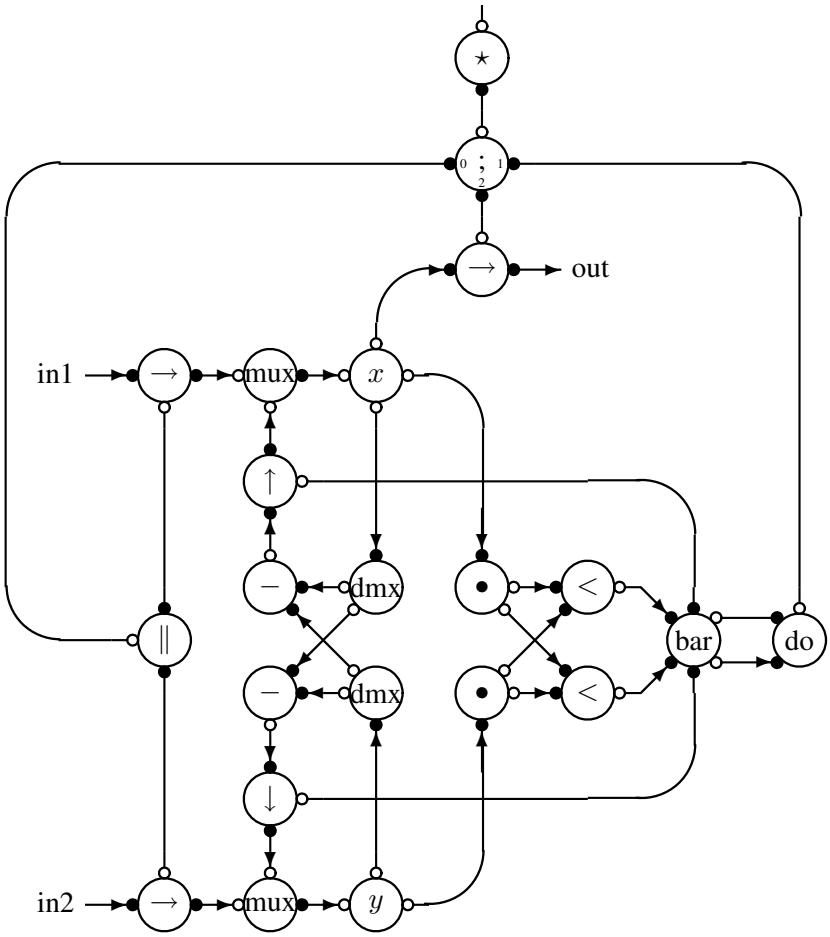


Figure 8.9. Compiled handshake circuit for the GCD program using guarded repetition.

The bar component can be activated either through its passive data port, or through its passive control port. (The do component, for example, sequences these two activations.) When activated through the data port, it collects the value of two guards through a handshake on the active data ports, and then sends the disjunction of these values along the passive data port, thus completing that handshake. When activated through the control port, the bar component activates an active control port of which the associated data port returned a 'true' value in the most recent data cycle. (For simplicity, this selection is typically implemented in a deterministic fashion, although this is not required at the level of the program.) One may observe that bar components can be

combined in a tree or list to implement a guarded command list of arbitrary length. Furthermore, not every data cycle has to be followed by a control cycle.

The 'mux', 'demux', and 'fork' components The program for GCD in figure 8.4 has two occurrences of variable `x` in which a value is written into `x`, namely input action `in1?x` and assignment `x:=x-y`. In the handshake circuit of figure 8.9, these two write actions for Tangram variable `x` are merged by the multiplexer component so as to arrive at the write port of handshake variable `x`.

Variable `x` occurs at five different locations in the program as an expression, once in the output expression `out!x`, twice in the guard expressions `x<y` and `y<x`, and twice in the assignment expressions `x-y` and `y-x`. These five inspections of variable `x` could be implemented as five distinct read ports on the handshake variable `x`, which is shown in the handshake circuit in [101, Fig. 2.7, p.34]. In figure 8.9, a different compilation is shown, in which handshake variable `x` has three read ports:

- A read port dedicated to the occurrence in the output action.
- A read port dedicated to the guard expressions. Their evaluation is *mutually inclusive*, and hence can be combined using a synchronizing fork component.
- A read port dedicated to the assignment expressions. Their evaluation is *mutually exclusive*, and hence can be combined using a demultiplexer.

The GCD example is discussed in further detail in chapter 13.

8.5. Martin's translation process

The work of Martin and his group at Caltech has made fundamental contributions to asynchronous design and it has influenced the work of many other researchers. The methods have been used at Caltech to design several significant chips, most recently and most notably an asynchronous MIPS R3000 processor [63]. As the following presentation of the design flow hints, the design process is elaborate and sophisticated and is probably only an option to a person who has spent time with the Caltech group.

The mostly manual design process involves the following steps (semantics-preserving transformations):

(1) *Process decomposition* where each process is refined into a collection of interacting simpler processes. This step is repeated until all processes are simple enough to be dealt with in the next step in the process.

(2) *Handshake expansion* where each communication channel is replaced by explicit wires and where each communication action (e.g. send or receive) is replaced by the signal transitions required by the protocol that is being used.

For example a receive statement such as:

$$C?y$$

is replaced by a sequence of simpler statements – for example:

$$[C_{req}];\ y := data;\ C_{ack}\uparrow;\ [\neg C_{req}];\ C_{ack}\downarrow$$

which is read as: "wait for request to go high", "read the data", "drive acknowledge high", "wait for request to go low", and "drive acknowledge low".

At this level it may be necessary to add state variables and/or to reshuffle signal transitions in order to obtain a specification that satisfies a condition similar to the CSC condition in chapter 6.

(3) *Production rule expansion* where each handshaking expansion is replaced by a set of production rules (or guarded commands), for example:

$$a \wedge b \mapsto c\uparrow \quad \text{and} \quad \neg b \wedge \neg c \mapsto c\downarrow$$

A production rule consist of a condition and an action, and the action is performed whenever the condition is true. As an aside we mention that the above two production rules express the same as the set and reset functions for the signal c on page 96. The production rules specify the behaviour of the internal signals and output signals of the process. The production rules are themselves simple concurrent processes and the guards must ensure that the signal transitions occur in program order (i.e. that the semantics of the original CHP program are maintained). This may require strengthening the guards. Furthermore, in order to obtain simpler circuit implementations, the guards may be modified and made symmetric.

(4) *Operator reduction* where production rules are grouped into clusters and where each cluster is mapped onto a basic hardware component similar to a generalized C-element. The above two production rules would be mapped into the generalized C-element shown in figure 6.17 on page 100.

8.6. Using VHDL for asynchronous design

8.6.1 Introduction

In this section we will introduce a couple of VHDL packages that provide the designer with primitives for synchronous message passing between processes – similar to the constructs found in the CSP-family of languages (send, receive and probe).

The material was developed in an M.Sc. project and used in the design of a 32-bit floating-point ALU using the IEEE floating-point number representation [84], and it has subsequently been used in a course on asynchronous circuit design. Others, including [70, 89, 113, 55], have developed related VHDL packages and approaches.

The channel packages introduced in the following support only one type of channel, using a 32-bit 4-phase bundled-data push protocol. However, as VHDL allows the overloading of procedures and functions, it is straightforward to define channels with arbitrary data types. All it takes is a little cut-and-paste editing. Providing support for protocols other than the 4-phase bundled-data push protocol will require more significant extensions to the packages.

8.6.2 VHDL versus CSP-type languages

The previous sections introduced several CSP-like hardware description languages for asynchronous design. The advantages of these languages are their support of concurrency and synchronous message passing, as well as a limited and well-defined set of language constructs that makes syntax-directed compilation a relatively simple task.

Having said this there is nothing that prevents a designer from using one of the industry standard languages VHDL (or Verilog) for the design of asynchronous circuits. In fact some of the fundamental concepts in these languages – concurrent processes and signal events – are "nice fits" with the modeling and design of asynchronous circuits. To illustrate this figure 8.10 shows how the Tangram program from figure 8.2 could be expressed in plain VHDL. In addition to demonstrating the feasibility, the figure also highlights the limitations of VHDL when it comes to modeling asynchronous circuits: most of the code expresses low-level handshaking details, and this greatly clutters the description of the function of the circuit.

VHDL obviously lacks built-in primitives for synchronous message passing on channels similar to those found in CSP-like languages. Another feature of the CSP family of languages that VHDL lacks is statement-level concurrency within a process. On the other hand there are also some advantages of using an industry standard hardware description language such as VHDL:

- It is well supported by existing CAD tool frameworks that provide simulators, pre-designed modules, mixed-mode simulation, and tools for synthesis, layout and the back annotation of timing information.
- The same simulator and test bench can be used throughout the entire design process from the first high-level specification to the final implementation in some target technology (for example a standard cell layout).
- It is possible to perform mixed-mode simulations where some entities are modeled using behavioural specifications and others are implemented using the components of the target technology.
- Many real-world systems include both synchronous and asynchronous subsystems, and such hybrid systems can be modeled without any problems in VHDL.

```
library IEEE;
use IEEE.std_logic_1164.all;

  type T is std_logic_vector(7 downto 0)

  entity ShiftReg is
    port ( in_req   : in  std_logic;
           in_ack   : out std_logic;
           in_data  : in  T;
           out_req  : out std_logic;
           out_ack  : in  std_logic;
           out-data : out T );
  end ShiftReg;

  architecture behav of ShiftReg is
  begin
    process
      variable x, y: T;
    begin
      loop
        out_req <= '1' ;           --  out!y
        out_data <= y ;
        wait until out_ack = '1';
        out_req <= '0';
        wait until out_ack = '0';
        y := x;                    --  y := x
        wait until in_req = '1';   --  in?x
        x := in_data;
        in.ack <= '1';
        wait until ch_req = '0';
        ch_ack <= '0';
      end loop;
    end process;
  end behav;
```

Figure 8.10. VHDL description of the 2-place shift register FIFO stage from figure 8.2.

8.6.3 Channel communication and design flow

The design flow presented in what follows is motivated by the advantages mentioned above. The goal is to augment VHDL with CSP-like channel communication primitives, i.e. the procedures `send(<channel>, <variable>)` and `receive(<channel>,<variable>)` and the function `probe(<channel>)`. Another goal is to enable mixed-mode simulations where one end of a channel connects to an entity whose architecture body is a circuit implementation and the other end connects to an entity whose architecture body is a behavioural description using the above communication primitives, figure 8.11(b). In this way a *manual* top-down stepwise refinement design process is supported, where the same test bench is used throughout the entire design process from high-level specification to low-level circuit implementation, figure 8.11(a-c).

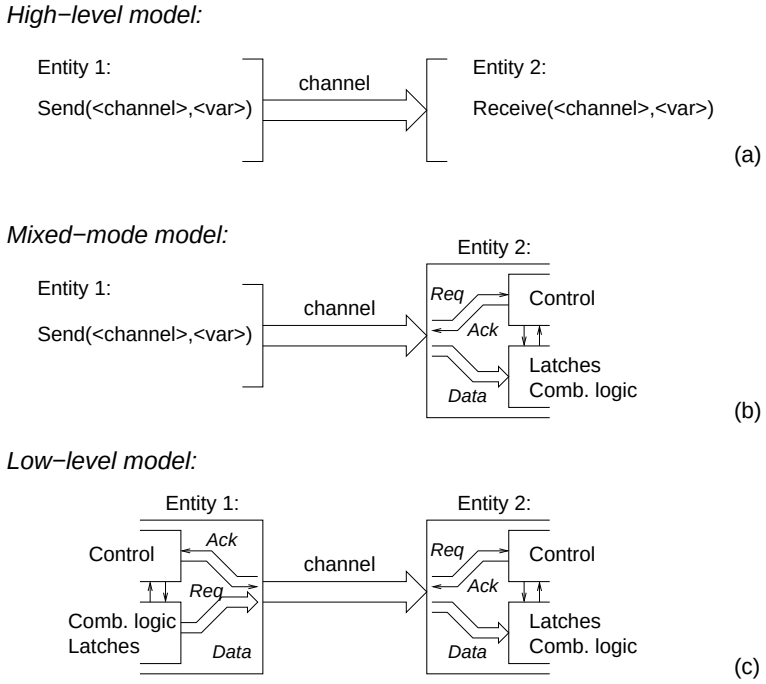


Figure 8.11. The VHDL packages for channel communication support high-level, mixed-mode and gate-level/standard cell simulations.

In VHDL all communication between processes takes place via signals. Channels therefore have to be declared as signals, preferably one signal per channel. Since (for a push channel) the sender drives the request and data part of a channel, and the receiver drives the acknowledge part, there are two drivers to one signal. This is allowed in VHDL if the signal is a resolved signal. Thus, it is possible to define a channel type as a record with a request, an acknowledge and a data field, and then define a resolution function for the channel type which will determine the resulting value of the channel. This type of channel, with separate request and acknowledge fields, will be called a *real channel* and is described in section 8.6.5. In simulations there will be three traces for each channel, showing the waveforms of request and acknowledge along with the data that is communicated.

A channel can also be defined with only two fields: one that describes the state of the handshaking (called the "handshake phase" or simply the "phase") and one containing the data. The type of the phase field is an enumerated type, whose values can be the handshake phases a channel can assume, as well as the values with which the sender and receiver can drive the field. This type of

channel will be called an *abstract channel.* In simulations there will be two traces for each channel, and it is easy to read the phases the channel assumes and the data values that are transfered.

The procedures and definitions are organized into two VHDL-packages: one called "abstpack.vhd" that can be used for simulating high-level models and one called "realpack.vhd" that can be used at all levels of design. Full listings can be found in appendix 8.A at the end of this chapter. The design flow enabled by these packages is as follows:

- The circuit and its environment or test bench is first modelled and simulated using abstract channels. All it takes is the following statement in the top level design unit: "`usepackage work.abstpack.all`".
- The circuit is then partitioned into simpler entities. The entities still communicate using channels and the simulation still uses the abstract channel package. This step may be repeated.
- At some point the designer changes to using the real channel package by changing to: "`usepackage work.realpack.all`" in the top-level design unit. Apart from this simple change, the VHDL source code is identical.
- It is now possible to partition entities into control circuitry (that can be designed as explained in chapter 6) and data circuitry (that consist of ordinary latches and combinational circuitry). Mixed mode simulations as illustrated in figure 8.11(b) are possible. Simulation models of the control circuits may be their actual implementation in the target technology or simply an entity containing a set of concurrent signal assignments – for example the Boolean equations produced by Petrify.
- Eventually, when all entities have been partitioned into control and data, and when all leaf entities have been implemented using components of the target technology, the design is complete. Using standard technology mapping tools an implementation may be produced, and the circuit can be simulated with back annotated timing information.

Note that the same simulation test bench can be used throughout the entire design process from the high-level specification to the low-level implementation using components from the target technology.

8.6.4 The abstract channel package

An abstract channel is defined in figure 8.12 with a data type called `fp` (a 32-bit standard logic vector representing an IEEE floating-point number). The actual channel type is called `channel_fp`. It is necessary to define a channel

```
type handshake_phase is
(
  u,          -- uninitialized
  idle,       -- no communication
  swait,      -- sender waiting
  rwait,      -- receiver waiting
  rcv,        -- receiving data
  rec1,       -- recovery phase 1
  rec2,       -- recovery phase 2
  req,        -- request signal
  ack,        -- acknowledge signal
  error       -- protocol error
);

subtype fp is std_logic_vector(31 downto 0);

type uchannel_fp is
  record
    phase : handshake_phase;
    data  : fp;
  end record;

type uchannel_fp_vector is array(natural range <>) of uchannel_fp;

function resolved(s : uchannel_fp_vector) return uchannel_fp;

subtype channel_fp is resolved uchannel_fp;
```

Figure 8.12. Definition of an abstract channel.

for each data type used in the design. The data type can be an arbitrary type, including record types, but it is advisable to use data types that are built from `std_logic` because this is typically the type used by target component libraries (such as standard cell libraries) that are eventually used for the implementation.

The meaning of the values of the type `handshake_phase` are described in detail below:

u: Uninitialized channel. This is the default value of the drivers. As long as either the sender or receiver drive the channel with this value, the channel stays uninitialized.

idle: No communication. Both the sender and receiver drive the channel with the `idle` value.

swait: The sender is waiting to perform a communication. The sender is driving the channel with the `req` value and the receiver drives with the `idle` value.

rwait: The receiver is waiting to perform a communication. The sender is driving the channel with the `idle` value and the receiver drives with the

`rwait` value. This value is used both as a driving value and as a resulting value for a channel, just like the `idle` and `u` values.

rcv: Data is transfered. The sender is driving the channel with the `req` value and the receiver drives it with the `rwait` value. After a predefined amount of time (`tpd` at the top of the package, see later in this section) the receiver changes its driving value to `ack`, and the channel changes its phase to `rec1`. In a simulation it is only possible to see the transfered value during the `rcv` phase and the `swait phase`. At all other times the data field assumes a predefined default data value.

rec1: Recovery phase. This phase is not seen in a simulation, since the channel changes to the `rec2` phase with no time delay.

rec2: Recovery phase. This phase is not seen in a simulation, since the channel changes to the `idle` phase with no time delay.

req: The sender drives the channel with this value, when it wants to perform a communication. A channel can never assume this value.

ack: The receiver drives the channel with this value when it wants to perform a communication. A channel can never assume this value.

error: Protocol error. A channel assumes this value when the resolution function detects an error. It is an error if there is more than one driver with an `rwait`, `req` or `ack` value. This could be the result if more than two drivers are connected to a channel, or if a `send` command is accidentally used instead of a `receive` command or vice versa.

Figure 8.13 shows a graphical illustration of the protocol of the abstract channel. The values in large letters are the resulting values of the channel, and the values in smaller letters below them are the driving values of the sender and receiver respectively. Both the sender and receiver are allowed to initiate a communication. This makes it possible in a simulation to see if either the

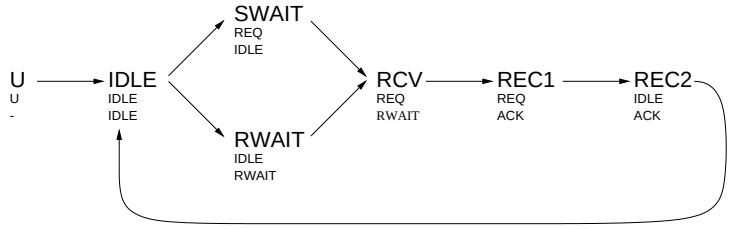


Figure 8.13. The protocol for the abstract channel. The values in large letters are the resulting resolved values of the channel, and the values in smaller letters below them are the driving values of the sender and receiver respectively.

sender or receiver is waiting to communicate. It is the procedures `send` and `receive` that follow this protocol.

Because channels with different data types are defined as separate types, the procedures `send`, `receive` and `probe` have to be defined for each of these channel types. Fortunately VHDL allows overloading of procedure names, so it is possible to make these definitions. The only differences between the definitions of the channels are the data types, the names of the channel types and the default values of the data fields in the channels. So it is very easy to copy the definitions of one channel to make a new channel type. It is not necessary to redefine the type `handshake_phase`. All these definitions are conveniently collected in a VHDL package. This package can then be referenced wherever needed. An example of such a package with only one channel type can be seen in appendix A.1. The procedures `initialize_in` and `initialize_out` are used to initialize the input and output ends of a channel. If a sender or receiver does not initialize a channel, no communications can take place on that channel.

A simple example of a subcircuit is the FIFO stage `fp_latch` shown in figure 8.14. Notice that the channels in the entity have the mode `inout`, and

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.abstract_channels.all;

entity fp_latch is
  generic(delay : time);
  port ( d      : inout channel_fp;   -- input data channel
  port ( q      : inout channel_fp;   -- output data channel
         resetn : in std_logic      );
end fp_latch;

architecture behav of fp_latch is
begin

  process
    variable data : fp;
  begin
    initialize_in(d);
    initialize_out(q);
    wait until resetn = '1';
    loop
      receive(d, data);
      wait for delay;
      send(q, data);
    end loop;
  end process;

end behav;
```

Figure 8.14. Description of a FIFO stage.

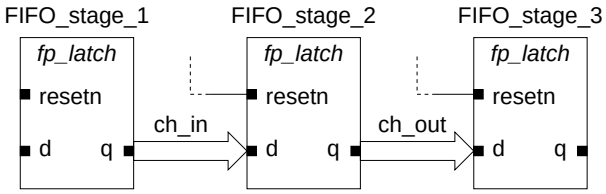


Figure 8.15. A FIFO built using the latch defined in figure 8.14.

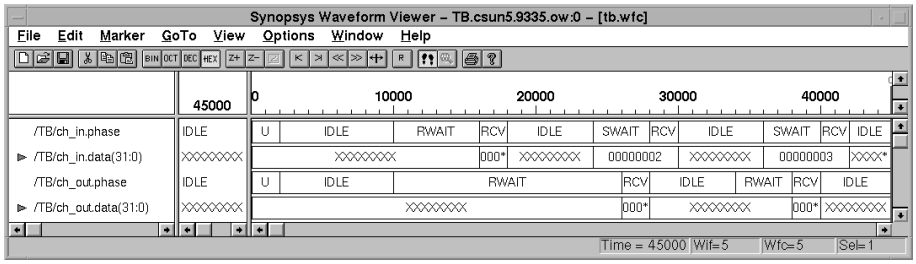


Figure 8.16. Simulation of the FIFO using the abstract channel package.

the FIFO stage waits for the reset signal `resetn` after the initialization. In that way it waits for other subcircuits which may actually use this reset signal for initialization.

The FIFO stage uses a generic parameter `delay`. This delay is inserted for experimental reasons in order to show the different phases of the channels. Three FIFO stages are connected in a pipeline (figure 8.15) and fed with data values. The middle section has a delay that is twice as long as the other two stages. This will result in a blocked channel just before the slow FIFO stage and a starved channel just after the slow FIFO stage.

The result of this experiment can be seen in figure 8.16. The simulator used is the Synopsys VSS. It is seen that `ch_in` is predominantly in the `swait` phase, which characterizes a blocked channel, and `ch_out` is predominantly in the `rwait` phase, which characterizes a starved channel.

8.6.5 The real channel package

At some point in the design process it is time to separate communicating entities into control and data entities. This is supported by the real channel types, in which the request and acknowledge signals are separate `std_logic` signals – the type used by the target component models. The data type is the same as the abstract channel type, but the handshaking is modeled differently. A real channel type is defined in figure 8.17.

```
subtype fp is std_logic_vector(31 downto 0);

type uchannel_fp is
  record
    req  : std_logic;
    ack  : std_logic;
    data : fp;
  end record;

type uchannel_fp_vector is array(natural range <>) of
       uchannel_fp;

function resolved(s : uchannel_fp_vector) return uchannel_fp;

subtype channel_fp is resolved uchannel_fp;
```

Figure 8.17. Definition of a real channel.

All definitions relating to the real channels are collected in a package (similar to the abstract channel package) and use the same names for the channel types, procedures and functions. For this reason it is very simple to switch to simulating using real channels. All it takes is to change the name of the package in the `use` statements in the top level design entity. Alternatively, one can use the same name for both packages, in which case it is the last analyzed package that is used in simulations.

An example of a real channel package with only one channel type can be seen in appendix A.2. This package defines a 32-bit standard logic 4-phase bundled-data push channel. The constant `tpd` in this package is the delay from a transition on the request or acknowledge signal to the response to this transition. "Synopsys compiler directives" are inserted in several places in the package. This is because Synopsys needs to know the channel types and the resolution functions belonging to them when it generates an EDIF netlist to the floor planner, but not the procedures in the package.

Figure 8.18 shows the result of repeating the simulation experiment from the previous section, this time using the real channel package. Notice the sequence of four-phase handshakes.

Note that the data value on a channel is, at all times, whatever value the sender is driving onto the channel. An alternative would be to make the resolution function put out the default data value outside the data-validity period, but this may cause the setup and hold times of the latches to be violated. The procedure `send` provides a broad data-validity scheme, which means that it can communicate with receivers that require early, broad or late data-validity schemes on the channel. The procedure `receive` requires an early data-validity scheme, which means that it can communicate with senders that provide early or broad data-validity schemes.

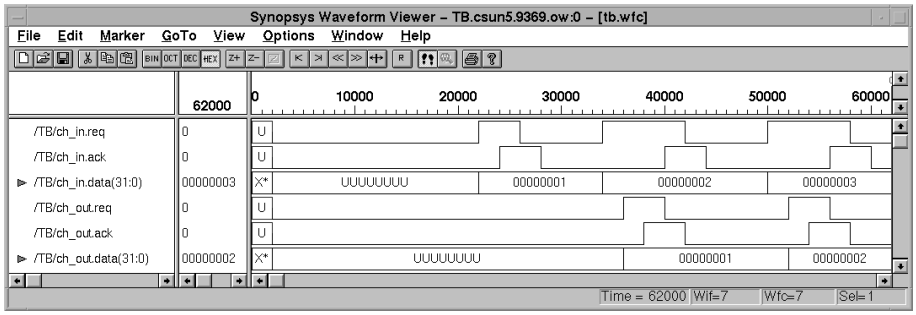


Figure 8.18. Simulation of the FIFO using the real channel package.

The resolution functions for the real channels (and the abstract channels) can detect protocol errors. Examples of errors are more than one sender or receiver on a channel, and using a `send` command or a `receive` command at the wrong end of a channel. In such cases the channel assumes the `X` value on the request or acknowledge signals.

8.6.6 Partitioning into control and data

This section describes how to separate an entity into control and data entities. This is possible when the real channel package is used but, as explained below, this partitioning has to follow certain guidelines.

To illustrate how the partitioning is carried out, the FIFO stage in figure 8.14 in the preceding section will be separated into a latch control circuit called `latch_ctrl` and a latch called `std_logic_latch`. The VHDL code is shown in figure 8.19, and figure 8.20 is a graphical illustration of the partitioning that includes the unresolved signals `ud` and `uq` as explained below.

In VHDL a driver that drives a compound resolved signal has to drive all fields in the signal. Therefore a control circuit cannot drive only the acknowledge field in a channel. To overcome this problem a signal of the corresponding unresolved channel type has to be declared inside the partitioned entity. This is the function of the signals `ud` and `uq` of type `uchannel_fp` in figure 8.17. The control circuit then drives only the acknowledge field in this signal; this is allowed since the signal is unresolved. The rest of the fields remain uninitialized. The unresolved signal then drives the channel; this is allowed since it drives all of the fields in the channel. The resolution function for the channel should ignore the uninitialized values that the channel is driven with. Components that use the `send` and `receive` procedures also drive those fields in the channel that they do not control with uninitialized values. For example, an output to a channel drives the acknowledge field in the channel with the `U` value. The fields

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.real_channels.all;

entity fp_latch is
  port ( d      : inout channel_fp;    -- input data channel
         q      : inout channel_fp;    -- output data channel
         resetn : in std_logic      );
end fp_latch;

architecture struct of fp_latch is

  component latch_ctrl
    port ( rin, aout, resetn : in  std_logic;
           ain, rout, lt : out std_logic      );
  end component;

  component std_logic_latch
    generic (width : positive);
    port ( lt : in  std_logic;
           d  : in  std_logic_vector(width-1 downto 0);
           q  : out std_logic_vector(width-1 downto 0)  );
  end component;

  signal lt : std_logic;
  signal ud, uq : uchannel_fp;

begin

  latch_ctrl1 : latch_ctrl
    port map (d.req,q.ack,resetn,ud.ack,uq.req,lt);
  std_logic_latch1 : std_logic_latch
    generic map (width => 32)
    port map (lt,d.data,uq.data);

  d <= connect(ud);
  q <= connect(uq);

end struct;
```

Figure 8.19. Separation of the FIFO stage into an ordinary data latch and a latch control circuit.

in a channel that are used as inputs are connected directly from the channel to the circuits that have to read those fields.

Notice in the description that the signals `ud` and `uq` do not drive `d` and `q` directly but through a function called `connect`. This function simply returns its parameter. It may seem unnecessary, but it has proved to be necessary when some of the subcircuits are described with a standard cell implementation. In a simulation a special "gate-level simulation engine" is used to simulate the standard cells [98]. During initialization it will set some of the signals to the value `X` instead of to the value `U` as it should. It has not been possible to get

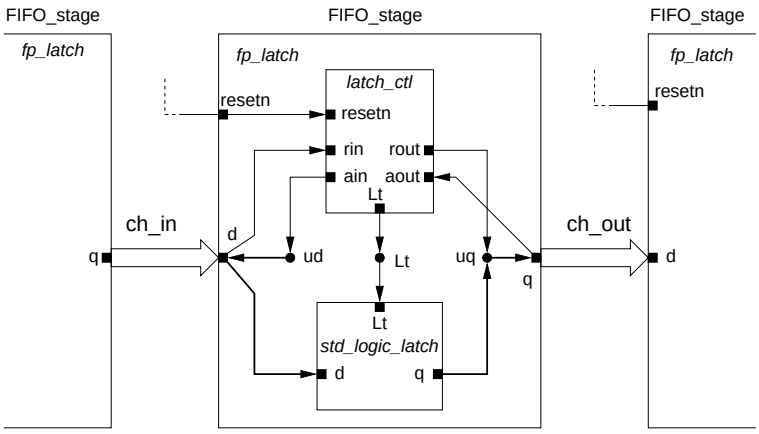


Figure 8.20. Separation of control and data.

the channel resolution function to ignore these `X` values, because the gate-level simulation engine sets some of the values in the channel. By introducing the `connect` function, which is a behavioural description, the normal simulator takes over and evaluates the channel by means of the corresponding resolution function. It should be emphasized that it is a bug in the gate-level simulation engine that necessitates the addition of the `connect` function.

8.7. Summary

This chapter addressed languages and CAD tools for high-level modeling and synthesis of asynchronous circuits. The text focused on a few representative and influential design methods that are based languages that are similar to CSP. The reason for preferring these languages are that they support channel based communication between processes (synchronous message passing) as well as concurrency at both process and statement level – two features that are important for modeling asynchronous circuits. The text also illustrated a synthesis method known as syntax directed translation. Subsequent chapters in this book will elaborate much more on these issues.

Finally the chapter illustrated how channel based communication can be implemented in VHDL, and we provided two packages containing all the necessary procedures and functions including: `send`, `receive` and `probe`. These packages supports a *manual* top-down stepwise-refinement design flow where the same test bench can be used to simulate the design throughout the entire design process from high level specification to low level circuit implementation.

This chapter on languages and CAD-tools for asynchronous design concludes the tutorial on asynchronous circuit design and it it time to wrap up: Chapter 2

presented the fundamental concepts and theories, and provided pointers to the literature. Chapters 3 and 4 presented an RTL-like abstract view on asynchronous circuits (tokens flowing in static data-flow structures) that is very useful for understanding their operation and performance. This material is probably where this tutorial supplements the existing body of literature the most. Chapters 5 and 6 addressed the design of datapath operators and control circuits. Focus in chapter 6 was on speed-independent circuits, but this is not the only approach. In recent years there has also been great progress in synthesizing multiple-input-change fundamental-mode circuits. Chapter 7 discussed more advanced 4-phase bundled-data protocols and circuits. Finally chapter 8 addressed languages and tools for high-level modeling and synthesis of asynchronous circuits.

The tutorial deliberately made no attempts at covering of all corners of the field – the aim was to pave a road into "the world of asynchronous design". Now you are here at the end of the road; hopefully with enough background to carry on digging deeper into the literature, and equally importantly, with sufficient understanding of the characteristics of asynchronous circuits, that you can start designing your own circuits. And finally; asynchronous circuits do not represent an alternative to synchronous circuits. They have advantages in some areas and disadvantages in other areas and they should be seen as a supplement, and as such they add new dimensions to the solution space that the digital designer explores. Even today, many circuits can not be categorized as either synchronous or asynchronous, they contain elements of both.

The following chapters will introduce some recent industrial scale asynchronous chips. Additional designs are presented in [80].

Appendix: The VHDL channel packages

A.1. The abstract channel package

```
-- Abstract channel package: (4-phase bundled-data push channel, 32-bit data)

library IEEE;
use IEEE.std_logic_1164.all;

package abstract_channels is

  constant tpd : time := 2 ns;

-- Type definition for abstract handshake protocol

  type handshake_phase is
  (
    u,          -- uninitialized
    idle,       -- no communication
    swait,      -- sender waiting
    rwait,      -- receiver waiting
    rcv,        -- receiving data
    rec1,       -- recovery phase 1
    rec2,       -- recovery phase 2
    req,        -- request signal
    ack,        -- acknowledge signal
    error       -- protocol error
  );

-- Floating point channel definitions

  subtype fp is std_logic_vector(31 downto 0);

  type uchannel_fp is
    record
      phase : handshake_phase;
      data  : fp;
    end record;

  type uchannel_fp_vector is array(natural range <>) of
         uchannel_fp;

  function resolved(s : uchannel_fp_vector) return uchannel_fp;

  subtype channel_fp is resolved uchannel_fp;

  procedure initialize_in(signal ch : out channel_fp);

  procedure initialize_out(signal ch : out channel_fp);

  procedure send(signal ch : inout channel_fp; d : in fp);

  procedure receive(signal ch : inout channel_fp; d : out fp);

  function probe(signal ch : in channel_fp) return boolean;

end abstract_channels;
```

```
package body abstract_channels is

-- Resolution table for abstract handshake protocol

  type table_type is array(handshake_phase, handshake_phase) of
         handshake_phase;

  constant resolution_table : table_type := (
-------------------------------------------------------------------------
-- 2. parameter:                                                |       |
-- u     idle  swait rwait rcv   rec1  rec2  req   ack   error     |1. par:|
-------------------------------------------------------------------------
  (u,    u,    u,    u,    u,    u,    u,    u,    u,    u    ), --| u     |
  (u,    idle, swait,rwait,rcv,  rec1, rec2, swait,rec2, error), --| idle  |
  (u,    swait,error,rcv,  error,error,rec1, error,rec1, error), --| swait |
  (u,    rwait,rcv,  error,error,error,error,rcv,  error,error), --| rwait |
  (u,    rcv,  error,error,error,error,error,error,error,error), --| rcv   |
  (u,    rec1, error,error,error,error,error,error,error,error), --| rec1  |
  (u,    rec2, rec1, error,error,error,error,rec1, error,error), --| rec2  |
  (u,    error,error,error,error,error,error,error,error,error), --| req   |
  (u,    error,error,error,error,error,error,error,error,error), --| ack   |
  (u,    error,error,error,error,error,error,error,error,error));--| error |

-- Fp channel

  constant default_data_fp : fp := "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";

  function resolved(s : uchannel_fp_vector) return uchannel_fp is
    variable result : uchannel_fp := (idle, default_data_fp);
  begin
    for i in s'range loop
      result.phase := resolution_table(result.phase, s(i).phase);
      if (s(i).phase = req) or (s(i).phase = swait) or
         (s(i).phase = rcv) then
           result.data := s(i).data;
      end if;
    end loop;
    if not((result.phase = swait) or (result.phase = rcv)) then
      result.data := default_data_fp;
    end if;
    return result;
  end resolved;

  procedure initialize_in(signal ch : out channel_fp) is
  begin
    ch.phase <= idle after tpd;
  end initialize_in;

  procedure initialize_out(signal ch : out channel_fp) is
  begin
    ch.phase <= idle after tpd;
  end initialize_out;

  procedure send(signal ch : inout channel_fp; d : in fp) is
  begin
    if not((ch.phase = idle) or (ch.phase = rwait)) then
      wait until (ch.phase = idle) or (ch.phase = rwait);
```

```
    end if;
    ch <= (req, d);
    wait until ch.phase = rec1;
    ch.phase <= idle;
  end send;

  procedure receive(signal ch : inout channel_fp; d : out fp) is
  begin
    if not((ch.phase = idle) or (ch.phase = swait)) then
      wait until (ch.phase = idle) or (ch.phase = swait);
    end if;
    ch.phase <= rwait;
    wait until ch.phase = rcv;
    wait for tpd;
    d := ch.data;
    ch.phase <= ack;
    wait until ch.phase = rec2;
    ch.phase <= idle;
  end receive;

  function probe(signal ch : in channel_fp) return boolean is
  begin
    return (ch.phase = swait);
  end probe;

end abstract_channels;
```

A.2. The real channel package

```
-- Low-level channel package (4-phase bundled-data push channel, 32-bit data)

library IEEE;
use IEEE.std_logic_1164.all;

package real_channels is

  -- synopsys synthesis_off
  constant tpd : time := 2 ns;
  -- synopsys synthesis_on

-- Floating point channel definitions

  subtype fp is std_logic_vector(31 downto 0);

  type uchannel_fp is
    record
      req  : std_logic;
      ack  : std_logic;
      data : fp;
    end record;

  type uchannel_fp_vector is array(natural range <>) of
         uchannel_fp;

  function resolved(s : uchannel_fp_vector) return uchannel_fp;
```

```
  subtype channel_fp is resolved uchannel_fp;

  -- synopsys synthesis_off
  procedure initialize_in(signal ch : out channel_fp);

  procedure initialize_out(signal ch : out channel_fp);

  procedure send(signal ch : inout channel_fp; d : in fp);

  procedure receive(signal ch : inout channel_fp; d : out fp);

  function probe(signal ch : in uchannel_fp) return boolean;
  -- synopsys synthesis_on

  function connect(signal ch : in uchannel_fp) return channel_fp;

end real_channels;

package body real_channels is

-- Resolution table for 4-phase handshake protocol

  -- synopsys synthesis_off
  type stdlogic_table is array(std_logic, std_logic) of std_logic;

  constant resolution_table : stdlogic_table := (
  --  ---------------------------------------------------------
  --  | 2. parameter:                                       |       |
  --  |  U    X    0    1    Z    W    L    H    -         |1. par:|
  --  ---------------------------------------------------------
      ( 'U', 'X', '0', '1', 'X', 'X', 'X', 'X', 'X' ),  -- |   U   |
      ( 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ),  -- |   X   |
      ( '0', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ),  -- |   0   |
      ( '1', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ),  -- |   1   |
      ( 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ),  -- |   Z   |
      ( 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ),  -- |   W   |
      ( 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ),  -- |   L   |
      ( 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ),  -- |   H   |
      ( 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ));  -- |   -   |
  -- synopsys synthesis_on

-- Fp channel

  -- synopsys synthesis_off
  constant default_data_fp : fp := "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
  -- synopsys synthesis_on

  function resolved(s : uchannel_fp_vector) return uchannel_fp is
  -- pragma resolution_method three_state
  -- synopsys synthesis_off
    variable result : uchannel_fp := ('U','U',default_data_fp);
  -- synopsys synthesis_on
  begin
  -- synopsys synthesis_off
    for i in s'range loop
      result.req := resolution_table(result.req,s(i).req);
      result.ack := resolution_table(result.ack,s(i).ack);
```

```
      if (s(i).req = '1') or (s(i).req = '0') then
        result.data := s(i).data;
      end if;
    end loop;
    if not((result.req = '1') or (result.req = '0')) then
      result.data := default_data_fp;
    end if;
    return result;
  -- synopsys synthesis_on
  end resolved;

  -- synopsys synthesis_off
  procedure initialize_in(signal ch : out channel_fp) is
  begin
    ch.ack <= '0' after tpd;
  end initialize_in;

  procedure initialize_out(signal ch : out channel_fp) is
  begin
    ch.req <= '0' after tpd;
  end initialize_out;

  procedure send(signal ch : inout channel_fp; d : in fp) is
  begin
    if ch.ack /= '0' then
      wait until ch.ack = '0';
    end if;
    ch.req <= '1' after tpd;
    ch.data <= d after tpd;
    wait until ch.ack = '1';
    ch.req <= '0' after tpd;
  end send;

  procedure receive(signal ch : inout channel_fp; d : out fp) is
  begin
    if ch.req /= '1' then
      wait until ch.req = '1';
    end if;
    wait for tpd;
    d := ch.data;
    ch.ack <= '1';
    wait until ch.req = '0';
    ch.ack <= '0' after tpd;
  end receive;

  function probe(signal ch : in uchannel_fp) return boolean is
  begin
    return (ch.req = '1');
  end probe;
  -- synopsys synthesis_on

  function connect(signal ch : in uchannel_fp) return channel_fp is
  begin
    return ch;
  end connect;

end real_channels;
```

Epilogue

Asynchronous technology has existed since the first days of digital electronics – many of the earliest computers did not employ a central clock signal. However, with the development of integrated circuits the need for a straightforward design discipline that could scale up rapidly with the available transistor resource was pressing, and clocked design became the dominant approach. Today, most practising digital designers know very little about asynchronous techniques, and what they do know tends to discourage them from venturing into the territory. But clocked design is beginning to show signs of stress – its ability to scale is waning, and it brings with it growing problems of excessive power dissipation and electromagnetic interference.

During the reign of the clock, a few designers have remained convinced that asynchronous techniques have merit, and new techniques have been developed that are far better suited to the VLSI era than were the approaches employed on early machines. In this book we have tried to illuminate these new techniques in a way that is accessible to any practising digital circuit designer, whether or not they have had prior exposure to asynchronous circuits.

In this account of asynchronous design techniques we have had to be selective in order not to obscure the principal goal with arcane detail. Much work of considerable quality and merit has been omitted, and the reader whose interest has been ignited by this book will find that there is a great deal of published material available that exposes aspects of asynchronous design that have not been touched upon here.

Although there are commercial examples of VLSI devices based on asynchronous techniques (a couple of which have been described in this book), these are exceptions – most asynchronous development is still taking place in research laboratories. If this is to change in the future, where will this change first manifest itself?

The impending demise of clocked design has been forecast for many years and still has not happened. If it does happen, it will be for some compelling reason, since designers will not lightly cast aside their years of experience in one design style in favour of another style that is less proven and less well supported by automated tools.

There are many possible reasons for considering asynchronous design, but no single 'killer application' that makes its use obligatory. Several of the arguments for adopting asynchronous techniques mentioned at the start of this book – low power, low electromagnetic interference, modularity, etc. – are applicable in their own niches, but only the modularity argument has the potential to gain universal adoption. Here a promising approach that will support heterogeneous timing environments is GALS (Globally Asynhronous Locally Synchronous) system design. An asynchronous on-chip interconnect – a 'chip area network' such as Chain (described on page 312) – is used to connect clocked modules. The modules themselves can be kept small enough for clock skew to be well-contained so that straightforward synchronous design techniques work well, and different modules can employ different clocks or the same clock with different phases.

Once this framework is in place, it is then clearly straightforward to make individual modules asynchronous on a case-by-case basis.

Here, perhaps unsurprisingly, we see the need to merge asynchronous technology with established synchronous design techniques, so most of the functional design can be performed using well-understood tools and approaches. This evolutionary approach contrasts with the revolutionary attacks described in Part III of this book, and represents the most likely scenario for the widespread adoption of the techniques described in this book in the medium-term future.

In the shorter term, however, the application niches that can benefit from asynchronous technology are important and viable. It is our hope in writing this book that more designers will come to understand the principles of asynchronous design and its potential to offer new solutions to old and new problems. Clocks are useful but they can become straitjackets. Don't be afraid to think outside the box!

For further information on asynchronous design see the bibliography at the end of this book, the *Asynchronous Bibliography* on the Internet [85], and the general information on asynchronous design available at the *Asynchronous Logic Homepage*, also on the Internet [36].

References

[1] T. Agerwala. Putting Petri nets to work. *IEEE Computer*, 12(12):85–94, December 1979.

[2] T.S. Balraj and M.J. Foster. Miss Manners: A specialized silicon compiler for synchronizers. In Charles E. Leierson, editor, *Advanced Research in VLSI*, pages 3–20. MIT Press, April 1986.

[3] A. Bardsley and D.A. Edwards. Compiling the language Balsa to delay-insensitive hardware. In C. D. Kloos and E. Cerny, editors, *Hardware Description Languages and their Applications (CHDL)*, pages 89–91, April 1997.

[4] A. Bardsley and D.A. Edwards. The Balsa asynchronous circuit synthesis system. In *Forum on Design Languages*, September 2000.

[5] P.A. Beerel, C.J. Myers, and T.H.-Y. Meng. Automatic synthesis of gate-level speed-independent circuits. Technical Report CSL-TR-94-648, Stanford University, November 1994.

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

[7] G. Birtwistle and A. Davis, editors. *Proceedings of the Banff VIII Workshop: Asynchronous Digital Circuit Design, Banff, Alberta, Canada, August 28–September 3, 1993*. Springer Verlag, Workshops in Computing Science, 1995. Contributions from: S.B. Furber, "Computing without Clocks: Micropipelining the ARM Processor," A. Davis, "Practical Asynchronous Circuit Design: Methods and Tools," C.H. van Berkel, "VLSI Programming of Asynchronous Circuits for Low Power," J. Ebergen, "Parallel Program and Asynchronous Circuit Design," A. Davis and S. Nowick, "Introductory Survey".

[8] E. Brinksma and T. Bolognesi. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1), 1987.

[9] E. Brunvand and R.F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *Proc. International Conf. Computer-Aided De-*

sign (ICCAD), pages 262–265. IEEE Computer Society Press, November 1989.

[10] J.A. Brzozowsky and C.-J.H. Seager. *Asynchronous Circuits*. Springer Verlag, Monographs in Computer Science, 1994. ISBN: 0-387-94420-6.

[11] S.M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, Computer Science Department, California Institute of Technology, 1991. Caltech-CS-TR-91-01.

[12] S.M. Burns. General condition for the decomposition of state holding elements. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

[13] S.M. Burns and A.J. Martin. Syntax-directed translation of concurrent programs into self-timed circuits. In J. Allen and F. Leighton, editors, *Advanced Research in VLSI*, pages 35–50. MIT Press, 1988.

[14] D.M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, October 1984.

[15] K.T. Christensen, P. Jensen, P. Korger, and J. Sparsø. The design of an asynchronous TinyRISC TR4101 microprocessor core. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 108–119. IEEE Computer Society Press, 1998.

[16] T.-A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, MIT Laboratory for Computer Science, June 1987.

[17] T.-A. Chu and R.K. Roy (editors). Special issue on asynchronous circuits and systems. *IEEE Design & Test*, 11(2), 1994.

[18] T.-A. Chu and L.A. Glasser. Synthesis of self-timed control circuits from graphs: An example. In *Proc. International Conf. Computer Design (ICCD)*, pages 565–571. IEEE Computer Society Press, 1986.

[19] B. Coates, A. Davis, and K. Stevens. The Post Office experience: Designing a large asynchronous chip. *Integration, the VLSI journal*, 15(3):341–366, October 1993.

[20] F. Commoner, A.W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *J. Comput. System Sci.*, 5(1):511–523, October 1971.

[21] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. In *XI Conference on Design of Integrated Circuits and Systems*, Barcelona, November 1996.

[22] U. Cummings, A. Lines, and A. Martin. An asynchronous pipelined lattice structure filter. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 126–133, November 1994.

[23] A. Davis. A data-driven machine architecture suitable for VLSI implementation. In *Proceedings of the First Caltech Conference on VLSI*, pages 479–494, Pasadena, CA, January 1979.

[24] A. Davis and S.M. Nowick. Asynchronous circuit design: Motivation, background, and methods. In G. Birtwistle and A. Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 1–49. Springer-Verlag, 1995.

[25] A. Davis and S.M. Nowick. An introduction to asynchronous circuit design. Technical Report UUCS-97-013, Department of Computer Science, University of Utah, September 1997.

[26] A. Davis and S.M. Nowick. An introduction to asynchronous circuit design. In A. Kent and J. G. Williams, editors, *The Encyclopedia of Computer Science and Technology*, volume 38. Marcel Dekker, New York, February 1998.

[27] J.B. Dennis. Data Flow Computation. In *Control Flow and Data Flow — Concepts of Distributed Programming, International Summer School*, pages 343–398, Marktoberdorf, West Germany, July 31 – August 12, 1984. Springer, Berlin.

[28] J.C. Ebergen and R. Berks. Response time properties of linear asynchronous pipelines. *Proceedings of the IEEE*, 87(2):308–318, February 1999.

[29] K.M. Fant and S.A. Brandt. Null Conventional Logic: A complete and consistent logic for asynchronous digital circuit synthesis. In *International Conference on Application-specific Systems, Architectures, and Processors*, pages 261–273, 1996.

[30] 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 TR CUCS-020-99, Columbia University, NY, July 1999. http://www.cs.columbia.edu/~nowick/minimalist.pdf.

[31] S.B. Furber and P. Day. Four-phase micropipeline latch control circuits. *IEEE Transactions on VLSI Systems*, 4(2):247–253, June 1996.

[32] S.B. Furber, P. Day, J.D. Garside, N.C. Paver, S. Temple, and J.V. Woods. The design and evaluation of an asynchronous microprocessor. In *Proc. Int'l. Conf. Computer Design*, pages 217–220, October 1994.

[33] S.B. Furber, D.A. Edwards, and J.D. Garside. AMULET3: a 100 MIPS asynchronous embedded processor. In *Proc. International Conf. Computer Design (ICCD)*, September 2000.

[34] S.B. Furber, J.D. Garside, P. Riocreux, S. Temple, P. Day, J. Liu, and N.C. Paver. AMULET2e: An asynchronous embedded controller. *Proceedings of the IEEE*, 87(2):243–256, February 1999.

[35] S.B. Furber, J.D. Garside, S. Temple, J. Liu, P. Day, and N.C. Paver. AMULET2e: An asynchronous embedded controller. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 290–299. IEEE Computer Society Press, 1997.

[36] J.D. Garside. The Asynchronous Logic Homepages. `http://www.cs.man.ac.uk/async/`.

[37] J.D. Garside, W.J. Bainbridge, A. Bardsley, D.A. Edwards, S.B. Furber, J. Liu, D.W. Lloyd, S. Mohammadi, J.S. Pepper, O. Petlin, S. Temple, and J.V. Woods. AMULET3i – an asynchronous system-on-chip. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 162–175. IEEE Computer Society Press, April 2000.

[38] B. Gilchrist, J.H. Pomerene, and S.Y. Wong. Fast carry logic for digital computers. *IRE Transactions on Electronic Computers*, EC-4(4):133–136, December 1955.

[39] L.A. Glasser and D.W. Dobberpuhl. *The Design and Analysis of VLSI Circuits*. Addison-Wesley, 1985.

[40] S. Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83(1):69–93, January 1995.

[41] L.G. Heller, W.R. Griffin, J.W. Davis, and N.G. Thoma. Cascode voltage switch logic: A differential CMOS logic family. *Proc. International Solid State Circuits Conference*, pages 16–17, February 1984.

[42] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

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

[44] D.A. Huffman. The synthesis of sequential switching circuits. *J. Franklin Inst.*, pages 161–190, 275–303, March/April 1954.

[45] D.A. Huffman. The synthesis of sequential switching circuits. In E. F. Moore, editor, *Sequential Machines: Selected Papers*. Addison-Wesley, 1964.

[46] K. Hwang. *Computer Arithmetic: Principles, Architecture, and Design*. John Wiley & Sons, 1979.

[47] S.C. Johnson and S. Mazor. Silicon compiler lets system makers design their own VLSI chips. *Electronic Design*, 32(20):167–181, 1984.

[48] G. Jones. *Programming in OCCAM*. Prentice-Hall international, 87.

[49] M.B. Josephs, S.M. Nowick, and C.H. van Berkel. Modeling and design of asynchronous circuits. *Proceedings of the IEEE*, 87(2):234–242, February 1999.

[50] J. Kessels, T. Kramer, G. den Besten, A. Peeters, and V. Timm. Applying asynchronous circuits in contactless smart cards. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 36–44. IEEE Computer Society Press, April 2000.

[51] J. Kessels, T. Kramer, A. Peeters, and V. Timm. DESCALE: a design experiment for a smart card application consuming low energy. In R. van Leuken, R. Nouta, and A. de Graaf, editors, *European Low Power Initiative for Electronic System Design*, pages 247–262. Delft Institute of Microelectronics and Submicron Technology, July 2000.

[52] Z. Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, 1978.

[53] A. Kondratyev, J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Logic decomposition of speed-independent circuits. *Proceedings of the IEEE*, 87(2):347–362, February 1999.

[54] J. Liu. *Arithmetic and control components for an asynchronous microprocessor*. PhD thesis, Department of Computer Science, University of Manchester, 1997.

[55] D.W. Lloyd. VHDL models of asychronous handshaking. (Personal communication, August 1998).

[56] A.J. Martin. The probe: An addition to communication primitives. *Information Processing Letters*, 20(3):125–130, 1985. Erratum: IPL 21(2):107, 1985.

[57] A.J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.

[58] A.J. Martin. Formal program transformations for VLSI circuit synthesis. In E.W. Dijkstra, editor, *Formal Development of Programs and Proofs*, UT Year of Programming Series, pages 59–80. Addison-Wesley, 1989.

[59] A.J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In W.J. Dally, editor, *Advanced Research in VLSI: Proceedings of the Sixth MIT Conference*, pages 263–278. MIT Press, 1990.

[60] A.J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C.A.R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64. Addison-Wesley, 1990.

[61] A.J. Martin. Asynchronous datapaths and the design of an asynchronous adder. *Formal Methods in System Design*, 1(1):119–137, July 1992.

[62] A.J. Martin, S.M. Burns, T.K. Lee, D. Borkovic, and P.J. Hazewindus. The first asynchronous microprocessor: The test results. *Computer Architecture News*, 17(4):95–98, 1989.

[63] A.J. Martin, A. Lines, R. Manohar, M. Nyström, P. Penzes, R. Southworth, U.V. Cummings, and T.-K. Lee. The design of an asynchronous MIPS R3000. In *Proceedings of the 17th Conference on Advanced Research in VLSI*, pages 164–181. MIT Press, September 1997.

[64] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[65] C.E. Molnar, I.W. Jones, W.S. Coates, and J.K. Lexau. A FIFO ring oscillator performance experiment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 279–289. IEEE Computer Society Press, April 1997.

[66] C.E. Molnar, I.W. Jones, W.S. Coates, J.K. Lexau, S.M. Fairbanks, and I.E. Sutherland. Two FIFO ring performance experiments. *Proceedings of the IEEE*, 87(2):297–307, February 1999.

[67] D.E. Muller. Asynchronous logics and application to information processing. In H. Aiken and W. F. Main, editors, *Proc. Symp. on Application of Switching Theory in Space Technology*, pages 289–297. Stanford University Press, 1963.

[68] D.E. Muller and W.S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching, Cambridge, April 1957, Part I*, pages 204–243. Harvard University Press, 1959. The annals of the computation laboratory of Harvard University, Volume XXIX.

[69] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[70] C.J. Myers. *Asynchronous Circuit Design*. John Wiley & Sons, July 2001. ISBN: 0-471-41543-X.

[71] C.D. Nielsen. Evaluation of function blocks for asynchronous design. In *Proc. European Design Automation Conference (EURO-DAC)*, pages 454–459. IEEE Computer Society Press, September 1994.

[72] C.D. Nielsen, J. Staunstrup, and S.R. Jones. Potential performance advantages of delay-insensitivity. In M. Sami and J. Calzadilla-Daguerre, editors, *Proceedings of IFIP workshop on Silicon Architectures for Neural Nets, StPaul-de-Vence, France, November 1990*. North-Holland, Amsterdam, 1991.

[73] L.S. Nielsen. *Low-power Asynchronous VLSI Design*. PhD thesis, Department of Information Technology, Technical University of Denmark, 1997. IT-TR:1997-12.

[74] L.S. Nielsen, C. Niessen, J. Sparsø, and C.H. van Berkel. Low-power operation using self-timed circuits and adaptive scaling of the supply voltage. *IEEE Transactions on VLSI Systems*, 2(4):391–397, 1994.

[75] L.S. Nielsen and J. Sparsø. A low-power asynchronous data-path for a FIR filter bank. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 197–207. IEEE Computer Society Press, 1996.

[76] L.S. Nielsen and J. Sparsø. An 85 μW asynchronous filter-bank for a digital hearing aid. In *Proc. IEEE International Solid State circuits Conference*, pages 108–109, 1998.

[77] L.S. Nielsen and J. Sparsø. Designing asynchronous circuits for low power: An IFIR filter bank for a digital hearing aid. *Proceedings of the IEEE*, 87(2):268–281, February 1999. Special issue on "Asynchronous Circuits and Systems" (Invited Paper).

[78] D.C. Noice. *A Two-Phase Clocking Dicipline for Digital Integrated Circuits*. PhD thesis, Department of Electrical Engineering, Stanford University, February 1983.

[79] S.M. Nowick. Design of a low-latency asynchronous adder using speculative completion. *IEE Proceedings, Computers and Digital Techniques*, 143(5):301–307, September 1996.

[80] S.M. Nowick, M.B. Josephs, and C.H. van Berkel (editors). Special issue on asynchronous circuits and systems. *Proceedings of the IEEE*, 87(2), February 1999.

[81] S.M. Nowick, K.Y. Yun, and P.A. Beerel. Speculative completion for the design of high-performance asynchronous dynamic adders. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 210–223. IEEE Computer Society Press, April 1997.

[82] International Standards Organization. LOTOS — a formal description technique based on the temporal ordering of observational behaviour. ISO IS 8807, 1989.

[83] N.C. Paver, P. Day, C. Farnsworth, D.L. Jackson, W.A. Lien, and J. Liu. A low-power, low-noise configurable self-timed DSP. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 32–42, 1998.

[84] M. Pedersen. Design of asynchronous circuits using standard CAD tools. Technical Report IT-E 774, Technical University of Denmark, Dept. of Information Technology, 1998. (In Danish).

[85] A.M.G. Peeters. The 'Asynchronous' Bibliography. `http://www.win.tue.nl/~wsinap/async.html`. Corresponding e-mail address: `async-bib@win.tue.nl`.

[86] A.M.G. Peeters. *Single-Rail Handshake Circuits*. PhD thesis, Eindhoven University of Technology, June 1996. `http://www.win.tue.nl/~wsinap/pdf/Peeters96.pdf`.

[87] J.L. Peterson. Petri nets. *Computing Surveys*, 9(3):223–252, September 1977.

[88] J. Rabaey. *Digital Integrated Circuits: A Design Perspective*. Prentice-Hall, 1996.

[89] M. Renaudin, P. Vivet, and F. Robin. A design framework for asynchronous/synchronous circuits based on CHP to HDL translation. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 135–144, April 1999.

[90] M. Roncken. Defect-oriented testability for asynchronous ICs. *Proceedings of the IEEE*, 87(2):363–375, February 1999.

[91] C.L. Seitz. System timing. In C.A. Mead and L.A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[92] N.P. Singh. A design methodology for self-timed systems. Master's thesis, Laboratory for Computer Science, MIT, 1981. MIT/LCS/TR-258.

[93] J. Sparsø, C.D. Nielsen, L.S. Nielsen, and J. Staunstrup. Design of self-timed multipliers: A comparison. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 165–180. Elsevier Science Publishers, 1993.

[94] J. Sparsø and J. Staunstrup. Delay-insensitive multi-ring structures. *INTEGRATION, the VLSI Journal*, 15(3):313–340, October 1993.

[95] J. Sparsø, J. Staunstrup, and M. Dantzer-Sørensen. Design of delay insensitive circuits using multi-ring structures. In G. Musgrave, editor, *Proc. of EURO-DAC '92, European Design Automation Conference, Hamburg, Germany, September 7-10, 1992*, pages 15–20. IEEE Computer Society Press, 1992.

[96] L. Stok. *Architectural Synthesis and Optimization of Digital Systems*. PhD thesis, Eindhoven University of Technology, 1991.

[97] I.E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[98] Synopsys, Inc. *Synopsys VSS Family Core Programs Manual*, 1997.

[99] S.H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley-Interscience, John Wiley & Sons, Inc., New York, 1969.

[100] C.H. van Berkel. Beware the isochronic fork. *INTEGRATION, the VLSI journal*, 13(3):103–128, 1992.

[101] C.H. van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.

[102] C.H. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, and F. Schalij. Asynchronous circuits for low power: a DCC error corrector. *IEEE Design & Test*, 11(2):22–32, 1994.

[103] C.H. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, and F. Schalij. A fully asynchronous low-power error corrector for the DCC player. In *ISSCC 1994 Digest of Technical Papers*, volume 37, pages 88–89. IEEE, 1994. ISSN 0193-6530.

[104] C.H. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, F. Schalij, and R. van de Viel. A single-rail re-implementation of a DCC error detector using a generic standard-cell library. In *2nd Working Conference on Asynchronous Design Methodologies, London, May 30-31, 1995*, pages 72–79, 1995.

[105] C.H. van Berkel, F. Huberts, and A. Peeters. Stretching quasi delay insensitivity by means of extended isochronic forks. In *Asynchronous Design Methodologies*, pages 99–106. IEEE Computer Society Press, May 1995.

[106] C.H. van Berkel, M.B. Josephs, and S.M. Nowick. Scanning the technology: Applications of asynchronous circuits. *Proceedings of the IEEE*, 87(2):223–233, February 1999.

[107] C.H. 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 Conference on Design Automation (EDAC)*, pages 384–389, 1991.

[108] C.H. van Berkel, C. Niessen, M. Rem, and R. Saeijs. VLSI programming and silicon compilation. In *Proc. International Conf. Computer Design (ICCD)*, pages 150–166, Rye Brook, New York, 1988. IEEE Computer Society Press.

[109] H. van Gageldonk, D. Baumann, C.H. van Berkel, D. Gloor, A. Peeters, and G. Stegmann. An asynchronous low-power 80c51 microcontroller. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 96–107. IEEE Computer Society Press, April 1998.

[110] P. Vanbekbergen. *Synthesis of Asynchronous Control Circuits from Graph-Theoretic Specifications*. PhD thesis, Catholic University of Leuven, September 1993.

[111] V.I. Varshavsky, M.A. Kishinevsky, V.B. Marakhovsky, V.A. Peschansky, L.Y. Rosenblum, A.R. Taubin, and B.S. Tzirlin. *Self-timed Control of Con-*

current Processes. Kluwer Academic Publisher, 1990. V.I.Varshavsky Ed., (Russian edition: 1986).

[112] T. Verhoeff. Delay-insensitive codes - an overview. *Distributed Computing*, 3(1):1–8, 1988.

[113] P. Viviet and M. Renaudin. CHP2VHDL, a CHP to VHDL translator - towards asynchronous-design simulation. In L. Lavagno and M.B. Josephs, editors, *Handouts from the ACiD-WG Workshop on Specification models and languages and technology effects of asynchronous design*. Dipartemento di Elettronica, Polytecnico de Torino, Italy, January 1998.

[114] J.F. Wakerly. *Digital Design: Principles and Practices, 3/e*. Prentice-Hall, 2001.

[115] N. Weste and K. Esraghian. *Principles of CMOS VLSI Design – A systems Perspective, 2nd edition*. Addison-Wesley, 1993.

[116] T.E. Williams. *Self-Timed Rings and their Application to Division*. PhD thesis, Department of Electrical Engineering and Computer Science, Stanford University, 1991. CSL-TR-91-482.

[117] T.E. Williams. Analyzing and improving latency and throughput in self-timed rings and pipelines. In *Tau-92: 1992 Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*. ACM/SIGDA, March 1992.

[118] T.E. Williams. Performance of iterative computation in self-timed rings. *Journal of VLSI Signal Processing*, 6(3), October 1993.

[119] T.E. Williams and M.A. Horowitz. A zero-overhead self-timed 160 ns. 54 bit CMOS divider. *IEEE Journal of Solid State Circuits*, 26(11):1651–1661, 1991.

[120] T.E. Williams, N. Patkar, and G. Shen. SPARC64: A 64-b 64-active-instruction out-of-order-execution MCM processor. *IEEE Journal of Solid-State Circuits*, 30(11):1215–1226, November 1995.

[121] C. Ykman-Couvreur, B. Lin, and H. de Man. Assassin: A synthesis system for asynchronous control circuits. Technical report, IMEC, September 1994. User and Tutorial manual.

[122] K.Y. Yun and D.L. Dill. Automatic synthesis of extended burst-mode circuits: Part II (automatic synthesis). *IEEE Transactions on Computer-Aided Design*, 18(2):118–132, February 1999.

Index