



Introduction to Asynchronous Circuit Design.

Sparsø, Jens

Publication date:
2020

Document Version
Publisher's PDF, also known as Version of record

[Link back to DTU Orbit](#)

Citation (APA):
Sparsø, J. (2020). *Introduction to Asynchronous Circuit Design*. DTU Compute, Technical University of Denmark.

General rights

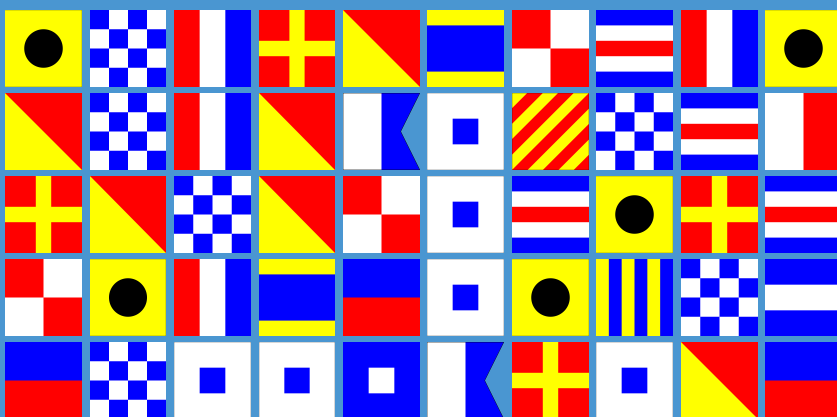
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Introduction to Asynchronous Circuit Design

Jens Sparsø



Introduction to Asynchronous Circuit Design

Jens Sparsø

Technical University of Denmark

Copyright © 2020 Jens Sparsø, Technical University of Denmark. Email: jspa@dtu.dk. Note: Original versions of eight chapters were previously published in 2001 and 2006.

Cataloging Data:

Sparsø, Jens
Introduction to Asynchronous Circuit Design

Pages 1-255
Includes bibliographical references and an index.

ISBN : 979-86-550-5385-4 (Paperback)
Publisher: Independently published (Kindle Direct Publishing)

ISBN : 978-87-643-2001-5 (PDF)
Publisher: DTU Compute, Technical University of Denmark.

The PDF-version of the book may be downloaded from the author's homepage <https://people.compute.dtu.dk/jspa> or from the research database of the Technical University of Denmark at <https://orbit.dtu.dk>

The electronic PDF-version is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
<https://creativecommons.org/licenses/by-nc-nd/4.0/>



Typeset by Jens Sparsø.

Contents

Preface	ix
Acknowledgments	xiii
1 Introduction	1
1.1 Why consider asynchronous circuits?	1
1.2 Aims and background	2
1.3 Clocking versus handshaking	3
1.4 Outline of the book	6
2 Fundamentals	9
2.1 Handshake protocols	9
2.1.1 Bundled-data protocols	9
2.1.2 The 4-phase dual-rail protocol	11
2.1.3 The 2-phase dual-rail protocol	13
2.1.4 Other protocols	13
2.2 Indication and the Muller C-element	14
2.3 The Muller pipeline	16
2.4 Circuit implementation styles	17
2.4.1 4-phase bundled-data	18
2.4.2 2-phase bundled data (Micropipelines)	19
2.4.3 4-phase dual-rail	20
2.5 Theory	23
2.5.1 The basics of speed-independence	23
2.5.2 Classification of asynchronous circuits	25
2.5.3 Isochronic forks	26
2.5.4 Relation to circuits	26
2.6 Test	27
2.7 Summary	28

3	Static data-flow structures	29
3.1	Introduction	29
3.2	Pipelines and rings	30
3.3	Building blocks	32
3.4	A simple example	35
3.5	Simple applications of rings	37
3.5.1	Sequential circuits	37
3.5.2	Iterative computations	37
3.5.3	Fibonacci sequence generator	38
3.6	When tokens spread	39
3.7	FOR, IF, and WHILE constructs	42
3.8	A more complex example: GCD	44
3.9	Pointers to additional examples	45
3.9.1	A low-power filter bank	45
3.9.2	An asynchronous microprocessor	46
3.9.3	A fine-grain pipelined vector multiplier	46
3.10	Summary	47
4	Performance	49
4.1	Introduction	49
4.2	A qualitative view of performance	50
4.2.1	Example 1: A FIFO used as a shift register	50
4.2.2	Example 2: A shift register with parallel load	52
4.3	Quantifying performance	55
4.3.1	Latency, throughput, and wavelength	55
4.3.2	Cycle time of a ring	57
4.3.3	Example 3: Performance of a 3-stage ring	59
4.3.4	Final remarks	60
4.4	Summary	60
5	Handshake circuit implementations (four-phase)	61
5.1	The latch, the sink, and the source	61
5.2	Fork, join and merge	63
5.3	MUX and DEMUX	65
5.4	Peephole optimizations	67
5.4.1	DEMUX with a sink on one output	67
5.4.2	A DEMUX with latches on both outputs	68
5.5	Memory cells	69
5.5.1	Introduction	69
5.5.2	A simple R-W data-flow memory cell	70
5.5.3	A R-W-RW data-flow memory cell	70
5.5.4	A R-W-WR data-flow memory cell	71
5.5.5	A more efficient R-W memory design	71
5.6	Function blocks – The basics	73

5.6.1	Introduction	73
5.6.2	Transparency to handshaking	74
5.6.3	Review of ripple-carry addition	77
5.7	Bundled-data function blocks	78
5.7.1	Using matched delays	78
5.7.2	Delay selection	79
5.8	Dual-rail function blocks	80
5.8.1	Delay insensitive minterm synthesis (DIMS)	80
5.8.2	Null Convention Logic	82
5.8.3	Transistor-level CMOS implementations	83
5.8.4	Martin's adder	84
5.9	Hybrid function blocks	86
5.10	Mutual exclusion and arbitration	88
5.10.1	Mutual exclusion	88
5.10.2	Arbitration	89
5.11	Summary	90
6	Speed-independent control circuits	91
6.1	Introduction	91
6.1.1	Asynchronous sequential circuits	92
6.1.2	Hazards	92
6.1.3	Delay models	93
6.1.4	Fundamental mode and input-output mode	94
6.1.5	Synthesis of fundamental mode circuits	94
6.2	Signal transition graphs	96
6.2.1	Petri nets and STGs	96
6.2.2	Some frequently used STG fragments	98
6.3	The basic synthesis procedure	101
6.3.1	Example 1: a C-element	102
6.3.2	Example 2: a circuit with choice	102
6.3.3	Example 2: Hazards in the simple gate implementation	104
6.4	Implementations using state-holding gates	106
6.4.1	Introduction	106
6.4.2	Excitation regions and quiescent regions	107
6.4.3	Example 2: Using state-holding elements	108
6.4.4	The monotonic cover constraint	108
6.4.5	Circuit topologies using state-holding elements	109
6.5	Initialization	111
6.6	Summary of the synthesis process	111
6.7	Petrify: A tool for synthesizing SI circuits from STGs	113
6.8	Design examples using Petrify	114
6.8.1	Example 2 revisited	114
6.8.2	A control circuit for a 4-phase bundled-data latch	117
6.8.3	A control circuit for a 4-phase bundled-data MUX	119

6.9	Summary	123
7	Performance analysis using timed Petri nets	125
7.1	Timed Petri nets	125
7.2	Sub-classes of Petri nets	127
7.3	Timing analysis of timed Petri nets	128
7.4	Example 3 revisited: Analysis using a TTPN	133
7.5	Example 3 revisited: Analysis using a simplified TPPN	135
7.6	Example 4: A four stage ring	136
7.7	Example 5: A pipeline with asymmetric delay elements	136
7.8	Worst-case timing analysis	139
8	Metastability, arbitration, and synchronization.	141
8.1	What is metastability?	141
8.2	Quantifying metastability	144
8.3	Dealing with metastability	150
8.3.1	Mutual exclusion and arbitration	150
8.3.2	Synchronization	151
8.3.3	Time-safe and value-safe systems	153
8.3.4	Additional comments and a word of warning	154
8.4	Synchronization in multi-clock systems	155
8.4.1	A simple handshake interface	155
8.4.2	Using a dual-ported memory	156
8.4.3	Using a dual-clock FIFO	157
8.4.4	Value-safe clocking with metastability	158
8.4.5	Value-safe clocking without metastability	159
8.5	A taxonomy of timing organizations	161
8.6	Examples of timing organizations	162
8.6.1	Plesiochronous bit-serial communication	162
8.6.2	Mesochronous communication links	164
8.6.3	Better than worst-case clocked circuits	167
8.7	Concluding remarks	169
9	Implementation of 2-phase bundled-data circuits	171
9.1	Templates for implementing 2-phase handshake latches	171
9.1.1	Recap of the Muller pipeline	172
9.1.2	Micropipelines	173
9.1.3	Mousetrap	174
9.1.4	Click elements	175
9.1.5	Loihi	176
9.2	2-phase static data-flow structures	177
9.2.1	A change of viewpoint	177
9.2.2	Phase-decoupled handshaking	179
9.2.3	Phase-decoupled handshake latches	181

9.3	Design examples: FIB and GCD	182
9.3.1	Fibonacci sequence generator (FIB)	182
9.3.2	Greatest common divisor (GCD)	183
9.4	Phase-decoupled click components	184
9.4.1	The handshake latch	184
9.4.2	Function blocks and delay elements	185
9.4.3	Join and Fork	185
9.4.4	Merge	186
9.4.5	MUX and DEMUX	187
9.4.6	Peephole optimizations	188
9.4.7	Mutual exclusion and arbitration	190
9.5	Prototyping using FPGAs	191
10	Advanced 4-phase bundled-data protocols and circuits	195
10.1	Channels and protocols	195
10.1.1	Channel types	195
10.1.2	Data-validity schemes	196
10.1.3	Discussion	196
10.2	Static type checking	198
10.3	More advanced latch control circuits	199
10.4	Summary	202
11	High-level languages and tools	203
11.1	Introduction	203
11.2	Concurrency and message passing in CSP	204
11.3	Tangram: program examples	206
11.3.1	A 2-place shift register	206
11.3.2	A 2-place (ripple) FIFO	207
11.3.3	GCD using while and if statements	207
11.3.4	GCD using guarded commands	207
11.4	Tangram: syntax-directed compilation	208
11.4.1	The 2-place shift register	209
11.4.2	The 2-place FIFO	210
11.4.3	GCD using guarded repetition	210
11.5	Martin's translation process	214
11.6	Using VHDL for asynchronous design	215
11.6.1	Introduction	215
11.6.2	VHDL versus CSP-type languages	215
11.6.3	Channel communication and design flow	217
11.6.4	The abstract channel package	219
11.6.5	The real channel package	223
11.6.6	Partitioning into control and data	224
11.7	Summary	227

11.8 The VHDL channel packages	227
11.8.1 The abstract channel package	227
11.8.2 The real channel package	230
References	233
Index	249

Preface

Background

A significant part of the material in this book originally appeared as:

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

The publisher held copyrights to these eight chapters for a limited time, and in 2006, I made the material available in the public domain (for non-commercial educational use). Apart from a few bug fixes, nothing was changed.

During the following years, both the field, as well as my knowledge, has evolved, and I have gradually collected more material. For quite some time, I have wanted to extend and update the text. The book you are holding is the result of an effort finally to do this.

Origin of the text

My interest in asynchronous circuit design was born when I read the seminal textbook “Introduction to VLSI Systems” by Carver Mead and Lynn Conway [97]. This book included an equally influential chapter 8, “System Timing,” authored by Chuck Seitz. Among other things, this chapter motivated and introduced so-called “self-timed circuits” – circuits that operate without a (global) clock or without clocks at all.

This material had a special appeal to me (and many others alike), and I wanted to design and implement such circuits. There were no textbooks back then, and the papers I read dealt with different and often narrow aspects of asynchronous design – I lacked the big picture and an engineering perspective, and this is what this book attempts to provide.

In such a context, the best way to learn is to do, and we set out to design a relatively small chip that was fabricated in 1991 in a 1.5 μm CMOS technology. In parallel, I started assembling pieces of knowledge into a big picture, and I began to look for the common ground behind the seemingly different approaches and methods I encountered and read about. My material

gradually evolved and was used for tutorials at several European conferences and summer schools as well as in courses taught at the Technical University of Denmark and elsewhere. In parallel, we designed and fabricated some more chips. In May 1999, I gave a one-week intensive course at Delft University of Technology, and it was when preparing for this I felt that the material was shaping up, and I set out to write the original text in 2000-2001.

Perspective and aim

Asynchronous design requires a mindset that is different from that generally employed in clocked design, and both my original tutorial and the book you are reading now tries to convey this different mindset. The book is intended as a beginner's text. The amount of formal notation is deliberately kept at a minimum, using instead plain English and graphical illustrations to explain the underlying intuition and reasoning

The book targets senior undergraduate and graduate students in Electrical and Computer Engineering and industrial designers with a background in conventional (clocked) digital design who wish to gain an understanding of asynchronous design. The book aims to enable its readers to design asynchronous control and data processing circuits of small and medium complexity, to read the literature on the topic, and to decide where/whether to use asynchronous circuits in some of their new designs.

At the Technical University of Denmark, the author is teaching a 5 ECTS credit point one-semester course using the material, supplemented by a few journal articles and a small design project.

Finally: If you use the material for regular class teaching, I would be grateful if you drop me an email. Any comments, bug reports, or suggestions for improvements or extensions, are also welcomed.

New material added

For those familiar with the original version of the text, the most significant changes and additions are the following:

1. The static data-flow handshake-component view presented in chapter 3 has proven very viable and is widely used. This chapter has been significantly updated, explaining better the spread token semantics of static data flow structures and providing additional design examples and peephole optimizations. As in the original version, 4-phase handshaking is assumed.
2. A similar static data-flow view of circuits using 2-phase handshaking has been lacking. Although conceptually a straightforward adoption/simplification from 4-phase handshaking, a closer look revealed some fundamental differences and challenges [86]. A completely new chapter (chapter 9) has been added covering 2-phase bundled-data circuits – both the

static data-flow view and circuit implementation styles introduced after the publication of the original 2001-version of this text. As 2-phase bundled data implementations have become increasingly popular, this new chapter fills a void.

3. The material on performance analysis using data-dependency graphs has been dropped from chapter 4 and replaced by a new chapter (chapter 7) that provides a more well-founded and more general approach using timed Petri nets. This new chapter is placed after the chapter on the design of speed independent control circuits where Petri nets and signal transition graphs are introduced.
4. Finally, a new chapter (chapter 8) has been added covering metastability, arbitration, and synchronization as well as the organization of different forms of multi-clock systems.

Jens Sparsø
Technical University of Denmark
June 2020
Email: jspa@dtu.dk

Acknowledgments

The material in this book has been collected over a period of more than 25 years. During these years, many people have helped me understand different bits and pieces, and I cannot possibly thank every individual. My primary source is all the many colleagues I have met at one of the so far 25 editions of the IEEE International Symposium on Asynchronous Circuits and Systems. Thank you all for many good and enlightening discussions over the years. Another source of learning and insight is teaching, and I also want to thank the many students who, over the years, have attended my course on design of asynchronous circuits at the Technical University of Denmark, and whose questions and comments have helped shape the material.

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 to perform the necessary synchronization, communication, and sequencing of operations. Expressed in “synchronous terms,” this results in a behavior that is similar to systematic fine-grain clock gating and local clocks that are not in phase and whose periods are determined by actual circuit delays. In essence, 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 a more comprehensive introduction to the mechanisms behind the advantages mentioned below in [11].

- Low power consumption, [154, 158, 45, 47, 114, 112]
... due to fine-grain clock gating and zero standby power consumption.
- High operating speed, [166, 167, 87]
... operating speed is determined by actual local latencies rather than global worst-case latency.

- Less emission of electro-magnetic noise, [154, 122]
...the local clocks tend to tick at random points in time.
- Robustness towards variations in supply voltage, temperature, and fabrication process parameters, [94, 109, 110]
...timing is based on matched delays (and can even be insensitive to circuit and wire delays).
- Better composability and modularity, [103, 89, 115, 147, 145]
...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 often represents 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 [104, 103]. Still, it was not until the late 1990s that projects in academia and industry demonstrated that it is possible to design asynchronous circuits that exhibit significant benefits in nontrivial real-life examples, and commercialization of the technology began to take place. Examples are presented in [133].

1.2 Aims and background

There are already several excellent articles and book chapters that introduce asynchronous design [57, 35, 36, 37, 11, 71, 145] as well as several monographs and textbooks devoted to asynchronous design including [133, 49, 23, 14, 107] – why then write yet another introduction to asynchronous design? There are several reasons:

- My experience from designing several asynchronous chips [144, 113], 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 first glance, they may seem different – an observation that is supported by different terminologies. Still, a closer look often reveals that the underlying principles and the resulting circuits are somewhat similar.
- Finally, most of the above-mentioned introductory articles and book chapters are comprehensive and heavy on mathematical formalism. 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, the reality is different. Today’s 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

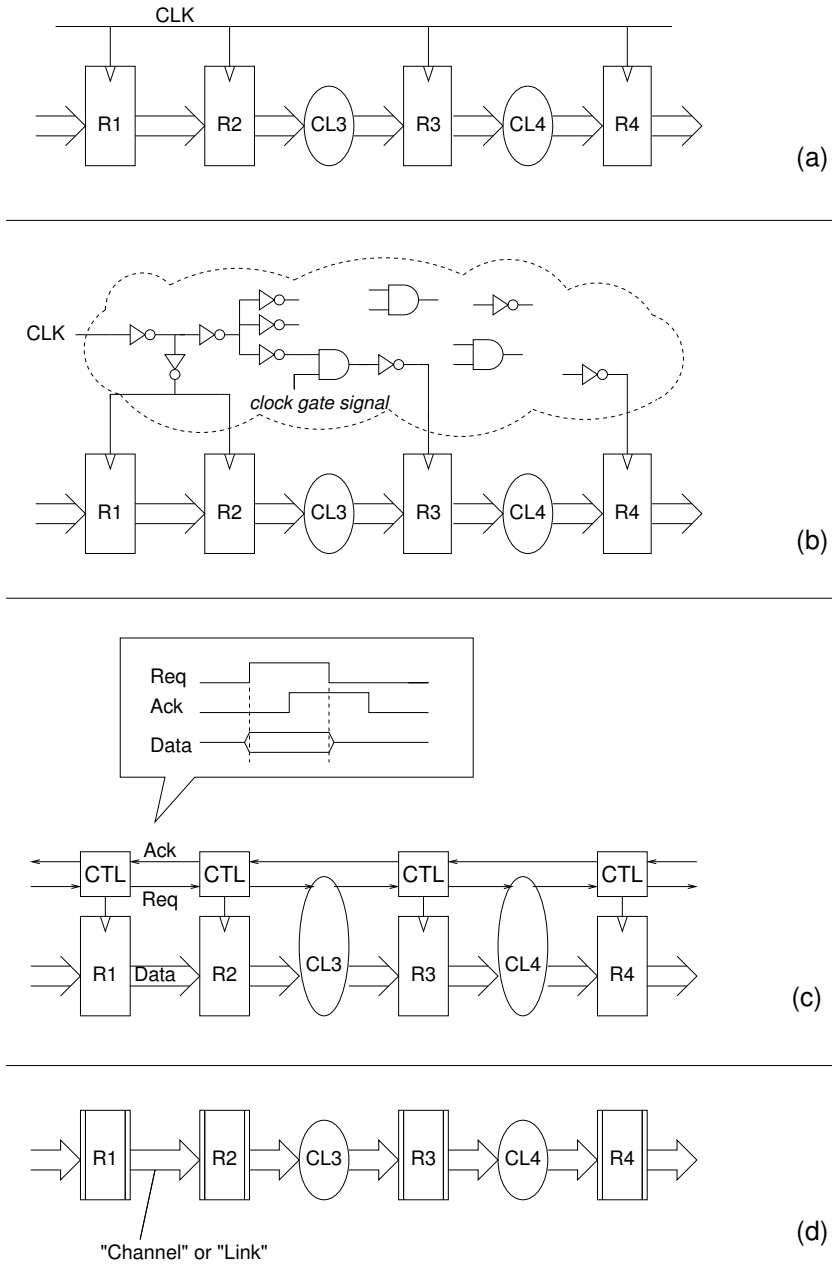


Figure 1.1: (a) A synchronous circuit, (b) a synchronous circuit with clock drivers and clock gating, (c) an equivalent asynchronous circuit, and (d) an abstract data-flow view of the asynchronous circuit. (The figure shows a pipeline, but it is intended to represent any circuit topology).

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 neighboring registers; for example, the simple request-acknowledge based handshake 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 [38]. 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 often hold 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 always encloses the transfer of exactly one data-token. Understanding this “token flow game” is crucial to the design of efficient circuits, and we 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 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 control the flow of tokens, whereas the combinational circuit blocks must be fully transparent to this handshaking. Ensuring this transparency is not always trivial, and it takes more than a traditional combinational circuit. For this reason, we 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 the book

Chapters 2 through 6 cover the basics of asynchronous design and constitute a whole. These chapters should be read in sequence. The remaining chapters 7 through 11 cover additional material, and these chapters can be read in any order depending on interest and need.

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. Chapter 4 addresses performance analysis and optimization of such structures, both qualitatively and quantitatively.

Chapter 5 addresses the circuit implementation of the handshake components introduced in chapter 3.

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.

Chapter 7 addresses performance analysis in the general case using timed Petri nets (benefiting from the introduction of Petri nets and signal transition graphs in chapter 6).

Chapter 8 covers metastability, arbitration, and synchronization in quite

some detail. The material is relevant for synchronous design and asynchronous design alike.

Chapter 9 addresses the design of asynchronous circuits using non-return-to-zero (NRZ) handshaking. This involves a new view on static data-flow structures that is different from that introduced in chapter 3. In addition, the chapter provides FPGA implementations of the basic asynchronous components, thereby enabling actual prototyping of asynchronous circuits.

Chapter 10 introduces more advanced topics related to the implementation of circuits using the 4-phase bundled-data protocol.

Finally, chapter 11 addresses hardware description languages and synthesis tools for asynchronous design. Chapter 11 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 as VHDL.

Chapter 2

Fundamentals

This chapter provides explanations 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 is 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. 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

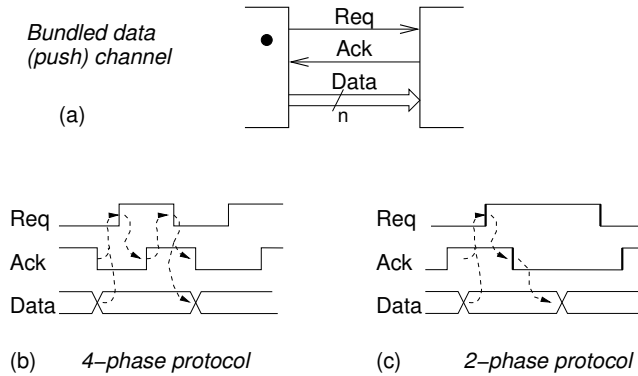


Figure 2.1: (a) A bundled-data channel. (b) A 4-phase bundled-data protocol. (c) A 2-phase bundled-data protocol.

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 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 protocol 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 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 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 10.1.1. In the following sections, we 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 sees 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 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.

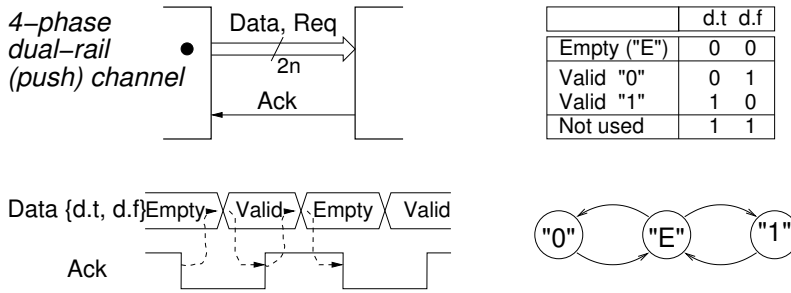


Figure 2.2: A delay-insensitive channel using the 4-phase dual-rail protocol.

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 [162], and it has some elegant 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 sees 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 sees 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.

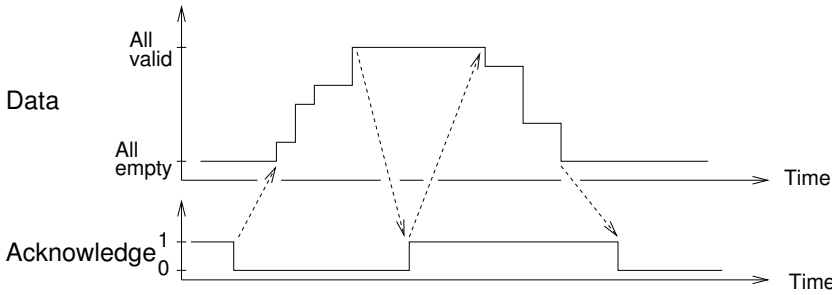


Figure 2.3: Illustration of the handshaking on a 4-phase dual-rail channel.

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.

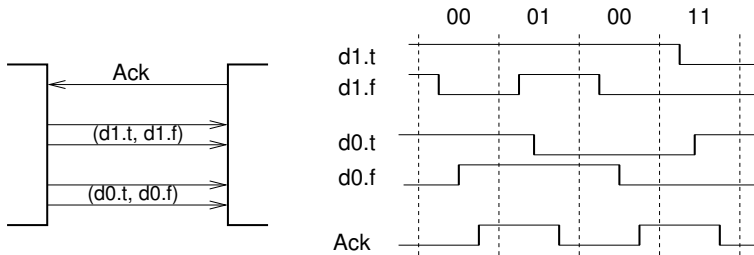


Figure 2.4: Illustration of the handshaking on a 2-phase dual-rail channel.

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 to use higher-radix encodings of data. 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:

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

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 acknowledgment, as well as a new component, the Muller C-element.

2.2 Indication and the Muller C-element

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 (usually including some feedback loops). 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).

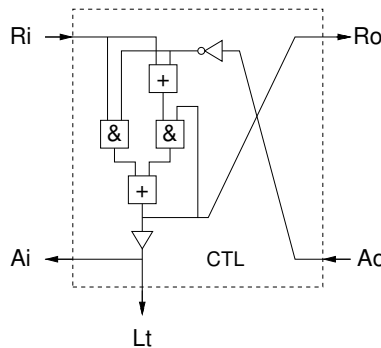
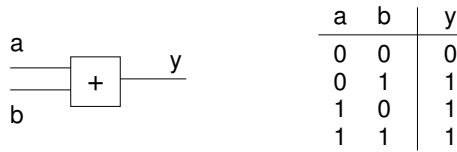
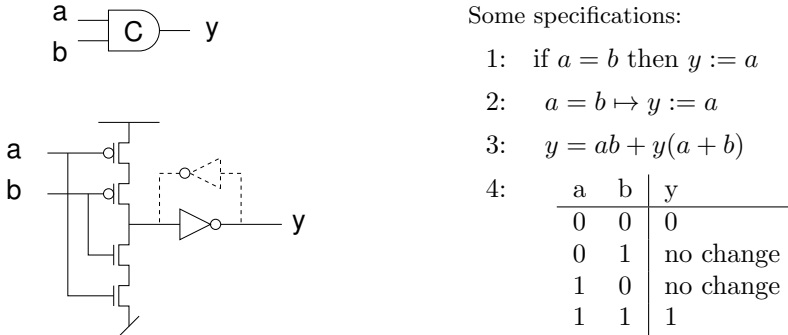


Figure 2.5: An example of an asynchronous control circuit. Lt is a “local” clock that is intended to control a latch.

**Figure 2.6:** A normal OR gate**Figure 2.7:** The Muller C-element: symbol, possible implementation, and some alternative specifications.

The concept of *indication* or *acknowledgment* plays an essential 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 address this issue in greater detail later in section 2.5.1 and 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

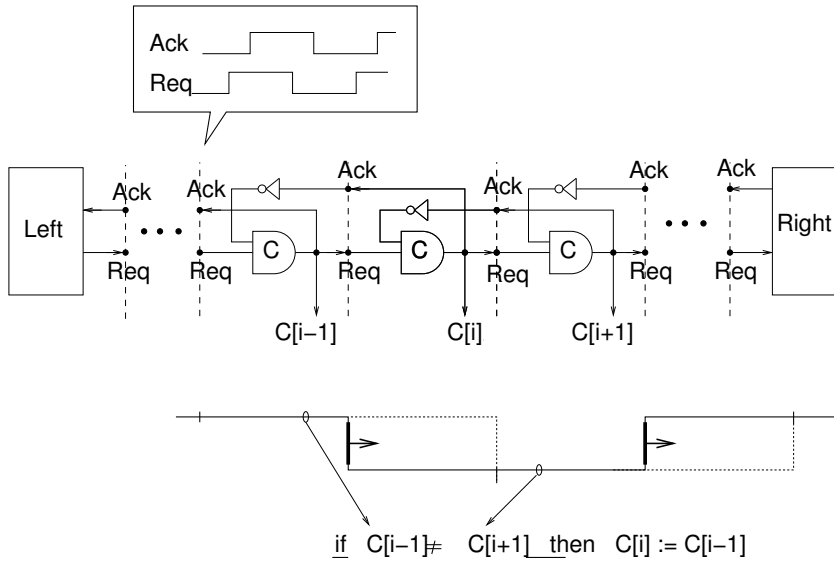


Figure 2.8: The Muller pipeline or Muller distributor.

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 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 behavior, and once you understand its behavior, 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 i th 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. Similarly, 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 reaches the right hand environment. If the right-hand environment does not respond to the handshake, the pipeline eventually fills. If this happens, the pipeline stops handshaking with the left-hand environment – the Muller pipeline behaves like a ripple through FIFO!

In addition to this elegant behavior, 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 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 latches as storage elements. More optimized and elaborate circuit implementations and more complex circuit structures are the topics of later chapters.

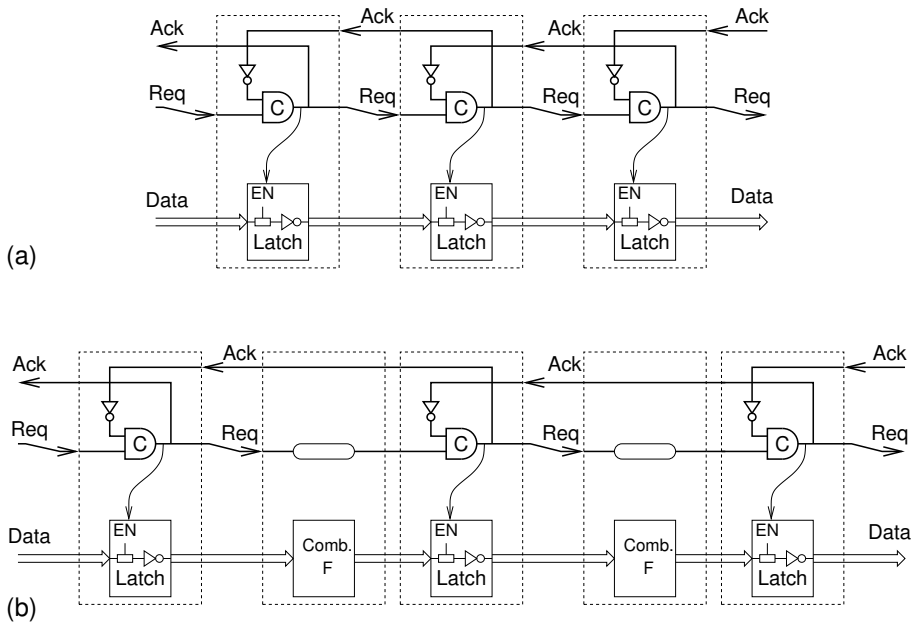


Figure 2.9: A simple 4-phase bundled-data pipeline.

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 neighboring 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 behavior, matching delays must be inserted in the request signal paths.

You may view this circuit as a traditional “synchronous” datapath, 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 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. For

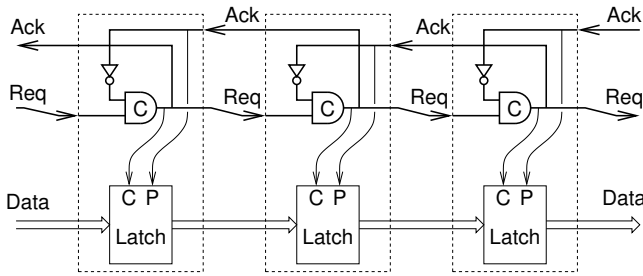


Figure 2.10: A simple 2-phase bundled-data pipeline.

the above implementation, this involves communication with both neighbours. Chapter 10 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 [147]. 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 signaling. 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 [147], but they emphasize 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 [147].

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-

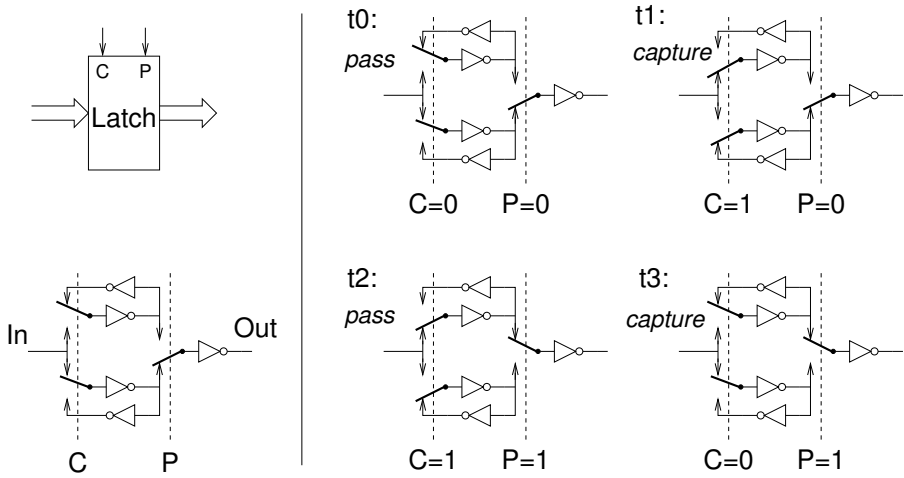


Figure 2.11: Implementation and operation of a capture-pass event controlled latch. At time t_0 , the latch is transparent (i.e., in pass mode), and signals C and P are both low. An event on the C input turns the latch into capture mode, etc.

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 [144], by the University of Manchester [45, 47] 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 requirements for speed. 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 due to the combined encoding of data and request. Figure 2.12 shows the implementation of a 1-bit wide and three stages 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 the empty codeword $\{d.t, d.f\} = \{0, 0\}$, causing the acknowledge signal out of

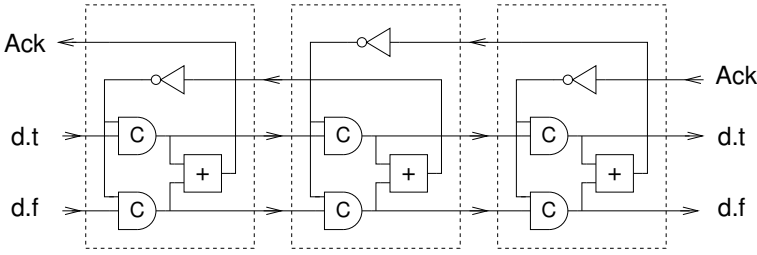


Figure 2.12: A simple 3-stage 1-bit wide 4-phase dual-rail pipeline.

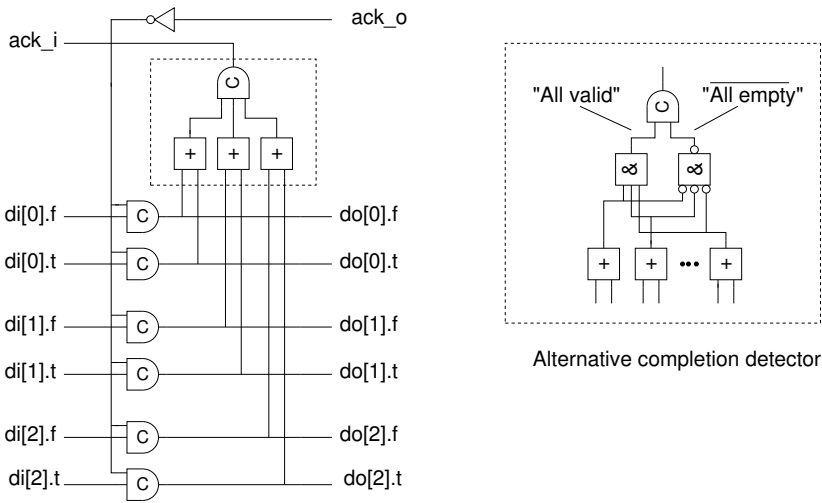


Figure 2.13: An N-bit latch with completion detection.

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. Towards a receiver, this does not guarantee that all bits in a word arrive at the same time. But often, the necessary synchronization is done in the function blocks. In [145, 143] we describe a design of this style using the DIMS combinational circuits explained below.

If bit-parallel synchronization is needed, the individual acknowledge signals can be combined into one global acknowledge using a C-element. Figure 2.13

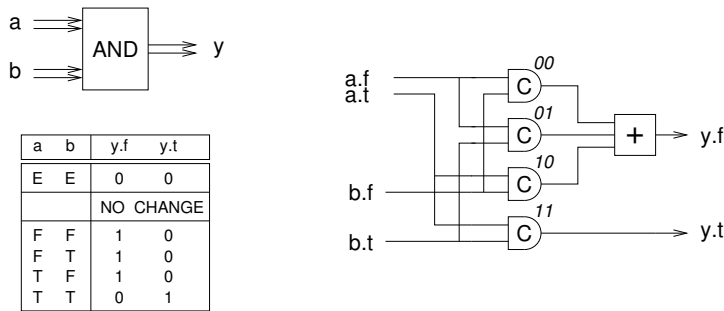


Figure 2.14: A 4-phase dual-rail AND gate: symbol, truth table, and implementation.

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). Similarly, 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 behavior 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 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 [104, 103]. While [104] develops the fundamental theory for the design of speed-independent circuits, [103] is a more practical introduction, including a design example: a bit-serial multiplier using latches and gates as explained above.

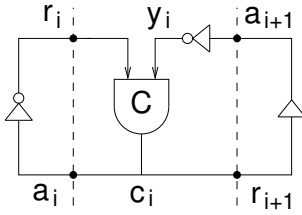
2.5 Theory

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. A reader who wishes to dig deeper into the theory is referred to the literature. Some recent starting points are [107, 57, 71, 37, 14].

2.5.1 The basics of speed-independence

We start by reviewing the basics of David Muller’s model of a circuit and the conditions for it being speed-independent [104]. 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 mimicking the handshaking behavior 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”, $z_i' = z_i$. A gate whose inputs have 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., $z_i' \neq z_i$. 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



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

Figure 2.15: Muller model of a Muller pipeline stage with “dummy” gates modeling the environment behavior.

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 restrict the discussion to an explanation of the fundamental ideas.

In the general case, several gates may be 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 other excited gates that 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 small 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 behavior of gate-level circuits, let us address the classification of asynchronous circuits.

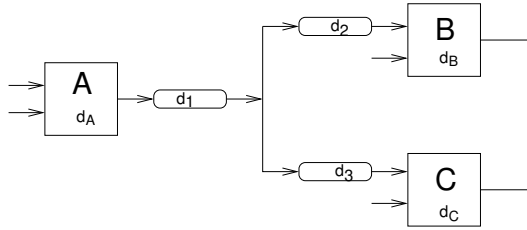


Figure 2.16: A circuit fragment with gate and wire delays. The output of gate A forks to inputs of gates B and C.

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 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 [91], and the Muller pipeline in figures 2.5, 2.8, and 2.15 is one important example. Circuits that are delay-insensitive, except for 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 evident 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 new outputs. 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 usually speed-independent (or in some cases even delay-insensitive), but the datapath 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 [48, 46, 50], 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 [113]. 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 most basic gates include forks.

In chapter 6, we 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 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 in their inputs, which in combination with slowly rising or falling signals can cause (and have caused!) circuits to malfunction. For static CMOS and circuits operating with low supply voltages (e.g. $V_{DD} \sim V_{tN} + |V_{tP}|$) 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 [152].

2.6 Test

When it comes to the commercial exploitation of asynchronous circuits, the problem of test comes to the fore. Testing 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 isochronic 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 neighboring 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 by the gates that take these signals as inputs, it follows that isochronic forks imply untestable stuck-at faults.

Testing asynchronous circuits incur 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 behavior, makes it difficult to test the feed-back circuitry that implements the state holding behavior. 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. Instead, the truth is that the techniques for testing synchronous circuits are not directly applicable. The situation is quite similar to the design of asynchronous circuits that we address in detail in the following chapters. Here a mix of new and well-known techniques is also needed. A good starting point for reading about the testing of asynchronous circuits is [131].

2.7 Summary

This chapter introduced a number of fundamental concepts. We now return to the main track of designing circuits. The reader may 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 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 behavior 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 assume a 4-phase protocol since this is most common. From a data-flow point of view this means that 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, and two-phase circuits are covered in depth in chapter 9. 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 (i.e., handshake 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 use a box symbol with double vertical lines to represent latches throughout the rest of this tutorial (see figure 3.1).

3.2 Pipelines and rings

Figure 3.1(a) shows a snapshot of a pipeline composed of five latches denoted L0, L1, ..., L4. The state of the C-elements controlling the latches is shown next to the C-elements. Figure 3.1(b) illustrates the state of the C-elements and indicate which C-elements are about to change state (similar to the explanation of the Muller pipeline in Section 2.3). With the given state of the C-elements, data latches L0, L3, and L4 are transparent and latches L1 and L2 are holding data. Data latch L1 holds a copy of data that has been copied into L2.

Figure 3.1(c) shows the more abstract *static data-flow view* of the pipeline. The “box arrows” represent channels or links consisting of request, acknowledge, and data signals (as explained on page 3). 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 be copied forward, and after this has happened, the bubble has moved backward.

This forward copying of tokens is illustrated in figure 3.1(d). For each step, figure 3.1(d) first shows “what happens” (tokens are copied forward) and then the resulting state of the circuit. This is to emphasize that a token-bubble pair is not simply swapped in an atomic operation; tokens are copied forward, leaving bubbles behind. It may be useful to realize the similarity between how (valid and empty) tokens are copied forward and how the rising and falling transitions (or the 0’s and 1’s) in the Muller pipeline control circuit moves forward.

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

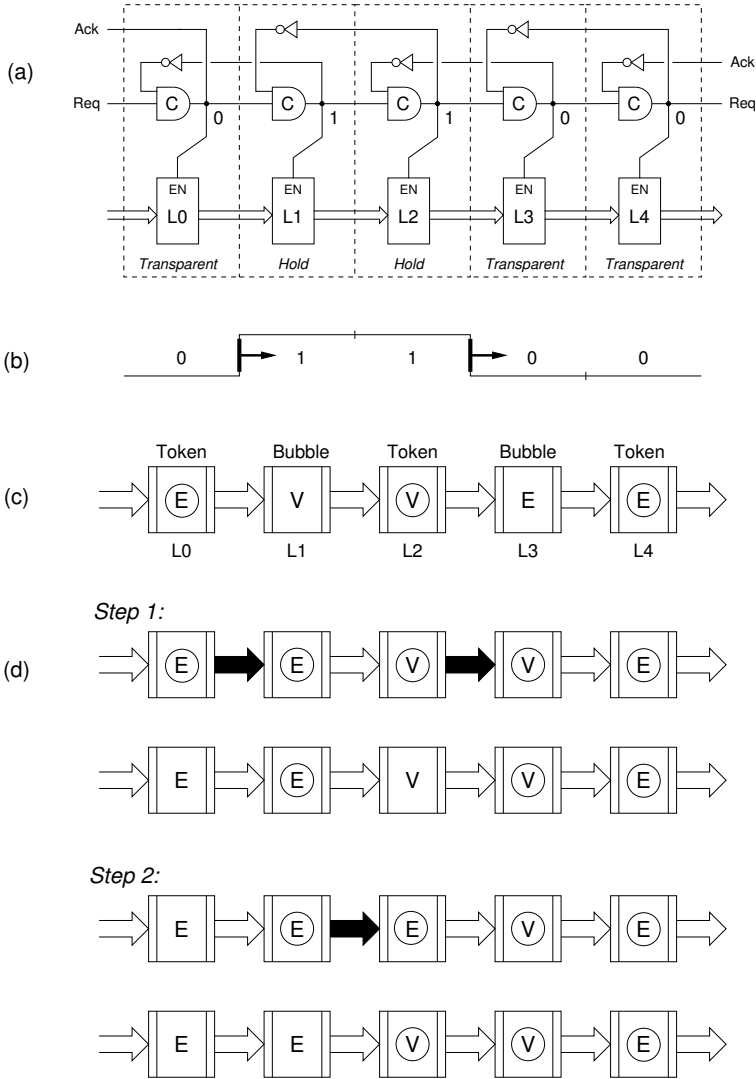


Figure 3.1: (a) A possible state of a five stage pipeline. (b) The state of the Muller pipeline control circuit. (c) The abstract “static data-flow structure” representation of the pipeline in figure 3.1(a). (d) Illustration of how the operation of the pipeline can be understood at this level: as tokens being copied forward (into stages holding bubbles).

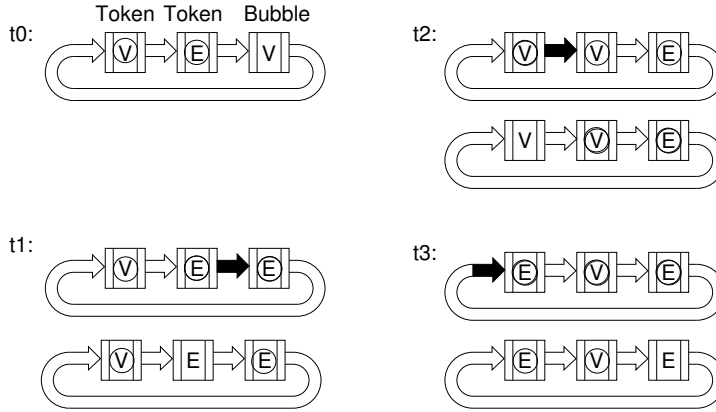


Figure 3.2: A three-stage ring. A possible initial state (t_0), and the following sequence of data transfers (t_0 , t_1 , and t_2).

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 at times t_1 , t_2 and t_3 are also shown in figure 3.2(b) – again emphasizing the tokens are *copied* forward. 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 is 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.

As a final note, the more theoretically inclined reader is referred to [140, 141] for a more formal and Petri-net-based discussion of the token-bubble semantics of static data-flow structures.

3.3 Building blocks

Figure 3.3 shows a minimum set of components that is sufficient to implement a large class of asynchronous circuits (static data-flow structures with deterministic behavior, i.e., without arbiters). The components can be grouped into five categories, as explained below. In the following sections, we will see examples of the token-flow behavior in structures composed of these components.

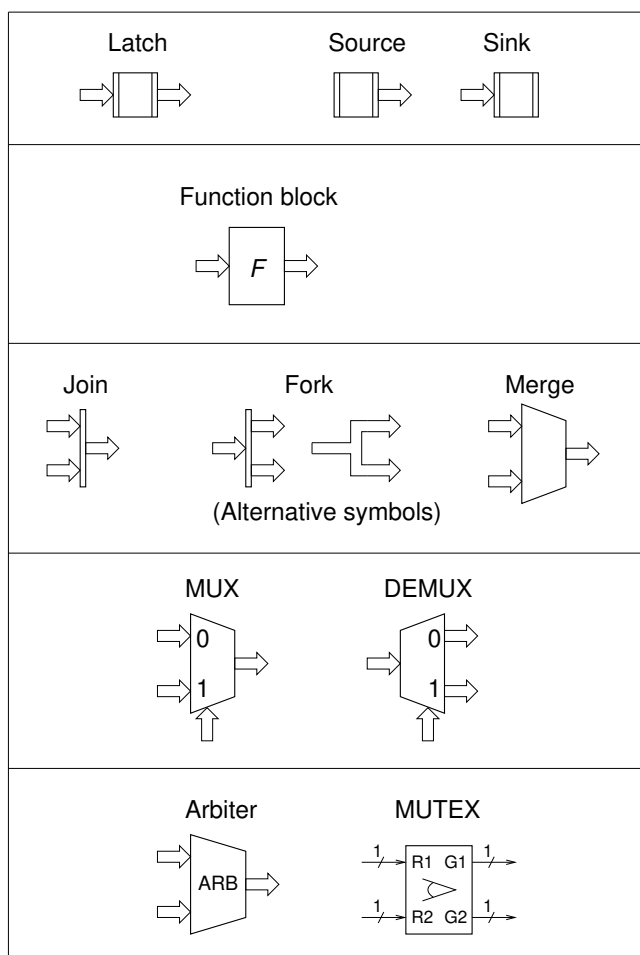


Figure 3.3: A basic and sufficient set of handshake components to implement data-flow style asynchronous circuits.

Latches provide storage for variables and implement the handshaking that supports the token flow. In addition to the normal handshake 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 shows 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: (1) waits for a token on its input, (2) computes the required combinatorial function, and (3) issues a token on its output. Both empty and valid tokens are handled in this way. The implementation of function blocks is a topic with much greater depths and insights than this simple description hints. The topic 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 often omit joins and forks from circuit 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 synchronizes the control channel and the relevant input channel and it sends the input data to the data output. The other input channel is ignored. Similarly, a DEMUX synchronizes the control and data input channels and steers the input to the selected output channel.

Mutual exclusion and arbitration: The merge component presented above assumes that handshakes on the two input channels are mutually exclusive. If this is not the case, an arbiter is needed. An arbiter selects the channel whose request is asserted first. The other input is ignored until the current input-to-output handshake completes. If the request signals on both inputs are asserted simultaneously, the arbiter must make a random choice. This involves metastability and may take an unbounded amount of time. A circuit implementing this core operation is a MUTEX that accepts two request inputs (R1 and R2) but never asserts more than one of its two outgoing grant signals (G1 or G2). Metastability, mutual exclusion, and arbitration are interesting and challenging topics that we discuss later in chapter 8.

As mentioned before, the latches implement the handshaking and thereby the token flow in a circuit. All other components must be transparent to the handshaking. This has significant implications for the implementation of these components!

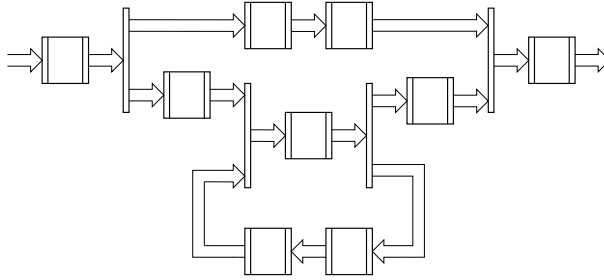


Figure 3.4: An example asynchronous circuit composed of latches, forks and joins.

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 behavior 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.

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. Likewise, the right-hand environment takes part in one handshake cycle and consumes a valid value and an empty value.

To reduce the amount of illustration in figure 3.5, we show only a single copy of the schematic for each “step,” combining what in figures 3.1 and 3.2 was illustrated as sub-steps *a* and *b*.

Because the flow of tokens is controlled by local handshaking, the circuit could exhibit many other behaviors. 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.

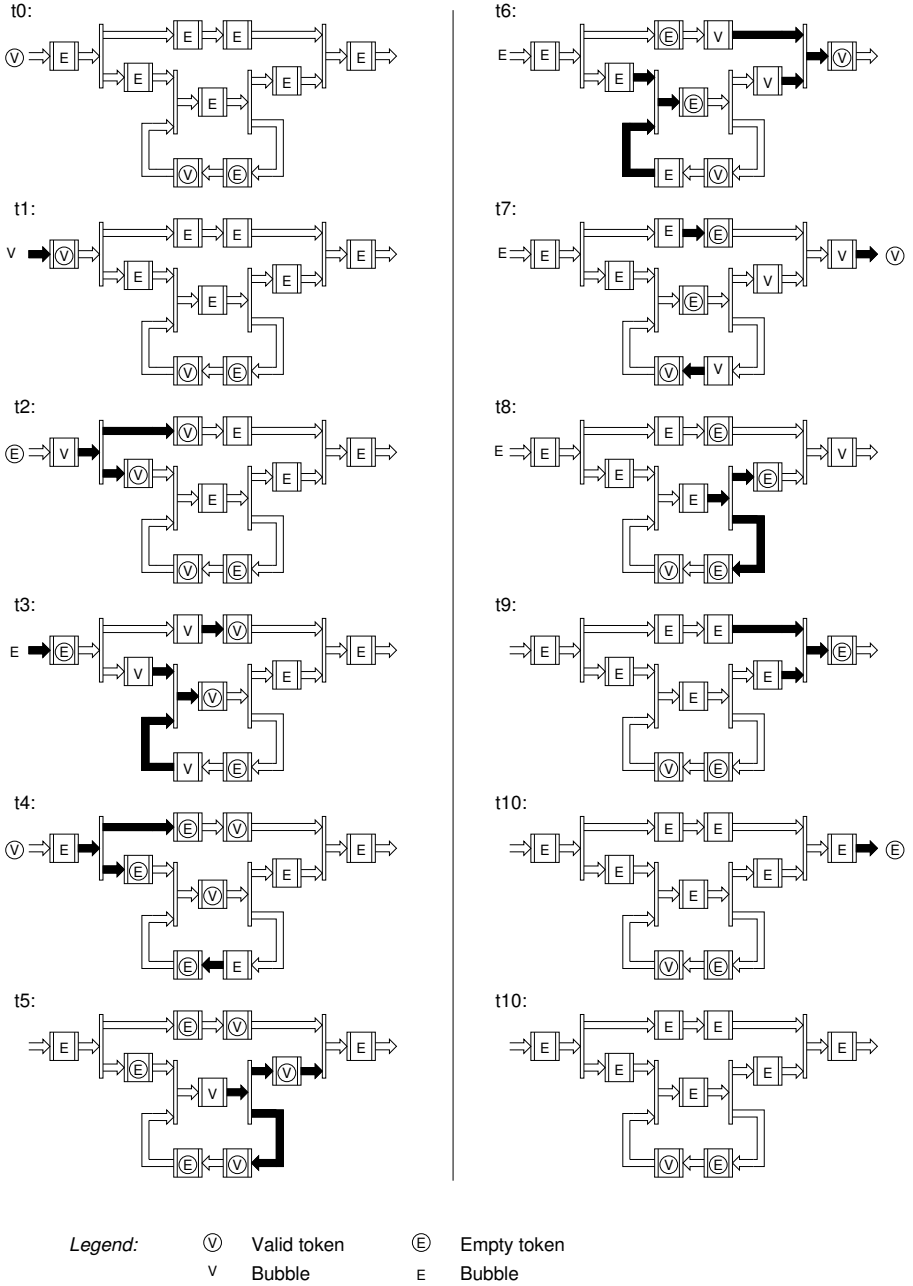


Figure 3.5: A possible operation sequence of the example circuit from Figure 3.4.

3.5 Simple applications of rings

This section presents a few simple circuits based on one or more rings.

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.”

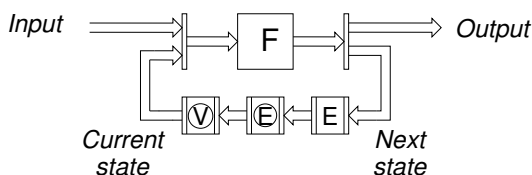


Figure 3.6: Implementation of an asynchronous finite state machine using a ring.

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 several times until the computation terminates, and (3) output the result. The necessary control is not shown. The figure shows one particular implementation. Possible 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

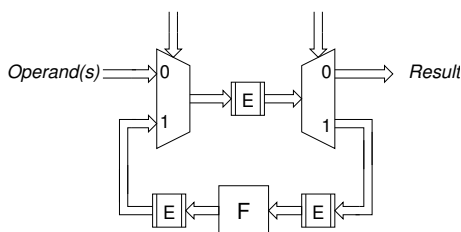


Figure 3.7: Implementation of an iterative computation using a ring.

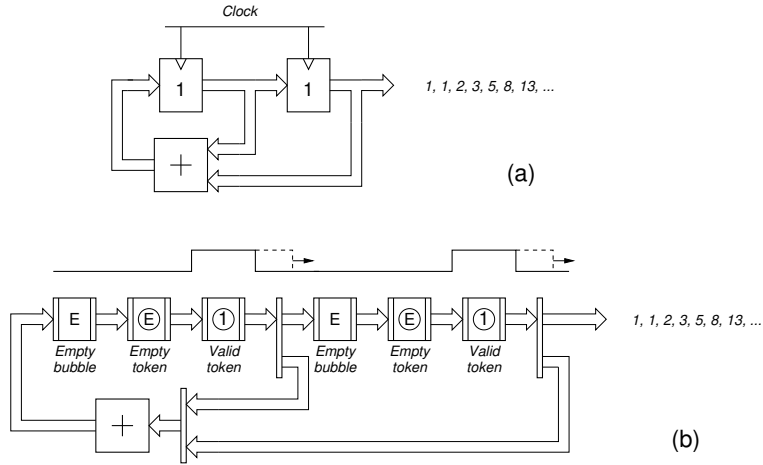


Figure 3.8: (a) A synchronous circuit producing the sequence of Fibonacci numbers. (b) A direct 4-phase asynchronous re-implementation consisting of two nested rings.

latches. In [166] 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 [167].

3.5.3 Fibonacci sequence generator

Figure 3.8(a) shows a clocked synchronous circuit that produces the sequence of Fibonacci numbers on its output port. An asynchronous re-implementation of the Fibonacci circuit is shown in figure 3.8(b). Starting from the initial state shown in the figure, the circuit will provide the first number in the sequence and assert request. When the environment acknowledges the receipt of a Fibonacci number, the circuit will compute and present the next number in the sequence. The circuit consists of two rings. The handshake components in an inner 3-stage ring with one token are also part of an outer 6-stage ring with two tokens. Assuming a 4-phase bundled-data implementation, the figure also shows the initial state of the C-elements in the Muller pipeline control path to illustrate the wave-analogy introduced in figure 3.1 on page 31.

We will revert to the asynchronous Fibonacci circuit later in two contexts. First, in the next section (3.6) where we introduce and explain the spread-token semantics of static data-flow structures, and later in chapter 9 where we develop a token-bubble-view of static data-flow structures using 2-phase handshaking. For the latter, we will use the waveform illustration as a stepping stone for a discussion of 2-phase handshaking.

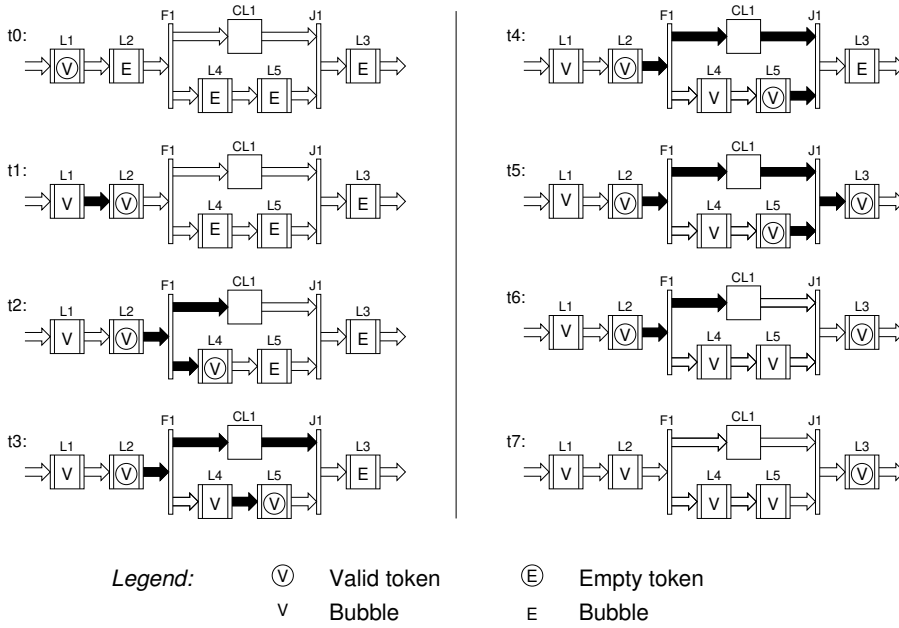


Figure 3.9: A circuit in which tokens spread.

3.6 When tokens spread

As explained in section 3.2, and illustrated in figures 3.1 and 3.2, tokens are copied forward, but the illustrations only showed situations where a token is stored in a single latch except for a short time interval where the token is copied forward. In the following, we will see that tokens may spread across more stages and for a longer time. It is important to understand this more elaborate spread-token semantics when designing and working with static data-flow asynchronous circuits.

A reader wanting to cover some ground quickly may skip this section for now and proceed with the remainder of this chapter. A reader who wants to dig deeper into the formal underpinnings may supplement this section with [140, 141].

Consider the circuit in figure 3.9. It is composed of five handshake latches (denoted L1, L2, L3, L4, and L5), a fork (denoted F1), a join (denoted J1), and a function block (denoted CL1). At time t_0 latch L1 holds a valid token, and all other latches hold empty bubbles.

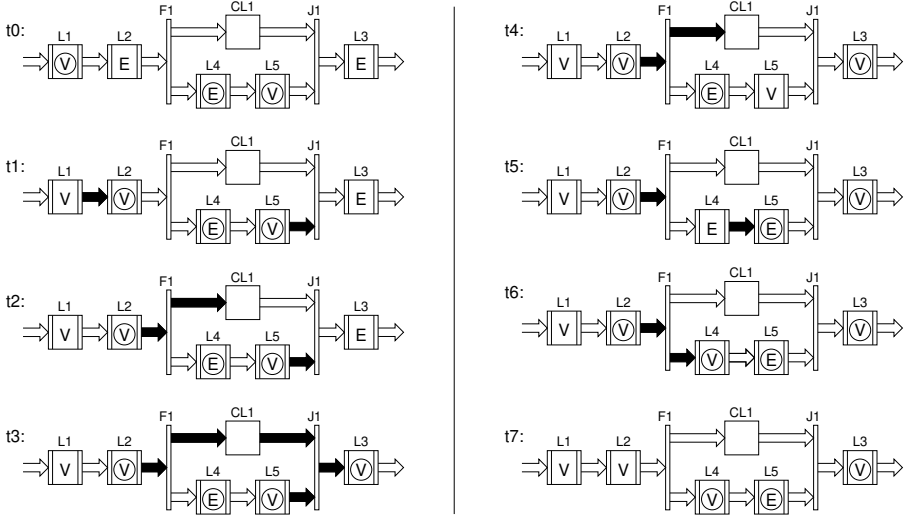


Figure 3.10: Another circuit in which tokens spread.

At time t_1 the valid token in $L1$ is copied into $L2$ and quickly after $L1$ becomes a valid bubble. Following this, at time t_2 , fork $F1$ forwards the valid token in $L2$ to both its output branches. In the lower branch, the token is copied into $L4$. At time t_3 , the token in $L2$ propagates through $CL1$ in the upper branch, and the token in $L4$ is copied into $L5$. Strictly speaking, $L4$ is now holding a valid bubble (an old duplicate), but $L2$ is still holding the same token as $L5$ – the token spreads across $L2$, $L4$ and $L5$. The token in $L2$ does not become a bubble until fork $F1$ receives an acknowledgment on both output channels. At time t_4 , join $J1$ receives a token on both its input ports, and it outputs a token that finally, at time t_5 , is copied into $L3$. At this moment, the same token now spreads across $L2$ - $L3$ in the upper branch and across $L2$ - $L4$ - $L5$ - $L3$ in the lower branch. Next, at time t_6 , join $J1$ passes the acknowledge from $L3$ towards $CL1$ and $L5$ and $L5$ now holds a valid bubble. Finally, at time t_7 , the upper output port of fork $F1$ receives an acknowledge from $CL1$. Synchronizing this with the acknowledge on the lower output that has been pending since time t_2 , the fork finally produces an acknowledgment towards $L2$, which now becomes a valid bubble.

It should be noted that the circuit in figure 3.9 has been devised for illustrative purposes. Latches $L4$ and $L5$ do no good. They could be deleted, and this would speed up the lower branch of the circuit and reduce the latency

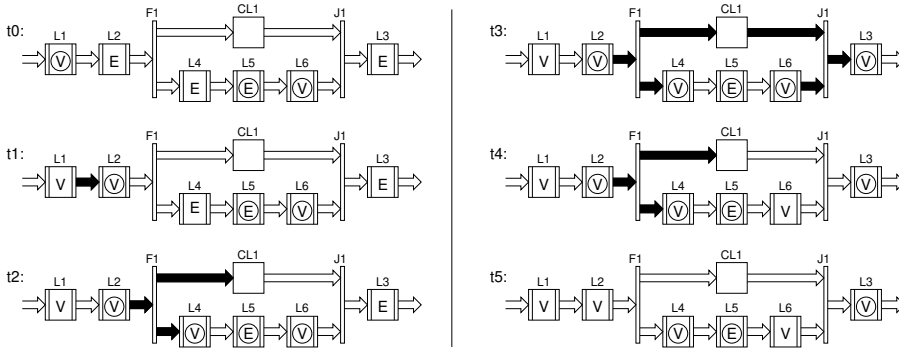


Figure 3.11: By adding an extra latch to the bottom branch of the circuit in figure 3.10, the bottom branch now has a bubble, and the whole circuit performs better.

from input to output.

The same circuit could be initialized differently, with a valid-empty token pair in latches L5 and L4, as shown in figure 3.10. The valid token in L5 changes the time-behavior of the circuit (and also its functionality, but this is not the point here). Again, fork F1 will forward the valid token in L2 to both its output branches (time t_2), but as latch L4 holds an empty token, and the valid token from L2 cannot (yet) propagate into the L4-L5-branch of the circuit. Like before, the token from L2 propagates through CL1 to the upper input of join J1. Here it is joined with the valid token in L5, and a valid token is copied into latch L3 (time t_3). The acknowledge signal from L3 is propagated to both input ports of join J1. In the upper branch, it quickly reaches the upper output port of fork F1. But as the bottom L4-L5-branch has not yet consumed the token from L2, fork F1 cannot issue an acknowledge towards latch L2. After some activity in the bottom L4-L5-branch, L4 eventually reaches a state where it holds an empty bubble (time t_5). Latch L4 can now finally receive the token from L2 (time t_6). And when L4 acknowledges, fork F1 synchronizes the acknowledge signals from CL1 and L4, and it produces an acknowledge signal to L2. Not until this point, does the state of L2 change from a valid token to a valid bubble (time t_7).

It should be noted that the circuit in figure 3.10 also has poor performance. A better design would have one extra latch, as shown in figure 3.11. This allows two concurrent actions: The token in L2, that is forked through the lower output of fork F1, is copied into L4. At the same time, the token in

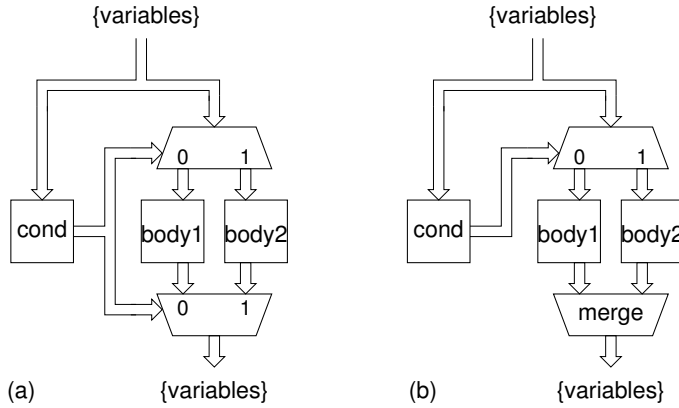


Figure 3.12: A template for implementing *if* statements.

L2 that is forked through the upper output of fork F1 will propagate through CL1, join with the token from L6, and the token thus produced by join J1 will be copied into L3. This circuit has better performance and the reason is the bubble in L4 in the L4-L5-L6 branch.

After this more detailed coverage of the token-bubble dynamics of static data-flow structures, let us now turn our attention towards building circuits with interesting functionality.

3.7 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 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, may recognize the great similarities between asynchronous circuits and control-data-flow graphs [38, 146].

if <cond> then <body1> else <body2>

An asynchronous circuit template for implementing an *if-statement* is shown in figure 3.12(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> are mutually exclusive, it is possible to replace the controlled MUX in the bottom of the circuit with a

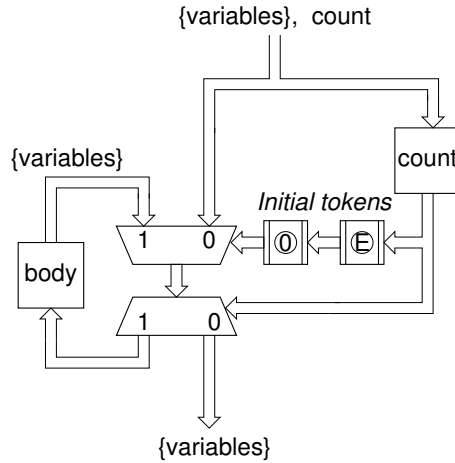


Figure 3.13: A template for implementing *for* statements.

simpler MERGE, as shown in figure 3.12(b). The circuit in figure 3.12 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.

for <count> do <body>

An asynchronous circuit template for implementing a *for statement* is shown in figure 3.13. 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>.

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).

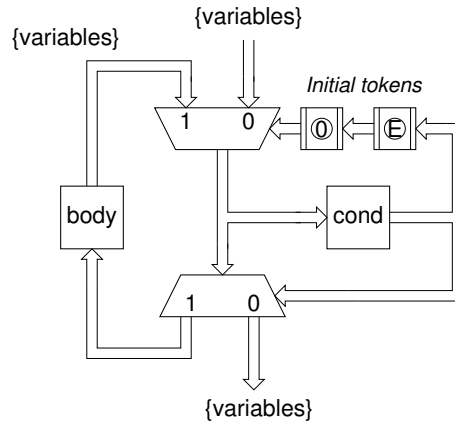


Figure 3.14: A template for implementing *while* statements.

while <cond> **do** <body>

An asynchronous circuit template for implementing a *while* statement is shown in figure 3.14. 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.

3.8 A more complex example: GCD

Using the templates just introduced, we 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 3.15 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 11, we use the same example to illustrate the Tangram language and the associated syntax-directed compilation process (section 11.3.3 on pages 208–208).

The implementation of GCD is shown in figure 3.16. It consists of a *while*-template whose body is an *if*-template. Figure 3.16 shows the circuit, including

```

input (a,b);
while  $a \neq b$  do
    if  $a > b$  then  $a \leftarrow a - b$ ;
    else  $b \leftarrow b - a$ ;
output (a);

```

Figure 3.15: A programming language specification of GCD.

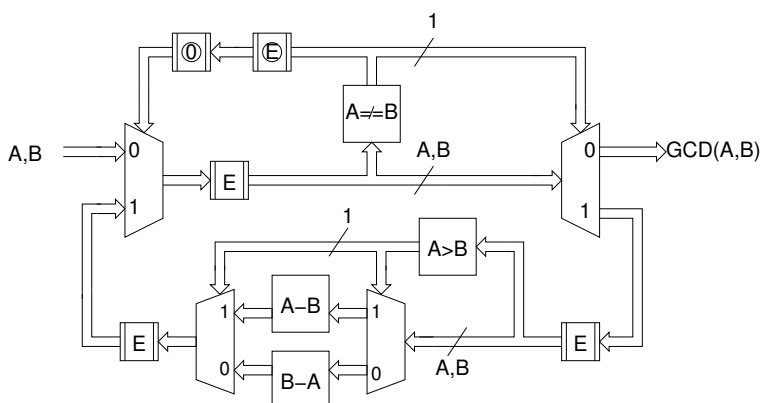


Figure 3.16: An asynchronous circuit implementation of GCD.

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.

3.9 Pointers to additional examples

3.9.1 A low-power filter bank

In [113], we reported on the design of a low-power IFIR filter bank for a digital hearing aid. The circuit 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.

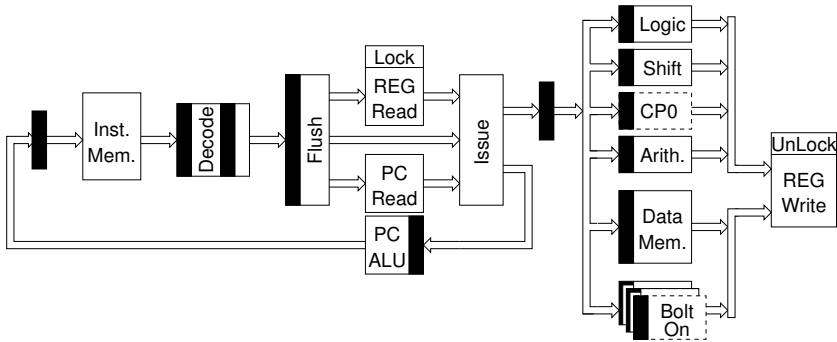


Figure 3.17: Architecture of the ARISC microprocessor.

3.9.2 An asynchronous microprocessor

In [20], 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 is shown in figure 3.17 can be understood as a simple data-flow structure. The solid-black rectangles represent latches, the box-arrows represent channels, and the text-boxes represent 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 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 [20]. Other asynchronous microprocessors are based on similar principles.

3.9.3 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 [145, 143]. 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.

3.10 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 addresses performance analysis at this level of abstraction.

Chapter 4

Performance

This chapter addresses the performance analysis and optimization of asynchronous circuits at the level of static data-flow structures, as introduced in the previous chapter. First, we develop a qualitative understanding, and then we introduce some fundamental quantitative performance parameters that characterize pipelines and rings built using identical stages.

4.1 Introduction

In a synchronous circuit, performance analysis and optimization is a matter of finding the critical path, i.e., the signal path between any two registers with the largest propagation delay. This determines the period of the clock signal. This process is known as static timing analysis, and because the global clock partitions the circuit into many combinatorial circuits that can be analyzed individually, it is a rather simple task, even for a large circuit.

For an asynchronous circuit, performance analysis and optimization is a global problem, and therefore a much more complex problem. The use of handshaking makes the timing in one component dependent on the timing of its neighbors, which again depends on the timing of their neighbors, 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, for example expressed in terms of the time it takes to complete a handshake cycle, can even exhibit transients and oscillations (as we will see in chapter 7).

We 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. Then we introduce some quantitative performance parameters that characterize pipelines and rings composed

of *identical* stages. Using these parameters, a designer can make first-level design decisions.

Analysis of more general and irregular structures – i.e., the general case – will be discussed later in chapter 4 when the necessary formal notation is in place.

Throughout the chapter, we assume 4-phase handshaking, and the examples we provide all use bundled-data circuits. The use of 4-phase handshaking means that the static data-flow model involves: valid tokens, valid bubbles, empty tokens, and empty bubbles.

It is left as an exercise for the reader to make the simple adaptations that are necessary for dealing with 2-phase handshaking that involves only valid tokens and valid bubbles.

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 number of tokens in the FIFO is invariant. This example is relevant because many designs use FIFOs in this way, and because it models the behavior of shift registers as well as rings – structures in which the number of tokens is also invariant.

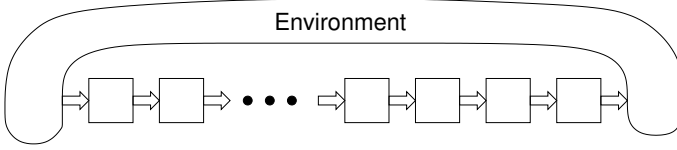
A relevant performance figure is the throughput, which is the rate at which tokens are input to or output from the shift register. This figure is proportional to the time it takes to shift the contents of the chain of latches one position to the right.

Figure 4.1(b) illustrates the behavior of an implementation in which there are $2N$ latches per valid token, and figure 4.1(c) illustrates the behavior 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.

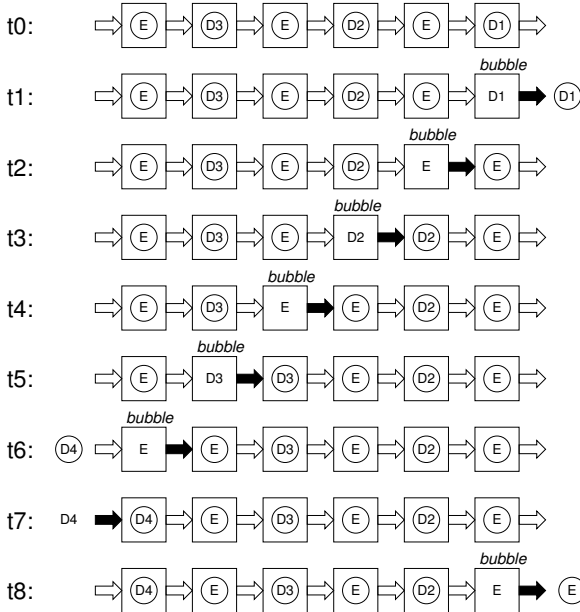
In figure 4.1(b), at time t_1 , the environment reads the valid token, $D1$, as indicated by the solid channel symbol. This introduces a bubble that enables data transfers to take place one at a time ($t_2 - t_5$). At time t_6 , the environment inputs a valid token, $D4$, and at this point, all elements 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 im-

(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:

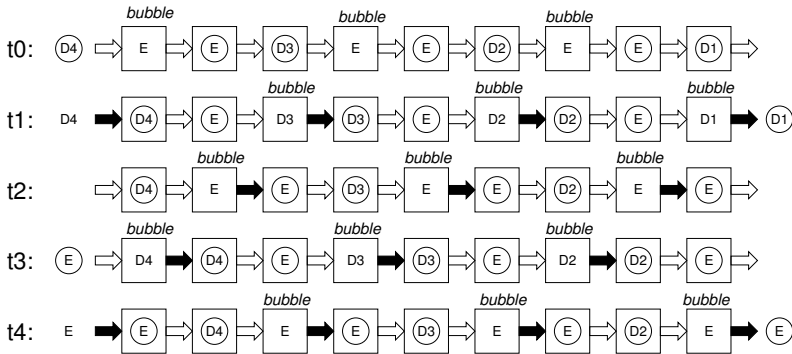


Figure 4.1: A FIFO and its environment. The environment alternates between reading a token from the FIFO and writing a token into the FIFO.

proving 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.

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 one 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 in expediciencies. 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

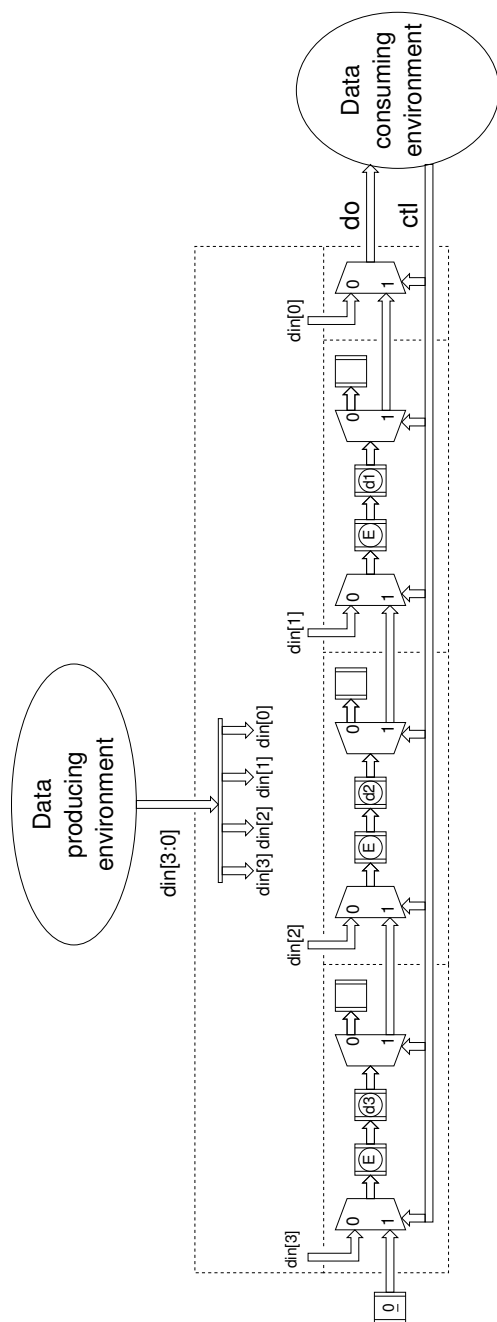


Figure 4.2: Initial design of the shift register with parallel load.

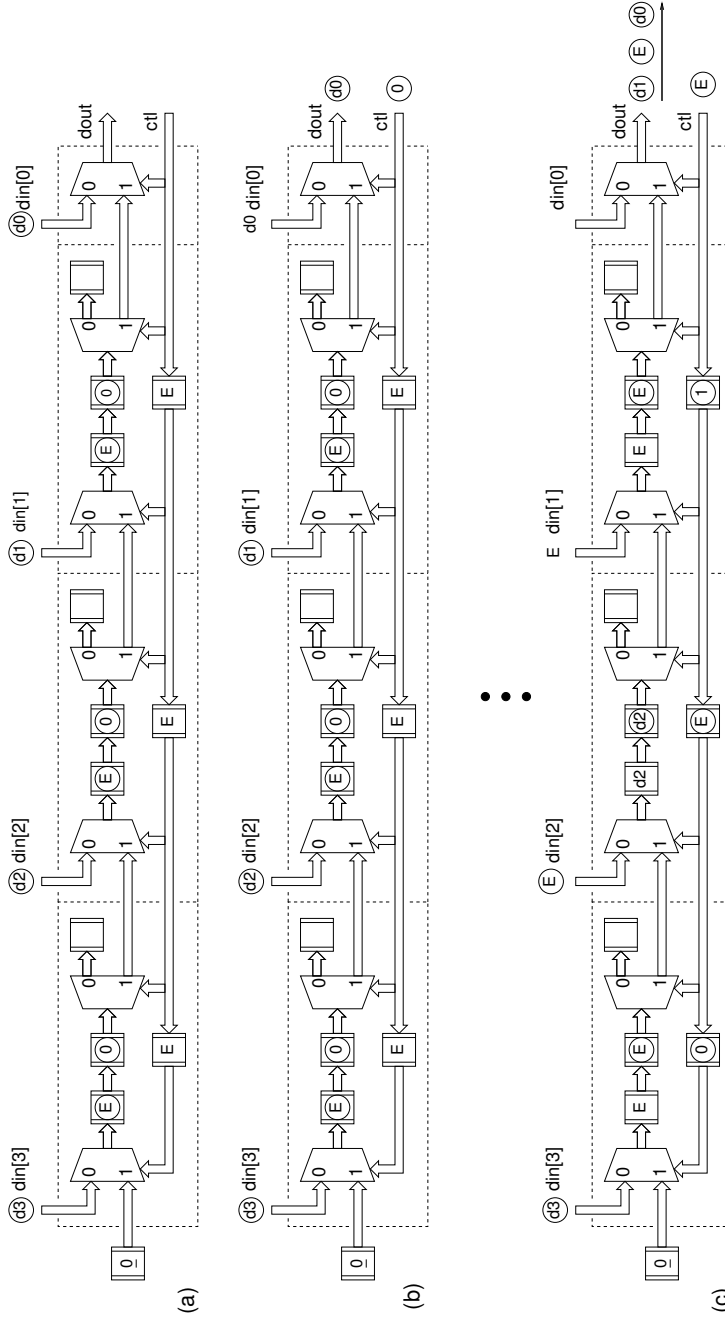


Figure 4.3: Improved design of the shift register with parallel load.

(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 to the datapath in each stage of the circuit corresponding to the situation in figure 4.1(c). However, if the extra latches are added to the control path instead, as shown in figure 4.3(a) on page 54, they will solve both problems.

This improved design exhibits an interesting and illustrative dynamic behavior: 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 is 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

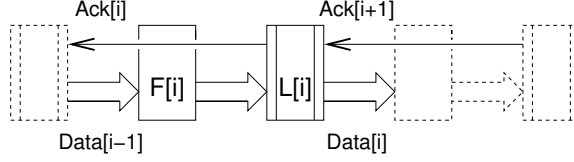
When the overall structure of a design is being decided, it is important to determine the optimal number of latches or pipeline stages in the rings and pipeline fragments from which the design is composed. In order to establish a basis for first-order design decisions, this section introduces some quantitative performance parameters. We restrict the discussion to 4-phase handshaking and bundled-data circuit implementations, and we only consider rings with 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 [168, 169, 170], these parameters are defined as follows:

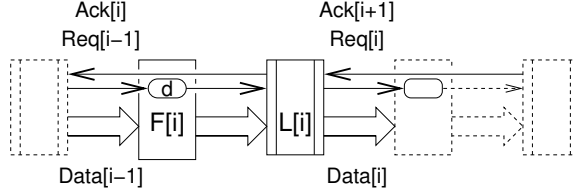
Latency: The latency is the delay from the input of a data item until the corresponding output data item is produced. When data flows in the forward direction, acknowledge signals propagate in the reverse direction. Consequently, two parameters are defined:

- 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

Dual-rail pipeline:



Bundled-data pipeline:

**Figure 4.4:** Generic pipelines for definition of performance parameters.

latencies for propagating a valid token and an empty token, respectively. It is assumed that these latencies are constants, i.e., that they are independent of the value of the data. [As the forward propagation of an empty token does not “compute” it may be desirable to minimize $L_{f.E}$. In the 4-phase bundled-data design, 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:

$$P = L_{f.V} + L_{r\uparrow} + L_{f.E} + L_{r\downarrow} \quad (4.1)$$

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:

$$P = 2L_f + 2L_r \quad (4.2)$$

We will also consider circuits where $L_{f,V} > L_{f,E}$ and, as we will see in section 7.7 and again in section 10.3, 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 7.7, we analyze a pipeline whose period is:

$$P = 2L_r + 2L_{f,V} \quad (4.3)$$

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 :

$$W_d = \frac{P}{L_f} \quad (4.4)$$

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 is limited by the lack of bubbles, and if there are more pipeline stages, the cycle time is limited by the forward latency through the pipeline stages. In [168, 169, 170] these two modes of operation are called *bubble limited* and *data limited*, respectively.

The cycle time of an N -stage ring in which there are 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):

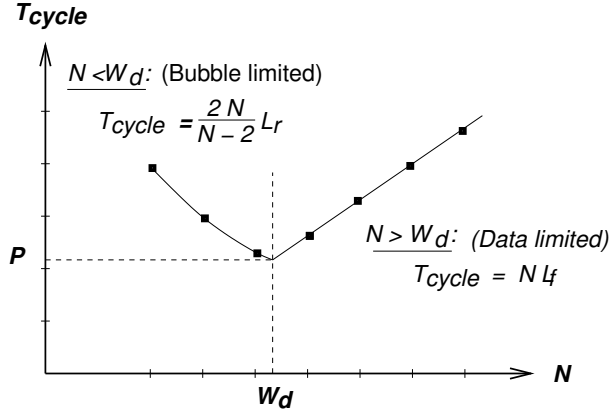


Figure 4.5: Cycle time of a ring as a function of the number of pipeline stages in it.

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

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

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:

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

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 [168, 169, 170]. 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 [28, 101, 102, 39] 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).

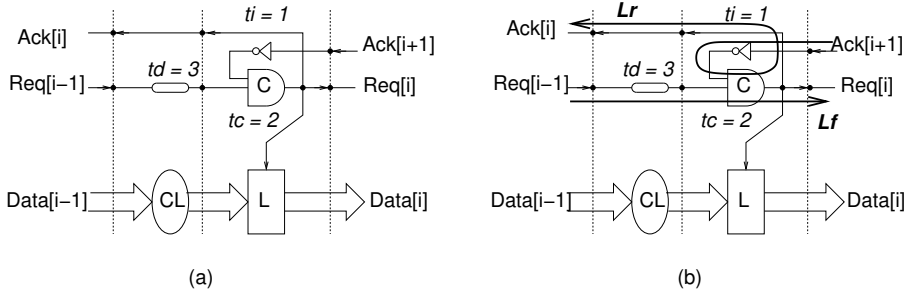


Figure 4.6: (a) A simple 4-phase bundled-data pipeline stage, and (b) an illustration of its forward and reverse latency signal paths.

4.3.3 Example 3: Performance of a 3-stage ring

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 combinational 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 combinational circuit. Without the combinational 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

Table 4.1: Performance of different simple ring configurations.

Parameter	FIFO		Pipeline	
	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

stages, and if W_d is not an integer, it is necessary to analyze rings with $\lfloor W_d \rfloor$ and $\lceil W_d \rceil$ stages 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. In later chapters, we will introduce (timed) Petri nets and use these to analyze more general structures involving rings and pipelines composed of non-identical stages.

4.4 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 (four-phase)

In this chapter, we address the implementation of handshake components using 4-phase bundled-data and 4-phase dual-rail handshaking. Implementation of circuits using 2-phase handshaking will be considered separately in chapter 9.

First, we consider the basic set of components, introduced in section 3.3 on page 33: (1) the latch and its degenerate forms, source and sink, (2) the unconditional data-flow control elements join, fork and merge, (3) the conditional flow control elements MUX and DEMUX and (4) function blocks.

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 5.6–5.9) 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, the sink, and the source

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 neighboring latches.

Possible implementations of the handshake latch were shown in chapter 2: Figure 2.9 on page 18 showed a 4-phase bundled-data latch, and figures 2.12–2.13 on page 21 show the implementation of a 4-phase dual-rail latch.

For convenience, these are shown again in figure 5.1 along with their degenerate forms: source and sink.

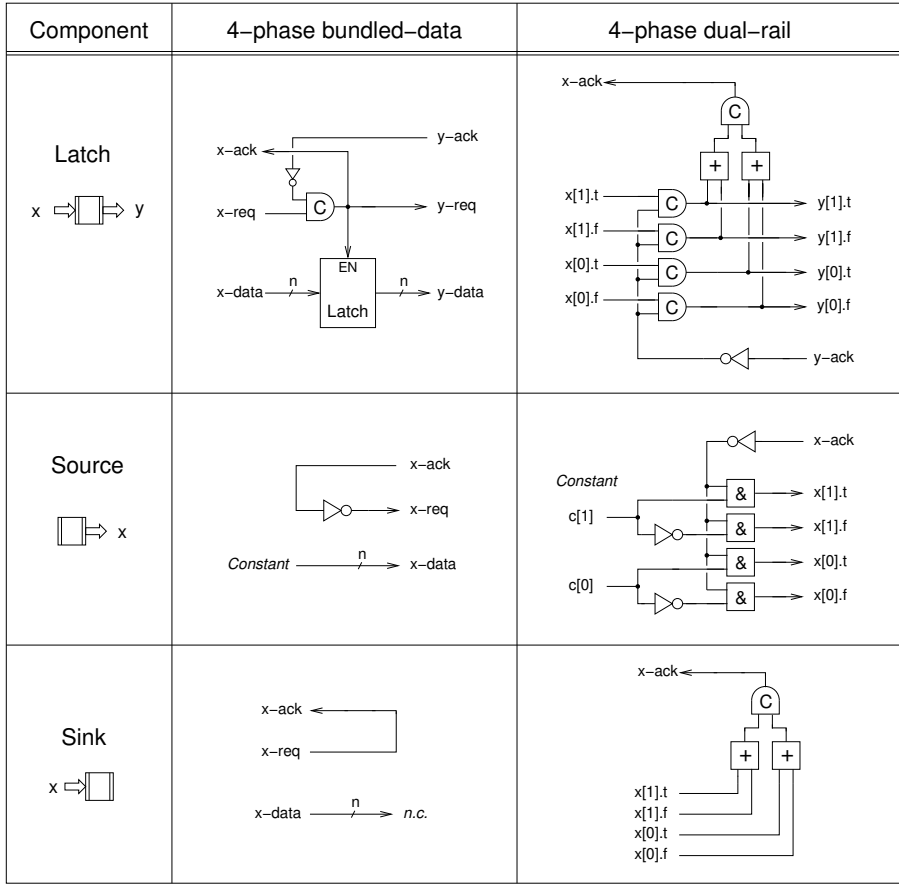


Figure 5.1: 4-phase bundled-data and 4-phase dual-rail implementations of the latch, the source, and the sink components – the active components that handshake with their neighbors.

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 51). Thus, the static spread for these FIFOs is $S = 2$.

Ideally, one would want to pack a valid token into every level-sensitive latch, and in chapter 10 we will address the design of 4-phase bundled-data handshake latches that have a smaller static spread.

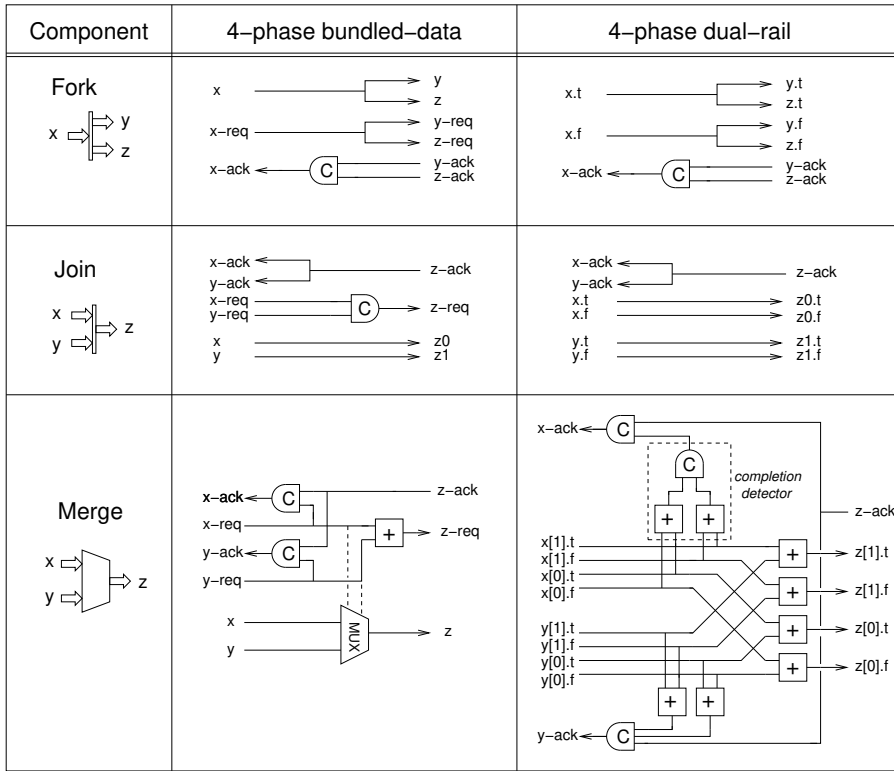


Figure 5.2: 4-phase bundled-data and 4-phase dual-rail implementations of the fork, join, and merge components.

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 5.2. 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 5.2 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 5.2. 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 OR'ed 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 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 suffices to reset the C-element in response to z_{ack} going low. This optimization is possible if asymmetric C-elements are available, figure 5.3. Similar arguments apply 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.

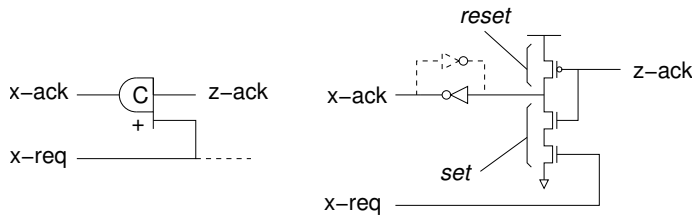


Figure 5.3: A possible implementation of the upper asymmetric C-element in the 4-phase bundled-data merge in figure 5.2.

The implementation of the 4-phase dual-rail merge is fairly similar. For illustration purposes, the figure shows an implementation with 2-bit channels. As request is encoded into the data signals, a completion detector is used to detect when an input channel has valid data. Acknowledge on an input channel is produced in response to an acknowledge on the output channel provided that the input channel has valid data. As shown for input channel y the two C-elements may be combined. Alternatively, when the channels have a large number of bits, it is possible to avoid C-elements with a corresponding high fan-in, by using the alternative completion detector design shown in figure 2.13 on page 21.

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 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. 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 MUX and DEMUX

Possible 4-phase bundled-data and 4-phase dual-rail implementations of the MUX and DEMUX components from figure 3.3 on page 33 are shown in figure 5.4.

Let us first recapitulate the function of the two components: a MUX synchronizes the control channel and the relevant input channel and relays the data and the handshaking of the selected input channel to the output channel. The other input channel is ignored (and may have a request pending). Similarly, a DEMUX synchronizes the control channel and the input channel and relay the data and the handshaking of the input channel to the selected output channel. The other output channel is silent.

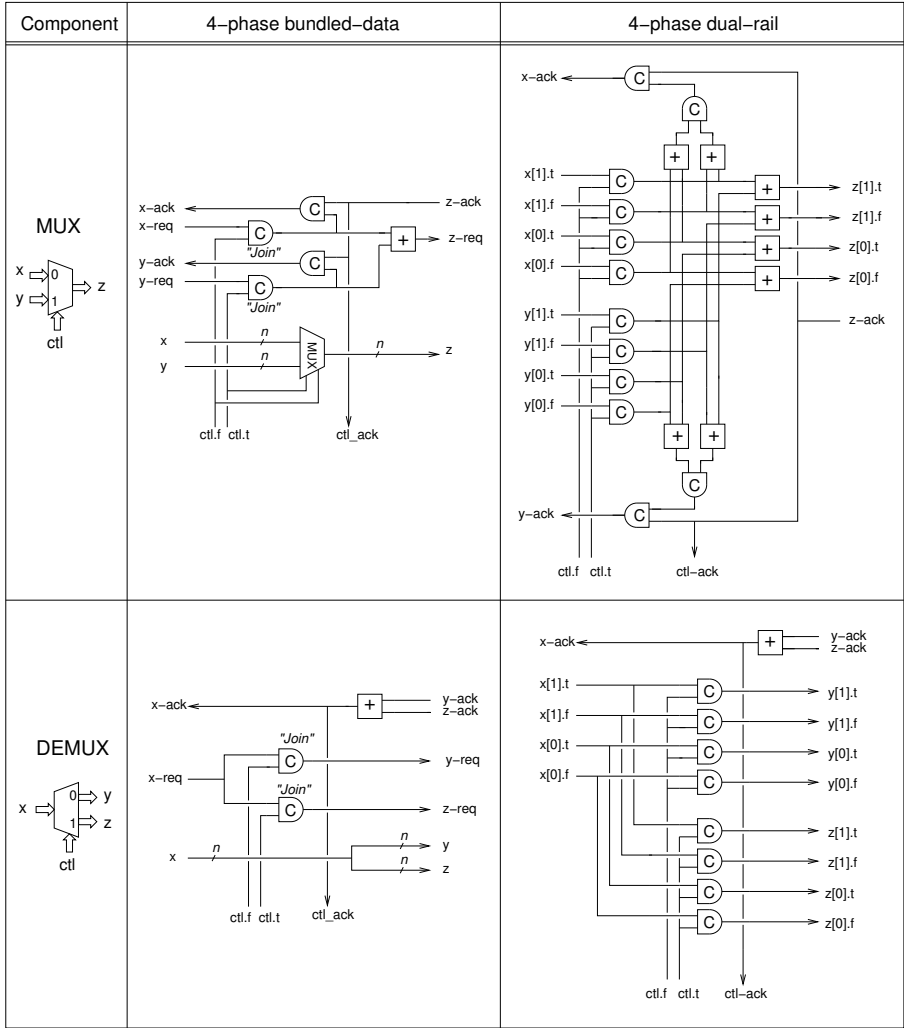


Figure 5.4: Implementation of the MUX and DEMUX handshake components. The input and output data channels x , y , and z use the 4-phase bundled-data protocol, and the control channel ctl uses the 4-phase dual-rail protocol (in order to simplify the design).

The schematics for the bundled-data designs show channels with an arbitrary number of bits, and the dual-rail designs show specific versions with 2-bit data, which is enough to illustrate the principles and to illustrate how to generalize to more bits. Note that the bundled-data designs both use a dual-rail control channel. The reason is that the one-hot behavior of the mutually exclusive signals $ctl.t$ and $ctl.f$ simplifies the designs. 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. Full 4-phase bundled-data implementations may be obtained by adding a 4-phase bundled-data to 4-phase dual-rail protocol conversion circuit to the control channel. 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.

The implementation of the handshake MUX shown in figure 5.4 can be understood as follows: The two inputs x and y are “conditioned” or joined with the control signals *ctl.t* and *ctl.f*, respectively. This enforces mutual exclusion among the two conditioned inputs. The rest of the circuit is then simply a Merge. For the 4-phase bundled-data MUX, this description only covers the control part. The datapath of the 4-phase bundled-data MUX is a conventional multiplexer selecting the proper input.

In a similar way, the handshake DEMUX can be understood as follows: The two outputs y and z are produced by “conditioning” the input x with the control signals *ctl.t* and *ctl.f* directly producing the output signals for the output channels. The acknowledge signals $y\text{-ack}$ and $z\text{-ack}$ are mutually exclusive and are merged using an OR-gate. The resulting signal drives $x\text{-ack}$ and *ctl-ack*.

At this point, we can take two paths moving forward. The most natural is to continue and dig deeper, as we will do in the subsequent section. Alternatively, a reader who wants to cover some ground quickly may jump to section 5.6.

5.4 Peephole optimizations

All the components presented so far can be classified as either driving the handshaking and being able to store tokens (latch, source and sink) or being transparent to handshaking (join, fork, merge, MUX and DEMUX). This clear separation is a simplification that helps in the design of data-flow circuits.

After a circuit using these components has been designed, the next step is to consider if combinations of components may be substituted by a smaller implementation – possibly a single component – with the same behavior.

Below we consider a number of such optimizations.

5.4.1 DEMUX with a sink on one output

As a first example of peephole optimization, we consider the shift register with parallel load presented in subsection 4.2.2. It contains demultiplexers where one output is connected to a sink, as shown in figure 5.5(a). The demux and the sink can be replaced by the circuit shown in figure 5.5(b), in some contexts known as a “conditional send” [82]. This circuit joins and consumes a token on the input channel and a token on the control channel, and depending on

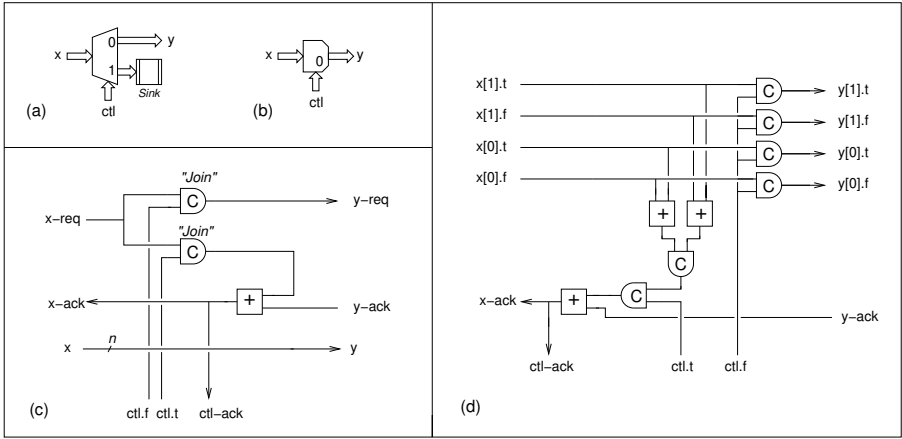


Figure 5.5: (a) A DEMUX with a sink on one of its outputs. (b) The schematic symbol for a “conditional send” component with the same behavior. (c) 4-phase bundled-data implementation. (d) 4-phase dual-rail implementation.

the value of the binary control, it either sends the input token to the output, or it consumes it, keeping the output channel silent.

A 4-phase bundled data implementation is shown in figure 5.5(c). It represents a minor optimization. For the 4-phase dual-rail implementation shown in figure 5.5(d), the gain is more substantial. Assuming N -bit wide input and output channels a direct implementation of a DEMUX and a sink costs $4N + 1$ C-elements and $N + 2$ OR-gates, whereas the implementation shown in figure 5.5(d) costs only $2N + 2$ C-elements and $N + 2$ OR-gates; in most cases close to a 50% reduction.

5.4.2 A DEMUX with latches on both outputs

Another example of a peephole optimization is a DEMUX with handshake latches on both outputs, as shown in figure 5.6(a). Fused implementations using 4-phase bundled-data and 4-phase dual-rail handshaking are shown in figure 5.6(b) and figure 5.6(c), respectively. The optimization uses a general idea: to avoid the C-element based conditioning of input signals (that we used in the DEMUX) and instead condition the acknowledge signal controlling the C-elements in the subsequent latch. In a 4-phase bundled-data design, there is no gain unless the two C-elements related to each of the two enable-latches are merged into 3-input C-elements. For the 4-phase bundled-data design, the gain is substantial: for an N bit channel, this optimization replaces $2N$ C-elements by a single C-element.

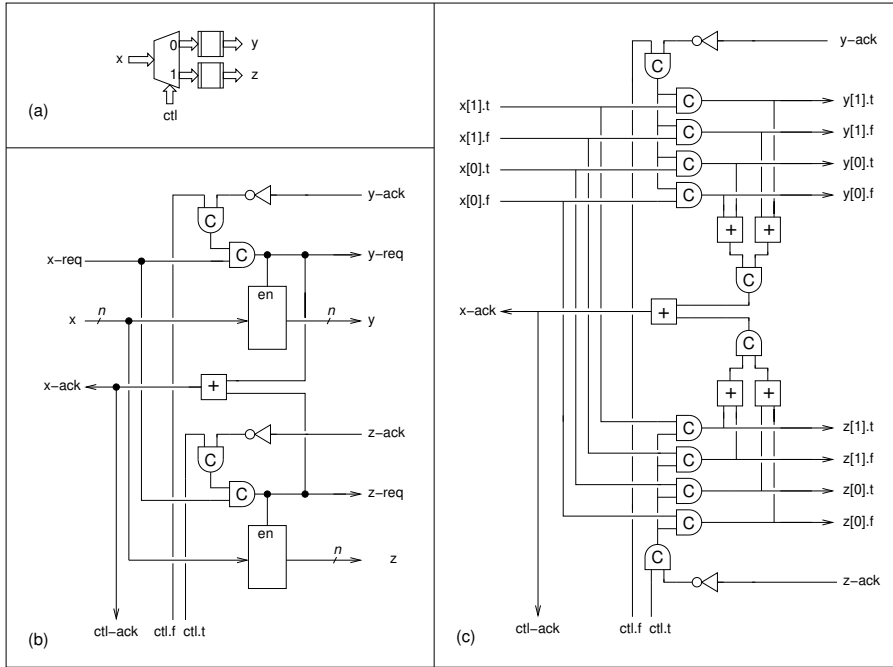


Figure 5.6: (a) A DEMUX with a latch on each of its output channels. (b) A fused 4-phase bundled-data implementation of the DEMUX. (c) An optimized 4-phase dual-rail implementation with the same behavior.

5.5 Memory cells

In this section, we explore the design of asynchronous memory cells. The designs presented can be used for small memories storing a few bits. For larger memories, efficient and smaller transistor-level designs are probably preferred. Such memories are a large topic in itself and are beyond the scope of this book.

5.5.1 Introduction

Figure 5.7 shows a schematic symbol of a memory cell. As seen, the memory has an input-data port, *Di*, an output data port, *Do*, and a control port, *Ctl*. The control channel is assumed to be a 4-phase 1-of-*n* channel. For a simple read-write memory, a 1-of-2 channel with signals *ctl.r*, *ctl.w* and acknowledge signal, *Ctl-ack* is used. A more advanced memory could use a 1-of-3 control channel and additionally offer a simultaneous read and write operation. If implementing read-before-write, the third control signal is called *Ctl.rw*, and if implementing write-before-read, it is called *Ctl.wr*.

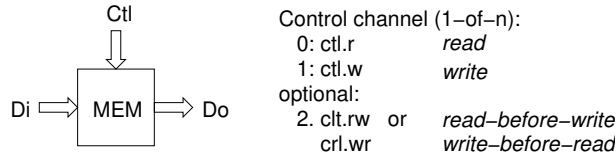


Figure 5.7: The interface of the asynchronous memory cell.

If the operation is a write, the input channel and the control channel are joined, and a token is consumed from both. The output channel is silent during a read operation. If the operation is a read, a token is consumed on the control channel, and a token is produced on the output channel. The input channel is ignored (and may have a pending token). Reading is an unbuffered operation; handshaking is relayed from the control port to the output port. If the operation is a combined read-write, the input channel and the control channel are joined, a token is consumed from both, and a token is emitted on the output port. A handshake on the control channel encloses this operation.

A memory cell can be read or written multiple times and in any order. This does not directly fit the data-flow paradigm where storage elements are written once and then read once in a repeating fashion. For this reason, a data-flow design of a memory cell needs a ring of handshake latches. Below we first present three data-flow memory designs. Following this, we explore an alternative and more area-efficient gate-level design that is based on a more conventional memory cell “wrapped” in circuitry that implements the necessary handshaking and completion detection.

5.5.2 A simple R-W data-flow memory cell

Figure 5.8 shows an implementation of a simple memory cell offering two operations: read and write.

The core of the design is a 3-stage ring. During a read, data is forked to the output and written back into the ring. During a write, the old value/token stored in the cell is silently consumed by the conditional send while the new data is being inputted.

5.5.3 A R-W-RW data-flow memory cell

A possible extension of the memory cell offers a third operation: read-before-write; i.e., a transaction where the old value is read and a new value is then written. The implementation is similar to the simple read-write memory cell from figure 5.8 except for a slight extension of the control path. The control channel is now 1-of-3. After the fork, the two 1-of-3 channels are converted to 1-of-2 channels by OR-ing two of the three mutually exclusive control signals, as indicated in figure 5.9.

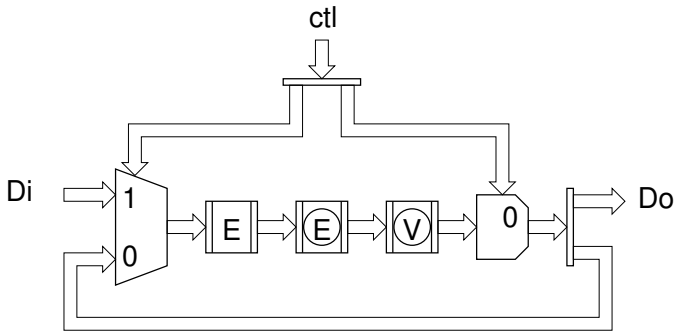


Figure 5.8: Implementation of the simple R-W data-flow memory cell.

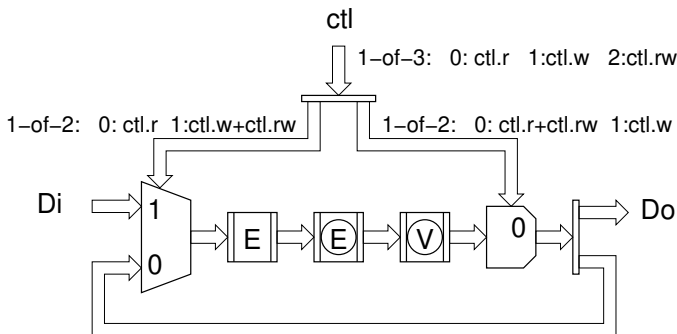


Figure 5.9: Implementation of a R-W-RW data-flow memory cell.

5.5.4 A R-W-WR data-flow memory cell

Another extension of the simple memory cell from figure 5.8 is a write-before-read transaction. This design is more elaborate. It can be implemented by first writing and then reading (all enclosed by a handshake on the control channel), but this is slow and complex to implement. A preferred alternative is to forward the write data directly from the input port to the output port at the same time as it is written into the cell. An implementation of this idea is shown in figure 5.10.

5.5.5 A more efficient R-W memory design

More efficient memory designs use a conventional bistable memory cell and “wraps” the memory cells in control and handshake circuitry. The design we present in the following uses a gate-level memory cell from [153, Sec. 8.5] and adapts it to the data-flow context. Figure 5.11 shows an implementation of

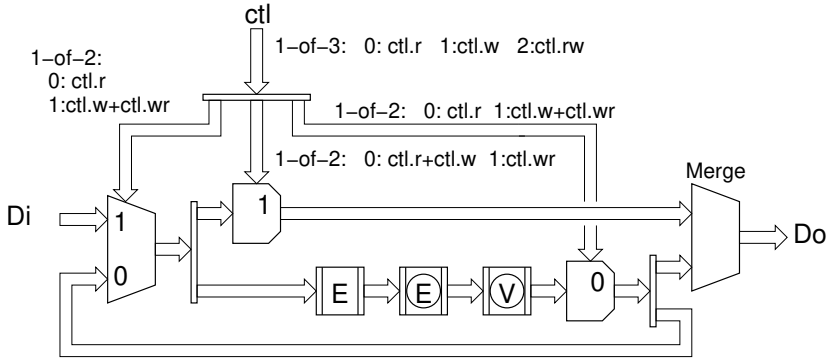


Figure 5.10: Implementation of a R-W-WR data-flow memory cell.

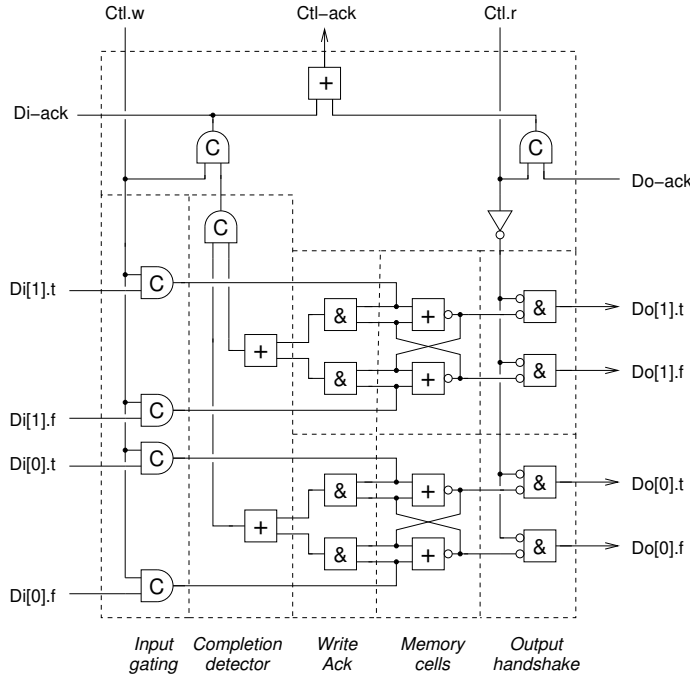


Figure 5.11: A dual-rail 4-phase implementation of a read-write memory using memory cells based on a pair of cross-coupled NOR-gates.

a data-flow read-write memory using this cell. For illustration purposes, the figure shows a memory storing one 2-bit word.

One bit of information is stored in a pair of cross-coupled NOR-gates. Reading involves “adding” handshaking, similar to what was used in a source (anding the dual-rail output signals with the *Ctl.r* signal). Writing to the cell is performed by an active high signal on the input of one of the NOR-gates, and completion of the write operation is indicated by the feedback signal from the other NOR-gate also going high. This is detected by the AND-gates (in the figure marked “write ack”). A completion detector then detects when all cells in a word have been written.

A memory with multiple words can be obtained by adding address signals to the control port. The address must first be decoded to a one-hot form, and then each one-hot address signal is gated with *Ctl.r* and *Ctl.w* signals producing mutually exclusive *Ctl.r* and *Ctl.w* for each word. In addition, a merge is needed on the data outputs, and the data inputs must be connected to all words in the memory (before the input gating).

The design can be extended with a third operation, read-after-write, or write-after-read, by extending the control circuitry with functionality that explicitly sequences the read and write operations in the required order.

5.6 Function blocks – The basics

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

5.6.1 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 purely functional behavior.

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 neighboring latches. This transparency to handshaking is what 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 as is the case when using dual-rail or other forms of delay-insensitive signaling.

The most general scenario is where a function block receives its operands on separate channels and produces its results on separate channels, as seen in figure 5.12. 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

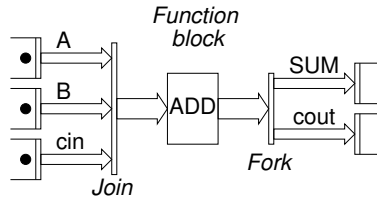


Figure 5.12: A function block whose operands and results are provided on separate channels requires a join of the input and a fork on the output.

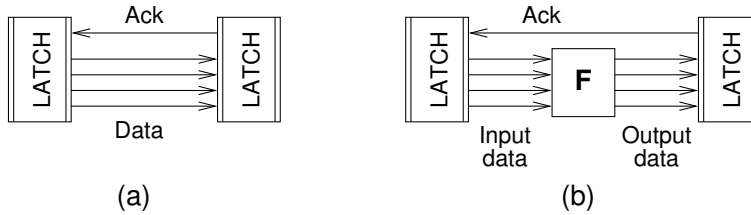


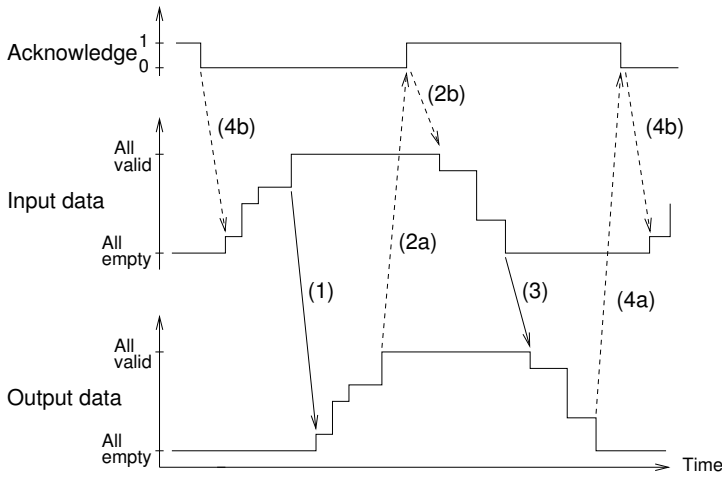
Figure 5.13: (a) Two latches connected directly by a handshake channel and (b) the same situation with a function block added between the latches. The handshaking, as seen by the latches in the two situations, should be the same, i.e., the function block must be designed such that it is transparent to the handshaking.

they can be integrated into the function block circuitry. In the following we 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 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 [108].

5.6.2 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 5.13(a) shows two handshake latches connected directly, and figure 5.13(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.



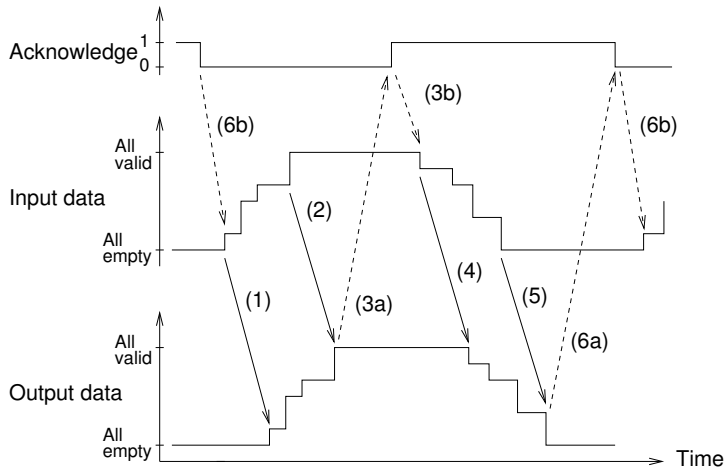
- | | | |
|------------------------------------|------------|---------------------------------|
| (1) “All inputs become defined” | $\searrow$ | “Some outputs become defined” |
| (2) “All outputs become defined” | $\searrow$ | “Some inputs become undefined” |
| (3) “All inputs become undefined” | $\searrow$ | “Some outputs become undefined” |
| (4) “All outputs become undefined” | $\searrow$ | “Some inputs become defined” |

Figure 5.14: Signal traces and event orderings for a strongly indicating function block.

A function block is obviously not allowed to produce a request on its output before receiving a request on its input. 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. (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 signaling that can be observed on the channel between the two latches in figure 5.13(a) was illustrated in figure 2.3 on page 13. We can illustrate the handshaking for the situation in figure 5.13(b) in a similar way.

- A function block is *strongly indicating*, as illustrated in figure 5.14, 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 5.15, if (1)



- | | | |
|------------------------------------|---|---------------------------------|
| (1) “Some inputs become defined” | ⌊ | “Some outputs become defined” |
| (2) “All inputs become defined” | ⌊ | “All outputs become defined” |
| (3) “All outputs become defined” | ⌊ | “Some inputs become undefined” |
| (4) “Some inputs become undefined” | ⌊ | “Some outputs become undefined” |
| (5) “All inputs become undefined” | ⌊ | “All outputs become undefined” |
| (6) “All outputs become undefined” | ⌊ | “Some inputs become defined” |

Figure 5.15: Signal traces and event orderings for a weakly indicating function block.

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 behavior is identical to Seitz’s weak conditions in [136]. In [136] 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 behavior 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.

5.6.3 Review of ripple-carry addition

Figure 5.16 illustrates the implementation principle of a ripple-carry adder.

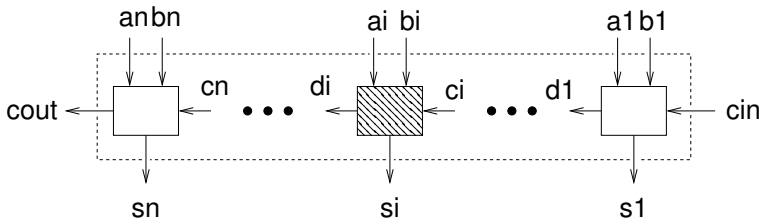


Figure 5.16: A ripple-carry adder. The carry output of one stage d_i is connected to the carry input of the next stage c_{i+1} .

A 1-bit full adder stage implements:

$$s = a \oplus b \oplus c \quad (5.1)$$

$$d = ab + ac + bc \quad (5.2)$$

In many implementations inputs a and b are recoded as:

$$p = a \oplus b \quad (\text{“propagate” carry}) \quad (5.3)$$

$$g = ab \quad (\text{“generate” carry}) \quad (5.4)$$

$$k = \bar{a}\bar{b} \quad (\text{“kill” carry}) \quad (5.5)$$

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

$$s = p \oplus c \quad (5.6)$$

$$d = g + pc \quad \text{or alternatively} \quad (5.7)$$

$$\bar{d} = k + p\bar{c} \quad (5.8)$$

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:

$$\begin{aligned} (d.t, d.f) = (0,0) &= \text{Empty} && \text{"The carry has not been computed yet"} \\ &&& \text{(possibly because it depends on } c) \\ (d.t, d.f) = (1,0) &= \text{True} && \text{"The carry is 1"} \\ (d.t, d.f) = (0,1) &= \text{False} && \text{"The carry is 0"} \end{aligned}$$

Consequently, 1-bit adder can 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 [51]. The same idea is explained in [68, pp. 75-78] and in [136].

5.7 Bundled-data function blocks

5.7.1 Using matched delays

A bundled-data implementation of the adder in figure 5.16 is shown in figure 5.17. 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 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

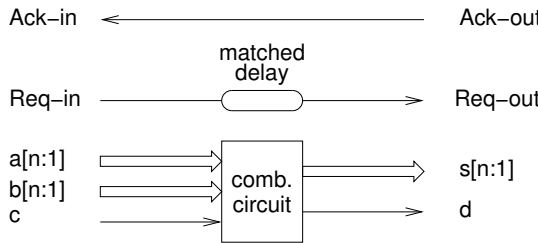


Figure 5.17: A 4-phase bundled data implementation of the N -bit handshake adder from figure 5.16.

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 an identical layout but with weaker transistors. In an automatic standard-cell 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 [20], the delay elements consumed 10 % of the total power.

5.7.2 Delay selection

In [117], Nowick proposed a scheme called “speculative completion.” The basic principle is illustrated in figure 5.18. 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 5.16) may be used for the estimation. As an example of the idea, consider a 16-bit adder. If $p_8 = 0$, the 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

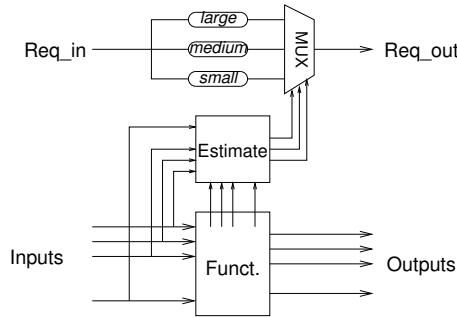


Figure 5.18: The basic idea of “speculative completion.”

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 referred to [117, 118].

5.8 Dual-rail function blocks

5.8.1 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 and EXOR. 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 of 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 5.19(b)-(c) shows the implementation of a 1-bit full adder. We will discuss the circuit in figure 5.19(d) shortly.

The PLA-like structure of the circuit in figure 5.19(c) illustrates a general principle for implementing arbitrary Boolean functions. In [145], 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

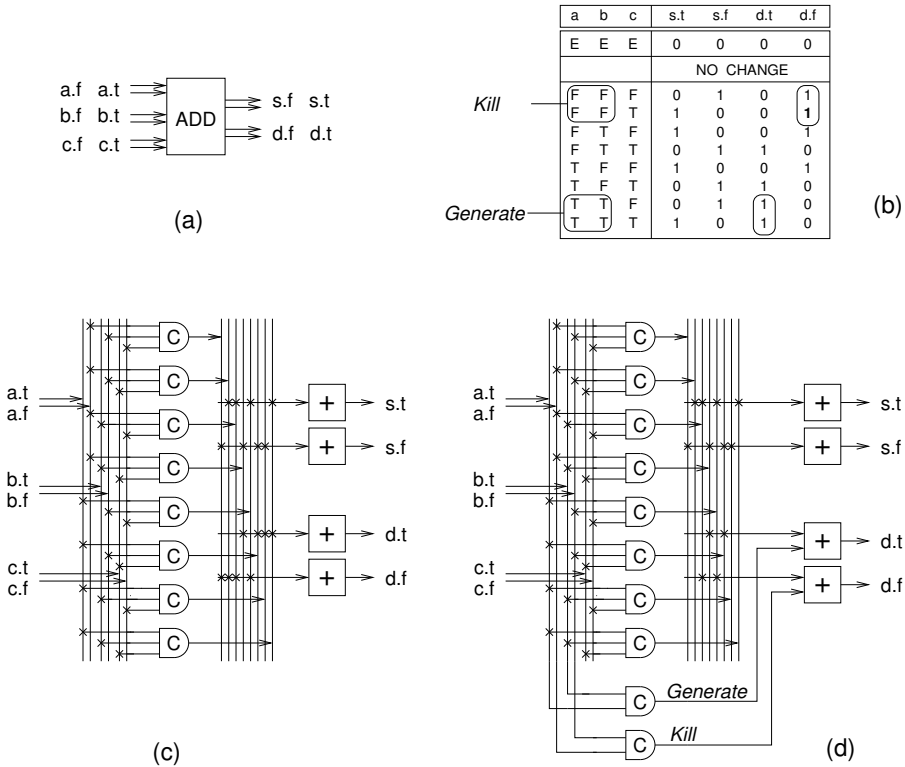


Figure 5.19: A 4-phase dual-rail full-adder: (a) Symbol, (b) truth table, (c) DIMS implementation, and (d) an optimization that makes the full adder weakly indicating.

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 [104, 103]. While [104] develops the fundamental theorem for the design of speed-independent circuits, [103] is a more practical introduction, including a design example: a bit-serial multiplier using latches and gates as explained above.

Referring to section 5.6.2, 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 5.19(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 5.19(c) and 5.19(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 5.8.4 we will introduce an elegant design that propagates empty values in constant time (with the latency of 2 full adder cells).

5.8.2 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 5.20. By using arbitrary m -of- n threshold gates with hysteresis – an idea proposed by Theseus Logic, Inc., [41] – it is possible to reduce the implementation complexity. An m -of- n threshold gate with hysteresis sets its output high when any m inputs have gone high, and it sets 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 significant similarities to the circuit design styles presented in [103, 139, 145, 108]. Figure 5.20 shows that OR 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 5.21 shows the implementation of a dual-rail full adder using NCL threshold gates. The circuit is weakly indicating.

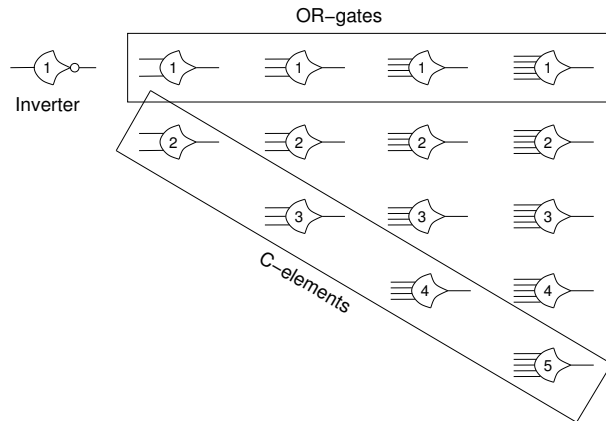


Figure 5.20: NCL gates: m -of- n threshold gates with hysteresis ($1 \leq m \leq n$).

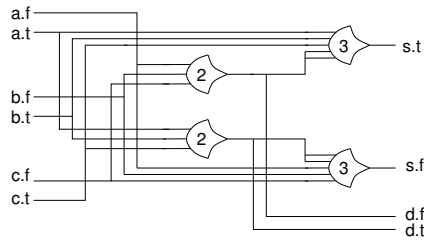


Figure 5.21: A full adder using NCL gates.

5.8.3 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 5.22 [119, 58].

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 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 5.23 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

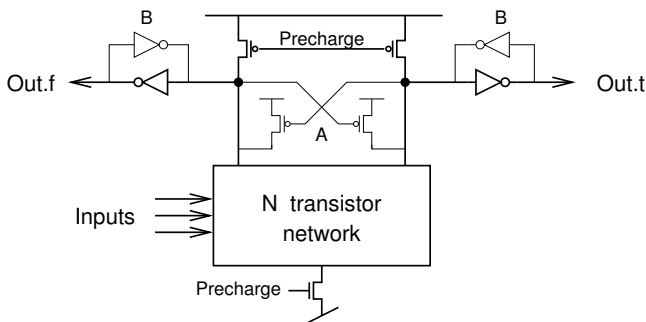


Figure 5.22: A precharged differential CMOS combinatorial circuit. By adding the cross-coupled p-type transistors labeled “A” or the (weak) feedback-inverters labeled “B,” the circuit becomes (pseudo)static.

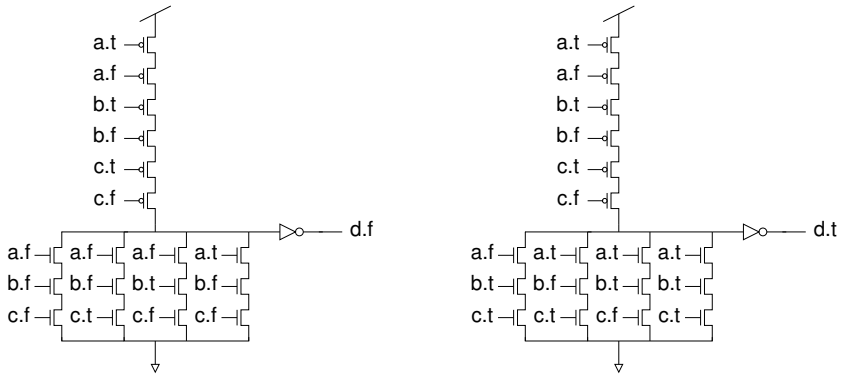


Figure 5.23: Transistor-level implementation of the carry signal for the strongly indicating full adder from figure 5.19(c).

networks, but in this particular case, it has not been done in order to better illustrate the relationship between the transistor implementation and the gate implementation in figure 5.19(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.

5.8.4 Martin’s adder

In [93], 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 5.23, 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 5.15 only require that one output remains 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 [93], 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 5.24(a)

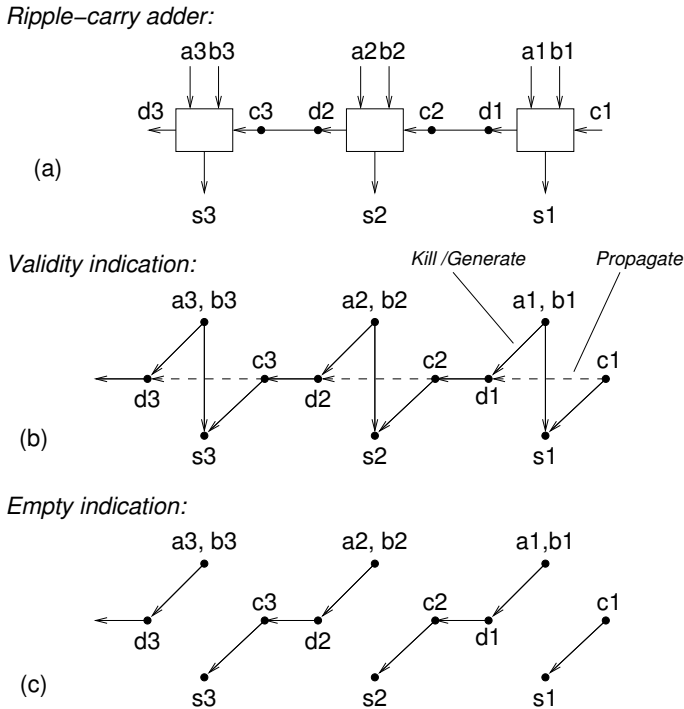


Figure 5.24: (a) A 3-stage ripple-carry adder and graphs illustrating how valid data (b) and empty data (c) propagate through the circuit (Martin [93]).

shows three full adder stages of a ripple-carry adder, and figures 5.24(b) and 5.24(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.

The corresponding transistor implementation of a full adder is shown in figure 5.25. 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 5.24, are a very useful technique for understanding and designing circuits with low latency and the weakest possible indication.

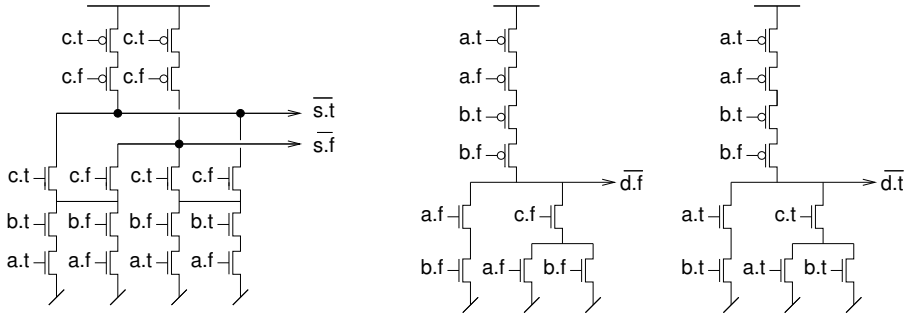


Figure 5.25: The CMOS transistor implementation of Martin's adder [93, Fig. 3].

5.9 Hybrid function blocks

The final adder we 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 5.26. Each full adder is composed of a carry circuit and a sum circuit. Figure 5.27(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 5.26.

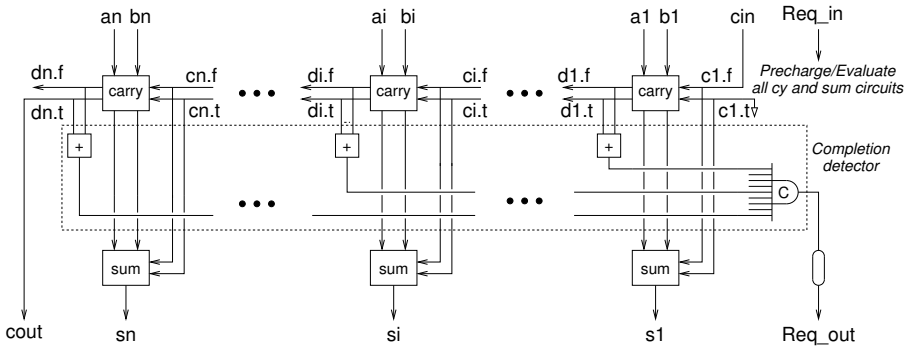


Figure 5.26: Block diagram of a hybrid adder with 4-phase bundled-data input and output channels and with an internal dual-rail carry chain.

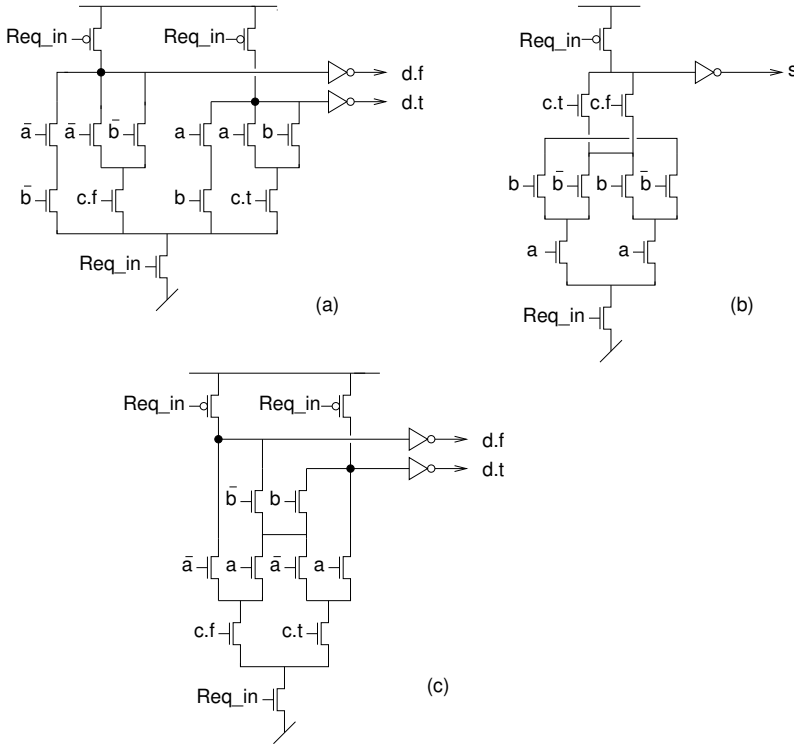


Figure 5.27: The CMOS transistor implementation of a full adder for the hybrid adder in figure 5.26: (a) a weakly indicating carry circuit, (b) the sum circuit, and (c) a strongly indicating carry circuit.

The size and latency of the completion detector in figure 5.26 grow with the size of the adder. 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 [111]. Following the naming convention established in figure 5.16 on page 77, we could make, for example, 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 $i = 3, 6, 9, \dots$ 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 [111] 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 the system performance. Hence, 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 [113] 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.

5.10 Mutual exclusion and arbitration

5.10.1 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). A possible implementation is shown in figure 5.28.

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.

A problem arises when both input signals are asserted at the same time. The two cross-coupled gates in the first stage of the mutex is a bi-stable circuit

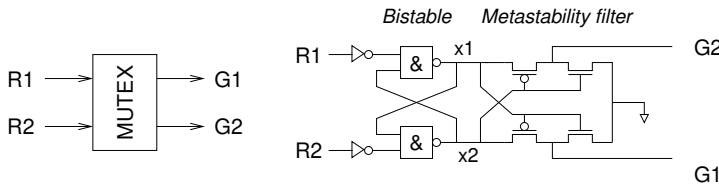


Figure 5.28: The mutual exclusion element: symbol and possible implementation.

that now receives requests to enter each of its two stable states at the same time. This may cause the circuit to produce non-digital output signals $x1$ and $x2$ hovering halfway between logic 0 and logic 1. This is a metastable state. In general, the circuit will recover to one of the two valid states quickly, but it is important to note that there is no upper limit on how long it may take. The purpose of the second stage, the metastability filter, is to prevent the undefined values on $x1$ and $x2$ from propagating to the outputs. It does this by driving signals $G1$ and $G2$ low until the bistable enters a valid digital state.

We will discuss metastability in much greater detail in chapter 8. The above should suffice in this chapter where focus is on implementation of the set of handshake components.

5.10.2 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 5.29.

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

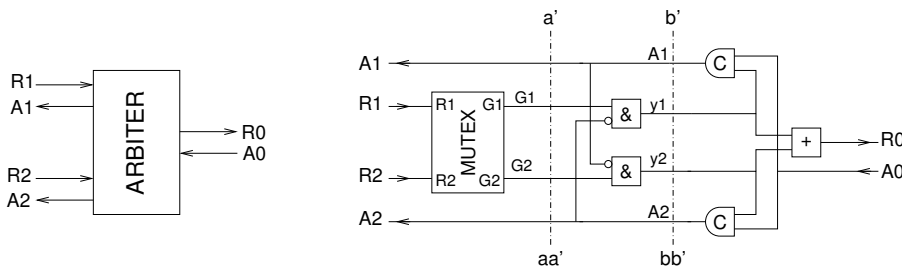


Figure 5.29: The handshake arbiter: symbol and possible implementation.

$y1$ 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 5.2 on page 63. 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 $y2$.

5.11 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. It 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. We start by presenting some of the basic assumptions and characteristics of the various design methods and give pointers to relevant literature. Then we 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, Petrifly.

A good starting point for further reading is a book by Myers [107]. 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.

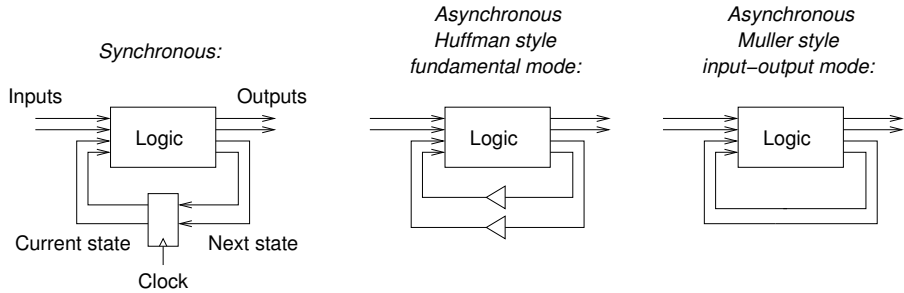


Figure 6.1: (a) A synchronous sequential circuit. (b) A Huffman style asynchronous sequential circuit with buffers in the feedback path, and (c) a Muller style asynchronous sequential circuit with wires in the feedback path.

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 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

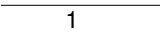
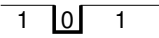
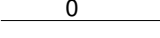
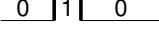
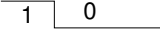
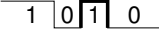
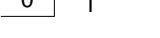
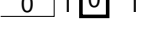
	<i>Desired signal</i>	<i>Actual signal</i>
Static-1 hazard:		
Static-0 hazard:		
Dynamic-10 hazard:		
Dynamic-01 hazard:		

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

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 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.

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 flavors, depending on how the delay time parameter is specified. The simplest is a *fixed delay*, where the delay is 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 [63, 64].

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 [104, 103]. 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

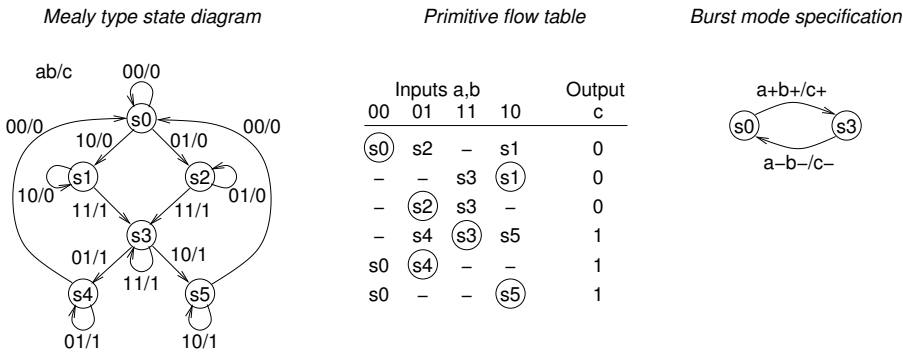


Figure 6.3: Some alternative specifications of a Muller C-element: a Mealy state diagram, a primitive flow table, and a burst-mode state diagram.

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 reaches a stable state where the environment is again allowed to make a single input change. Another way of expressing this behavior is to say that the circuit starts in a stable state (which is defined to be a state that persists until an input signal changes). In response to an input signal change, the circuit steps 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 [79], [151], or [107] and to specify and synthesize a C-element. The following gives a flavor 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* [34, 24]. When in a stable state, a burst-mode circuit waits 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 [43, 172]. These tools are available in the public domain.

6.2 Signal transition graphs

The rest of this chapter is 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 explain in the following consists of: (1) Capturing the behavior 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 [2, 128, 106] 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

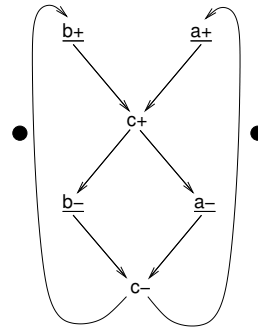
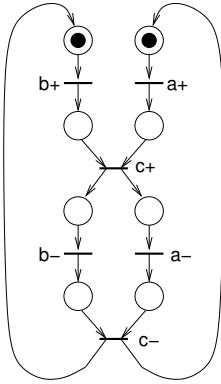
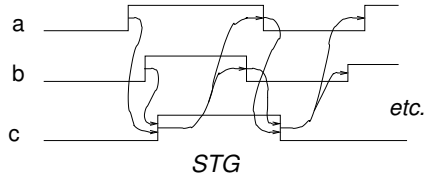
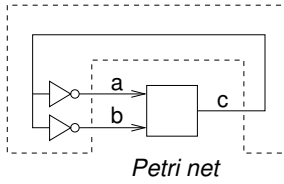
*C-element and dummy environment**Timing diagram*

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

make the analysis for certain properties practical. The STGs we 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 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 behavior 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 does 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

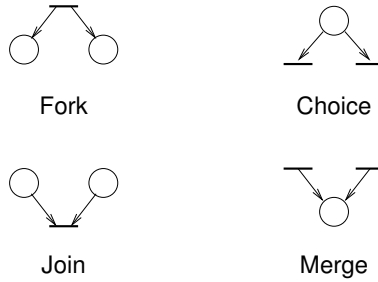


Figure 6.5: Petri net fragments for fork, join, free choice and merge constructs.

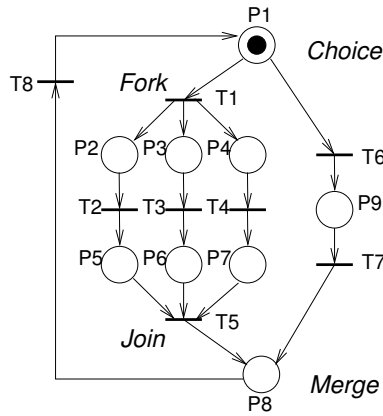


Figure 6.6: An example Petri net that illustrates the use of fork, join, free choice, and merge.

the choice place must represent mutually exclusive input signal transitions. This requirement is quite natural; we 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 constructs. The example system will either perform transitions $T6$ 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 33. For this, we 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

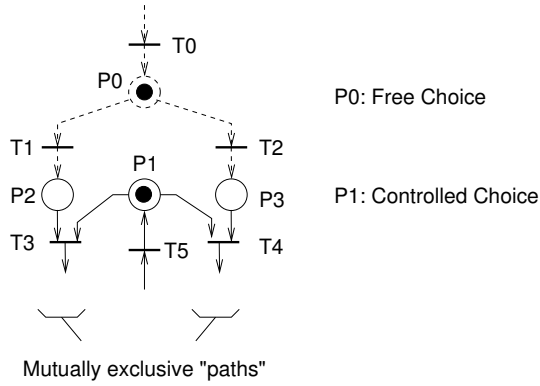


Figure 6.7: A Petri net fragment including a controlled choice.

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 6.8 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 33. 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 an 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

the CAD tools that has been developed – for example Petrify that we introduce later.

6.3.1 Example 1: a C-element

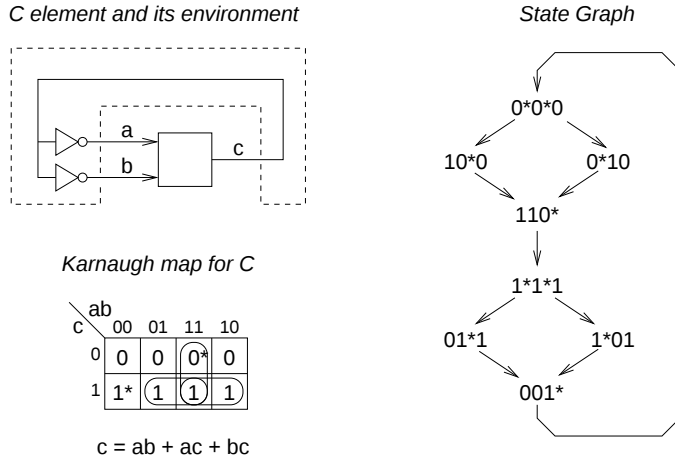


Figure 6.9: State graph and Boolean equation for the C-element STG from figure 6.4.

Figure 6.9 shows the state graph corresponding to the STG specification in figure 6.4 on page 97. 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). 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 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 revert 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 [10, 9].

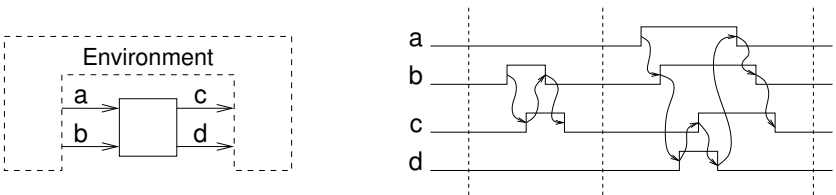


Figure 6.10: The example circuit from [10, 9].

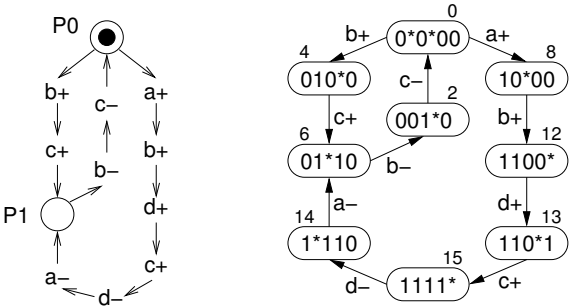


Figure 6.11: The STG specification and the corresponding state graph.

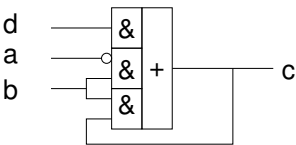
Karnaugh map:

a b	c d			
	00	01	11	10
00	0	x	x	F
01	R	x	x	1
11	0	R	1	1
10	0	x	x	x

Boolean equation for c:

$$c = d + \bar{a}b + bc$$

An atomic complex gate:



Using simple gates:

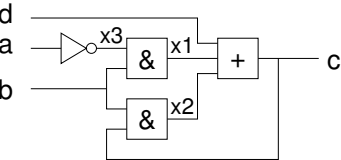


Figure 6.12: The Karnaugh map, the Boolean equation, and two alternative gate-level implementations of output signal c .

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 behaviors, 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 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 98, 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 [15, 80].

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.

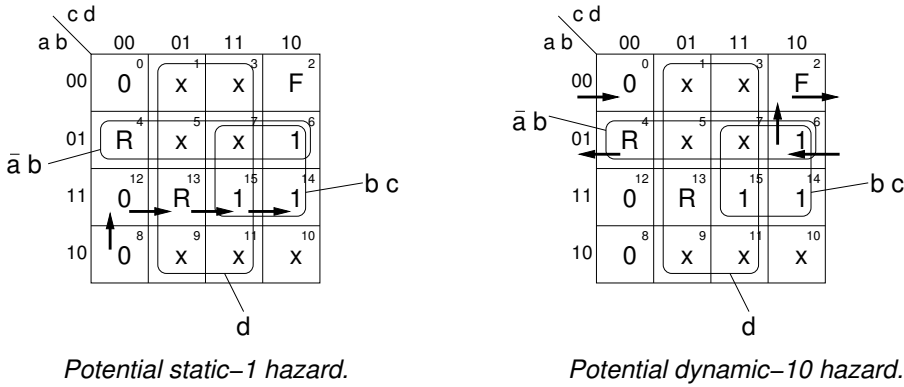


Figure 6.13: The Karnaugh maps for output signal c showing state sequences that may lead to hazards.

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 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 x_2 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 x_1) and the OR gate relaying $b+$ into $c+$ and $b-$ into $c-$. However, after the $c+$ transition, the lower AND gate, x_2 , 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 (x_2) fires, it will later become excited (F) in response to $c-$. The net effect of this is that the lower AND gate (x_2) may superimpose a 0-1-0 pulse onto the c output after the intended $c-$ transition has occurred.

In the above, we did not consider the inverter with input signal a and output signal x_3 . Since a is not an input to any other gate, this decomposition is SI.

In summary, both types of hazards 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,

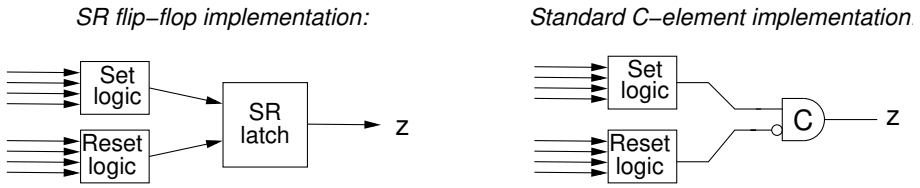


Figure 6.14: Possible implementation templates using (simple) state holding elements.

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 leads 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 discuss alternative and more elaborate implementations, but for the following discussion, the basic topologies in figure 6.14 suffice.

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:

$$z = \text{“Set”} + z \cdot \overline{\text{“Reset”}} \quad (6.1)$$

For signal c in the above example (figure 6.12 on page 103), we would get the following *set* and *reset functions*: $c_{set} = d + \bar{a}b$ and $c_{reset} = \bar{b}$ (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:

- $\text{ER}(z+)$ denotes a region of states where $z = R = 0^*$
- $\text{ER}(z-)$ 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:

- $\text{QR}(z+)$ denotes a region of states where $z = 1$
- $\text{QR}(z-)$ 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 $\text{ER}(z+)$ regions
- may contain states from the $\text{QR}(z+)$ regions
- may contain states not reachable by the circuit

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

- must contain all states in the $\text{ER}(z-)$ regions
- may contain states from the $\text{QR}(z-)$ 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.

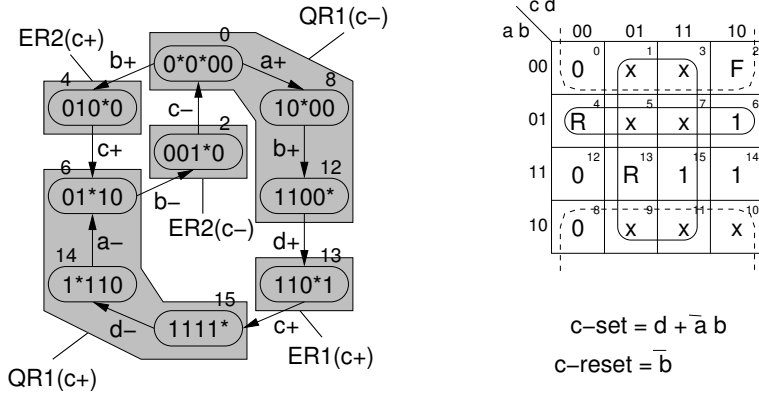


Figure 6.15: Excitation and quiescent regions in the state diagram for signal c in the example circuit from figure 6.10, and the corresponding derivation of equations for the set and reset functions.

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.

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 $\bar{a}b$ 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 creates a pulse on the c_{set} signal that causes 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 $\bar{a}b$ (that covers states 4, 5, 7, and 6) is reduced to include only states 4 and 5, corresponding to $c_{set} = d + \bar{a}b\bar{c}$, we avoid the problem.

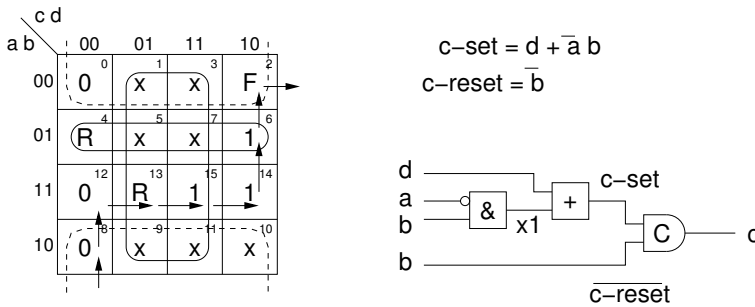


Figure 6.16: Implementation of c using a standard C-element and simple gates, along with the Karnaugh map from which the set and reset functions were derived.

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.

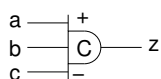
6.4.5 Circuit topologies using state-holding elements

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 $z_{reset} = \bar{b}\bar{c}$ 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

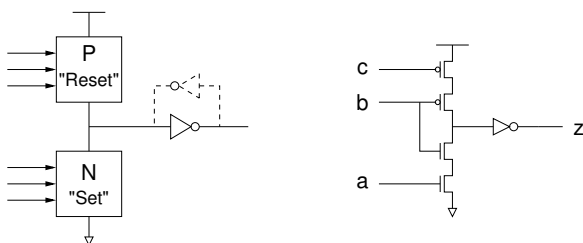
Gate level symbol:



$$z\text{-set} = a \cdot b$$

$$z\text{-reset} = \bar{b} \cdot \bar{c}$$

Dynamic (and pseudostatic) CMOS implementation:



Static CMOS implementation:

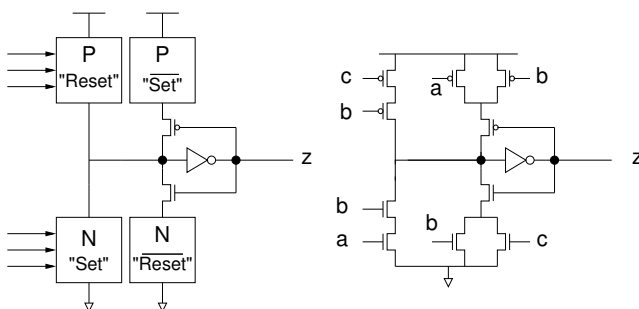


Figure 6.17: A generalized C-element: gate-level symbol, and some CMOS transistor implementations.

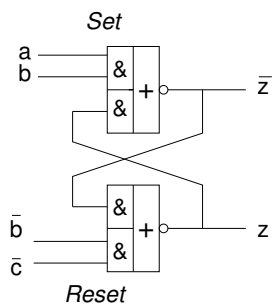


Figure 6.18: An SR implementation based on two complex And-Or-Invert gates.

the desired signal z and its complement $\bar{z}$, and during transitions, it *never* produces $(z, \bar{z}) = (1, 1)$. Again, the example is a circuit where $z_{set} = ab$ and $z_{reset} = \bar{b}\bar{c}$.

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 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, the initialization of asynchronous circuits based on handshake components may also be achieved by an implicit approach that exploits the function of the circuit to “propagate” initial signal values into the circuit. This is called self-initialization, [153].

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 [104], MIT [21, 22], Stanford [9], IMEC [160, 171], St. Petersburg Electrical Engineering Institute [161], and the multinational group of researchers who have developed the Petrify tool [27] that we 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 we 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 [107].

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

1. Specify the desired behavior of the circuit and its (dummy) environment using an STG.
2. Check that the STG satisfies properties 1-5 on page 98: 1-bounded, consistent state assignment, liveness, only input free choice, 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 98: 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 the result of significant international research collaboration [27, 26]. It is available from <http://www.cs.upc.es/~jordicf/petrify/>.

Petrify is a typical command-line UNIX program with many options and switches. STGs are specified in ascii text-files, that can subsequently be displayed graphically.

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 provide 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) was developed at the Technical University of Denmark. It is available from VSTGL <http://vstgl.sourceforge.net/>, but by now, it is dated and may not install easily.

More recently the, researchesr at University of Newcastle has developed a tool called WorkCraft that is an elaborate framework for working with a range of graph-based models used for synthesizing and analyzing asynchronous circuits as well as for analyzing models of concurrent systems in a broader and more general context. WorkCraft has an easy to use graphical user interface, and Petrify is embedded within WorkCraft along with a related synthesis tool called MPSAT and many other useful plug-ins. WorkCraft is available from <https://workcraft.org>

Petrify (and MPSAT) can solve CSC violations by inserting state variables, and 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 not be combined with the **-cg** and **-gc** options.

In the following section, we will show some example circuits drawn from the previous chapters of the book. The STGs and the solutions are produced using VSTGL and Petrify. The reader is encouraged to redo some of the examples using WorkCraft.

6.8 Design examples using Petrify

In the following we 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 33, and (c) a control circuit for the 4-phase bundled-data implementation of the MUX from figure 3.3 on page 33. For all of the examples, we assume push channels only.

6.8.1 Example 2 revisited

As a first example, we 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 Petrify may visualize it, 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.

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
```

```
....
```

```
# 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
```

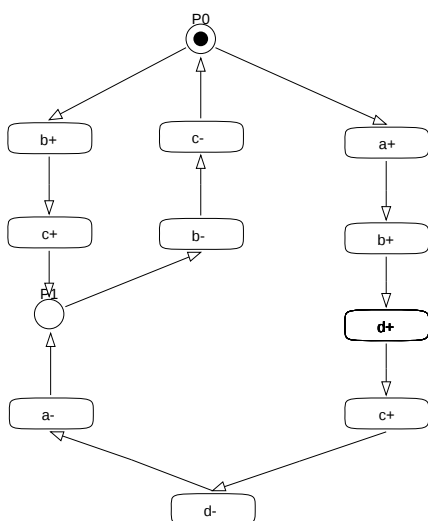


Figure 6.19: The STG of example 2 as it is entered into VSTGL.

```
# 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';
```

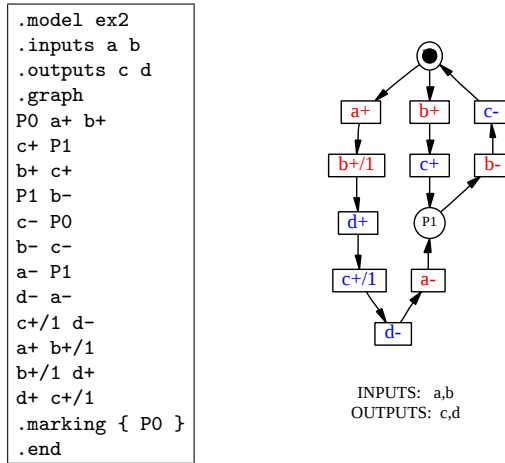


Figure 6.20: The textual description of the STG for example 2 and the drawing of the STG that is produced by Petrify.

```

[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 106

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 106.

6.8.2 A 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 behavior of the latch controller. Initially, Lt is low, and the latch is transparent, and when new input data arrives, they 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. 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
```

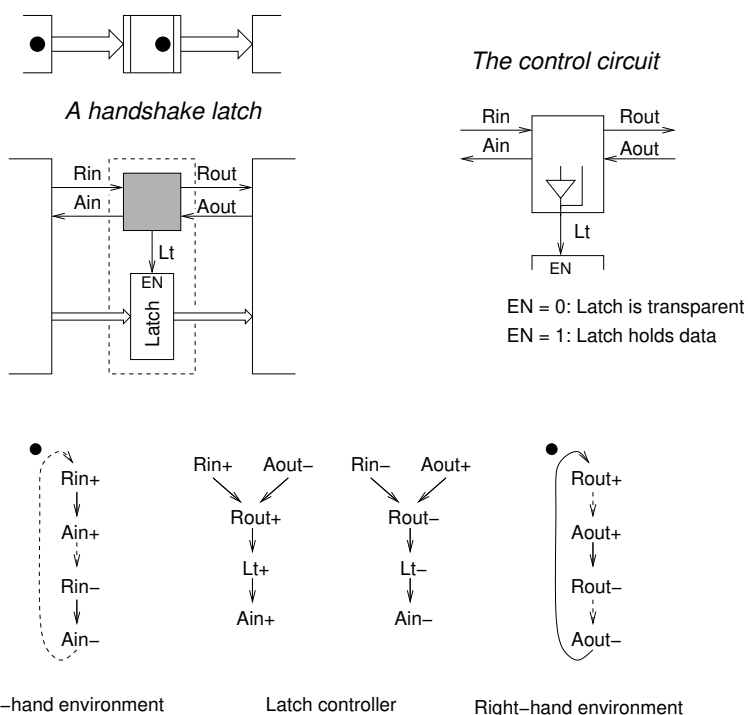


Figure 6.21: A 4-phase bundled-data handshake latch and some STG fragments that capture ideas about its behavior.

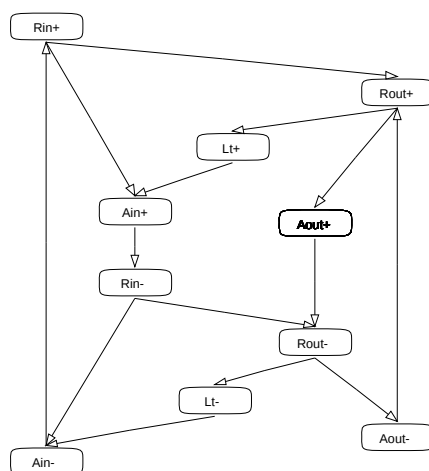


Figure 6.22: The resulting STG for the latch controller (as input to VSTGL).

```
> 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 A 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 33. It also shows how the handshake MUX can be implemented by a “regular” combinational circuit multiplexer and a control circuit. Below we design a speed-independent control circuit for a 4-phase bundled-data MUX.

The MUX has three input channels, and we *must* assume they are connected to three *independent* dummy environments. The dots remind us that

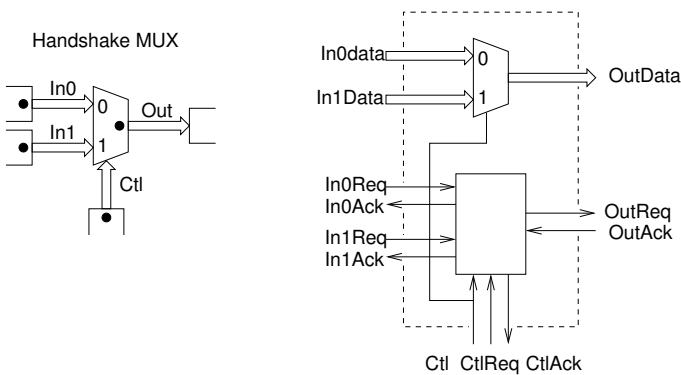


Figure 6.23: The handshake MUX and the structure of a 4-phase bundled-data implementation.

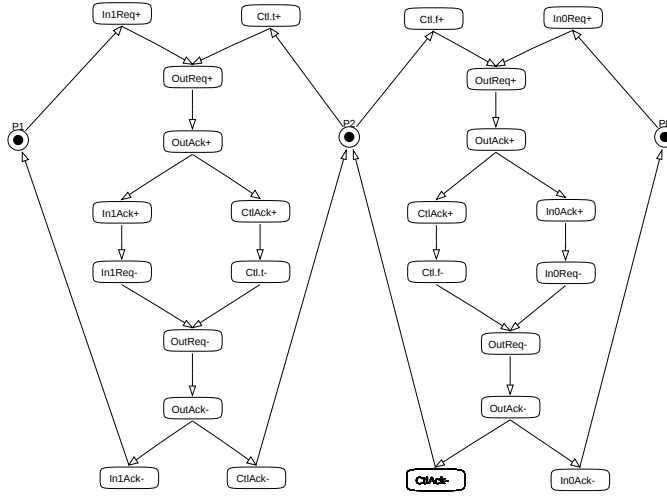


Figure 6.24: The STG specification of the control circuit for a 4-phase bundled-data MUX using a 4-phase dual-rail control channel. Combined with the STG fragment for a bundled-data (control) channel, the resulting STG for an all 4-phase dual-rail MUX is obtained (figure 6.25).

the channels are push channels. When specifying the behavior of the MUX control circuit and its (dummy) environment, it is essential to keep this in mind. A typical error when drawing STGs is to specify an environment with a more limited behavior than the real environment. For each of the three input channels, the STG has cycles involving $(Req+; Ack+; Req-; Ack-; \text{etc.})$, 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 101, 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.

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.

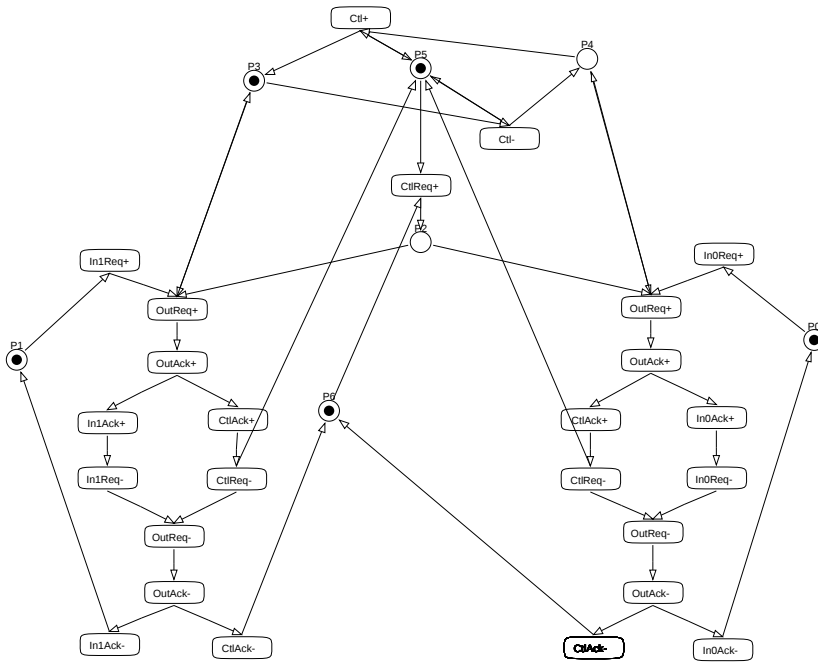


Figure 6.25: The final STG specification of the control circuit for the 4-phase bundled-data MUX. All channels, including the control channel, are 4-phase bundled-data.

```
> 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.

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);

```

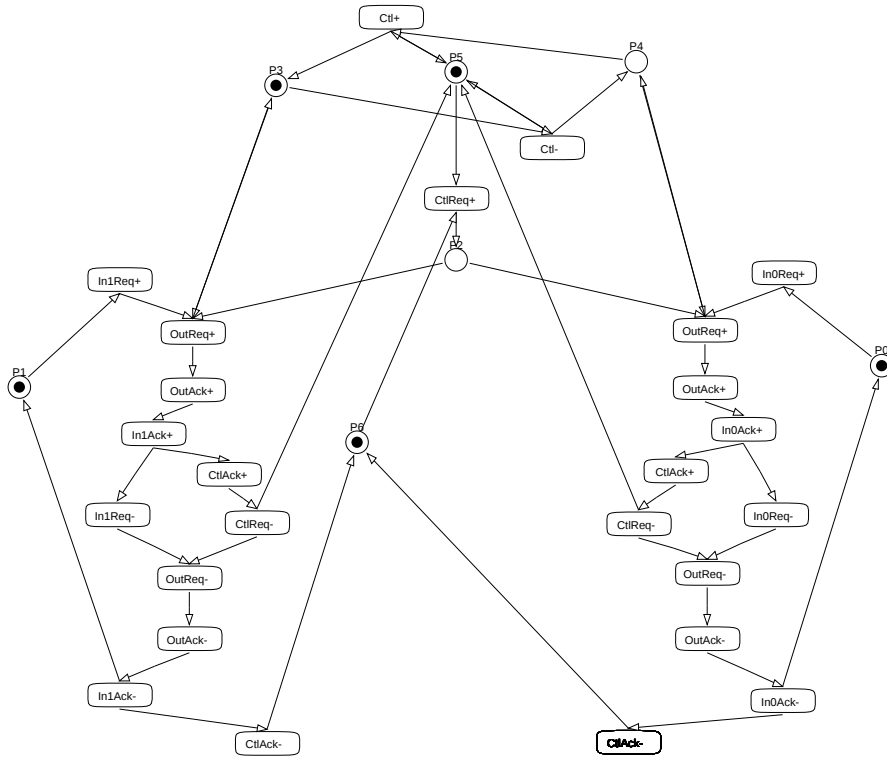


Figure 6.26: The modified STG specification of the 4-phase bundled-data MUX control circuit.

```
[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

Performance analysis using timed Petri nets

We now continue the thread we left at the end of chapter 4 and address quantitative performance analysis of more general structures involving rings and pipelines composed of non-identical pipeline stages.

7.1 Timed Petri nets

In chapter 6, we introduced Petri nets [128, 2, 106] as a means to specify the behavior of a desired circuit, i.e., as a starting point for synthesis. Petri nets are also useful for modeling and analysis of existing circuits. In this chapter, we explore the use of timed Petri nets [129, 164] – Petri nets annotated with delay information – for performance analysis of asynchronous circuits.

The delay information can be annotated to either transitions or places, and such Petri nets are called timed-transition Petri nets (TTPN) and timed-place Petri nets (TPPN), respectively.

Figure 7.1 shows timed Petri nets for the C-element and dummy environment example from figure 6.4 in section 6.2.1. The annotated propagation delays of the inverters are 1 ns and 2 ns respectively, and the propagation delay of the C-element is 4 ns. For the timed transition Petri net, the shaded boxes indicate which components are modeled by which Petri net fragments. As seen, each gate gives rise to two transitions and their input places; one for the up-going and one for the down-going transition of the signal.

In figure 7.1 delays are fixed constant values. It is possible to use other delay models including those discussed in section 6.1.3 as well as stochastic delay models. In the following, we limit to constant delays.

In a TTPN, when a transition is enabled, it fires after the designated delay. In a TPPN, when a token flows into a place, it experiences a delay before

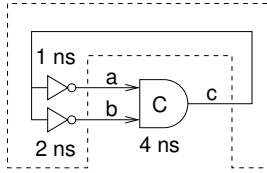
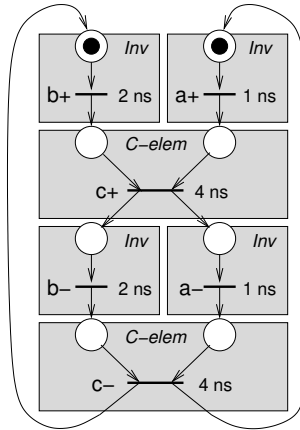
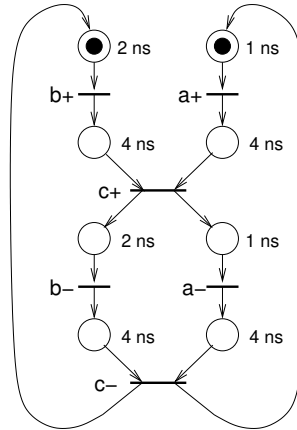
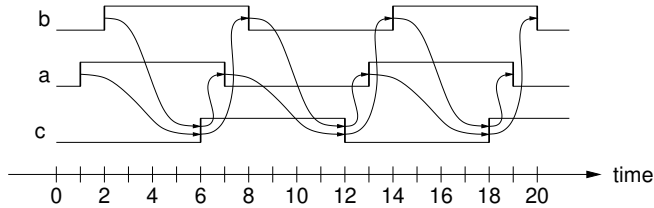
(a) *C-element and dummy environment*(b) *Timed transition Petri net*(c) *Timed place Petri net*(d) *Timing diagram*

Figure 7.1: Timed Petri net models of the C-element with dummy environment example from section 6.2.1 and figure 6.4.

it can be consumed by a transition. A TPPN, like the one in figure 7.1(c), allows modeling of (single output) gates with different propagation delays from different inputs to the output. In a TTPN, on the other hand, the delay is the same from all inputs to the output. It should be noted that a TPPN can be expanded by adding a transition and a place before every transition in the original TTPN in order to model different delays from different inputs. In the

following, when considering TTPNs, we sometimes omit “trivial places” and draw STGs instead of Petri nets, as we also did in chapter 6.

By inspecting the TTPN model in figure 7.1, it is relatively easy to realize that the output of the C-element oscillates with a period of 12 ns. For just marginally more complex timed Petri nets this may no longer be possible and the execution of a timed Petri net may even be different for different initial markings. Below we will review some classic results and elaborate on this. But first we need to introduce a number of Petri-net sub-classes whose properties affect their timing analysis.

7.2 Sub-classes of Petri nets

Petri nets come in many flavors and can be very general and very expressive. In this section, we present several *restricted* classes of Petri nets [106] that are simpler to analyze. The restrictions are defined by structural limitations. We have already discussed one such restriction, one-bounded Petri nets, whose practical implication is a close correspondence between the Petri net model and the circuit itself.

A Petri net graph is a 3-tuple (S, T, W) where S is a finite set of places, T is a finite set of transitions, and W is a finite set of directed arcs. Each arc connects from a transition to a place, or from a place to a transition. This means that S and T are disjoint and therefore, that no object can be both a place and a transition.

In a *state machine (SM)*, every transition has one incoming arc and one outgoing arc, and all markings have exactly one token. As a consequence, there can not be concurrency (the single token follow one path through the Petri net), but there can be choice (i.e., nondeterminism). In the Petri net literature “choice” is often called “confusion”.

In a *marked graph (MG)*, [25] sometimes called an *event graph (EG)*, every place has one incoming arc, and one outgoing arc. This means that there can not be conflict, but there can be concurrency.

In a *free choice net (FC)*, every arc from a place to a transition is either the only arc from that place or the only arc to that transition. This means that there can be both concurrency and conflict, but not at the same time.

In chapter 6, we discussed free choice and controlled choice. Controlled choice is more general than free choice in the sense that it involves concurrency (multiple tokens) in combination with choice.

To illustrate the above definitions, we mention that the Petri net in Figure 7.1 is a marked graph. All markings have two tokens, and transitions on signals a and b are concurrent.

An example of a state machine is shown in Figure 7.2. The observant reader may recognize that the figure shows a full Petri net corresponding to the signal transition graph in Figure 6.11 in subsection 6.3.2 (Example 2). Any marking

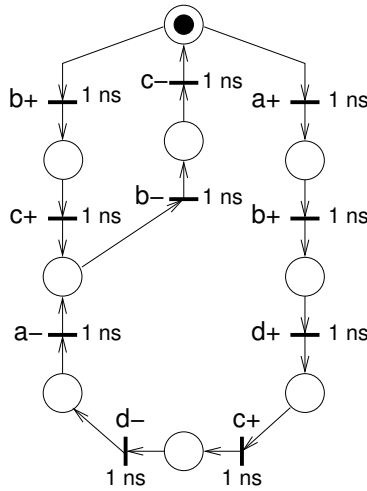


Figure 7.2: Timed transition Petri net corresponding to the STG in Figure 6.11 in subsection 6.3.2 (Example 2). The Petri net is a state machine.

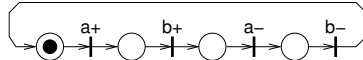


Figure 7.3: A Petri net that is both a marked graph and a state machine.

has exactly one token and depending on the input free choice made after place P_0 , the token cycles through the shorter cycle $C_S = C(b+, c+, b-, d-)$ or the longer cycle $C_L = (a+, b+, d+, c+, d-, a-, b-)$. The Petri net is a TTPN where a delay of 1 ns is associated with each transition.

A Petri net can be both a state machine and a marked graph if there is neither choice nor concurrency. Figure 7.3 shows an example.

The relation between the different classes of Petri nets is illustrated in Figure 7.4. In this book, we limit to the following classes: state machines, marked graphs, and Petri nets with free choice and controlled choice, hence the category “Other forms of choice.” The interested reader is referred to [106] for an exhaustive classification.

7.3 Timing analysis of timed Petri nets

For a clocked circuit, performance is often characterized by the maximum possible clock frequency, and this frequency corresponds to the “critical path” of the circuit – the maximum propagation delay along any combinational circuit

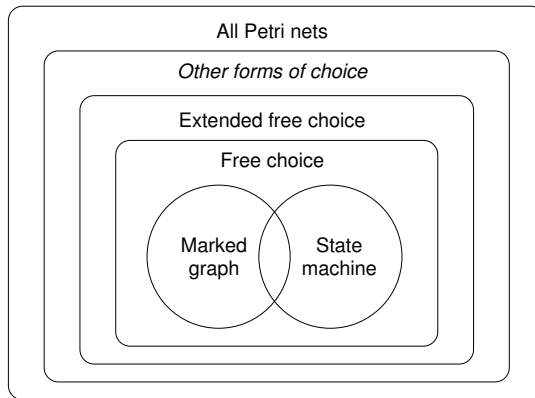


Figure 7.4: Graphical illustration of Petri net classes and their relationship.

path from some flip-flop and into some flip-flop. In an asynchronous circuit, that has no clock, an equivalent measure is the cycle time for a handshake, or expressed more precisely, the maximum time separation between successive (rising) transitions of one of the handshake signals, for example request.

For the marked graph in Figure 7.1, it is relatively easy to realize that the time separation between successive $c+$ transitions is 12 ns corresponding to a token cycling the longest path ($b+$, $c+$, $b-$, $c-$).

For the state machine in Figure 7.2, the situation is more complicated. Depending on the outcome of the free choice, the token cycle through the shorter sequence ($b+$, $c+$, $b-$, $d-$) or the longer sequence ($a+$, $b+$, $d+$, $c+$, $d-$, $a-$, $b-$) producing an execution trace that can be any interleaving of long and short sequence. This means that there are four answers to a question like “what is the time separation between successive $b+$ transitions.” The four answers correspond to the following combination of long (L) and short (S) sequences: 9 ns for L;S, 8 ns for L;L, 5 ns for S;L, and 4 ns for S;S. If instead, we ask for the maximum or the minimum time separation between successive $b+$ transitions, we can get unique answers.

The general message to take from this is that it is not possible to perform a mechanical analysis of a timed Petri net with choice (i.e., a state machine or a free choice net). In many cases, it is possible to work with separate Petri nets, one for each possible choice, and then analyze these separately.

Most of the circuits we are interested in can be modeled using Petri nets that are one-bounded, strongly connected, marked graphs. “One-bounded” means that places never contain more than one token and “strongly connected” means that for any two places, there exists a path that connects them. In such restricted Petri nets, the number of tokens in a cycle remains the same after any firing sequence, and all the transitions have the same cycle time [129].

The minimum average cycle time is given by:

$$T_{cycle} = \max \left\{ \frac{T_k}{N_k}; k = 1, 2, \dots, q \right\} \quad (7.1)$$

where q is the number of simple cycles in the graph (simple meaning that nodes are only visited once). For a given cycle, T_k is the sum of the delays associated with the transitions (or places) in the cycle, and N_k is the number of tokens in the cycle.

For the marked graph in Figure 7.1, there are four cycles, all with a single token:

$C1 = C(a+; c+; a-, c-)$ with a total delay of 10 ns

$C2 = C(a+; c+; b-, c-)$ with a total delay of 11 ns

$C3 = C(b+; c+; a-, c-)$ with a total delay of 11 ns

$C4 = C(b+; c+; b-, c-)$ with a total delay of 12 ns

Thus, the average cycle time is $T_{cycle} = 12$ ns.

The above is quite intuitive, but in the general case, things are more complex than our simple example suggests. The time separation between successive rising or falling transitions may exhibit some variation. When the circuit starts operating after being reset, an initial aperiodic timing behavior may be observed before the timing behavior settles into a periodic mode. This periodic behavior may show a constant time separation between successive transitions on the same signal, or it may show a repeating pattern of different time separations. The cycle time computed using equation 7.1 is the average time separation in a steady-state – or starting from time $t = 0$, the average as the number of executed cycles approaches infinity.

A few examples derived from [96] illustrates this. The examples are timed place Petri nets (TPPN). In a TPPN tokens wait for the specified amount of time before eventually engaging in the firing of the transition that they enable. It is assumed that the tokens have initially been waiting longer than the specified delay time and therefore, that the enabled transitions fire immediately at time $t = 0$.

Figure 7.5 shows a TPPN that has a single critical cycle, $C(c, d)$. Regardless of the initial marking, the firing sequence settles into the same repeating sequence of transitions, and in the steady-state, the time separation between subsequent occurrences of the same transition is constant.

Figure 7.6 shows the same TPPN with the same two initial markings, but with different delays such that there are now two critical cycles $C(a, b)$ and $C(c, d)$ both with a cycle time of 5. For both initial markings, the timing behavior settles into a mode where the time separation between successive occurrences of the same transition is constant, but the repeating steady-state sequences of transitions are different.

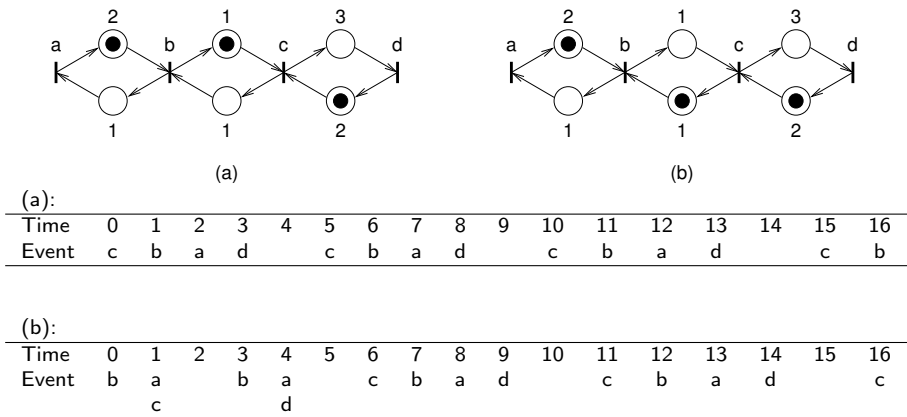


Figure 7.5: A Timed place Petri net with different initial markings. With one dominating cycle, the TPPN settles into the same repeating sequence regardless of the initial marking. In the steady-state, the time separation between successive occurrences of a given transition is constant.

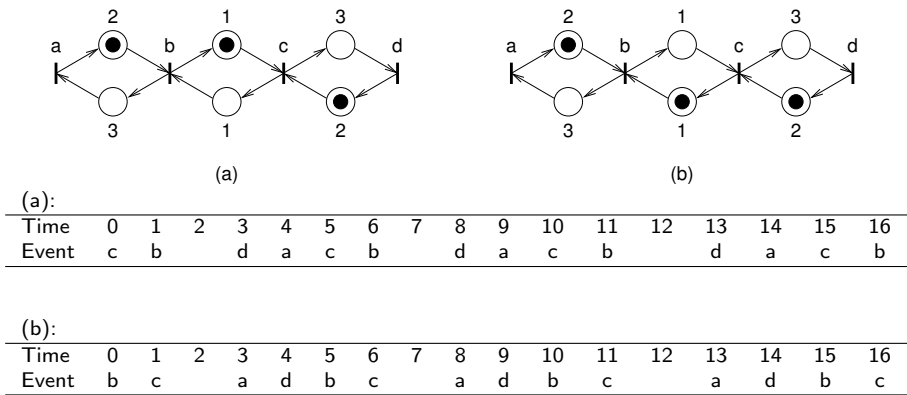


Figure 7.6: A Timed place Petri net with different initial markings. It has two dominating cycles, and the two initial markings in (a) and (b) result in different steady-state sequences. The time separation between successive occurrences of a given transition is constant and identical in both cases.

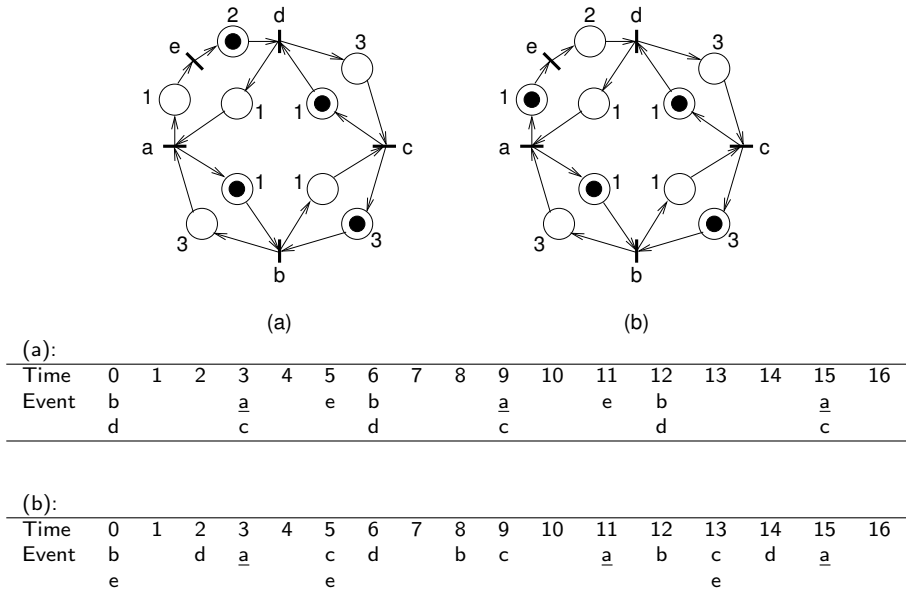


Figure 7.7: A Timed place Petri net with different initial markings; (a) and (b). The different markings lead to different timing behaviors.

Finally, figure 7.7 shows another TPPN with different initial markings and with multiple critical cycles. The two different markings illustrated in Figure 7.7(a) and (b) result in the execution traces shown below the Petri nets. For the TPPN in figure 7.7(a), the steady-state time separation between successive transitions of some signal is constant, 6 time-units. For the TPPN in figure 7.7(b), the steady-state time separation between successive transitions of some signal alternates between 4 and 8 time-units.

The average is still 6 time-units as would be computed using equation 7.1, but if a circuit with such oscillating behavior is used in a context where the worst-case is important, it is the 8 time-units that matter. This could be the case if the asynchronous circuit is used in combination with a clocked circuit. Similar issues need to be considered for a possible initial non-steady-state behavior. In such scenarios, it is the worst-case time separation between events that matters, but let us first explore how far we can go with the simpler average cycle time and equation 7.1.

7.4 Example 3 revisited: Analysis using a TTPN

Let us now revisit the last example circuit we considered in section 4.3.3. This circuit, called “example 3,” is a ring composed of 3 identical pipeline stages. The full circuit is shown again in figure 7.8(a). The relationship between circuit fragments and fragments of a corresponding TTPN is shown in figure 7.8(b). Each gate output signal in the circuit causes two transitions in the Petri net: one for the up-going and one for the down-going transition of the signal. The delay elements are considered as single input single output gates (like the inverters). The full TTPN for the control circuit is shown in figure 7.8(c). To ease reading this figure, signals are named after the gates that drive the signals (“d” for delay element, “C” for C-element, and “i” for inverter).

For the TTPN model, we need an initial marking. This should correspond to a snapshot of a possible state that can be observed during the normal operation of the circuit. A natural choice would be the reset state. The cyclic behavior of the circuit means that the circuit “visits” this state during operation as well. Figure 7.8(d) shows an abstract data-flow representation of the 3-stage ring. An empty token has just been copied into the third stage. This corresponds to a situation where the three C-elements in the control circuit (C1, C2, C3) are (1, 0, 0), i.e., C3 has just performed a falling transition. This corresponds to the two tokens in the bottom right corner of Figure 7.8(c). The next C-element transition will be C2 that will make a rising transition. This transition will be enabled by the token on arc ($c1+$, $d2+$) after it has fired transition $d2+$, and by the token on arc ($c3-$, $i2+$) after it has fired transition $i2+$.

The cycle time can be computed by using equation 7.1. The graph has many (simple) cycles all with exactly one token:

One cycle, $C(d1+, c1+, d2+, c2+, d3+, c3+)$, corresponds to the forward propagation of rising transitions and it has cycle time $P = 15$ ns. A similar cycle with the same cycle time is $C(d1-, c1-, d2-, c2-, d3-, c3-)$ that corresponds to the forward propagation of falling transitions. The longest cycle turns out to be: $C(i2+, c2+, i1-, c1-, i3+, c3+, i2-, c2-, i1+, c1+, i3-, c3-)$. It corresponds to the single bubble in the data-flow representation that supports the three forward propagations of a valid token and the three forward propagations of an empty token that is needed to completely cycle the pattern of tokens and bubbles as discussed in section 3.2. The cycle time of this longest cycle is $P = 18$ ns. The TTPN has six more cycles, all with a similar structure and all with the same cycle time, $P = 16$ ns. Three we can call “3-up-1-down” (referring to the C-element transitions), for example $C(i1+, c1+, d2+, c2+, d3+, c3+, i2-, c2-)$. By horizontal mirroring we get three similar cycles that we can call “3-down-1-up”.

It is comforting to see that the maximum cycle time $C = 18$ ns corresponds to what we calculated in table 4.1 in section 4.3.3.

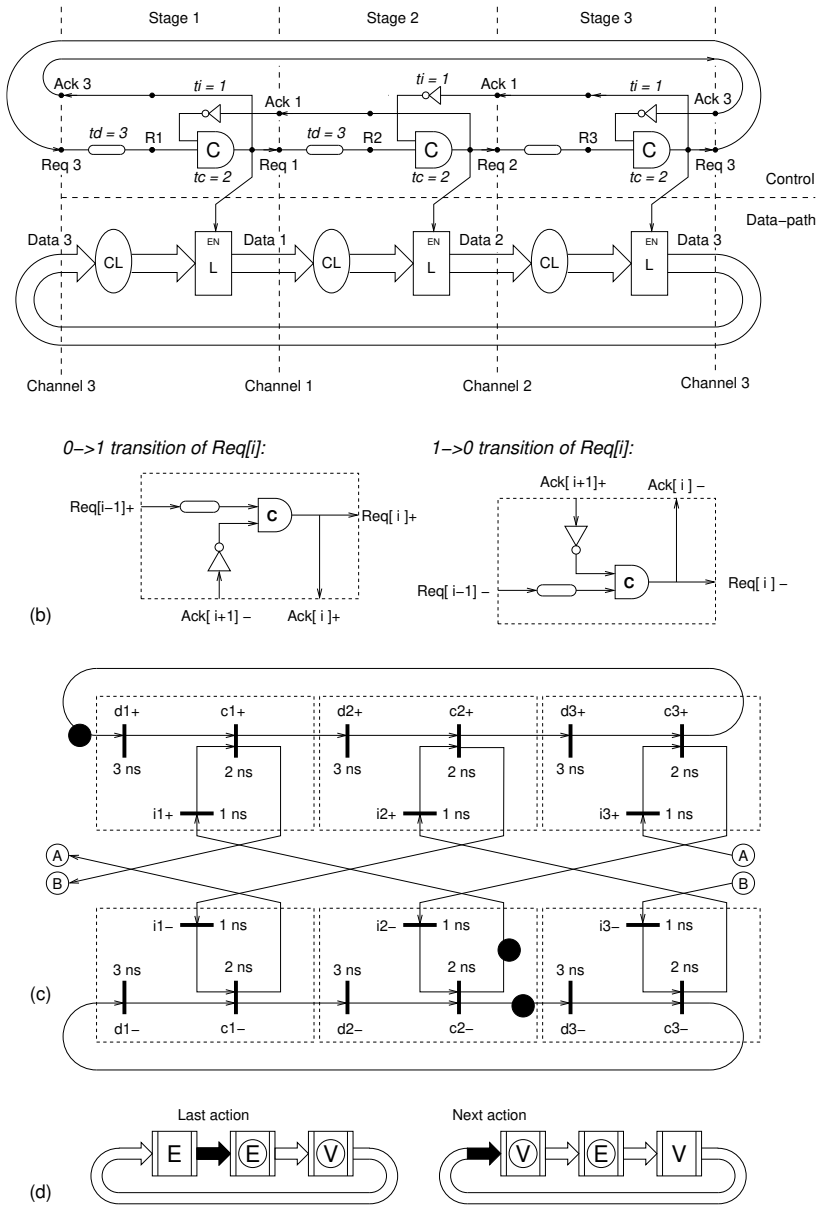


Figure 7.8: (a) A simple 3-stage ring composed of identical pipeline stages (example 3 from section 4.3.3). (b) Relationship between fragments in the schematic and fragments of the corresponding TTPN. (c) The full TTPN model of the control circuit of the 3 stage ring. (d) The initial marking of the TTPN corresponds to the state after “last action” and before “next action.”

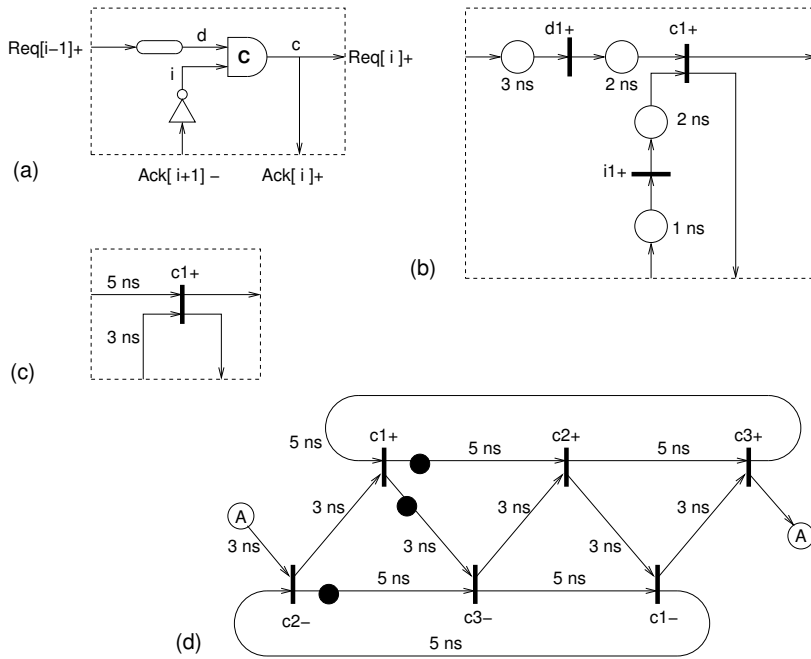


Figure 7.9: Simplifying the TPPN model of example 3. (a) Circuit fragment for a rising transition. (b) The corresponding full TPPN fragment. (c) Abstracting away internal signals i and d . The delays of their input places are added to the delays of the input places of signal c . As a final simplification, trivial-places (places with one input arc and one output arc) are omitted as in an STG. (d) The final simplified TPPN for example 3 with the same initial marking as used in figure 7.8.

7.5 Example 3 revisited: Analysis using a simplified TPPN

The Timed transition Petri net in figure 7.8 is quite heavy. If we use a timed place Petri net instead and apply some trivial simplifications, we get a simpler Petri net that is easier to work with and analyze. This is illustrated in figure 7.9. The simplification consists of substituting a straight sequence of transitions and places by a single transition and a single place. To this place, we associate a delay that corresponds to the sum of the delays in all the places in the straight sequence of transitions and places. By “straight sequence” we mean that the transitions and places all have a single input and a single output edge. In this way, we hide the signals corresponding to the omitted transitions.

The resulting simplified TPPN has the same (average) cycle time (com-

puted using equation 7.1), but it may not exhibit all the possible timing behaviors as of the full TPPN. To realize this, consider the TPPN in figure 7.6. If we hide signal e we will have a place with a delay of 3 in the a -to- b path, similar to the other clockwise connections, d -to- c , c -to- d , and b -to- a . And in the initial marking, there will be a token in the place on the (new direct) a -to- d path. This gives us the same timing behavior as in figure 7.6(a). The oscillating behavior of figure 7.6(b) cannot be produced by the simplified TPPN, so a designer should be careful when working with circuits where the worst-case cycle-time matters.

7.6 Example 4: A four stage ring

As discussed earlier, a 4-stage ring with one valid token, one empty token, and two bubbles may have a better performance. Figure 7.10(a) shows the abstract static data-flow view, and Figure 7.10(b) shows the corresponding TTPN.

There are now two “bubble cycles” marked A and B, and they each have one token. This supports rising and falling C-element transitions to happen concurrently. The dominating cycle is now the forward propagation of the rising (or the falling) transitions, $P = 20$ ns. As we already knew from table 4.1 on page 59, the extra stage did not help in this particular case.

7.7 Example 5: A pipeline with asymmetric delay elements

In all the previous examples, we have analyzed rings. Let us now consider a pipeline. Here the typical performance-related question is: At what rate will data-tokens flow through the pipeline? Or alternatively: What is the time separation between successive data-tokens flowing through the pipeline? To analyze this, we need to analyze a “sufficiently long” pipeline. We will discuss the meaning of “sufficiently long” shortly.

Intuitively we need to study and analyze a pipeline stage sandwiched between identical pipeline stages to have a realistic model of the environment of the pipeline stage we consider. Figure 7.11(a) shows a pipeline consisting of three identical stages connected to an ideal source and an ideal sink. We assume a four-phase bundled data implementation using a Muller pipeline as the control circuit, and we assume that we use asymmetric matched delays as illustrated in figure 7.11(b). By using asymmetric delay elements we hope to improve performance by minimizing the forward and reverse latencies for empty-tokens. The full TTPN model is shown in figure 7.11(c).

In order to obtain a strongly connected TTPN model, we add a dummy producer and a dummy consumer. A dummy producer in the input handshake channel can be modeled by an inverter (i.e., by connecting Ack- to Req+ and by

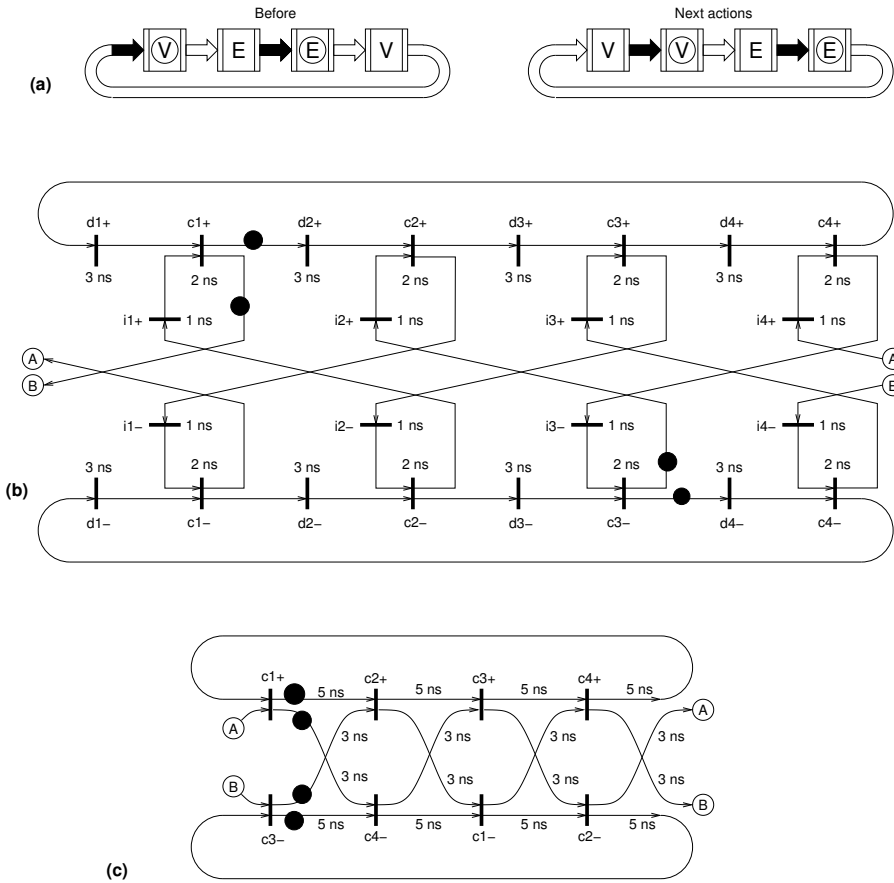


Figure 7.10: (a) A simple 4-stage ring composed of identical pipeline stages similar to those in figure 7.8. (a) Data flow structure view to help identify the initial marking of the TTPN. (b) The resulting full TTPN. (c) A simplified TTPN.

connecting Ack+ to Req-), and a dummy consumer on the output channel can be modeled by a buffer (i.e., by connecting Req+ to Ack+ and by connecting Req- to Ack-). A possible initial state is that the pipeline is completely empty. An incoming Valid-token will then fall through the pipeline. In the TTPN in figure 7.11(c), this corresponds to the C-elements making rising transitions. To enable this, all $c+$ transitions have a token on the input place corresponding to the acknowledge signal from the successor pipeline state.

Two cycles are marked in figure 7.11(c):

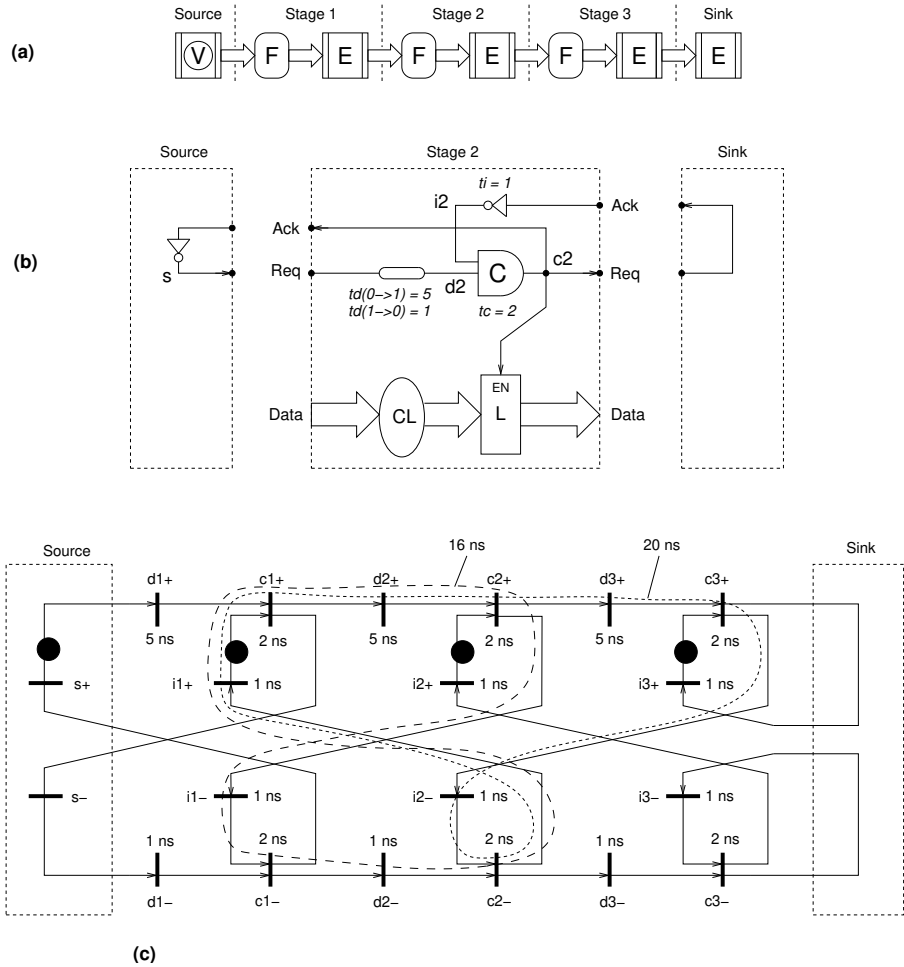


Figure 7.11: (a) A pipeline composed of identical stages. The pipeline is initially empty, and the source is about to inject a valid token. (b) The pipeline stages are identical and use a 4-phase bundled data implementation with asymmetric matched delays. The source is modeled using an inverter, and the sink is modeled by simply setting $\text{Ack} \leq \text{Req}$ (c) A detailed TTPN model of the pipeline and its environment. The marking corresponds to the initial state described above; all C'elements are low, and the source has just raised Req.

$$C1 = C(d2+, c2+, i1-, c1-, d2-, c2-, i1+, c1+) \text{ with } T_{Cyclc} = 16 \text{ ns}$$

$$C2 = C(d2+, c2+, d3+, c3+, i2-, c2-, i1+, c1+) \text{ with } T_{Cyclc} = 20 \text{ ns}$$

Cycle $C2$ is the critical cycle, and hence the time separation between successive data-tokens flowing through the pipeline is 20 ns. Using the terminology from section 4.3.1, we say that the pipeline has a period $P = 20$ ns.

Recalling the definition of per-stage forward and reverse latencies from section 4.3.1, the pipeline stage we consider here has the following parameters:

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

Using these parameters we see that cycles $C1$ and $C2$ corresponds to the following periods respectively:

$$\begin{aligned} P_1 &= \underbrace{t_{d2(0 \rightarrow 1)} + t_{c2}}_{L_{f.V}} + \underbrace{t_{i1} + t_{c1}}_{L_r \downarrow} + \underbrace{t_{d2(1 \rightarrow 0)} + t_{c2}}_{L_{f.E}} + \underbrace{t_{i1} + t_{c1}}_{L_r \uparrow} \\ &= L_{f.V} + L_{f.E} + 2L_r = 16 \text{ ns} \\ P_2 &= \underbrace{t_{d2(0 \rightarrow 1)} + t_{c2}}_{L_{f.V}} + \underbrace{t_{d3(0 \rightarrow 1)} + t_{c3}}_{L_{f.V}} + \underbrace{t_{i2} + t_{c2}}_{L_r \downarrow} + \underbrace{t_{i1} + t_{c1}}_{L_r \uparrow} \\ &= 2L_r + 2L_{f.V} = 20 \text{ ns} \end{aligned}$$

Note that period P_1 is the minimum possible period given by equation 4.1 on page 56, and that period P_2 , the actual period of the pipeline, is the period given in equation 4.3 on page 57. This example illustrates an important lesson: that for some (simple) handshake latch implementations (here using a Muller pipeline stage), it may not be possible to reduce the cycle time by using function blocks with asymmetric delays ($L_{f.E} < L_{f.V}$).

In later chapters, we will consider other 4-phase latch controllers that can take advantage of a lower $L_{f.E}$, and we will consider two-phase designs where there is no $L_{f.E}$ and no $L_r \uparrow$ and where therefore, this issue does not exist.

7.8 Worst-case timing analysis

So far, we have discussed timing analysis based on equation 7.1 that gives us the minimum average cycle time of a timed maked graph. This is very useful to make first-order design decisions and optimizations. In figure 7.6, we showed an example where the cycle time steadily alternates between 4 and 8 time-units with an average of 6 time-units as calculated using equation 7.1.

We also mentioned that similar (perhaps aperiodic) fluctuations may occur after reset before a steady-state behavior is reached.

In some cases, for example when the asynchronous circuit in question is used in combination with a synchronous circuit, it is necessary to design for the worst-case cycle time, or in the more general case, the worst-case time separation between some important events. In our context, events are signal transitions.

This is a harder problem to solve than the average case problem. Early work in this area was performed by Hulgaard *et al.* [67, 65]. Later works with a focus on asynchronous circuits are [96, 62]. A circuit designer may be more interested in analyzing a given circuit than in the underlying theories and algorithms and should look for software packages that implement the proposed analysis algorithms. The author has experience with a tool [95] based on the methods presented in [96] and with a tool [74] based on Hulgaard's algorithm. Work using the latter is presented in [73, 72, 75].

Chapter 8

Metastability, arbitration, and synchronization.

Digital circuits operate on 0s and 1s, and this binary abstraction rests on the assumption that the setup-time and hold-time of all state holding elements (e.g., flip-flops and latches) are satisfied. This cannot always be guaranteed and may cause non-digital values persisting for some unbounded amount of time. This phenomenon is called metastability, it is unavoidable, and it must be properly dealt with.

In this chapter, we address three issues: (i) What is metastability and what causes it. (ii) How to quantify the effects of metastability. (iii) How to properly deal with metastability in order to design reliable circuits that perform synchronization and arbitration.

8.1 What is metastability?

In a clocked circuit, metastability can occur if an input signal to a flip-flop makes a transition at a point in time that violates the setup and hold times required by the flip-flop. In this case, where the flip-flop is sampling an input signal in transition, the flip-flop literally has to decide if the input signal made its transition before or after the rising edge of the clock, i.e., decide who was first. In an asynchronous circuit, a similar situation may occur if two input signals to a set-reset bistable (as used in the MUTEX in section 5.10) are asserted at almost the same time. Like the flip-flop we just described, the mutex has to decide who was first. The two situations are illustrated in figure 8.1, and as indicated, it is unclear what value the output will take.

The crux of the matter is that there is no upper limit on how long time it may take to make the required discrete (e.g., binary) decision, and during this period, where the circuit is undecided, it may produce a non-digital output,

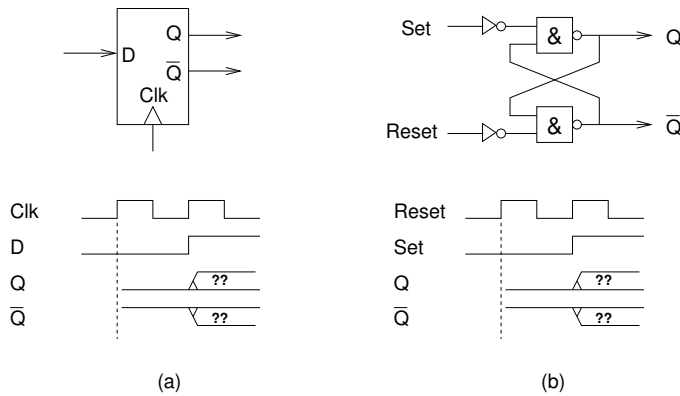


Figure 8.1: (a) A D flip-flop exposed to a data signal that transition simultaneously with the clock. (b) an asynchronous set-reset bistable where at one point set and reset are both asserted at the same time.

which, if not properly dealt with, may cause a system failure.

Figure 8.2(a) shows a simple bistable element composed of two cross-coupled inverters and figure 8.2(b) shows a plot combining the DC-transfer characteristic of the two inverters. The latter shows the two stable states (0 and 1) as well as a metastable state where the two DC-transfer curves cross.

A mechanical equivalent of a bistable is shown in figure 8.2(c). Here a ball can lie stable in one of two lows, representing states '0' and '1'. To flip the state, the ball has to be rolled over a top. If we start pushing the ball towards the opposite state, but stop pushing when the ball is somewhere near the top, the ball may hesitate for some time before falling to one or the other side. In rare cases, the ball is left almost perfectly balanced at the top, and it may hesitate for a very long time, before randomly falling into one of the two stable states. It is even possible that the ball is perfectly balanced at the top and stays there forever. The ball hesitating at the top for some time is a metastable state.

Before discussing circuits in more detail, let us look at some additional examples from every-day life in order to consolidate our understanding of the nature of the problem.

Imagine that you are presented to pictures taken at the finish line of different 100-meter runs (for simplicity involving only two contestants), and imagine that your task is to identify the winner based on the picture shown to you. Imagine further that the time you take to decide is measured. In cases where the decision is obvious, you are fast. In cases where it is less obvious who won, you dwell longer, analyzing the picture more carefully, and in some few cases where it is almost impossible to make a decision, you dwell very long, perhaps

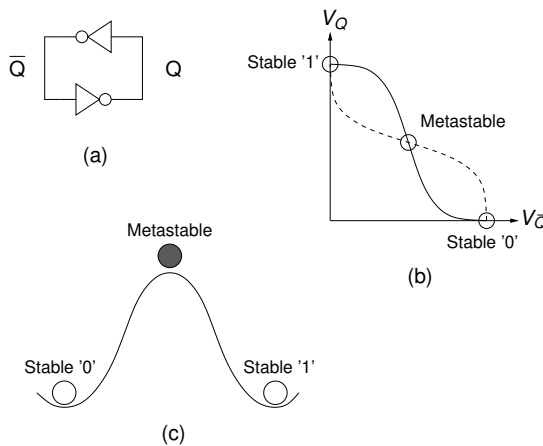


Figure 8.2: (a) A bistable built from a pair of cross-coupled inverters. (b) A plot combining the DC-transfer characteristics of the two inverters. (c) A mechanical equivalent of a bistable.

even indefinitely. These extended periods of time, where you “have not yet decided,” is metastability.

Another situation, that we have all experienced, is when two people are walking towards each other in a sidewalk or corridor. In order to pass each other, one or both has to yield by making a step to the side. Sometimes both persons step to the same side, leaving the situation unresolved. This may continue more times before the two persons eventually manage to pass each other. This is an example of an oscillating metastable behaviour, and as before, it results in an additional delay before a well-defined digital output is reached.

Early examples of metastability in electronic circuits appeared in the first digital computers where signals from pushbuttons were input to clocked circuitry. If such an asynchronous input signal is switching close to the rising edge of the clock, and if the signal is forked to several subsystems, then there is a possibility that one subsystem may see the signal as being 0 and that another subsystem may see it as being 1. Such inconsistent interpretations of a signal may result in faulty behavior. The situation can be countered by connecting each asynchronous input signal to a single flip-flop whose output is then used inside the circuit. Early computers like ENIAC are reported to have used such “input-buffering” flip-flops [85]. During the 1940’s-1970’s many digital designers believed that the use of such buffer flip-flops was enough, reasoning that if a buffer-flip-flop just missed seeing an input signal being asserted at a given clock tick, then it would see it at the next clock tick.

The fact that a flip-flop whose setup time or hold time is violated may

take an unbounded time to settle into a stable digital state was identified and addressed as early as in 1952 [85]. This paper contributed an analysis showing that the probability of a flip-flop not having settled to a stable state reduces exponentially with the time allocated to wait for this. As mentioned, this fact and its implications were not widely understood, and it was not until 1973 that measurements confirmed the existence of metastability, i.e., unbounded settling time [18].

Interestingly, versions of this paper were rejected with reviewer comments like [100]: *“Of course. So what”* and *“this can’t be true because flip-flops only deal with zeros and ones”* and *“if this problem really existed, it would be so important that everybody knowledgeable in the field would have to know about it ... I’m an expert and I don’t know about it, so therefore it must not exist.”*

A flip-flop is a bistable circuit, and often both the output value Q and its complement $\overline{Q}$ are available. The measurements presented in [18] show that metastability can take two forms: (1) *“halfway metastability”* where both outputs (Q and $\overline{Q}$) start switching but then hang halfway between 0 and 1 for some extended time. (2) *“oscillating metastability”* where the two outputs oscillate in phase: $(Q, \overline{Q}) = (0, 0); (1, 1); (0, 0); \dots$, before some imbalance eventually causes the circuit to settle into a stable state; $(0, 1)$ or $(1, 0)$. The two situations are shown in figure 8.3.

The mechanical bistable illustrated in figure 8.2(c) and the task of declaring winners in the 100-meter run are examples of “halfway metastability.” Considering the cross-coupled inverters in figure 8.2(a), halfway metastability tend to occur when the propagation delays of the inverters are dominated by the rise and fall times of the inverters, as is usually the case in CMOS technology. Oscillating metastability tend to occur when the propagation delays of the inverters are larger than the rise and fall times of the inverters. The example with two people meeting in a narrow corridor is an illustration of oscillating metastability.

Using a transistor-level simulator like SPICE, it is possible to initialize Q and $\overline{Q}$ with voltage levels very close to the metastable point in figure 8.2(b) and perform a transient simulation showing how the circuit works its way out of metastability. In a similar way, it is possible to initialize both Q and $\overline{Q}$ close to logic 1 (or logic 0) and observe oscillating metastability. In both cases, a small imbalance is needed in order for the simulation to reach a stable state, and the closer to the balancing point the circuit is initialized, the longer it takes for the circuit to reach a stable state.

8.2 Quantifying metastability

Now that we have developed an intuitive understanding of metastability, the next step is to try and quantify it, i.e., to quantify the probability that metastability occurs and persists beyond some given time-interval.

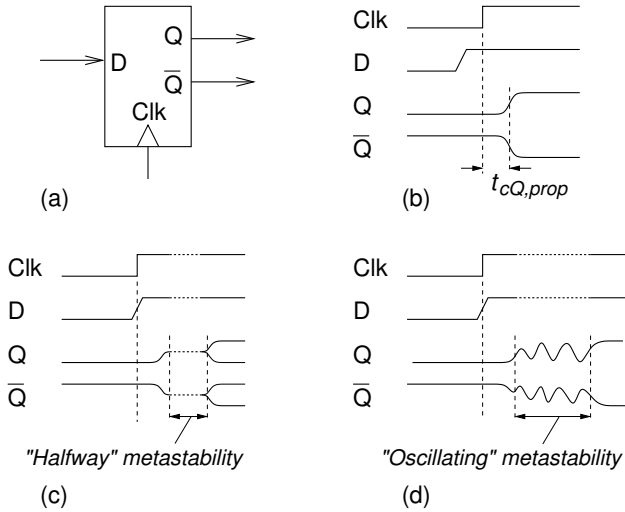


Figure 8.3: (a) A D-flip-flop with outputs Q and $\bar{Q}$. (b) A normal situation where the input signal D satisfies the required setup and hold times. The propagation delay from Clk to Q is $t_{cQ,prop}$. If the input signal D makes a transition almost simultaneously with the rising edge of the clock, the flip-flop becomes metastable for some time. This can be in the form of “halfway metastability” as shown in (c) or “oscillating metastability” as shown in (d).

Figure 8.4 shows the response time of a flip-flop as a function of the time separation between the arrival time of the input data and the rising edge of

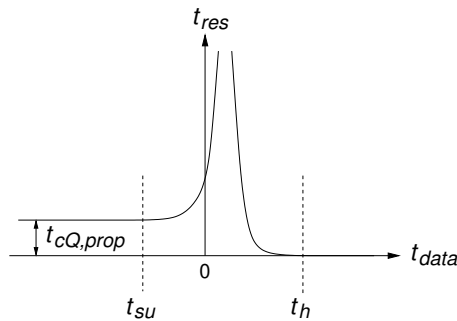


Figure 8.4: (a) Response time (t_{res}) of a flip-flop as a function of the arrival time of data relative to the clock. If the input satisfies the setup and hold times of the flip-flop, the response time is simply the propagation delay ($t_{cQ,prop}$).

the clock. When the input changes more than the setup time, t_{su} , before the clock, the response time is the propagation delay of the flip-flop, t_{pd} , and when the input changes after the hold time, t_h the flip-flop remains stable. If the input transitions within the setup-time to hold-time interval, the response can increase steeply, and as illustrated in figure 8.4, the input may arrive at a time where the response time is unbounded. These situations are metastability.

A similar situation exists for a MUTEX (section 5.10). Here the arrival time is the arrival time of one of the request signals relative to the other. The analysis below is for a flip-flop, but it applies for a MUTEX as well if signal Clk is substituted by signal $R1$, and signal $Data$ is substituted by signal $R2$.

If $P(met_t)$ denotes the probability of a flip-flop 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:

$$MTBF = \frac{1}{P(met_t)} \quad (8.1)$$

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

$$P(met_t) = P(met_t | met_{t=0}) \cdot P(met_{t=0}) \quad (8.2)$$

where:

- $P(met_t | met_{t=0})$ is the probability that the flip-flop is still metastable at time t given that it was metastable at time $t = 0$ (i.e., at the beginning of a clock period, right after the rising edge of the clock).
- $P(met_{t=0})$ is the probability that the flip-flop enters metastability within an observation interval of one second.

The probability $P(met_{t=0})$ can be calculated as follows: the flip-flop will go metastable if the inputs, $Data$ and Clk , are exposed to transitions that occur almost simultaneously, i.e., within some small time window $\Delta \sim t_{su} + t_h$.

If we assume that the data signal and the clock signal are uncorrelated and that they have average switching frequencies f_{Data} and f_{Clk} respectively, then:

$$P(met_{t=0}) = \Delta \cdot f_{Clk} \cdot f_{Data} \quad (8.3)$$

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

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

$$P(met_t | met_{t=0}) = e^{-t/\tau} \quad (8.4)$$

where τ expresses the ability of the flip-flop 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

	f_{Clk}	Δ	τ
180 nm CMOS ASIC [53]	200 MHz	50 ps	10 ps
28 nm CMOS ASIC [52]	1 GHz	20 ps	10 ps
180 nm CMOS [8]			17 ps
65 nm CMOS [8]			6.0 ps
22 nm CMOS [8]			7.12 ps
65 nm CMOS [7]		10 ps	90 ps
Textbook [30, Ch. 28]	500 MHz	200 ps	100 ps

Table 8.1: Published values for τ and Δ for different process nodes. Field left blank when information not available.

coupled inverters 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 inverters at the metastable point has a single dominating pole.

Combining equations 8.1–8.4 we obtain

$$P(\text{met}_t) = \Delta \cdot f_{Data} \cdot f_{Clk} \cdot e^{-t/\tau} \quad (8.5)$$

expressing the probability that a metastable situation occur and persist for longer than time t within an observation interval of one second.

If we allocate time t_r for recovery from metastability, and consider it a failure if metastability lasts longer, then the mean time between failure (MTBF) is the inverse of equation 8.5:

$$MTBF = \frac{e^{t_r/\tau}}{\Delta \cdot f_{Data} \cdot f_{Clk}} \quad (8.6)$$

Experiments and simulations have shown that this equation is reasonably accurate provided that t_r is not very small, and experiments or simulations may be used to determine the two parameters Δ and τ .

Actual values for Δ and τ are not that easy to find in the open literature. Table 8.1 lists some values from a selection of sources. Due to the nature of the problem, selecting conservative values is advised. A bit simplistic, and very much on the safe side, one can set $\Delta = t_{su} + t_h$ [30]. Realistic values are significantly smaller. The other parameter, τ , appears in the exponent and plays a dominating role. In technologies down to 65 nm CMOS, τ scaled proportionally to $FO4$ (the typical propagation delay in an inverter driving four equally sized inverters) [8]: $\tau \approx 0.45 \cdot FO4$. For technologies smaller than 65 nm, τ no longer scales proportionally to $FO4$, and in a modern 22 nm CMOS technology $\tau \approx 1.0 \cdot FO4$. The “scaling gap” between $FO4$ and τ is expected to increase as technologies shrink further. The values for τ stated in table 8.1 are for typical process parameters, a nominal supply voltage, and

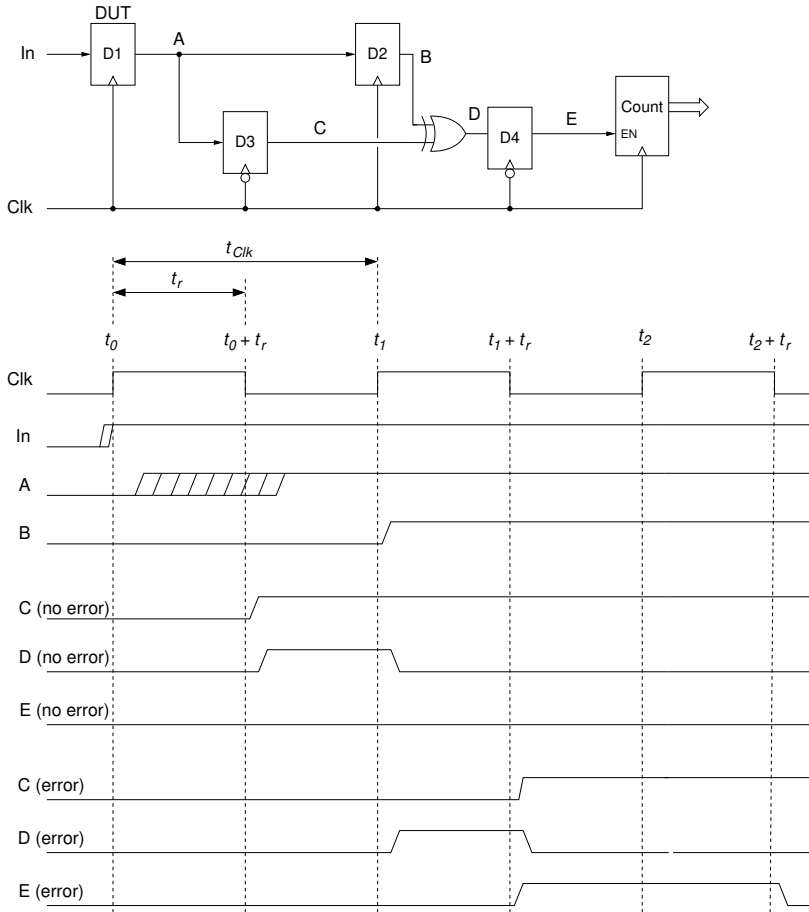


Figure 8.5: A circuit used for metastability characterization.

room temperature. In worst-case process-voltage-temperature corners, τ may vary considerably. Intuitively variations in τ should track variations in $FO4$, but this is not always the case. Results have been published, showing that in some situations, τ is larger when the temperature is lower [173, 7].

A possible experiment, from which τ and Δ can be determined, is shown in figure 8.5 [3, 8]. Two independent (uncorrelated) pulse generators generate Clk and $Data$. Flip-flop D1 is the device under test (DUT) that occasionally goes metastable. The output of D1 is fed to two flip-flops: D3 that is clocked after some delay, t_r , (the allowed resolution time, here when Clk makes a falling transition), and D2, that is clocked at the next clock tick (at which time D1 is assumed to have recovered from metastability). If the value in

D2 is considered “correct,” then it may be considered an error, caused by metastability that has not yet resolved, if the value in D3 is different. The XOR gate compares the two signals, and the result is clocked into D4, whose output can be used as an enable signal for a counter. The timing diagram in figure 8.5 shows the operation of the circuit when metastability has resolved before t_r (no error) and when metastability persists beyond t_r (error).

The experiment is left for hours or days, and at the end the count value is read. The MTBF for a given value of t_r , is the time the experiment has been running divided by the count of metastable events. By changing the duty cycle of the clock signal, it is possible to alter t_r . If the duty cycle cannot be changed, a controllable delay element may be used to produce a delayed clock for D3 and D4. Alternatively, the clock period may be varied, thereby varying the time where the clock is high. Another option when using an FPGA is to use a digital clock manager component to produce a clock signal and a 90-degree phase-shifted version of the clock signal.

The MTBF determined using this circuit can be used in conjunction with equation 8.6 to determine τ and Δ . By taking the natural logarithm of both

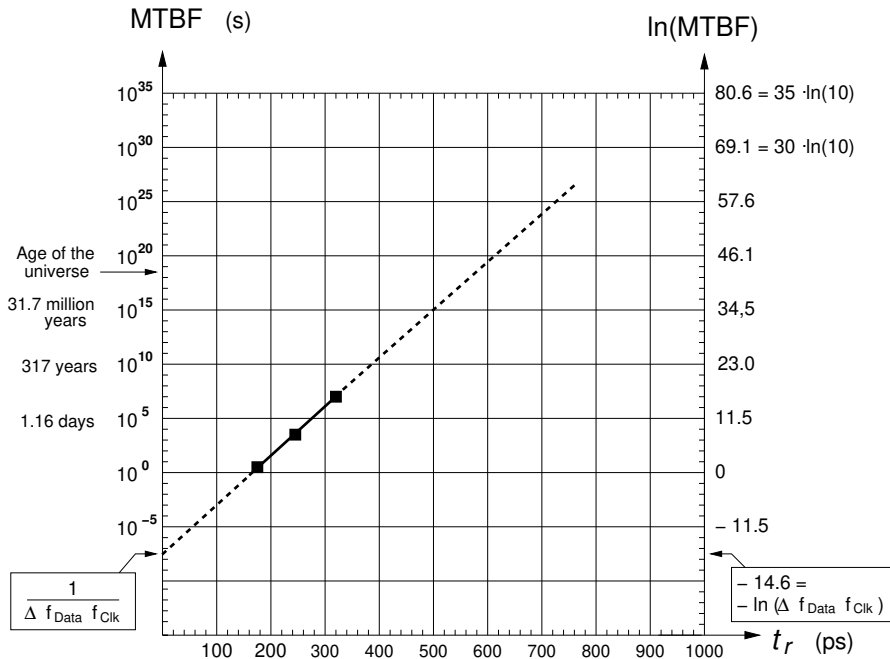


Figure 8.6: A plot of $MTBF$ (and $\ln(MTBF)$) as a function of the allowed resolution time t_r . From the plot τ and Δ can be determined.

sides of equation 8.6, we get:

$$\ln(MTBF) = \frac{t_r}{\tau} - \ln(\Delta \cdot f_{Data} \cdot f_{Clk}) \quad (8.7)$$

For a given experiment, the term $\ln(\Delta \cdot f_{Data} \cdot f_{Clk})$ is constant, and consequently $\ln(MTBF)$ is a linear function of t_r . Figure 8.6 shows a semilogarithmic plot of $MTBF$ for different values of t_r . By fitting these measured points to a straight line, Δ and τ can be determined. The slope of the line is $1/\tau$, and $\ln(\Delta \cdot f_{Data} \cdot f_{Clk})$ is the extrapolated intersection with the y-axis from which Δ is easily obtained. Knowing Δ , τ can be computed from any point on the line. The experiment ignores many subtle details of the internals of a master-slave flip-flop, but it is adequate when t_r is not too small. As we are interested in reliable circuits with a high $MTBF$, we do not deal with small values of t_r .

The graph in figure 8.6 is for a circuit using $f_{Clk} = 1.0$ GHz and $f_{Data} = 0.1$ GHz. From the graph we obtain $\Delta = 20$ ps and $\tau = 10$ ps.

8.3 Dealing with metastability

Now that we understand that metastability is unavoidable, the next step is how to deal with it. Below we cover the two fundamental situations: mutual exclusion in asynchronous circuits and synchronization in clocked circuits. In a later section, we address communication and synchronization in multi-clock systems.

8.3.1 Mutual exclusion and arbitration

Figure 8.7 shows the MUTEX circuit from figure 5.28 on page 89. 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 grant outputs $G1$ and $G2$ in such a way that at most one output is active at any given time.

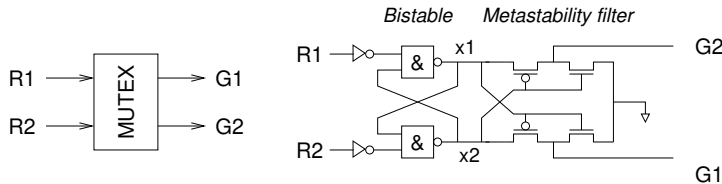


Figure 8.7: The mutual exclusion element: symbol and possible implementation (copy of figure 5.28 on page 89).

The circuit consists of a pair of cross-coupled NAND gates implementing an asynchronous set-reset bi-stable 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 – signals $G1$ and $G2$ are both driven low, until signals $x1$ and $x2$ differ by more than a transistor threshold voltage.

The metastability filter in figure 5.28 is a CMOS transistor-level implementation from [92]. An NMOS predecessor of this circuit appeared in [136]. 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 ([119, 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 [4, 156].

In conclusion the MUTEX deals with metastability by taking as long as it needs to decide, and it always outputs a well-defined digital signals.

If an asynchronous circuit containing one or more MUTEX'es is required to perform some computation, one can calculate the response time using nominal propagation delays. And for a certain amount of extra time allocated for recovery from metastability, it is possible to calculate the corresponding MTBF. Or vice versa.

8.3.2 Synchronization

Figure 8.8 shows different ways of dealing with an asynchronous input signal. In figure 8.8(a) nothing is done, the input is just fed to the logic in the clocked system. In figure 8.8(b), the asynchronous input is synchronized using one synchronizer flip-flop, and in figure 8.8(c), a pair of synchronizer flip-flops.

The solution in figure 8.8(c) using a pair of flip-flops is the “standard solution.” The first flip-flop, $S1$, may go metastable quite often, but only if it stays metastable for an entire clock period will the second flip-flop, $S2$, go metastable and cause problems. If the period of the clock signal is t_{Clk} , the clock-to-Q propagation delay of a flip-flop is $t_{cQ,prop}$ and the setup time of a flip-flop is t_{su} and if we do not want synchronizer flip-flop $S2$ to go metastable, then the time available for flip-flop $S1$ to recover from metastability is:

$$t_r = t_{Clk} - t_{cQ,prop} - t_{su} \quad (8.8)$$

As an example, we consider a circuit with a clock period of 1.0 ns (corresponding to a frequency of 1.0 GHz), and an asynchronous input signal that switches with an average frequency of 1.0 MHz. If we assume $t_{cQ,prop} = 100$ ps,

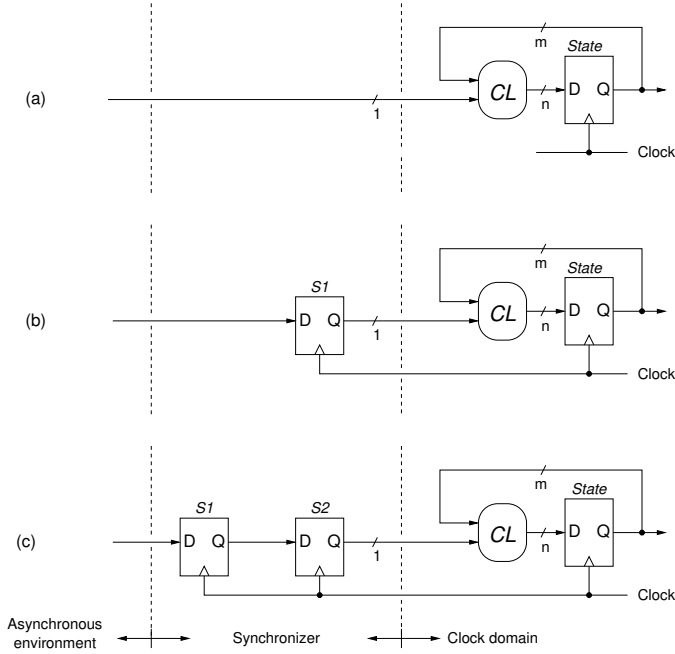


Figure 8.8: Different synchronizer organizations: (a) No synchronizer. (b) A single flip-flop synchronizer. (c) A “standard” synchronizer using a pair of flip-flops.

$t_{su} = 100\text{ ps}$, $\tau = 10\text{ ps}$ and $\Delta = 20\text{ ps}$, then the MTBF when using a pair of synchronizer flip-flops can be calculated as follows:

$$\begin{aligned}
 MTBF &= \frac{e^{t_r/\tau}}{\Delta \cdot f_{Data} \cdot f_{Clk}} \\
 &= \frac{e^{800/10}}{20\text{ ps} \cdot 1,0\text{ MHz} \cdot 500\text{ MHz}} \\
 &= 2.5 \cdot 10^{30}\text{ sec} \\
 &= 8.0 \cdot 10^{22}\text{ years}
 \end{aligned} \tag{8.9}$$

In situations where the clock signal has a low frequency, it may be OK to use just a single flip-flop for synchronization, as shown in Figure 8.8(b). In this case, the time for recovery from metastability is reduced by the propagation delay in the combinational logic, CL : $t_r = t_{clk} - (t_{CL,prop} + t_{cQ,prop} + t_{su})$. Using $f_{Clk} = 500\text{ MHz}$, $t_{CL,prop} = 500\text{ ns}$ and the flip-flop timing parameters from before we (again) get $MTBF = 8.0 \cdot 10^{22}\text{ years}$.

The MTBF can be made arbitrarily high by extending the circuit in figure 8.8(a) with more flip-flops. For example, the MTBF for a three-stage

synchronizer (allowing two cycles for recovery) becomes:

$$MTBF = \frac{2 \cdot e^{t_r/\tau}}{\Delta \cdot f_{Data} \cdot f_{Clk}} \quad (8.10)$$

Finally, figure 8.8(a) shows a flawed circuit that completely ignores synchronization. Not only may the register marked *State* go metastable quite often. It is also very likely that this causes the combinational logic, marked *CL*, to compute inconsistent output signals. An example is a state transition that flips the value of multiple flip-flops in the state register. In addition to the frequent occurrence of metastability, this may cause inconsistencies and, most likely, a faulty behavior.

In section 8.4 we will address how to perform synchronization when data is communicated from one clock domain to another, but first, it is time to address a more fundamental issue and distinguish between time-safe and value-safe systems.

8.3.3 Time-safe and value-safe systems

The previous two subsections showed two basic ways of dealing with metastability: A mutex always produces a valid digital output, but its propagation delay is unbounded.

A synchronizer, on the other hand, always responds after a fixed time, but there is a non-zero probability that its output is a non-digital value.

In general, when a decision has to be made and metastability is unavoidable, the resulting systems are either time-safe or value-safe [19] depending on how metastability is dealt with:

Time-safe systems are systems that guarantee to produce a result in bounded time, but where there is a non-zero probability that the result is not a proper digital value.

Value-safe systems are systems that guarantee to produce a proper digital result, but where there is no upper bound on the time it takes.

The distinction between value-safe and time-safe systems is a fundamental one, but in practice, both may lead to the same fault/problem: An airplane may crash into a mountain if a binary obstacle detecting device responds in time but with a metastable sensor reading (“maybe”), or if the sensor is too late at producing a binary reading. However, there is one important difference. When metastability is unavoidable, a value-safe organization may be preferable over a time-safe organization. This is because the time it takes for a value-safe circuit to compute a result is only prolonged by the time that the mutexes and arbiters in the circuit actually spend being metastable. In contrast, in a time-safe system, extra time is allocated for synchronization every time a signal is synchronized.

If we revert to our example with the 100 meter runs: If you are allowed to look at each picture for as long as you need to make a decision – at which point you ask to see the next picture – then you have a value-safe setup (and a setup that mimics an asynchronous circuit): You always identify the winner, but the time you take is unknown. There is no upper bound on how long you take to process a set of pictures. In an alternative time-safe setup, you are shown new pictures at a fixed rate. Here you may not have decided when shown the next picture, but the time taken to process a set of pictures is constant. This situation resembles a synchronous circuit with a free-running clock. If your task is to process a set of 100 pictures, and if you are assigned a certain time for the job, then you may do a better job with a value-safe arrangement than if you operate with a time-safe arrangement. The reason is that you only take longer time when metastability actually occurs. You may still have some pictures left unprocessed at the end of the assigned time interval, but it is far fewer than the number of pictures where you fail to declare a winner in a time-safe arrangement where you spend exactly one-hundreds of the allocated time processing each of the 100 pictures. If metastability does not occur, you waste time waiting for the next picture.

Finally, we mention that asynchronous systems are typically value-safe and that synchronous systems involving arbitration or synchronization are usually time-safe because of the free-running clock. However, if the clock generator being used allow individual clock periods to be stretched by an arbitrary amount of time when metastability occurs, then it is possible to build value-safe clocked systems. We address this later in sections 8.4.4 and 8.4.5.

8.3.4 Additional comments and a word of warning

The cross-coupled inverters in figure 8.2(a) represent a quite accurate model for analyzing a mutex coming out of metastability. For a flip-flop, that is composed of a master latch and a slave latch, i.e., two bi-stables, it may be too simplistic. However, experience and analysis show that if the allowed recovery time (t_r) is not too small (which you anyway do not want), then the model is still a reasonable “fit.”

As the MTBF is sensitive to the value of τ , this parameter must be evaluated conservatively to be on the safe side. In order to make τ small, the bistable circuitry in mutex'es and in flip-flops have to be designed with a high gain in the feedback loop, with fast low V_T transistors, and with minimum parasitic capacitance on the two internal nodes. For this reason, flip-flops designed for use in synchronizers are different from flip-flops used for storing data, and some cell libraries contain both types of flip-flops. Finally, we observe that some data flip-flops are designed using pass-transistors and weak feedback bleeder-inverters. These have a small loop gain and are not suited for use as synchronizers.

So far, we have only discussed metastability related to an incoming data

signal relative to a clock signal. Metastability can also occur when a reset signal is de-asserted. Reset is typically an asynchronous input produced from a push-button. Flip-flops may have synchronous reset or asynchronous reset. In both cases, a decision has to be made: was reset de-asserted before or after the clock? Therefore, flip-flops with synchronous or asynchronous reset, are all vulnerable to metastability. For this reason, the reset signal to a flip-flop must always be synchronized to the clock. And as is the case for data signals, the reset signal should only be synchronized once. If not, counters in different modules may not show the same values.

Finally, you should keep in mind that there is no way that a simulation can be used to verify that a system containing synchronizers always works correctly. You have to have faith in your design and the analysis done – be a careful and conservative skeptic.

8.4 Synchronization in multi-clock systems

For many reasons, it is preferable to organize large and complex systems as a collection of individually clocked modules and to use asynchronous communication among these modules: It simplifies clock distribution and timing closure, and it enables voltage and frequency scaling of individual modules. The term globally-asynchronous locally-synchronous (GALS) is widely used to denote such systems, and there is a rich literature on the topic. In the following, we address the fundamentals. For the interested reader, we provide pointers to additional literature [105, 19].

8.4.1 A simple handshake interface

A 2-phase or 4-phase bundled data handshake channel connecting two independent clock domains can be implemented by synchronizing the incoming handshake signals to the local clock. Figure 8.9 shows an implementation of a push channel where the synchronization of a signal is done using a pair of flip-flops.

A novice into synchronization and metastability may think that the data signals should also be synchronized. This is a huge mistake! When flip-flops become metastable, the output eventually and randomly becomes 0 or 1. This happens for each bit and consequently may result in corrupted data. The transmitter may assert the request signal and provide valid data in the same cycle, but as the synchronizer delays the request signal, the data has been stable for more than a clock period when the receiver sees the synchronized request signal. A state machine in the receiver can thus simply produce a clock-enable signal for an input register, or it can postpone the acknowledgment and directly use the incoming data.

All the channel types (nonput, push, pull, and biput) introduced later in section 10.1.1 can be implemented using variations of the circuit in figure 8.9.

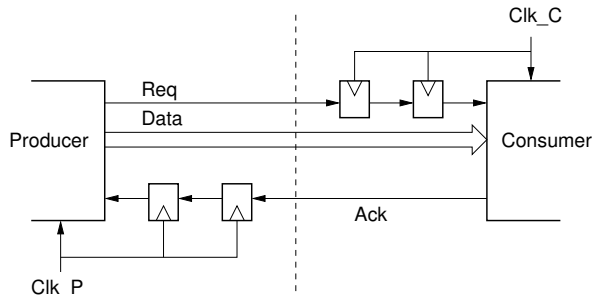


Figure 8.9: A bundled-data channel connecting two clock domains. Signals *Req* and *Ack* are synchronized.

The design in figure 8.9 is simple, reliable, and widely used, but because every transition of a handshake signal is synchronized, the maximum throughput is in the order of one data item for every 4-6 clock cycles. This assumes that the clock frequencies of the transmitter and the receiver are roughly the same, and whether it is 4 or 6 cycles depends on the implementation of the state machines (Mealy or Moore).

To increase the throughput, the number of signal transitions/events that need synchronization must be reduced. This can be done in a number of ways. The simplest is to increase the bit-width of the data. An extension of this is to introduce multiple buffers allowing the transmitter and receiver to operate in parallel. One option is to use two buffers in an alternating fashion.

A more advanced setup using three buffers can completely hide the synchronization latency. The transmitter can be filling one buffer, the receiver can be emptying another buffer, and in parallel with this, ownership of the third buffer may be transferred from the transmitter to the receiver or from the receiver to the transmitter. The transfer of ownership requires synchronization of signals. A straightforward design use a pair of handshake signals per buffer. Alternatively, it can be noted that the buffers are used in a cyclic fashion, and that the request and acknowledge signals in figure 8.9 can be interpreted as “I have filled a new buffer” and “I have emptied oldest buffer”. As is often the case, this organization is sacrificing increased latency in return for improved throughput.

8.4.2 Using a dual-ported memory

A simple and widely used approach is a dual-ported (or multi-ported) random access memory that allows individual clocks on the different ports. The block RAMs found in most FPGAs support this. As long as a write operation to a memory location does not collide with another read or write to the same address, everything is fine. Coordination among different clocked modules

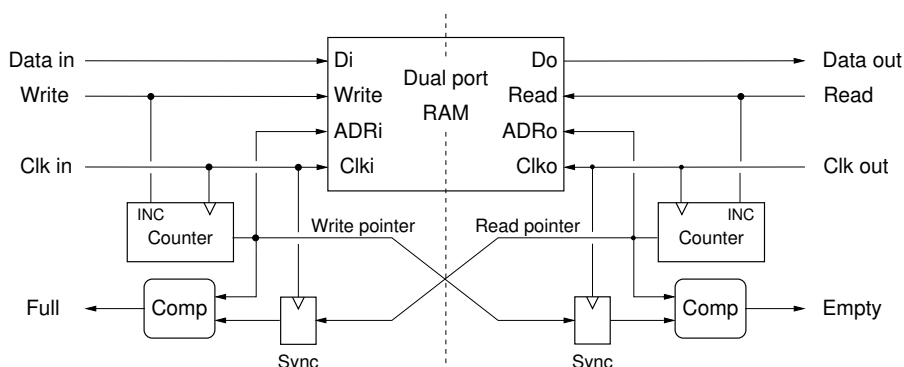


Figure 8.10: A dual-clock FIFO built using a dual-ported memory and read and write pointers.

accessing the memory can be enforced by separate flags or handshake signals that must be synchronized, as described in the previous subsection.

8.4.3 Using a dual-clock FIFO

Another way of reducing the number of synchronization events is to use a dual-clock FIFO queue. A shallow FIFO can be implemented using a small set of discrete registers. A FIFO with more capacity can be built using a random access memory accessed using read and write pointers. The pointers wrap around and the memory is used as a circular buffer. The read pointer points at the memory address holding the oldest data, and the write pointer points to the first free memory location. Figure 8.10 shows the interface and the implementation of such a dual-clock FIFO.

The write port of the FIFO, including the write pointer, uses the clock of the transmitter, and the read port, including the read pointer, uses the clock of the receiver. To produce the signals *Full* and *Empty*, it is necessary to communicate the value of the read and write pointers to the opposite clock domain. As the pointers always increment (occasionally wrapping around), and assuming that the capacity of the FIFO is a power of two, it is possible to use Gray-encoding of the counters. This ensures that from one count to the next, only a single bit changes. In this case, it is safe to pass the count value through a synchronization register. By doing this, and by using only a single layer of synchronization (as in figure 8.8(b)), it is possible to reduce the latency for producing the signals *Full* and *Empty*.

The design in figure 8.10 is so widely used that in some FPGA chips, every block RAM comes with dedicated circuitry implementing address pointers, the necessary synchronization circuitry, and the full/empty detection logic.

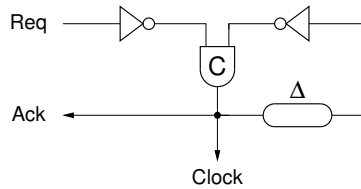


Figure 8.11: A ring-oscillator-based clock-generator that allows stretching of individual clock periods as well as stopping the clock completely during certain time periods.

8.4.4 Value-safe clocking with metastability

The MUTEX element in 8.7 consists of a bi-stable circuit and a metastability filter. If we add a similar metastability filter to the outputs Q and $\bar{Q}$ of a D-flip-flop, it becomes possible to detect when the flip-flop is metastable. This information can be used to stretch those clock-periods where metastability occurs. Such an arrangement is a value-safe clocked system. This idea was proposed in [124] and has been explored in many subsequent works. A more recent paper [105] contains a unified presentation of a range of systems using stretchable and stoppable clocks that we adopt here.

Figure 8.11 shows a fundamental circuit that can be used to design systems with stretchable and stoppable clocks. The right-hand side is a ring oscillator (a delay element and inverter) extended with a C-element. The C-element allows a left hand environment to control the operation through signals Req and Ack . The C-element waits for both its inputs meaning that if the left-hand environment is fast, the circuit works as a free-running oscillator with a duty cycle of 50% and a period of two times the delay of the delay element, the inverter, and the C-element. This could be implemented by connecting Ack to Req through an inverter. It is interesting to note that the resulting circuit is identical to the circuit in figure 2.15.

If the Req input to the asymmetric C-element in figure 8.11 is the last input to switch, then the current clock-period is extended until Req transitions. This can be exploited to occasionally stretch a clock period in case one or more flip-flops in the circuit go metastable. In this way, the next clock pulse can be postponed until metastability has ceased. The result is a value-safe clocked circuit.

Figure 8.12 shows an arrangement performing value-safe synchronization of an asynchronous input. The signal Met from the metastability detector is asserted when the first synchronizer flip-flop, $S1$, is metastable, and it is de-asserted when metastability has ceased. Note that it is the rising transition of the clock signal that is controlled. This means that only the phase where the clock is low is stretched.

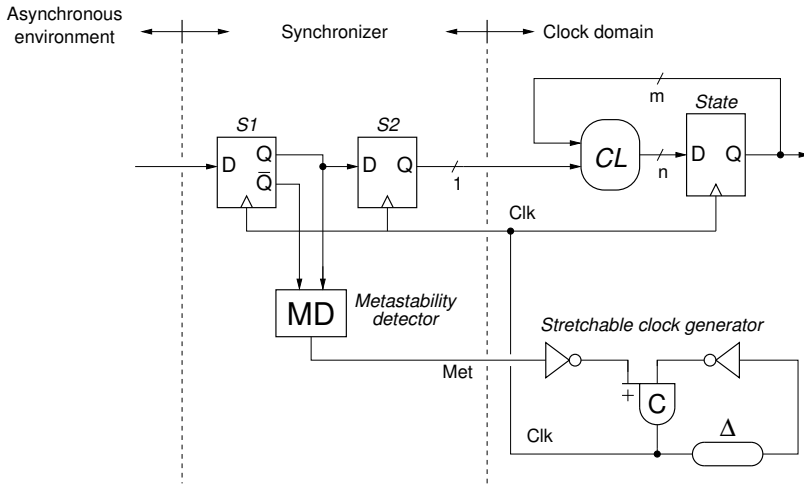


Figure 8.12: Value-safe synchronization using a metastability detector and a clock generator capable of stretching the low phase of the clock.

8.4.5 Value-safe clocking without metastability

In some situations, we can control a stretchable clock generator, like the one in figure 8.12, using circuitry that does not involve any metastability.

Metastability is related to making a binary decision based on two or more unrelated signals. In many situations, a synchronous module is only waiting for a single input signal to transition. In a system with a free-running clock, this leads to the question: “*did the input transition before or after the clock?*”? As we know by now, this may cause metastability. If it is possible to stretch individual clock periods, for example by using the stretchable clock generator from figure 8.12, then the situation reduces to a “*wait for the input condition and then proceed*” – a situation that does not involve a choice and therefore avoids metastability. In [19], an early work that covers most of the fundamentals, this class of systems is called *escapement systems*, named after the mechanism in a mechanical watch where the pendulum interacts with the rest of the mechanism to allow it to advance one step.

We will use a concrete example to explain how an escapement system works. Figure 8.13(a) shows a module with an asynchronous interface that computes some function in a sequential manner. The module has an internal clock but it is not exposed to the interface. A *Start* signal indicates the validity of the input, and a *Finish* signal indicate the validity of the output. As shown, the *Start* and *Finish* signals follow a 4-phase protocol.

The operation of the module is controlled by a state machine with six states, and the state graph for the machine is shown in figure 8.13(b). The

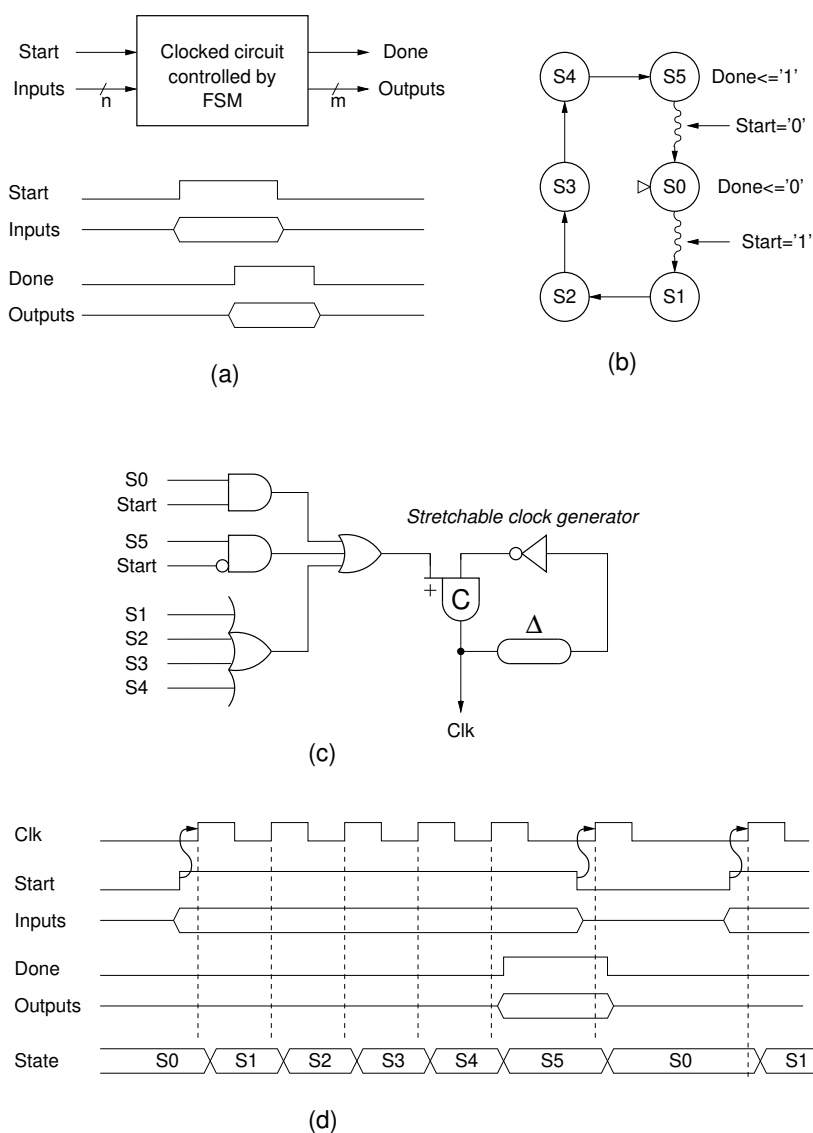


Figure 8.13: An asynchronous module implemented as an internally clocked circuit. By stretching specific clock periods, synchronization and metastability is avoided. Such an organization is called an escapement system [19].

state graph is used for illustration purposes; it has no intended use. In state S0 the machine waits for start to go high, and then it transitions to state S1. Likewise, in state S5 the machine waits for start to go low, and then it transitions to state S0. In a normal state graph, states S5 and S0 would have a self-loop that is taken when *Start* has not yet taken the value that the machine is waiting for. In the circuit we consider here, the machine will never spend more than one clock cycle in S5 and S0. Instead, as we will see below, the clock periods where the machine is in states S5 and S0 may be stretched while the machine is waiting for the start to go low or high, respectively. This is the reason for the new notation using the wave-style state-transition-arcs annotated with the condition that the machine is waiting for. This notation is from [19].

For simplicity, we assume a one-hot state encoding. Figure 8.13(c) shows how a stretchable clock generator can be controlled by the state, and the *Start* signal, and figure 8.13(d) shows a timing diagram providing details on how the circuit operates.

The example circuit in figure 8.13 has a single asynchronous input signal, *Start*. In the general case, there may be more asynchronous inputs. As long as the machine is only waiting for a single asynchronous input in any given state, there is no metastability. Hence, the circuit in figure 8.13 can easily be extended to handle more input signals or additional handshake channels.

8.5 A taxonomy of timing organizations

So far, if a system involves more than one clock, we have assumed that the clocks are uncorrelated, i.e., that input signals from the environment of a clocked module are asynchronous with respect to the clock used in the module.

In many cases, there is some relation between the different clocks, and this can be exploited and/or may need to be considered when several clocked modules are composed to build larger systems. The following classes of timing organizations can be identified [98]:

Synchronous: Systems with a single global clock that is available everywhere without any skew.

Mesochronous: Systems with a single clock-oscillator where different branches of the clock distribution network have different delays causing some phase difference among the different instances of the clock signal. As transmitters and receivers operate at the exact same rate, flow control is not needed. Some works [29, 149] assume a “fixed” or “static” phase difference. A more general assumption is that the phase difference is unknown but upwards bounded. Solutions typically involve delay elements used to align the data and the clock, or “time elasticity” implemented using dual-clock FIFOs.

Plesiochronous: Systems where each clocked module has its own clock oscillator. All the clock oscillators have the same nominal frequency, but there may be a small frequency mismatch, such as a few parts per million.

An example is several clock oscillators built using crystals with the same nominal frequency. In a linear pipeline-arrangement of clocked modules, the frequency difference can be dealt with by adjusting the phase of the clock and by occasionally dropping or adding bits at specific points in the data-stream, typically between data-frames.

Periodic: The periods of the clocks have some mutual ratio, and this knowledge can be exploited to identify where it is safe to transfer data and avoid metastability.

Asynchronous: The clocks in the different modules are independent and uncorrelated.

8.6 Examples of timing organizations

To illustrate the fundamentals introduced in this chapter, this section provides examples of synchronization and timing organizations: (i) a bit-serial plesiochronous communication interface, (ii) several FIFO-based mesochronous communication links that completely avoid metastability, and (iii) clocked circuits achieving better than worst-case performance (including both time-safe and value-safe solutions).

8.6.1 Plesiochronous bit-serial communication

In the field of telecommunication, bit-serial transmission of data is widely used. Ideally, the clocks in the different networked units should have the same frequency, but because each unit has its own clock oscillator, the whole system is plesiochronous. The connection between two plesiochronous modules is usually a bit-serial link without an associated clock signal. Instead, by using some form of encoding of the data, the receiver can regenerate the clock from the incoming signal. An example of such an encoding is an 8 bit - to -10 bit code where a byte of data is converted into a 10 bit symbol. The 10 bit symbols are encoded in a way that ensures an almost equal number of 0s and 1s in a 10 bit symbol (resulting in zero DC voltage on the cable) as well as frequent switching between 0 and 1 allowing a clock recovery circuit, typically a phase-locked loop (PLL), to regenerate the senders clock.

In this way, the input side of a plesiochronous module operates using a regenerated version of the sender's clock, and somewhere inside the module, data has to be transferred to the domain using the locally generated clock.

The frequency of the bit serial communication is often very high, making it impossible to synchronize and implement clock domain crossing of individual

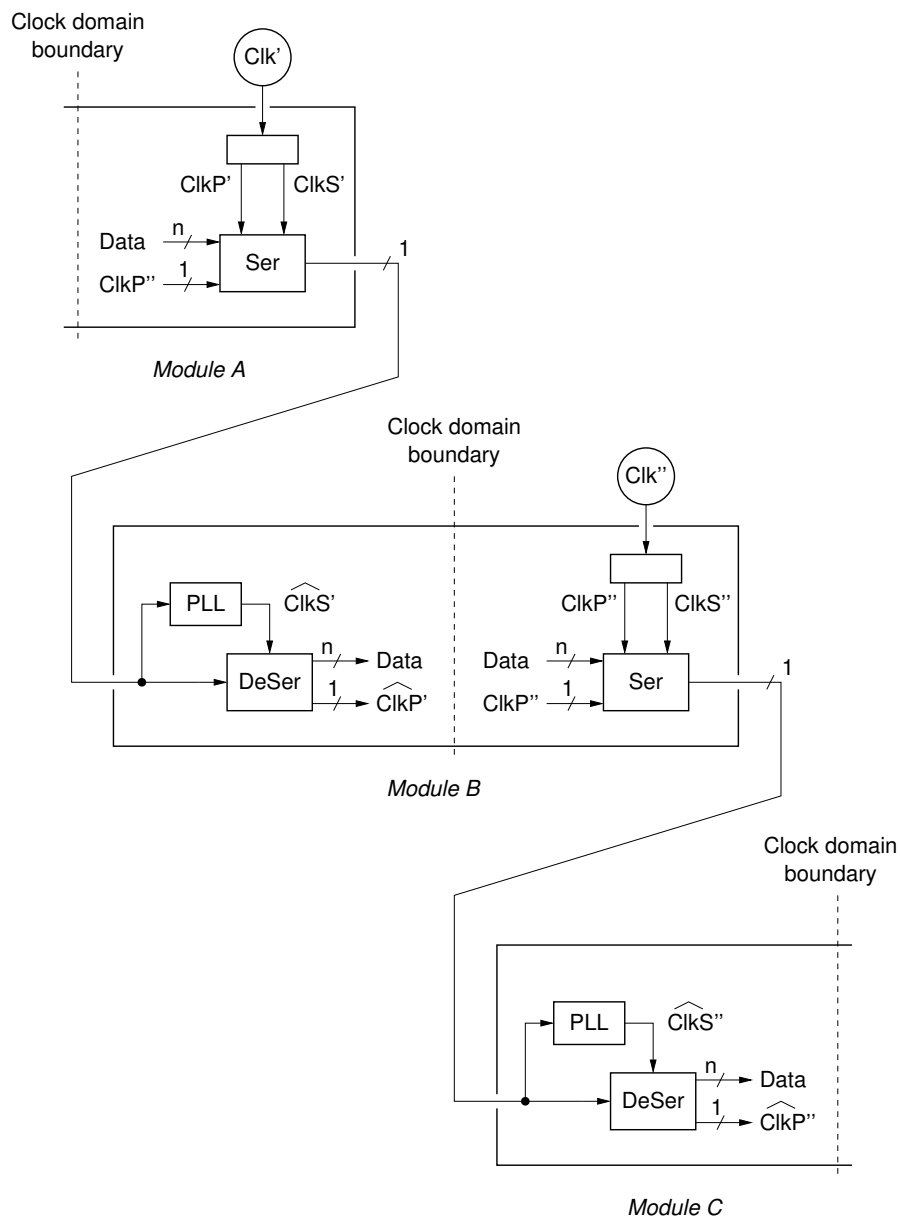


Figure 8.14: A number of plesiochronous modules (A, B, and C) connected in series. Module B is shown in its entirety. For modules A and C, only the part interfacing with module B is shown.

bits. An alternative possibility is to synchronize data after the 10 bit-to-8 bit decoder. If this is still too fast, larger blocks of data must be synchronized as a single atom.

Figure 8.14 shows an example with three plesiochronous modules connected in series. Module A is driven by clock Clk' . From this, clocks $ClkS'$ and $ClkP'$ are generated. $ClkS'$ is used for the high speed serial transmission. $ClkP'$ has a frequency that is n times lower than $ClkS'$ and is used by the part of the circuit that operates on parallelized data.

Similarly, module B is driven by clock Clk'' . The PLL in module B extracts the clock from the incoming bitstream. This extracted version of $ClkS'$ is called $\widehat{ClkS'}$. This extracted clock is used by the de-serializer that produces data in parallel form along with an extracted version of $ClkP'$ called $\widehat{ClkP'}$. The parallelized data is then transferred from the $\widehat{ClkP'}$ domain to the $ClkP''$ domain. The details of this are not shown in the figure. A possible solution is to use a dual-clock FIFO supplemented by the ability to drop or insert data in-between frames of data, as explained previously. Further details are beyond the scope of this text; our aim here is only to illustrate the basics of a plesiochronous organization.

8.6.2 Mesochronous communication links

If the phase difference is constant, the incoming data can be delayed such that it satisfies the setup and hold times of the register receiving the data. For a wide datapath, it may become too expensive to delay all the data signals, and an alternative solution that delays the clock signal instead may be preferred. More advanced solutions use variable delay elements in combination with some control that dynamically adapts to the phase of the incoming data. For examples of such circuits, the interested reader is referred to [29, Section 10.3.1]. Some of these adaptive solutions involve metastability, and therefore cannot be 100 % reliable.

It is possible to avoid metastability by using a FIFO to provide time-elasticity between a transmitter and a receiver. Figure 8.15 illustrates this. As we saw in section 8.4.3, the generation of the full and empty signals may involve metastability. However, if the FIFO is initialized to be half full, and if the mesochronous transmitter and receiver writes and reads a data item in every clock cycle, then the FIFO remains roughly half full. A phase difference between the transmitter and the receiver will fill or drain the FIFO relative to the initial state. If the phase difference is upwards bounded, it is possible to select the capacity of the FIFO such that it never runs full or empty. In this case, the full and empty signals remain un-asserted, and data can be streamed one item per clock and without metastability. In general, the FIFO need only have a capacity of 3-5 words to absorb a phase difference of half a clock period, and the RAM and pointer-based implementation shown in figure 8.10 may be too expensive. More efficient implementations of shallow FIFO's are presented

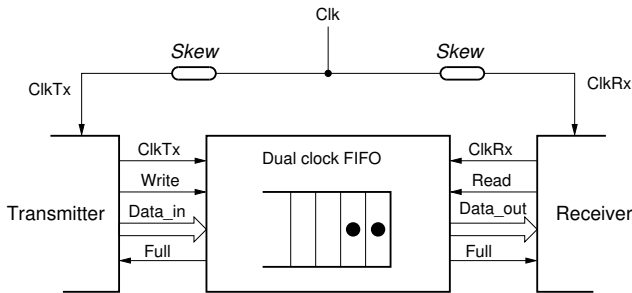


Figure 8.15: A dual clock FIFO can be used to connect a mesochronous transmitter and receiver. If it initialized half full, the FIFO can absorb a certain phase difference without becoming completely full or empty.

in [121, 165, 1].

An interesting alternative is to replace the dual-clock FIFO in figure 8.10 by an asynchronous FIFO. This idea was first presented in [54, 56], where the arrangement is called STARI (self-timed at receivers input). Figure 8.16(a) shows an implementation using a 2-phase bundled-data FIFO (the divide by two circuits toggles the request and acknowledge signals when the clock signal makes a rising transition). The key idea in the design is to avoid metastability by making two timing assumptions: (1) the time for a request-acknowledge handshake on the ports of the FIFO is always shorter than the clock period, and (2) the phase difference between the transmitter and the receiver is upwards bounded such that the FIFO never runs full or empty.

As seen in figure 8.16, the clock signal of the transmitter is used to drive the request signal on the input port of the FIFO, and the corresponding acknowledge signal is ignored based on the assumption it precedes the next clock tick and the fact that the FIFO is never full. In a similar way, the clock of the receiver is used to drive the acknowledge signal on the output port of the FIFO. And as seen, the corresponding request signal is ignored based on the assumption that it precedes the next clock tick, and based on the fact that the FIFO is never empty. The FIFO is assumed to use 2-phase handshaking, and the “:2” components toggle their outputs every time the clock makes a rising transition.

It is crucial that the timing assumptions regarding the speed and capacity of the FIFO are thoroughly verified. A timed-transition Petri net model of the STRAI communication link is analyzed in [66]. This analysis is somewhat involved. In the following, we offer an alternative analysis offering more intuitive insights.

Firstly, we note that the mesochronous environment of the FIFO means that the FIFO can be understood as a shift register or ring (as analyzed in section 4.3). A FIFO or ring has minimum handshake cycle-time when the num-

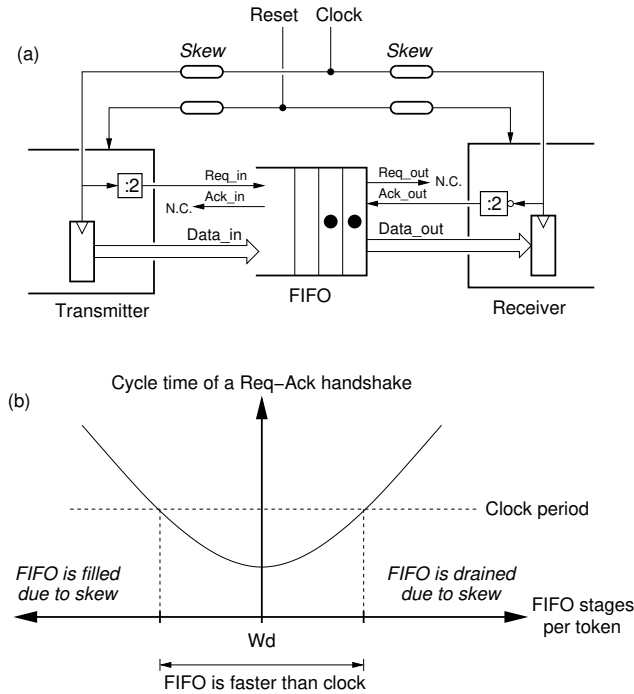


Figure 8.16: (a) An asynchronous FIFO used to connect a mesochronous transmitter and receiver. If initialized half full, the FIFO can absorb some phase difference between the transmitter and the receiver. (b) The design assumes that the cycle-time of a handshake is smaller than the period of the clock.

ber of stages per data item (token) matches the dynamic wavelength (W_d). If the phase difference is such that the transmitter clock is ahead of the receiver clock, then the FIFO fills and operates in the bubble-limited region. Likewise, if the phase difference is such that the receiver clock is ahead of the transmitter clock, the FIFO drains and operates in the data-limited region. Both scenarios cause a slowdown of the FIFO. This is illustrated in figure 8.16(b). The maximum skew that the circuit can tolerate is when the cycle-time of a handshake becomes as large as the period of the clock (with some appropriate safety margin).

The rings we analyzed in section 4.3 had an integer number of stages and an integer number of tokens. A FIFO embedded in a mesochronous environment also has a fixed number of stages. But the number of tokens is not limited to integer values, because the phase difference between the transmitter and the receiver can be arbitrary.

Finally, a comment on reset and initialization: After reset, the transmitter first inputs a number of empty or invalid data items in order to bring the FIFO to a state where it is half full. From this point, both the transmitter and the receiver are active. In order to implement this, the transmitter and the receiver may use counters that are reset by the reset signal. This means that the “phase difference” mentioned above is the phase difference between the counters in the transmitter and the receiver. This difference is created by clock skew as well as reset skew, and we note that the counters could potentially be off by more than one count.

8.6.3 Better than worst-case clocked circuits

To provide more insight into time-safe and value-safe designs, we briefly look at techniques that have been proposed to *overclock* circuits by exploiting the fact that the actual case latency of a (combinatorial) circuit is typically much lower than the worst-case latency. An early and well-known example of work in this direction is Razor [40] that we discuss here. Later improvements and extensions are “Razor II” [31], “Bubble Razor” [42], and SafeRazor [17].

The key idea in Razor, as well as in other so-called *better than worst-case schemes*, is to clock a given circuit faster than what a worst-case critical-path analysis would allow and to combine such over-clocking with a mechanism that is capable of detecting when a timing error occurs, and capable of performing some rollback-and-fix operation that can be implemented in hardware or software or a combination of the two. If this corrective measure – which may consume one or more extra clock cycles as well as some corresponding additional energy – is not performed too frequently, then there is a net gain. This gain can be used to achieve a faster design or to scale the supply voltage and save power. The scaling can be performed adaptively until the number of errors increases to a point where it starts to become costly in terms of energy and time to detect and fix errors. In this way, the circuit can reach a minimum-energy operating point.

In the original Razor I paper, this idea is implemented in a 64-bit processor, and figure 8.17 that is adapted from [17] shows the principle. The critical path in the circuit is in the *CL1*-logic and t_{CL1} denotes its actual case delay. In most cases, t_{CL1} is well below the period of the clock, t_{clk} , and the data that is clocked into *R1* is correct. But occasionally, depending on the exact operating conditions, $t_{CL1} \geq t_{clk}$. In this case, *R1* is storing a wrong value, or flip-flops in *R1* may enter a metastable state. The correct value is always available in *RA* that is clocked by a delayed copy of the clock.

Ignoring the setup and hold-times for the registers, the following timing requirements are assumed for correct operation:

$$t_{clk} + \Delta > t_{CL1,pd} \quad (8.11)$$

$$\Delta < t_{CL1,cd} \quad (8.12)$$

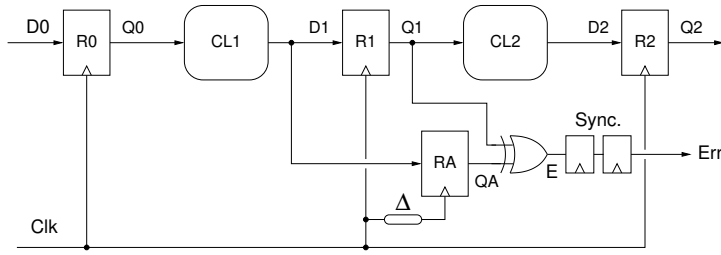


Figure 8.17: The basic principle of the Razor design.

where $t_{CL1,pd}$ is the (worst-case) propagation delay of $CL1$ and where $t_{CL1,cd}$ is the contamination delay of $CL1$.

An error situation is detected (by the XOR gate) when $R1$ and RA contain different values. As $R1$ may go metastable, its output may change at any time, meaning that also the signal E can change at any time. For this reason, E must be synchronized using a chain of two or more flip-flops. The synchronized error signal, Err , is used to perform a rollback and restart of the pipeline. In this process, the correct result in RA is used instead of the incorrect result in $R1$.

This may all sound fine, but there is a subtle issue. If the first flip-flop in the synchronizer becomes metastable, it eventually and randomly resolves to 1 or 0. This means that it is possible to have $Err = 0$ (indicating no error) even if $R1$ was metastable. In this case, $R1$ outputs corrupt/random values that propagate down the pipeline, causing the processor to compute erroneous results. In principle, this is no different from other metastability-related issues, provided of course, that the MTBF is acceptable. But it must be stressed that the essence of the mechanism is to drive the circuit towards a point where metastability happens rather frequently, and most likely, this results in an unacceptably high failure rate.

The details of how to perform the roll-back-and-restart of the pipeline, as well as low level circuit details of special flip-flops used, are beyond the scope of this text. For more information about this, the reader is referred to the original sources cited above.

Our perspective here is the fundamentals, and in this perspective, the design, as presented above, is a time-safe design; it uses a free running-clock, and it involves making a decision: to give the current computation in $CL1$ one more clock cycle or not. Therefore it cannot avoid metastability and be 100 % correct.

An alternative is to opt for a value-safe organization as presented in sections 8.4.4 and 8.4.5. In this way, the circuit can avoid making a decision: It merely waits for an event that signals that it is now safe to proceed and generate the next clock-tick. This can be implemented using a stretchable clock

generator. The signal being waited for can be a completion signal from the combinatorial logic itself, in which case there is no metastability (the organization is an escapement system), or it can be a completion signal produced by a circuit that somehow involves metastability, but *in combination with* a metastability filter, such that metastability has resolved before the completion signal is asserted.

Considering the circuit in figure 8.17, a first fix is to augment register *R1* with a metastability detector that can stretch the clock using a design similar to the one in figure 8.12 on page 159. In this way, at the end of a clock period, *R1* either contains wrong data that it safely sampled too early, or, if flip-flops in *R1* go metastable, it contains whatever random value the metastability in the flip-flops have resolved to. The point is that *R1* always contains a digitally well-defined value and that this can safely be compared to the value in *RA*, producing an error signal that does not suffer from metastability.

8.7 Concluding remarks

Clocking, metastability, synchronization, and timing organizations is a vast topic with a very large body of literature. This chapter has covered the basics. Good starting points for exploring more is [78, 52, 29, 30]. A final advice is: Be careful when dealing with these matters. It is easy to make mistakes, and it is very hard to debug and identify design errors because of the intermittent nature of the resulting system failures.

Chapter 9

Implementation of 2-phase bundled-data circuits

In this chapter, we take a closer and more comprehensive look at 2-phase bundled-data circuits. We start by reviewing a number of alternative implementations of a handshake register. Following this, we address the static data-flow structures view of circuits using 2-phase handshaking. As will become clear, there are more depths to this than one would immediately think: A change of viewpoint is required allowing us to decouple the phases of the handshake signals on the input and output channels of select handshake registers in the circuits. In the last part of the chapter, we present a new implementation-template called phase-decoupled click-elements that implements such phase-decoupling, and we present implementations of all the data-flow handshake components introduced in chapter 3, figure 3.3. As these phase-decoupled components are implemented using conventional gates and edge-triggered flip-flops only, they are suitable for prototyping of asynchronous circuits using conventional FPGAs.

9.1 Templates for implementing 2-phase handshake latches

In this section, we review a range of circuit design templates for 2-phase bundled-data pipelines: The classic Muller pipeline, Micropipelines [147], Mouse-trap [137, 138], Click elements [126] and the pipeline stage used in Intel’s Loihi neuromorphic chip [83].

This chapter contains material from [86]. © IEEE 2019. Reprinted with permission from: A. Mardari, Z.Jelčicová and J. Sparsø, Design and FPGA-implementation of Asynchronous Circuits Using Two-phase Handshaking. In. *Proc. IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 9-18, 2019.

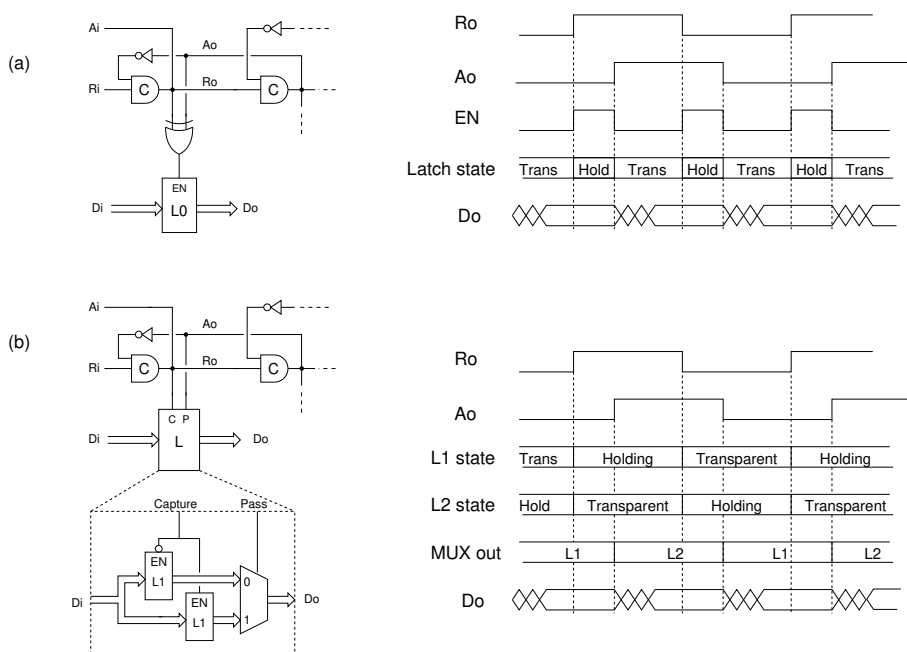


Figure 9.2: The latch-based Micropipeline 2-phase bundled data pipeline templates presented in [147].

9.1.2 Micropipelines

In his Turing award paper “Micropipelines” [147], Ivan Sutherland presents a couple of more efficient 2-phase bundled data designs that use latches in the datapath. As latches need a pulse to control the enable input, we need a circuit that can produce a pulse for every transition on the R_o signal. This can be achieved using an XOR-gate, as shown in figure 9.2(a). The leading edge of the enable pulse (turning the latch into hold mode) is produced when R_o toggles (indicating the availability of new data), and the trailing edge of the pulse is produced when A_o toggles (indicating that the downstream pipeline stage has received and stored the data). At this point, the latch becomes transparent again. Aiming to emphasize the similarities between the different bundled-data templates, the design in figure 9.2(a) omits some details of the corresponding design in [147, Fig. 14]; details that give the circuit more robust timing properties at the cost of added latency in the control circuit. The circuit uses simple enable latches in the datapath resulting in a relatively small circuit.

An alternative and faster design uses what we could call a “dual-level-sensitive latch,” i.e., a pair of latches operated in a similar way as the two flip-flops in the dual-edge-triggered flip-flop in figure 9.1(c). This circuit is

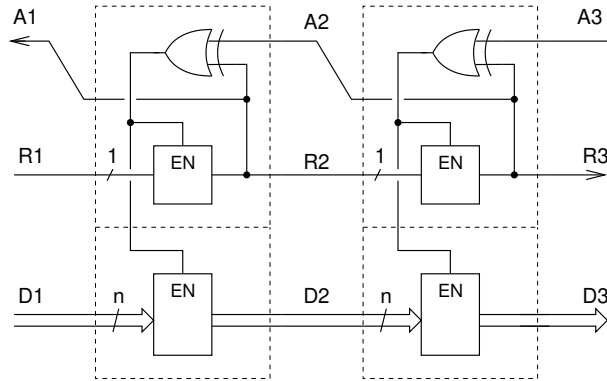


Figure 9.3: A pipeline composed of two Mousetrap-based handshake latches. The Mousetrap latch controller consisting of an XOR-gate and an enable-latch implements the same behavior as the Micropipeline latch controller in figure 9.2(a).

shown in figure 9.2(b). Again the schematic omits some elegant details of the corresponding design in [147, Fig. 12] in order to emphasize the similarities between the different bundled-data templates. In this design, the data-path consists of two latches and a multiplexer, and in comparison with the design in figure 9.2(a), it has a large area-overhead.

Finally, we mention that the latches in the datapath of the circuit figure 9.2(a) could be replaced by flip-flops similar to the step we took going from figure 9.1(a) to figure 9.1(b).

Sutherlands Micropipelines paper [147] also provides implementations of a number of additional components, for example, the toggle that we will use in section 9.4.7, and it includes a design example – a pipelined multiplier.

9.1.3 Mousetrap

Mousetrap [137, 138] is a very elegant implementation of a 2-phase latch controller capable of controlling a data-path with a normally-transparent enable-latch. This means that the functionality is similar to the Micropipeline design shown in figure 9.2(a). Figure 9.3 shows the implementation of a pipeline with two Mousetrap-based handshake latches. As seen, the control circuit consists of only an ordinary XOR-gate and an ordinary 1-bit enable latch.

The name “Mousetrap” hints to the operation of the circuit: If a transparent enable-latch is seen as an open, i.e., armed mousetrap, and if incoming data and request is seen as a mouse, then the trap flaps when the mouse protrudes its head through it. And the trap opens again, when the mouse is trapped in the next downstream handshake-latch in the pipeline.

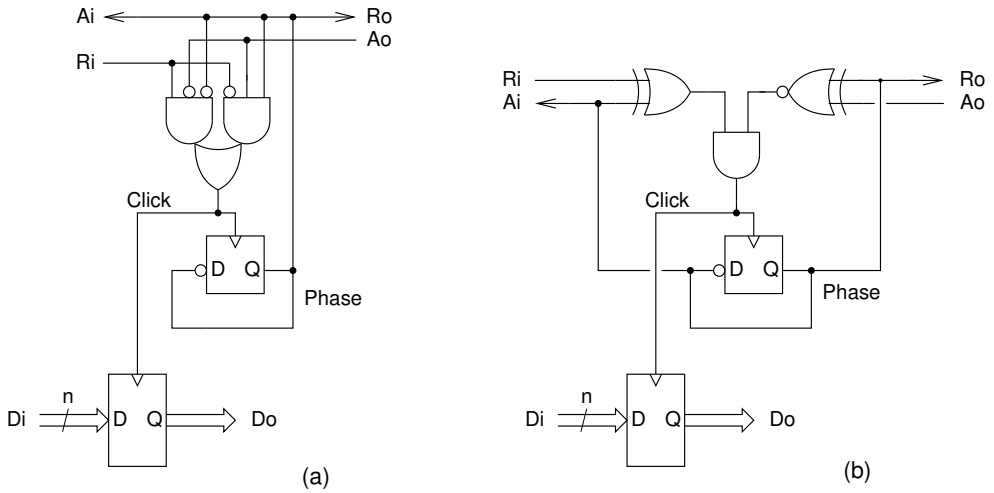


Figure 9.4: A pipeline stage (i.e., handshake latch) based on the Click-element template.

The circuit is very fast and it avoids the use of C-elements – a component that may not be available in some cell libraries. However, Mousetrap is only an implementation of a 2-phase handshake latch. C-elements are still needed to implement all the handshake components that are transparent to handshaking: join, fork merge etc. The design of these components is not addressed in [137, 138].

9.1.4 Click elements

The click template introduced in [126] is based on an idea originally described in [150]. It excels by using only regular gates and edge-triggered D-flip-flops – avoiding both C-elements and latches. There are no combinational circuit loops; all signal paths start and end in edge-triggered flip flops. According to the authors, this makes it more compliant with conventional CAD-tools for synthesis and timing analysis, and the overhead of introducing scan-chains for post-fabrication test is significantly smaller than for Micropipelines and Mousetrap.

The implementation of a click pipeline stage is shown in figure 9.4(a). It can be understood as follows: Assuming that the pipeline stage sits in an initially empty pipeline where all handshake signals are low, it captures a first data element when R_i makes a transition to 1 (i.e., when $A_o = 0 \wedge R_i = 1$). The second data item is captured after both A_o and R_i have transitioned (i.e., when $A_o = 1 \wedge R_i = 0$) and the third when again $A_o = 0 \wedge R_i = 1$. Using the output of a toggle flip-flop, the two and-gates detects the two alternating

conditions, each time producing a rising edge of a pulse on the click-signal. This clocks/toggles the phase flip-flop, and through the feedback signal to the and-gates this produces the falling edge of the pulse on the click signal. In this way, a click-pulse is produced every time a new data item has to be captured. The duration of the click-pulse is determined by the propagation delays in the AND and OR gates and the flip-flop producing the phase signal.

Figure 9.4(b) shows an alternative implementation providing identical functionality. The circuit can be understood by focusing on the channels. When a channel is conveying a token from a sender to a receiver, this is indicated by $Req \neq Ack$. And when the receiver acknowledges and the channel is no longer conveying a token, this is indicated by $Req = Ack$. The circuit in figure 9.4(b) produces a click-pulse when a new token is present on the input channel, *and* a token is no longer present on the output channel.

Implementation of click-based versions of additional handshake components (join, fork, merge, MUX, etc.) is also covered in [126]. We will get back to these in section 9.4.

9.1.5 Loihi

Intel’s neuromorphic chip named Loihi uses a 2-phase bundled-data template shown in figure 9.5 [83, 33]. It shares circuit fragments with the alternative Click template in figure 9.4(b), and with the Mousetrap template from figure 9.3. The logic producing the clock pulse is similar to the control logic in the alternative Click template, and the use of a transparent latch to propagate Req from the input to the output is similar to Mousetrap. It uses latches in the datapath, but where Micropipelines and Mousetrap implement normally-transparent handshake latches, Loihi implements a normally-opaque handshake latch. This trades some speed for a lower power consumption by avoiding propagation of glitching signals. As we observed in section 10.3, the power saving can be substantial.

Briefly, the operation of the circuit is as follows: When a data token is to be received and stored in the pipeline stage, CLK goes high, and the latch in the data path becomes transparent. After some delay (set by the PW delay element), $DCLK$ becomes high as well, and this makes the latch in the control path transparent. This allows Ri to propagate and drive Ro . At the same time, the XOR-XNOR-AND circuit drives CLK and $DCLK$ low again.

Finally, we note that the schematic in figure 9.5 shows where delay elements (PW, SETUP and HOLD) can be added, to increase the width of the latching pulse and to increase setup and hold times towards a downstream neighbor pipeline stage.

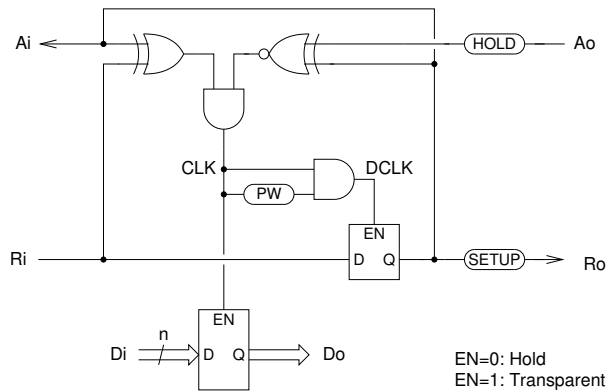


Figure 9.5: The normally-opaque two-phase bundled-data pipeline stage used in Intel’s Loihi chip.

9.2 2-phase static data-flow structures

In this section, we develop a token-bubble view of 2-phase static data-flow structures. As will be clear from the opening discussion we have to overcome some fundamental issues, abandon the wave propagation analogy, and take a new viewpoint that focuses on the state of the individual handshake channels.

9.2.1 A change of viewpoint

The handshake-latch implementation-templates presented in section 9.1 are quite different in the way they control the datapath that may use different forms of enable latches or edge-triggered D-flip-flops. Despite these differences, the handshaking they implement on their input and output ports is identical to a stage in a Muller pipeline (i.e., a C-element and an inverter).

It is well known that a ring composed of Muller pipeline stages needs at least 3 stages (C-elements) to oscillate [55, 142]. The three C-elements repeatedly cycles through the following sequence of states (010; 011; 001; 101; 100; 110)*. This oscillation can be seen as a standing wave that propagates by copying both the crest and the trough forward with the restriction that the circuit cannot enter states 111 and 000. Figure 9.6(a) shows such a 3-stage ring, and Figure 9.6(b) shows the corresponding standing wave that rotates in the ring. The ring is in state (010), and the next state is (011). The corresponding token-bubble view is different for a 2-phase design and a 4-phase design.

In a 4-phase design, see Figure 9.6(c), the ring contains a valid token, an empty token, and a bubble. The forward propagation of a valid token corresponds to the C-element in the Muller pipeline control circuit transitioning

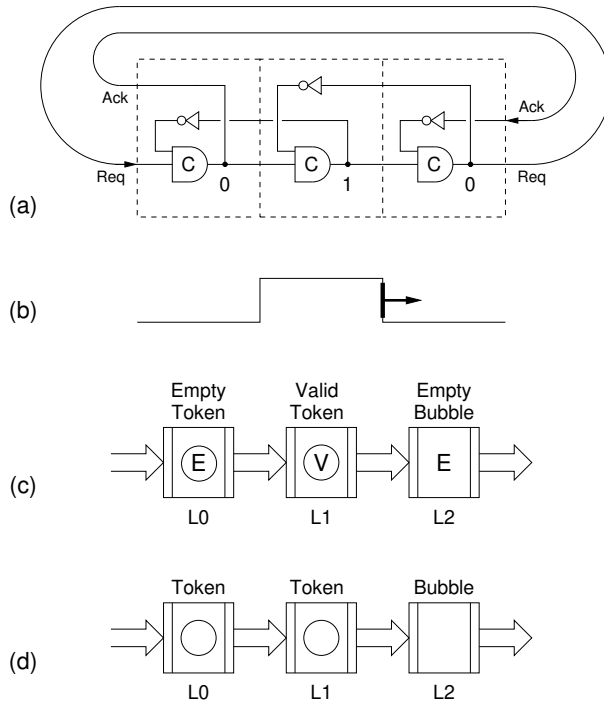


Figure 9.6: (a) A three-stage Muller-pipeline ring. (b) A snapshot of the standing wave rotating in the ring. (c) When used in a circuit using 4-phase handshaking, the ring contains a valid token, an empty token, and a bubble (valid or empty). (d) When used in a circuit using 2-phase handshaking, the ring contains two tokens and one bubble.

to '1', and the forward propagation of an empty token corresponds to the C-element transitioning to '0'. Seen this way, a 3-stage ring may contain one permanently rotating wave. Such a three-stage ring containing one valid token can be used to implement iterative computations, where the result from the current step depends on the result from the previous step. In a 2-phase design, see Figure 9.6(d), the ring contains two tokens and one bubble. The forward propagation of a token is associated with a transition (rising or falling) of the C-element in the Muller pipeline control circuit. Consequently, a three-stage ring using C-element based control circuits – or any of the other of the 2-phase templates discussed in this section – will contain two tokens and one bubble.

For rings in general, the message is that they can only contain an even number of tokens, an observation originally made in [132]. This is a severe restriction. Especially the observation that rings with a single token are not possible, as this *precludes implementation of iterative/recursive computations*

as for example the Fibonacci circuit from section 3.5.3 on page 38. The root of the problem is that in all the 2-phase pipeline templates discussed in the previous section, the request signal towards the downstream neighbor and the acknowledge signal towards the upstream neighbor is the same signal (the output of the C-element).

One way of implementing rings with a single token is to alternately propagate a rising transition and a falling transition through the ring. This could be done by inverting the request and acknowledge signals when the input and output ports of a pipeline are connected to form a ring. As static data-flow structures typically involve many coupled rings and short pipeline segments, possibly shared by several nested rings, there are severe constraints on where such inverters can be inserted and on how the circuits should be initialized.

An alternative and simpler approach [86] is to focus on the state of the individual handshake channels in the circuit and to decouple the generation of the output request and input acknowledge signals produced by a pipeline stage. In the next section, we explore and develop this viewpoint.

9.2.2 Phase-decoupled handshaking

We now abandon the “wave propagation viewpoint” in favour of a “channel viewpoint” that will allow us to design and implement rings with any number of tokens.

Conceptually, in two-phase design, there is no difference between rising and falling signal transitions. However, when it comes to implementation, the designer has to decide on the initial signal levels. We note that in data-flow circuits, all handshake channels are push channels, and in the following, we adopt two policies:

- P1: For all channels in a circuit, a transition on the request wire (signaling that the sender conveys token) is followed by a transition on the acknowledge wire *with the same polarity* (signaling that the token has been received).
- P2: All channels conveying tokens are initialized with $Req = 1$ and $Ack = 0$. All channels *not* conveying tokens are initialized with $Req = 0$ and $Ack = 0$.

Following these policies, we see that the input channel to a pipeline stage conveys a new token (from an upstream neighbor) when $In_req \neq In_Ack$ and that a token on an output channel has been received by the downstream neighbor when $Out_Req = Out_Ack$. The use of XOR and XNOR gates in the click control circuit in Fig. 9.4(b) implements this in a very explicit way. However, the function of the circuits generating the click signals in Figs. 9.4(a) and 9.4(b) are identical.

As the 2-phase static data-flow structure abstraction involves only tokens and bubbles, rings with just two stages should be possible, as should rings with

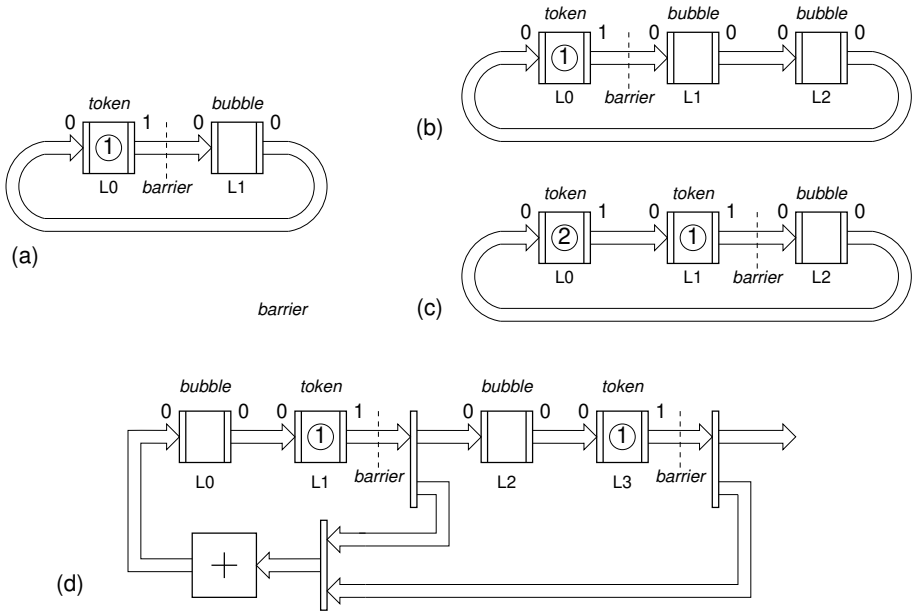


Figure 9.7: (a) a two-stage ring with one token and one bubble, (b) a three-stage ring with one token and two bubbles, (c) a three-stage ring with two tokens and one bubble and (d) a two-phase version of the Fibonacci circuit.

an odd number of tokens. To illustrate this, and to illustrate the use of policies P1 and P2, figure 9.7 shows a number of possible 2-phase data flow structures: (a) a two-stage ring with one token and one bubble, (b) a three-stage ring with one token and two bubbles, (c) a three-stage ring with two tokens and one bubble, and (d) a two-phase version of the Fibonacci circuit. The channels are annotated with the initial values of the handshake signals – request at the sender’s end and acknowledge at the receiver’s end.

Finally, a note on initialization: A circuit must be reset to a state where state-holding elements produce handshake signals with the initial values (as shown in for example figure 9.7) and where latches in the data path that initially hold tokens are set to the desired initial values. In order to safely bring the circuits out of reset and into normal operation, we introduce (where needed in order to keep the circuit “frozen” in its initial state), a *barrier* on channels that initially propagate tokens – the dashed lines in figure 9.7. These barriers are controlled by a global go-signal, and they block the request signals on the corresponding channels. In this way, reset can be de-asserted, possibly with some skew, before the go-signal is asserted, and the circuit starts operating. According to policy P2, a channel propagating a token has the request signal

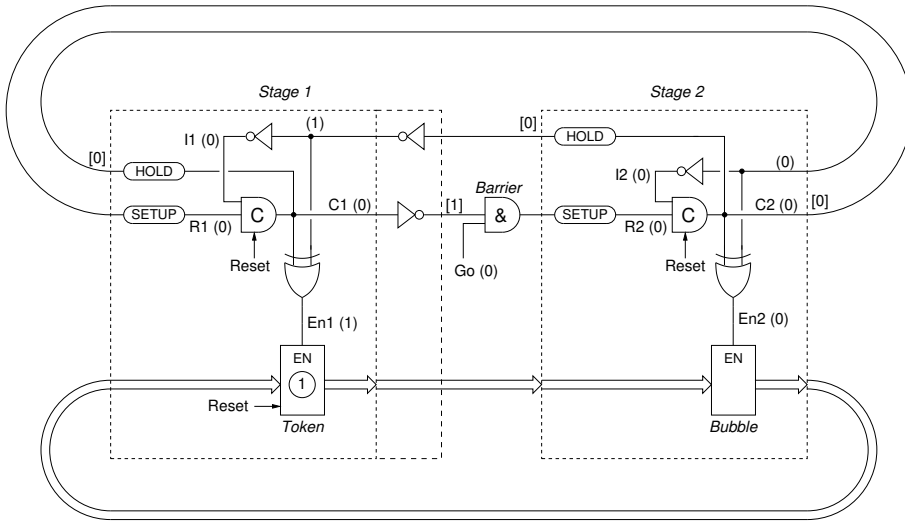


Figure 9.8: Micropipeline implementation of a phase-decoupled 2-stage ring.

set to high. This means that the barrier must output a request signal that is low. An AND-gate is used to implement this forcing to zero.

9.2.3 Phase-decoupled handshake latches

The structures shown in figure 9.7 can be built from two types of pipeline stages: those containing tokens and those containing bubbles. The latter can be implemented using the templates shown in figures 9.2, 9.3, 9.4, and 9.5, initialized such that all handshake signals and state-holding elements driving these handshake signals are initialized low. For stages holding tokens, we use the very same implementation and initialization, with inverters added to the request and acknowledge signals in the output channel, and with an and gate to implement a barrier that blocks the propagation of the token until the signal go is asserted. Figure 9.8 shows such a phase-decoupled Micropipeline implementation of the two-stage ring from figure 9.7(a). The latch in the datapath of the first stage is holding data (a token) and must be reset to contain the desired value. The latch in the datapath of the second stage is transparent.

It is interesting to note that the control circuit is delay insensitive and capable of iterating with only two stages. The reader may want to verify delay insensitivity, by showing that a corresponding circuit where all ends of forking wires are modeled by components (delay elements and inverters) is speed independent. This can be done using the tool WorkCraft (see <http://workcraft.org/>). A model of the circuit involves six signals: Ignoring inversions output signal C1 from the C-element in stage 1 is forked as signals I2 and R2, and output

signal C2 from the C-element in stage 2 is forked as signals I1 and R1.

As all the implementation styles we have described, handshakes in the exact same way they could all be used in place of the Micropipeline stages 1 and 2 in figure 9.8. There are, however, fundamental differences when it comes to controlling the latches or flip-flops in the datapath.

The Mousetrap and Micropipeline templates use transparent latches in the datapath, and there is a potential risk that the data-latches in Stage 1 and Stage 2 are both transparent at the same time. In the circuit in figure 9.8 when go is asserted, the C-element in stage 2 will switch to “1”. This causes two simultaneous events: the data latch in Stage 2 becomes opaque (holding data), and the data latch in Stage 1 becomes transparent. In order to tolerate some timing skew, it would be desirable if the latch receiving a token, becomes opaque sometime before the latch that provides the token becomes transparent. This can be controlled by increasing the delay of the delay elements marked “hold” in figure 9.8.

For Loihi and Click, the situation is more relaxed. Loihi uses a normally-opaque latch in the data-path, and the width of the pulse controlling the latch is short and determined by a delay element in the circuit itself. Click uses edge-triggered flip-flops in the data path, and these are clocked by a pulse whose width is determined by gate-delays within the circuit itself.

For Click, there is an alternative implementation offering phase-decoupled handshaking and initialization in a way that is more intuitive and simpler to use than inserting inverters in the circuit. This phase-decoupled click circuit will be introduced in the next section.

9.3 Design examples: FIB and GCD

In this section, we illustrate the use of the components and the design methodology by showing and explaining the design and implementation of two small circuits that contain multiple coupled rings and pipeline segments. Both circuits have been implemented and tested on the Nexys4DDR FPGA-board, and the code is available in a Git-repository [61].

9.3.1 Fibonacci sequence generator (FIB)

The Fibonacci circuit has no inputs; it simply computes and outputs the sequence of Fibonacci numbers (0, 1, 1, 2, 3, 5, ...). Our implementation using phase-decoupled two-phase handshake components is shown in Fig. 9.9. The figure also shows the initial state of the circuit and the use of fused components. A matched delay is only required in the function block (CL0) since the LUTs generating the click signals in the other components provide sufficient delay margins.

The circuit is initialized with a token in each of the two Register+Fork components. The design consists of two nested rings: an inner ring containing

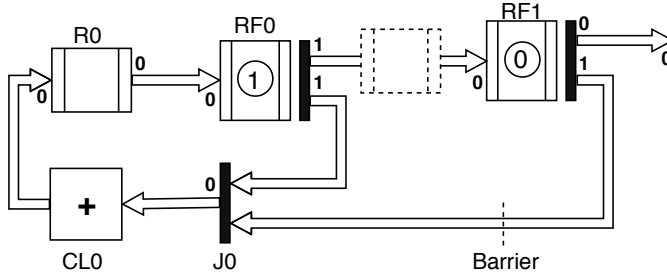


Figure 9.9: Schematic of the Fibonacci circuit. The component in dashed lines shows a more straight-forward implementation of the circuit.

$RF0 \rightarrow J0 \rightarrow CL0 \rightarrow R0$ and an outer ring containing the same components *and* handshake register $RF1$. The inner ring has two handshake registers and one token. The outer ring has three handshake registers and two tokens. By following policy P2, we can ensure the correct initialization of both rings. Notice that the schematic shows no annotations on the input channels of join $J0$, our passive join simply forwards the acknowledge signal from the output channel to the two input channels (without any buffering). This acknowledge signal is produced by handshake register $R0$.

When the `go` signal is asserted, and the barrier opens, the circuit starts: $J0$ joins the tokens from $RF0$ and $RF1$, and the resulting (single) token spreads across the $J0$, $CL0$ (the adder) and into $R0$. At the same time, the environment consumes (a forked copy of) the token in $RF1$. This spread-token operation is mentioned/assumed in [142], and studied in detail in [141]. We have chosen this design instead of the more straight-forward implementation (with an additional click stage shown using dashed lines in Fig. 9.10) to illustrate better the spread token semantics (illustrated in the Git repository [61]).

9.3.2 Greatest common divisor (GCD)

The greatest common divisor (GCD) circuit shown in Fig. 9.10 was designed after [142, Sec. 3.7] with small modifications. As we use two-phase handshaking, we need fewer handshake registers. In addition, we use a Merge instead of a MUX at the end of the if-then-else construct. The circuit is initialized according to policies P1 and P2 with a token (with value '1') in handshake register $R0$ and with bubbles in the remaining handshake registers. The circuit has no barrier since, after reset, it waits for a token on the input channel.

Notice that $R0$ has different phases on the input and output channels. This is because the ring $MX0 \rightarrow RF0 \rightarrow F0 \rightarrow R0$ has a single token. The other rings in this circuit are: $MX0 \rightarrow RF0 \rightarrow DX0 \rightarrow RF1 \rightarrow DX1 \rightarrow ME0$ and $MX0 \rightarrow RF0 \rightarrow F0 \rightarrow DX0 \rightarrow RF1 \rightarrow DX1 \rightarrow ME0$. In all of the rings, the tokens eventually get spread

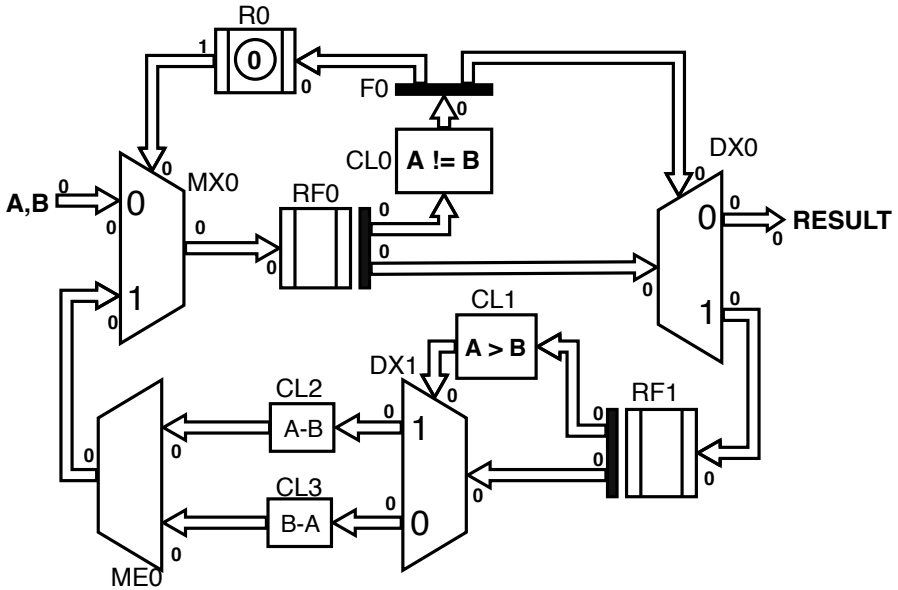


Figure 9.10: Schematic of the GCD circuit.

across several components, as seen in the step-wise illustration provided in the Git repository [61].

9.4 Phase-decoupled click components

In this section, we present phase-decoupled Click implementations of all the handshake components from figure 3.3 on page 32. The material is based on [86]. As all the components are implemented using “ordinary” gates and edge-triggered D-flip-flops, it is relatively straightforward to provide FPGA-implementations and allowing easy prototyping of asynchronous circuits.

9.4.1 The handshake latch

Instead of adding inverters to stages initially holding tokens, a second phase flip-flop is added, see figure 9.11. In this way, the request on the input channel and the acknowledgement on the output channel are decoupled and driven directly by phase-flip-flops initialized according to the values annotated to the data-flow schematics, see figure 9.7. The shaded rectangles indicate combinational logic that is implemented in a single LUT in an FPGA. Three LUTs are needed: one for computing the click-signal and two to invert the two phase-signals.

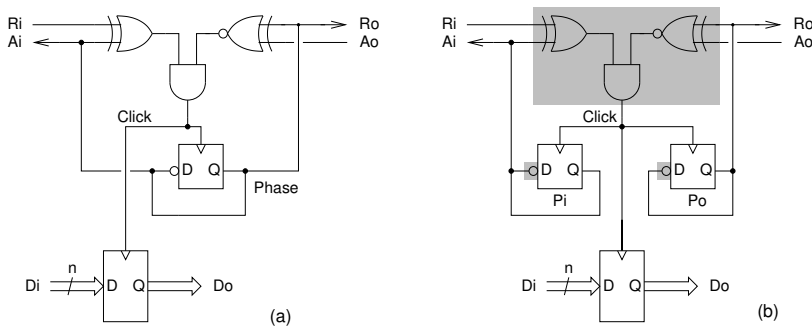


Figure 9.11: (a) The Click template from figure 9.4. (b) The phase-decoupled Click template using separate phase flip-flops to generate *In_Ack* and *Out_Req*. The two phase flip-flops, *Pi* and *Po* are initialized to produce the handshake signals with the desired initial values, e.g., the values annotated to the channels in data-flow schematics, as in figure 9.7.

The click pulse has a very short duration. If desired, the pulse-width can be increased by delaying the self-resetting of the control circuit. This can be done by adding a delay to the click signal (delaying the clocking of phase and data flip-flops) or by adding a delay after one of the phase-flip-flops (delaying *In_ack* or *Out_req*).

9.4.2 Function blocks and delay elements

A function block in a bundled data circuit is an ordinary combinatorial circuit extended with a request and an acknowledge signal. Request must be delayed for longer than the propagation delay in the combinatorial circuit. In 2-phase design, there is no return-to-zero signaling, all transitions of request are events signaling the availability of new data. For this reason, delay elements in function blocks must be symmetric delay elements.

9.4.3 Join and Fork

Simple and straightforward implementations of the join and fork components are shown in Fig. 9.12. They are textbook implementations [142, Sect. 5.2] using a click-circuit to implement the functionality of C-element.

Following design policies P1 and P2, the simple join in Fig. 9.12 can always be used. The phase flip-flop is initialized according to the state of the input and output channels. Because the component is transparent to handshaking, these are guaranteed to be in phase. Again, the shaded rectangles indicate combinational logic that is implemented in a single LUT in an FPGA.

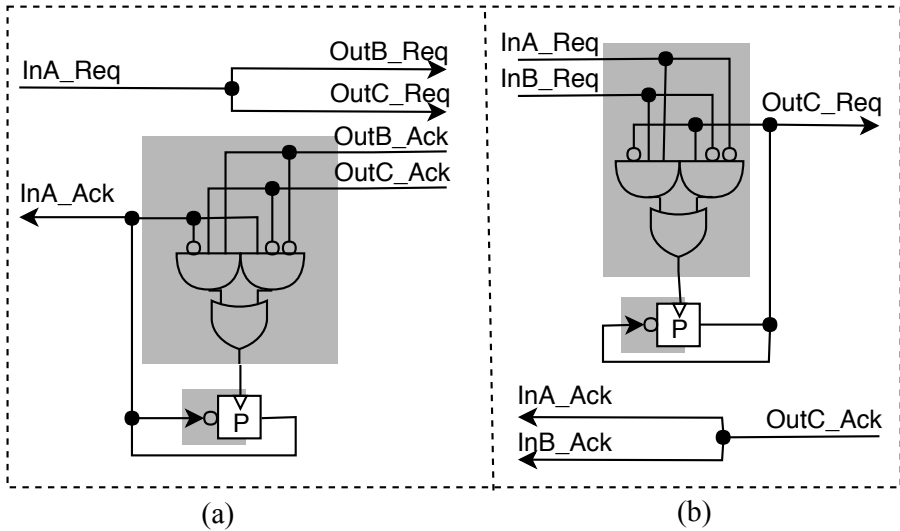


Figure 9.12: (a) Fork. (b) Join.

9.4.4 Merge

The implementation of the Merge is shown in Fig. 9.13. It assumes mutually exclusive inputs and therefore uses separate phase-flip-flops (denoted `Pa` and `Pb`) in the input ports. As the input and output phase flip-flops are clocked by separate signals, the circuit also needs a separate phase flip-flop (denoted `Pc`) in the output port.

The circuit functions as follows: A transition on either `InA_Req` or `InB_Req` asserts either `Sel_A` or `Sel_B`, and the multiplexor propagates the proper input data to the output (`Out_Data`). This also creates a rising edge on the signal `click_out`, which causes a transition on `Out_Req`. Finally, this creates a (silent) falling transition on signal `click_in`. When the right-hand environment later acknowledges by transitioning signal `Out_Ack`, this causes a rising edge on signal `click_in`. This clocks both `Pa` and `Pb` and causes a transition on `InA_Ack` if the operation of the merge started by a transition `InA_Req` or a transition on `InB_Ack` if the operation of the merge started by a transition of `InB_Req`.

This way of using a phase-flip-flop to produce an acknowledge based on the corresponding request is a small variation that we prefer instead of the clock-gating used in the buffered Merge in [126] and the plain Merge described in [81]. The problem with a gated clock is that it is produced by an AND-gate requiring the gating signal to be stable in a time window enclosing the period where the clock signal is high. Our solution avoids this timing requirement.

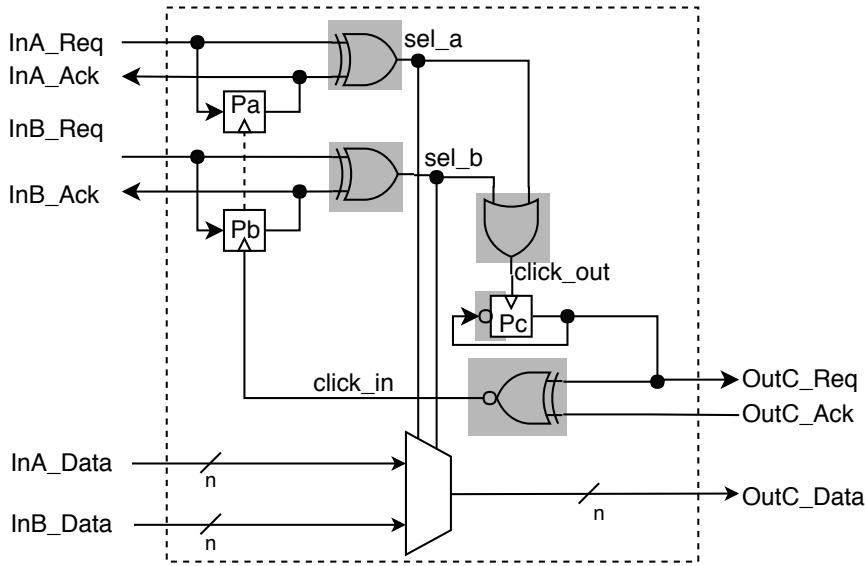


Figure 9.13: Merge

9.4.5 MUX and DEMUX

The MUX and DEMUX components are used to implement conditional flow control, and their implementation is shown in Figs. 9.14 and 9.15. The MUX has two input channels (InA, InB), a selection (input) channel for choosing between InA and InB, and an output channel (OutC). Figure 9.14 shows the implementation of the MUX. The phase flip-flops Pa, Pb and Ps are all clocked on every transition of the incoming acknowledge by the same signal derived from the function $\text{OutC_Req} = \text{OutC_Ack}$. The phase flip-flop Pc drives the request signal of the output channel (signal OutC_req), and it is toggled whenever there is a token on the selector channel and the selected input. Similar to the Merge, the MUX has phase-decoupled channels due to the nature of its function.

Figure 9.15 shows the implementation of the DEMUX (inspired by [81]). It has two input channels (InA and InSel) and two output channels (OutB and OutC). The component joins the two inputs and produces an output on the selected channel. Similar to the MUX, the DEMUX has multiple internal phase flip-flops. The phase flip-flops Pb and Pc are clocked when both request signals on the input channels transition. Phase flip-flop Pa (participating on the input channel handshakes) is clocked whenever an acknowledge is received (as indicated by the following expression: $\text{OutB_Ack} = \text{OutB_Req} \wedge (\text{OutC_Ack} = \text{OutC_Req})$). Again, we prefer this style of clocking to the gated clocking used in the components described in [126, 81].

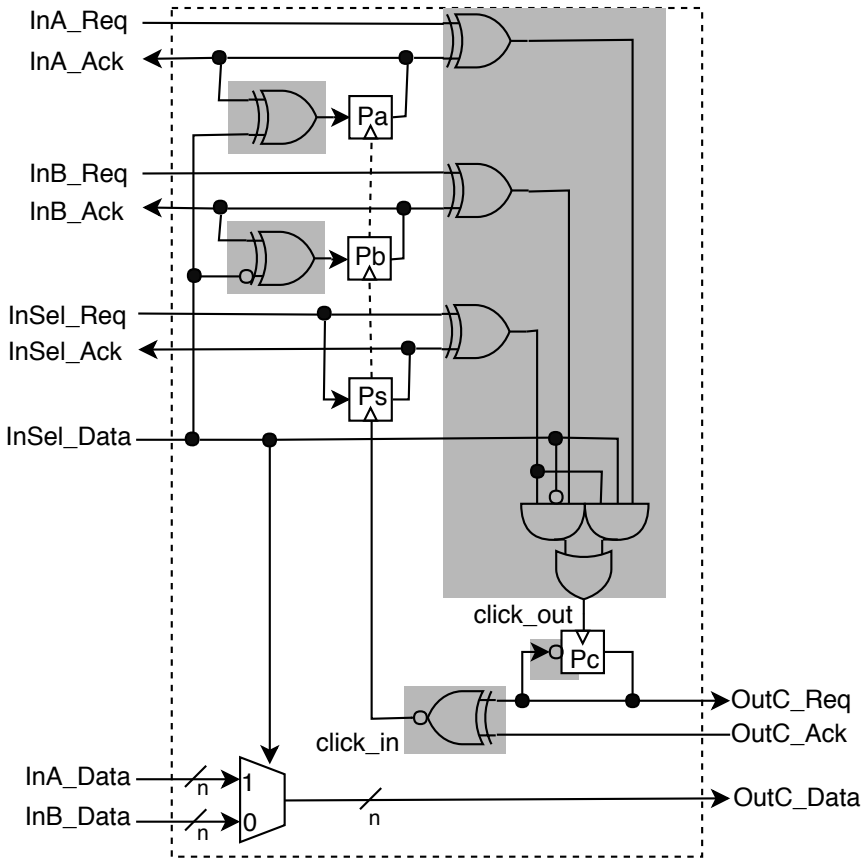


Figure 9.14: MUX

9.4.6 Peephole optimizations

It is possible to reduce the hardware cost of a circuit by performing peephole optimizations where certain combinations of handshake components are substituted by a single fused circuit. All of these optimizations involve merging handshake registers and one or more of the passive components. The original click-paper [126] showed how easy it is to extend the click-based handshake register with join-functionality on the input and fork functionality on the output. The same is the case for the phase-decoupled handshake register. Below we describe a range of such fused components.

The schematic symbols for a handshake register fused with a Join or with a Fork are shown in Fig. 9.16, and Fig. 9.17 shows the implementation of a fused Join+Register+Fork circuit with two input channels and two output channels. For simplicity, the figure shows a design with separate phase flip-

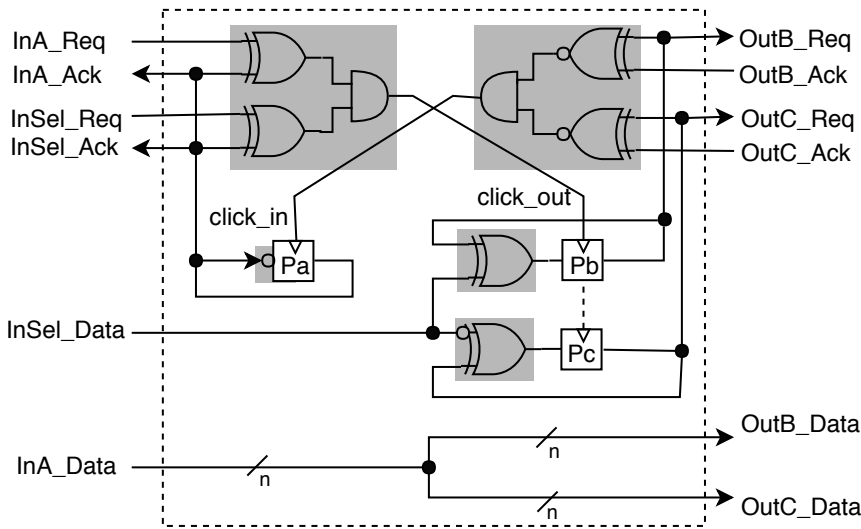


Figure 9.15: DEMUX



Figure 9.16: Schematic symbols for a handshake register fused with a Join and/or a Fork.

flops for each input and output channel. As all phase flip-flops are clocked by the same signal, at most two-phase flip-flops are needed – phase flip-flops initialized to the same value can share a single flip-flop. It is easy to see how input and output channels can be dropped from or added to the circuit by dropping or adding XOR or XNOR gates.

In a similar way, it is possible to fuse a handshake register with a Merge or a MUX or a DEMUX, but as discussed in [86] these fused components are only marginally smaller and faster than direct compositions of the basic components, and we do not describe them here. Another idea is to fuse components that implement a complete pipeline stage, i.e., to implement a fused Join+CombinatorialLogic+Register+Fork circuit. This could be done by fusing the Join and the Fork into the handshake register. In the general case, where nothing is known about the surrounding circuitry, this would require a matched delay element for each input channel, rather than a single delay element matching the combinatorial circuit. As matched delay elements are

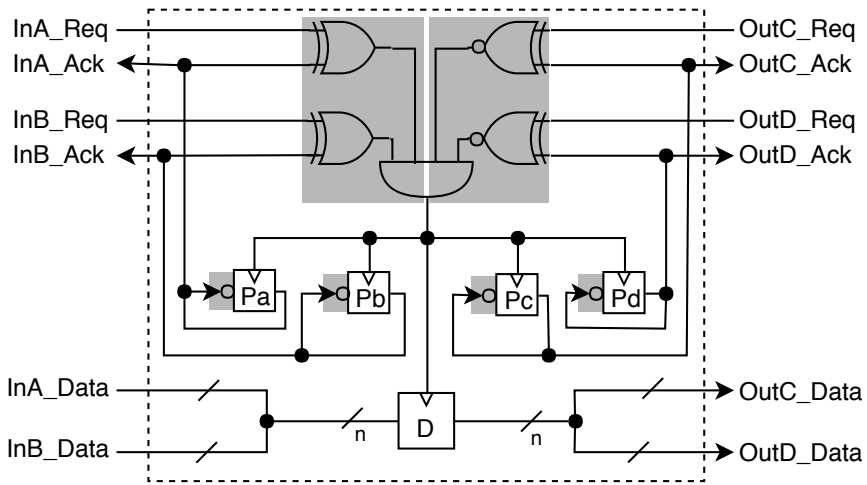


Figure 9.17: Implementation of a phase-decoupled fused Join+Register+Fork component with two input channels (A and B) and two output channels (C and D).

quite expensive to implement, we do not describe them here.

9.4.7 Mutual exclusion and arbitration

In section 5.10, we presented the implementation of 4-phase versions of a mutual exclusion element and a two-way arbiter. Below we provide 2-phase click-style implementations of these.

For the 4-phase Mutex we note that the request inputs convey two events: When a request signal is asserted, this signals that the client requests the shared resource administered by the Mutex, and when the same request signal is de-asserted, it signals that the client has released the resource. In a 2-phase design, lacking the return-to-zero signaling, a separate “done” signal is needed to signal the latter. In Kinniment’s book [78, p. 215], such a Mutex is called an RGD (request-grant-done) Mutex. The book also shows the design of a two-way arbiter using the RGD-mutex. The same RGD-mutex designs were studied earlier in [123, sec. 3.3.8]. Figure 9.18 shows the two implementations of the RGD-mutex presented in [123]. They are both built around the 4-phase mutex from figure 5.28. Figure 9.18(b) shows a simple design that assumes a client will not produce a new request until the 4-phase mutex has de-asserted the grant signal related to a previous request by the same client. A more robust implementation is shown in figure 9.18(c). This design gates the request from a client using a signal that explicitly detects when the 4-phase mutex has de-asserted the grant signal related to a previous request by the same client. The

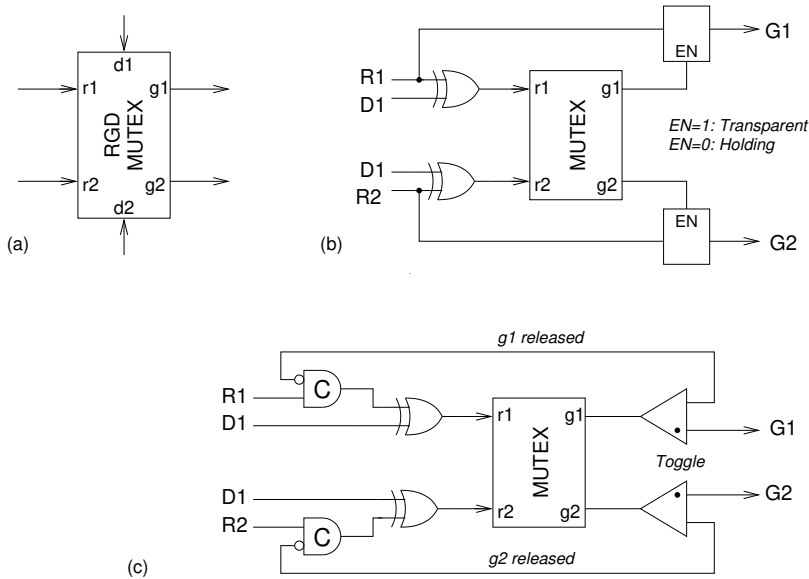


Figure 9.18: (a) Component symbol for a 2-phase RGD mutex. (b) A simple implementation assuming that a client does not issue another request until the mutex has released grant related to the previous request. (c) A safe design that explicitly enforces this ordering.

toggle is a component that relays events on the input to one of the outputs in an alternating fashion, starting with the output marked with a dot [147].

By combining a click-style implementation of the safe RGD-mutex and a click-based merge from figure 9.13, see figure 9.19, we obtain a 2-phase click-style implementation of a two-way arbiter, similar to the 4-phase version in figure 5.29.

9.5 Prototyping using FPGAs

If the material in this book is used to teach asynchronous design, it is desirable that students can build and operate circuits in practice. To support this, we provide an on-line repository [61] containing: (a) Schematics and synthesizable VHDL source code for all the handshake components. More details on this may be found in [86]. (b) Schematics and source code for the Fibonacci and GCD examples from section 9.3, including VHDL test-benches for simulation. (c) A sequence of snapshots of the schematics illustrating the token-flow operation of the circuits.

Both circuits have been simulated and synthesized using Xilinx Vivado and

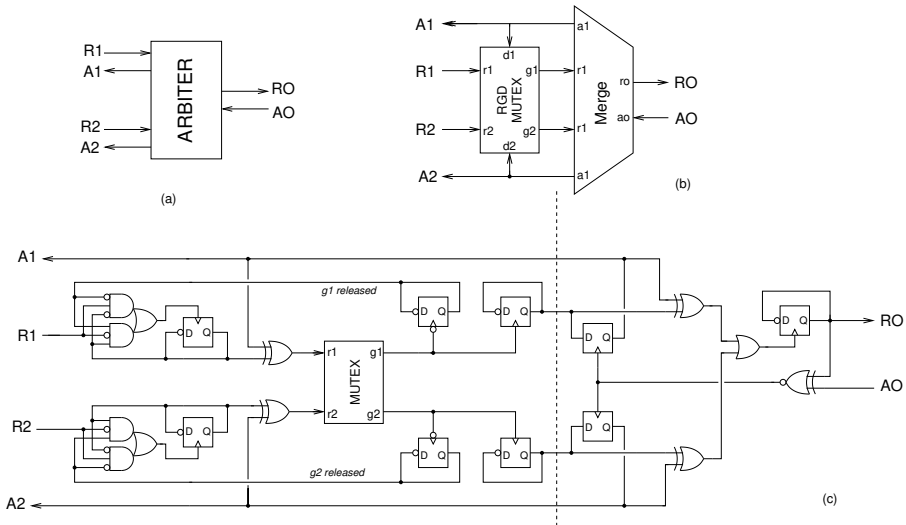


Figure 9.19: (a) A 2-phase click-style two-way arbiter can be implemented by combining a safe RGB-mutex and a click-style merge. (b) Schematic of the full arbiter circuit.

implemented on a Digilent Nexys4DDR FPGA-board and on a Basys 3 board (both boards have a Xilinx Artix 7 chip). The circuits have been operated manually. Input channels are implemented using a debounced pushbutton for the request signal, a set of switches for the data, and an LED for the acknowledge signal. Output channels are implemented using LEDs for the request signal and the data signals, and a debounced pushbutton for the acknowledge signal. The corresponding XDC-files (constraint files specifying the pinout) are included in the design sources in the GitHub repository. In the component source files, the *"DONT_TOUCH"* attribute is set for combinational signals and registers, to force the place and route tool to keep the signals. Therefore, a minimal project setup is necessary for using the designs.

A post-synthesis timing simulation of the Fibonacci circuit is shown in Figure 9.20. The first five signals show the environment signals. Below these, some selected internal signals are also plotted and grouped by component. A file showing a similar simulation of the GCD circuit is included in the GitHub repository.

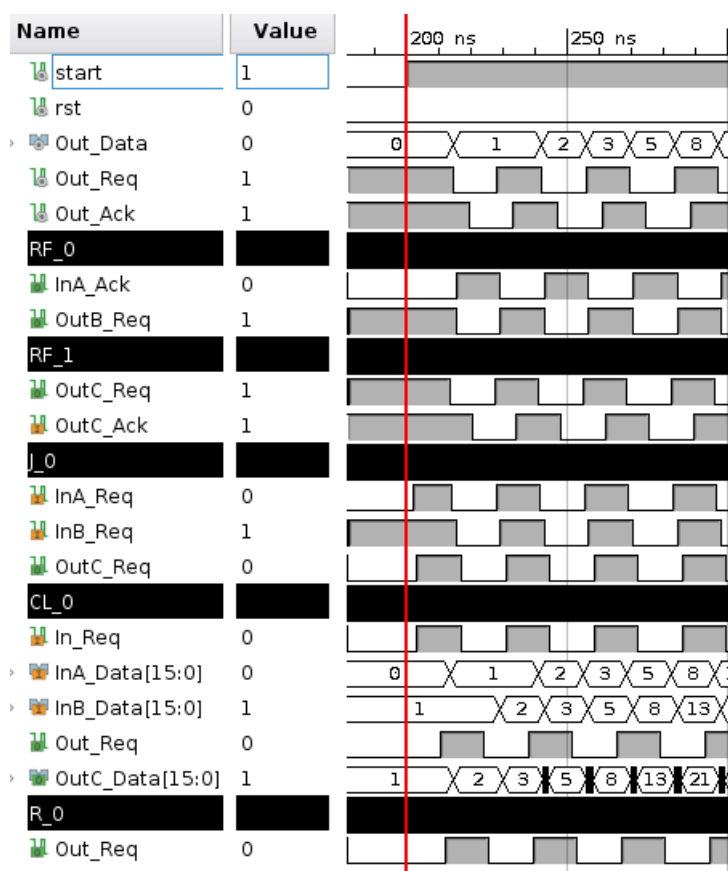


Figure 9.20: Fibonacci example - Post-synthesis timing simulation

Chapter 10

Advanced 4-phase bundled-data protocols and circuits

Earlier chapters have covered the basics of 4-phase bundled-data circuits. In this chapter, we broaden the perspective and introduces a variety of channel types and handshake protocols with different data-validity schemes. The corresponding latch control circuits 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 chapter 6.

Today, the newer 2-phase bundled-data circuit templates described in chapter 9 (e.g., Mousetrap, Click and Loihi) have gained popularity, and are more widely used.

10.1 Channels and protocols

10.1.1 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 acknowledge signal. Figure 10.1 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.

10.1.2 Data-validity schemes

For the bundled-data protocols, it is also relevant to define the time interval in which data is valid, and figure 10.2 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 10.2.

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 data 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 happens – the sender may prolong the data-validity interval. The receiver may even rely on this.

A typical example of this is the extended-early data-validity schemes in figure 10.2. On a push channel, the data-validity interval begins sometime before $Req \uparrow$ and ends sometime after $Req \downarrow$.

10.1.3 Discussion

The above classification of channel types and handshake protocols stems mostly from Peeters’ Ph.D. thesis [127]. The choice of channel type, handshake protocol, and data-validity scheme affects the implementation of the handshake components in terms of area, speed, and power. Just as a design may use a

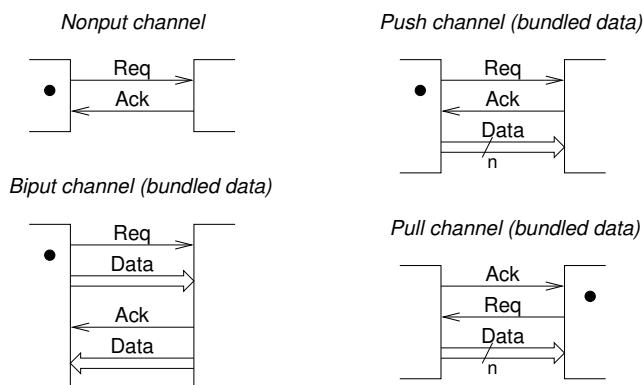


Figure 10.1: The four fundamental channel types: nonput, push, biput, and pull.

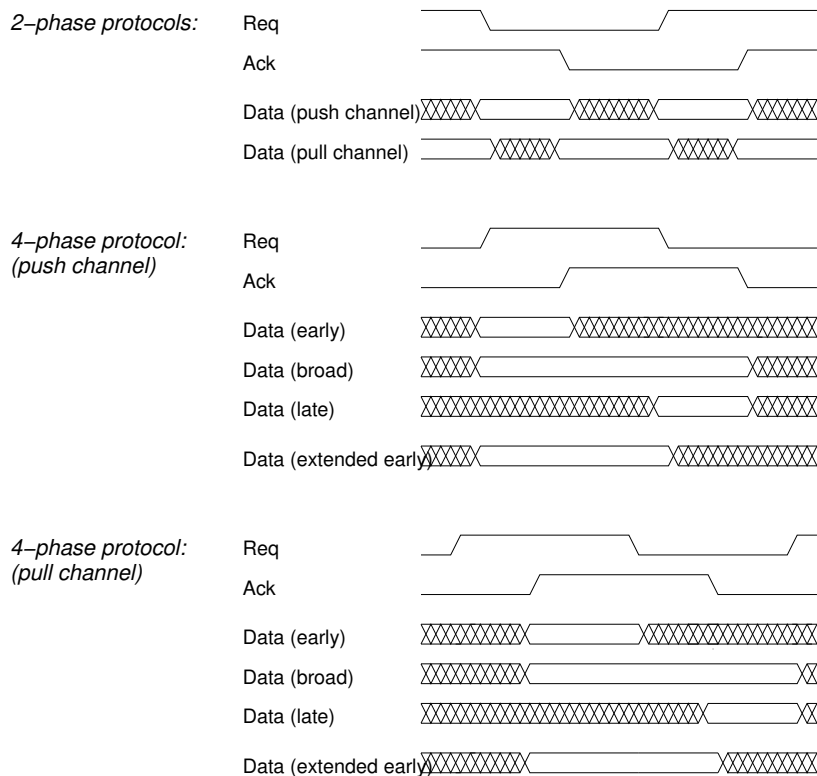


Figure 10.2: Data-validity schemes for 2-phase and 4-phase bundled data.

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.

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 [127, 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 [127, 84] for more details.

10.2 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 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 [116] and used in the Genesil silicon compiler [69].

Figure 10.3 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:

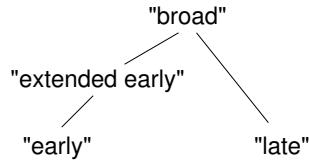


Figure 10.3: Hierarchical ordering of the four data-validity schemes for the 4-phase bundled-data protocol.

- 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 118, 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 121 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 erroneous behavior.

10.3 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 [44], this circuit is called a *simple* latch controller, and in [84], 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 the other half will be holding empty tokens, as illustrated in figure 10.4(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 resources.

Ideally, one would want to store a valid token in every level-sensitive latch as illustrated in figure 10.5 and just “add” the empty tokens to the data stream on the interfaces as part of the handshaking. In [44], Furber and Day explain

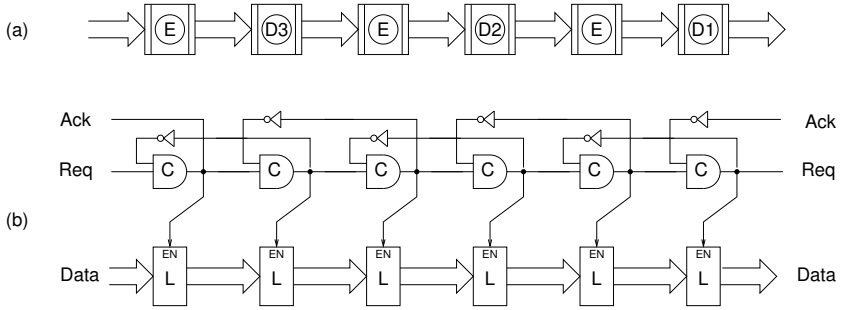


Figure 10.4: (a) A FIFO based on handshake latches, and (b) its implementation using simple latch controllers and level-sensitive latches. The FIFO fills with valid data in every other latch. A latch is transparent when $EN = 0$ and it is opaque (holding data) when $EN = 1$.

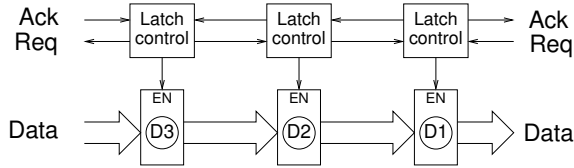


Figure 10.5: A FIFO where every level-sensitive latch holds valid data when the FIFO is full. The semi-decoupled and fully-decoupled latch controllers from [44] allow this behavior.

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: $A_{out}\uparrow \preceq A_{in}\uparrow$.

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	$2L_r + L_{f.V} + L_{f.E}$

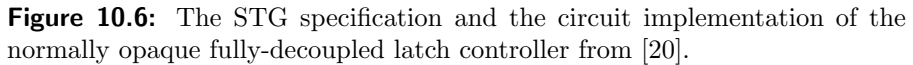
Table 10.1: Summary of the characteristics of the latch controllers in [44].

- 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 section 7.7 on page 136. The fully-decoupled latch controller presented in [44] does not have this problem. Due to the decoupling of the input and output channels, the critical cycle in a timed Petri net model only visits nodes related to two neighboring pipeline stages, and the period, P , becomes minimum (c.f. section 7.7). Table 10.1 summarizes the characteristics of the simple, semi-decoupled and fully-decoupled latch controllers.

All of the latch controllers mentioned above 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 to the opaque mode in which it will hold the data. In the design of the asynchronous MIPS processor reported in [20], we experienced approximately a 50 % power reduction when replacing normally transparent latch controllers by normally opaque latch controllers.

Figure 10.6 shows the STG specification and the circuit implementation of the normally opaque latch controller used in [20]. As seen from the STG, there is quite a strong interaction between the input and output channels, but the critical cycle in a timed Petri net model 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.



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 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 11

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.

11.1 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 [59, 60] 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 of concurrent systems in general: OCCAM [70], LOTOS [120, 12], and CCS [99], as well as languages defined specifically for designing asynchronous circuits: Tangram [115, 153], CHP [90], and Balsa [5, 6]. 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 flavor of this type of language. Following this, we briefly explain two somewhat 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. developed a proprietary language, Tangram, and an associated silicon compiler [115, 155, 153, 127]. 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 were designed within Philips [157, 158, 159, 76, 77]. The last of these was a smart-card circuit that is described in [142, Chap. 13].
- At Caltech, Prof. Martin developed the 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 [89, 92]. Recently, Prof. Manohar at Yale University has made similar tools available as open source.

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.

11.2 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.

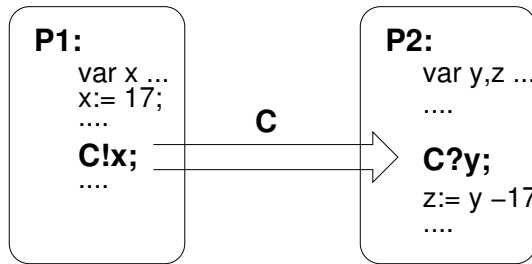


Figure 11.1: Two processes $P1$ and $P2$ connected by a channel C . Process $P1$ sends the value of its variable x to the channel C , and process $P2$ receives the value and assigns it to its variable y .

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 11.1, 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 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 [88], 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 of 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 previ-

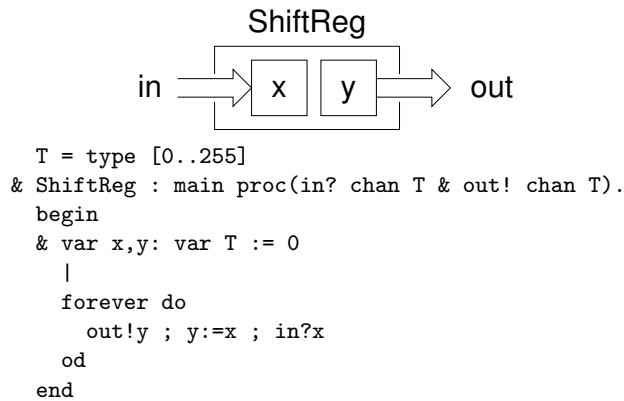


Figure 11.2: A Tangram program for a 2-place shift register.

ous 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 behavior of the circuit.

In the following section, we will provide some small program examples to give a flavor 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 developed a similar language and synthesis tool that is available in the public domain [6]. Other examples of related work are presented in [13] and [16].

11.3 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.

11.3.1 A 2-place shift register

Figure 11.2 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

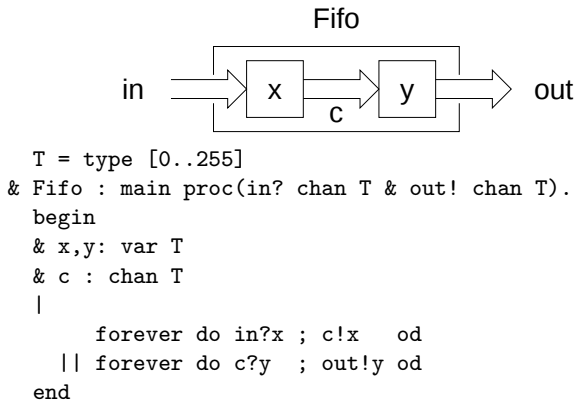


Figure 11.3: A Tangram program for a 2-place (ripple) FIFO.

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`.

11.3.2 A 2-place (ripple) FIFO

Figure 11.3 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.

11.3.3 GCD using while and if statements

Figure 11.4 shows the code for a module that computes the greatest common divisor, the example from section 3.8. The “`do x<>y then ...od`” is a while statement and apart from the syntactical differences, the code in figure 11.4 is identical to the code in figure 3.15 on page 45.

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

11.3.4 GCD using guarded commands

Figure 11.5 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

```

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 11.4: A Tangram for GCD using while and if statements.

```

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 11.5: A Tangram program for GCD using guarded repetition.

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.

11.4 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.

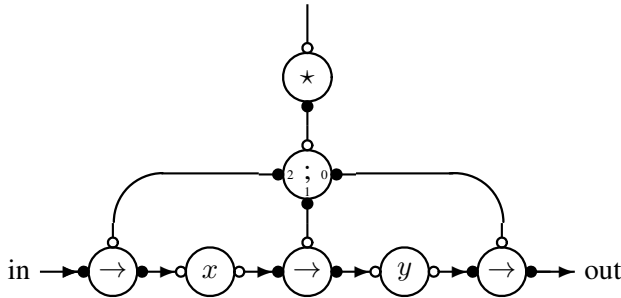


Figure 11.6: The compiled handshake circuit for the 2-place shift register.

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 flavor of “syntax-directed compilation” by showing handshake circuits corresponding to the example Tangram programs from the previous section.

11.4.1 The 2-place shift register

As a first example of syntax-directed compilation, figure 11.6 shows the handshake circuit corresponding to the Tangram program in figure 11.2.

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 11.6, 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 channel 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 output 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 ‘ $\rightarrow$ ’. 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.

11.4.2 The 2-place FIFO

Figure 11.7 shows the handshake circuit corresponding to the Tangram program in figure 11.3. The component labeled ‘psv’ in the handshake circuit of figure 11.7 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 ($c?y$).

An optimization of the handshake circuit for **Fifo** is shown in figure 11.8. 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 11.2. The only difference is in the control part of the circuits.

11.4.3 GCD using guarded repetition

As a more complex example of syntax-directed compilation, figure 11.9 shows the handshake circuit compiled from the Tangram program in figure 11.5.

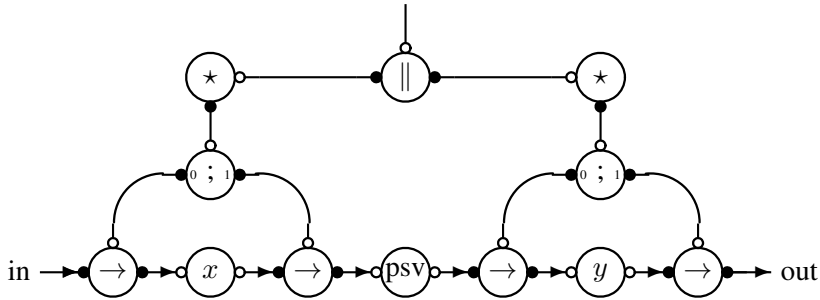


Figure 11.7: Compiled handshake circuit for the FIFO program.

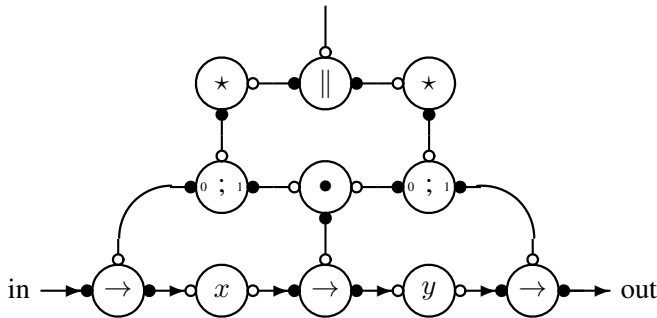


Figure 11.8: Optimized handshake circuit for the FIFO program.

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.

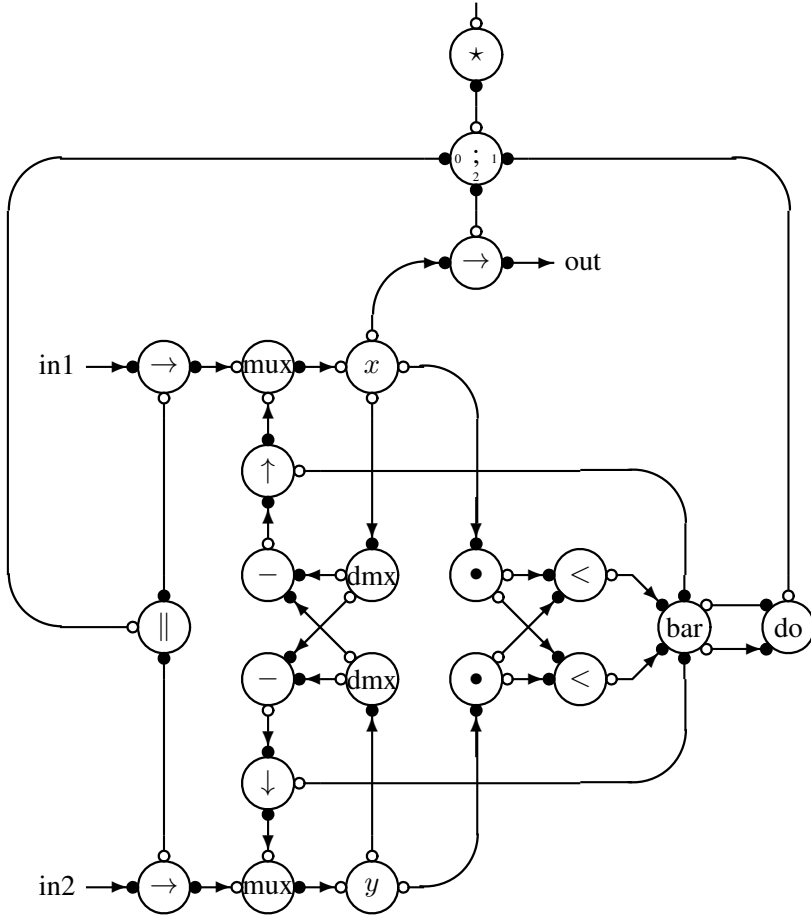


Figure 11.9: Compiled handshake circuit for the GCD program using guarded repetition.

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 non-

put 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.

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 with 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 11.4 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 11.9, 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 [153, Fig. 2.7, p.34]. In figure 11.9, 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.

11.5 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 [87]. 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:

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

A production rule consists 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 106. The production rules specify the behavior 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 is 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 110.

11.6 Using VHDL for asynchronous design

11.6.1 Introduction

In this section, we 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 [125], and it has subsequently been used in a course on asynchronous circuit design. Others, including [107, 130, 163, 32, 134, 135], have developed related 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.

11.6.2 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, Verilog, or System 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 11.10 shows how the Tangram program from figure 11.2 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

```

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 11.10: VHDL description of the 2-place shift register FIFO stage from figure 11.2.

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 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 behavioral 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.

11.6.3 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 behavioral description using the above communication primitives, figure 11.11(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 11.11(a-c).

In VHDL, all communication between processes takes place via signals. For this reason channels 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 11.6.5. 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 transferred.

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

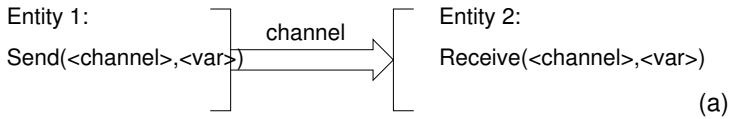
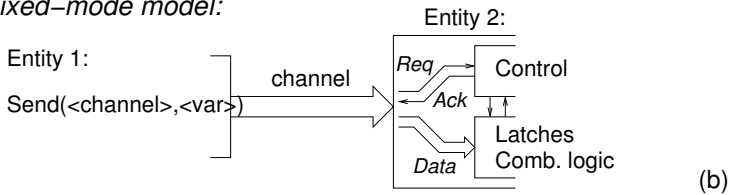
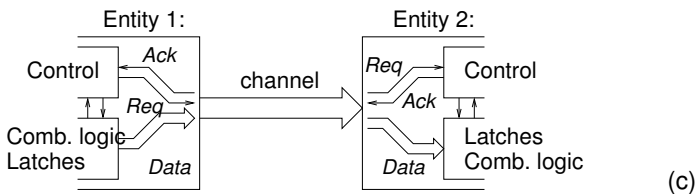
High-level model:*Mixed-mode model:**Low-level model:*

Figure 11.11: The VHDL packages for channel communication support high-level, mixed-mode, and gate-level/standard cell simulations.

listings can be found in section 11.8 at the end of this chapter. The design flow enabled by these packages is as follows:

- The circuit and its environment or test bench are first modeled 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 consists of ordinary latches and combinational circuitry). Mixed-mode simulations, as illustrated in figure 11.11(b), are possible. Simulation models of the

```

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 11.12: Definition of an abstract channel.

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 Petrifly.

- 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.

11.6.4 The abstract channel package

An abstract channel is defined in figure 11.12 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 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` is described in detail below:

- u:** Uninitialized channel. This is the default value of the drivers. As long as either the sender or receiver drives 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 transferred. 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 transferred 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 a `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.

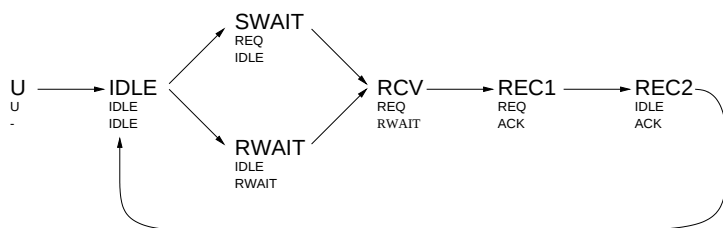


Figure 11.13: The protocol for the abstract channel. The values in large letters are the resulting resolved values of the channel, and the values in smaller letters below them are the driving values of the sender and receiver, respectively.

Figure 11.13 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 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 section 11.8.1. 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 11.14. Notice that the channels in the entity have the mode **inout**, and the FIFO stage waits for the reset signal **resetn** after the initialization. In this way, it waits for other subcircuits that 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 11.15) 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.

```

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
        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 11.14: Description of a FIFO stage.

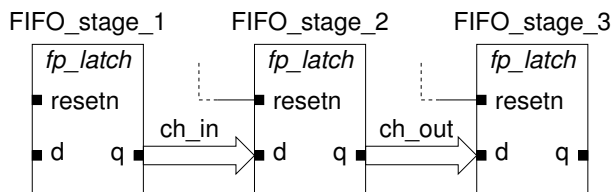


Figure 11.15: A FIFO built using the latch defined in figure 11.14.

The result of this experiment can be seen in figure 11.16. 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.

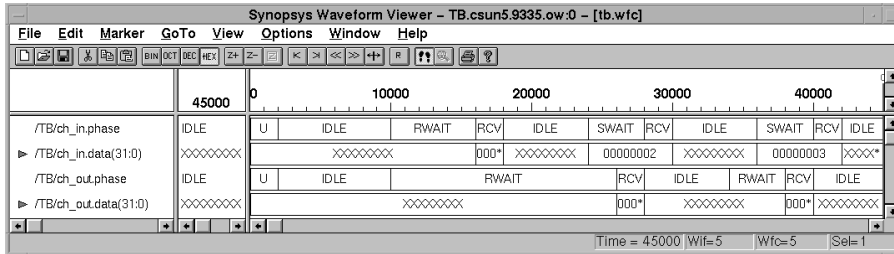


Figure 11.16: Simulation of the FIFO using the abstract channel package.

```

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 11.17: Definition of a real channel.

11.6.5 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 11.17.

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 section 11.8.2. This package defines a 32-bit standard logic 4-phase

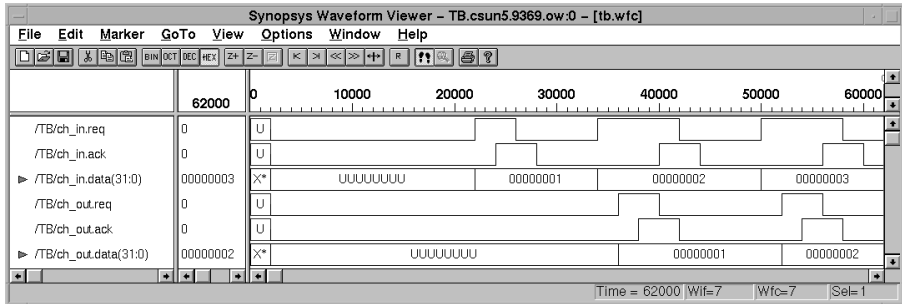


Figure 11.18: Simulation of the FIFO using the real channel package.

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 11.18 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.

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.

11.6.6 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 11.14 in the preceding section will be separated into a latch control circuit

```

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 11.19: Separation of the FIFO stage into an ordinary data latch and a latch control circuit.

called `latch_ctrl` and a latch called `std_logic_latch`. The VHDL code is shown in figure 11.19, and figure 11.20 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

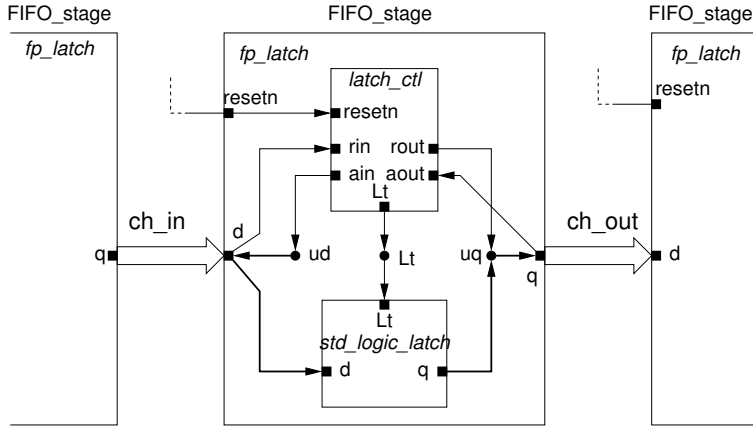


Figure 11.20: Separation of control and data.

is the function of the signals `ud` and `uq` of type `uchannel_fp` in figure 11.17. 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 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 [148]. 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 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 behavioral description, the normal simulator takes over and evaluates the channel using 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.

11.7 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 on languages that are similar to CSP. The reasons 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.

11.8 The VHDL channel packages

11.8.1 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;

```

11.8.2 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 := "XXXXXXXXXXXXXXXXXXXXXXXXXXXX";
-- 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;

```

Bibliography

- [1] Ameer M.S. Abdelhadi and Mark R. Greenstreet. Interleaved architectures for high-throughput synthesizable synchronization fifos. In *Proc. IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, volume 2017-, pages 41–48. IEEE Computer Society, 2017.
- [2] T. Agerwala. Putting petri nets to work. *IEEE Computer*, 12(12):85–94, December 1979.
- [3] Peeter Alfke. Metastable Recovery in Virtex-II Pro FPGAs, 2005. Xilinx application note XAPP094 (v3.0) February 10, 2005.
- [4] T.S. Balraj and M.J. Foster. Miss Manners: A specialized silicon compiler for synchronizers. In Charles E. Leieron, editor, *Advanced Research in VLSI*, pages 3–20. MIT Press, April 1986.
- [5] A. Bardsley and D. 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.
- [6] A. Bardsley and D. A. Edwards. The Balsa asynchronous circuit synthesis system. In *Forum on Design Languages*, September 2000.
- [7] Salomon Beer, Ran Ginosar, Jerome Cox, Tom Chaney, and David M. Zar. Metastability challenges for 65 nm and beyond; simulation and measurements. In *Proc. Design, Automation and Test in Europe (DATE)*, March 2013.
- [8] Salomon Beer, Ran Ginosar, Michael Priel, Rostislav Dobkin, and Avinoam Kolodny. The devolution of synchronizers. In *Proc. IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 94–103, 2010.
- [9] 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.

- [10] Peter A. Beerel, Chris J. Myers, and Teresa H.-Y. Meng. Automatic synthesis of gate-level speed-independent circuits. Technical Report CSL-TR-94-648, Stanford University, November 1994.
- [11] C. H. (Kees) van Berkel, Mark B. Josephs, and Steven M. Nowick. Scanning the technology: Applications of asynchronous circuits. *Proceedings of the IEEE*, 87(2):223–233, February 1999.
- [12] Ed Brinksma and Tommaso Bolognesi. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1), 1987.
- [13] Erik Brunvand and Robert F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *Proc. IEEE/ACM Int'l. Conf. Computer-aided Design (ICCAD)*, pages 262–265, November 1989.
- [14] J. A. Brzozowsky and C.-J. H. Seager. *Asynchronous Circuits*. Springer Verlag, Monographs in Computer Science, 1994. ISBN: 0-387-94420-6.
- [15] 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.
- [16] Steven M. Burns and Alain J. Martin. Syntax-directed translation of concurrent programs into self-timed circuits. In J. Allen and F. Leighton, editors, *Proceedings of the Fifth MIT Conference on Advanced Research in VLSI*, pages 35–50. MIT Press, 1988.
- [17] Marco Cannizzaro, Salomon Beer, Jordi Cortadella, Ran Ginosar, and Luciano Lavagno. SafeRazor: Metastability-robust adaptive clocking in resilient circuits. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 62(9):7177141, 2238–2247, 2015.
- [18] T. J. Chaney and C. E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, C-22(4):421–422, April 1973.
- [19] Daniel M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, October 1984. Report No. STAN-CS-84-1026.
- [20] K. T. Christensen, P. Jensen, P. Korgær, and J. Sparsø. The Design of an Asynchronous TinyRISC TR4101 Microprocessor Core. In *Proc. IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 108–119. IEEE Computer Society Press, 1998.

- [21] 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.
- [22] Tam-Anh Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, MIT Laboratory for Computer Science, June 1987.
- [23] Tam-Anh Chu and Rabinda K Roy (editors). Special issue on asynchronous circuits and systems. *IEEE Design & Test*, 11(2), 1994.
- [24] Bill Coates, Al Davis, and Ken Stevens. The Post Office experience: Designing a large asynchronous chip. *Integration, the VLSI journal*, 15(3):341–366, October 1993.
- [25] F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *J. Comput. System Sci.*, 5(1):511–523, October 1971.
- [26] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic Synthesis of Asynchronous Controllers and Interfaces*. Springer-Verlag, 2002.
- [27] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alexandre Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. In *XI Conference on Design of Integrated Circuits and Systems*, pages 205–210, Barcelona, November 1996.
- [28] Uri Cummings, Andrew Lines, and Alain Martin. An asynchronous pipelined lattice structure filter. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 126–133, November 1994.
- [29] W. J. Dally and J. W. Poulton. *Digital Systems Engineering*. Cambridge University Press, 1998.
- [30] William J. Dally, R. Curtis Harting, and Tor M. Aamodt. *Digital design using VHDL: A Systems Approach*. Cambridge University Press, 2016.
- [31] S. Das, C. Tokunaga, S. Pant, W. Ma, S. Kalaiselvan, K. Lai, D. M. Bull, and D. T. Blaauw. RazorII: In situ error detection and correction for PVT and SER tolerance. *IEEE Journal of Solid-State Circuits*, 44(1):32–48, 2009.
- [32] David Lloyd, Dept. of Computer Science, Manchester University. VHDL models of asynchronous handshaking. (Personal communication, August 1998).

- [33] M. Davies, N. Srinivasa, T. Lin, et al. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, 2018.
- [34] A. L. 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.
- [35] Al Davis and Steven M. Nowick. Asynchronous circuit design: Motivation, background, and methods. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 1–49. Springer-Verlag, 1995.
- [36] Al Davis and Steven M. Nowick. An introduction to asynchronous circuit design. Technical Report UUCS-97-013, Department of Computer Science, University of Utah, September 1997.
- [37] Al Davis and Steven 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.
- [38] Jack 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.
- [39] Jo Ebergen and Robert Berks. Response time properties of linear asynchronous pipelines. *Proceedings of the IEEE*, 87(2):308–318, February 1999.
- [40] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, and Trevor Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proc. IEEE/ACM International Symposium on Microarchitecture*, pages 7–18, 2003.
- [41] Karl M. Fant and Scott 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.
- [42] Matthew Fojtik, David Fick, Yejoong Kim, Nathaniel Pinckney, David Money Harris, David Blaauw, and Dennis Sylvester. Bubble Razor: Eliminating timing margins in an ARM cortex-M3 processor in 45 nm CMOS using architecturally independent error detection and correction. *IEEE Journal of Solid-State Circuits*, 48(1):66–81, 2013.

- [43] 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.
- [44] S. B. Furber and P. Day. Four-phase micropipeline latch control circuits. *IEEE Transactions on VLSI Systems*, 4(2):247–253, June 1996.
- [45] 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 (ICCD)*, pages 217–220, October 1994.
- [46] 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.
- [47] S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day, and N. C. Paver. AMULET2e: An Asynchronous Embedded Controller. In *Proc. IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 290–299. IEEE Computer Society Press, 1997.
- [48] Stephen B. Furber, James D. Garside, Peter Riocreux, Steven Temple, Paul Day, Jianwei Liu, and Nigel C. Paver. AMULET2e: An asynchronous embedded controller. *Proceedings of the IEEE*, 87(2):243–256, February 1999.
- [49] G. Birtwistle and A. Davis, editor. *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: Steve Furber, “Computing without Clocks: Micropipelining the ARM Processor,” Al Davis, “Practical Asynchronous Circuit Design: Methods and Tools,” C.H. van Berkel, “VLSI Programming of Asynchronous Circuits for Low Power,” Jo Ebergen, “Parallel Program and Asynchronous Circuit Design,” A. Davis and S. Nowick, “Introductory Survey”.
- [50] 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.
- [51] Bruce 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.

- [52] R. Ginosar. Metastability and synchronizers: A tutorial. *IEEE Design & Test of Computers*, 28(5):23–35, 2011.
- [53] Ran Ginosar. Fourteen ways to fool your synchronizer. In *Proc. IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 89–96, May 2003.
- [54] Mark R. Greenstreet. *STARI: A Technique for High-Bandwidth Communication*. PhD thesis, Princeton University, Department of Computer Science, 1993. TR-394-92.
- [55] Mark R. Greenstreet, Jørgen Staunstrup, and Ted E. Williams. Self-timed iteration. In Carlo H. Séquin, editor, *Proceedings of VLSI '87*, pages 269–282. IFIP, August 1987.
- [56] M.R. Greenstreet. Implementing a STARI chip. In *Proc. Int'l. Conf. Computer Design (ICCD)*, pages 38–43, 1995.
- [57] Scott Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83(1):69–93, January 1995.
- [58] 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.
- [59] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.
- [60] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, 1985.
- [61] <https://github.com/zuzkajelcicova/Async-Click-Library>.
- [62] Wenmian Hua and Rajit Manohar. Exact timing analysis for asynchronous systems. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, 37(1):203–216, 2018.
- [63] D. A. Huffman. The synthesis of sequential switching circuits. *J. Franklin Inst.*, pages 161–190, 275–303, March/April 1954.
- [64] D. A. Huffman. The synthesis of sequential switching circuits. In E. F. Moore, editor, *Sequential Machines: Selected Papers*. Addison-Wesley, 1964.
- [65] H. Hulgaard, S.M. Burns, T. Amon, and G. Borriello. An algorithm for exact bounds on the time separation of events in concurrent systems. *Computers, IEEE Transactions on*, 44(11):1306–1317, 1995.

- [66] Henrik Hulgaard, Steven M. Burns, Tod Amon, and Gaetano Borriello. Practical applications of an efficient time separation of events algorithm. In *Proc. IEEE/ACM Int'l. Conf. Computer-aided Design (ICCAD)*, pages 146–151, 1993.
- [67] Henrik Hulgaard, Steven. M. Burns, Tod Amon, and Gaetano Borriello. An algorithm for exact bounds on the time separation of events in concurrent systems. Technical Report TR #94-02-02, University of Washington, Department of Computer Science and Engineering, 1994. (to appear in IEEE Transaction on Computers).
- [68] Kai Hwang. *Computer Arithmetic: Principles, Architecture, and Design*. John Wiley & Sons, 1979.
- [69] S. C. Johnson and S. Mazor. Silicon compiler lets system makers design their own vlsi chips. *Electronic Design*, 32(20):167–181, 1984.
- [70] Geraint Jones. *Programming in occam*. Prentice-Hall international, 87.
- [71] Mark B. Josephs, Steven M. Nowick, and C. H. (Kees) van Berkel. Modeling and design of asynchronous circuits. *Proceedings of the IEEE*, 87(2):234–242, February 1999.
- [72] E. Kasapaki, M. Schoeberl, Rasmus Bo Sørensen, C. T. Müller, K. Goossens, and J. Sparsø. Argo: A Real-Time Network-on-Chip Architecture with an Efficient GALS Implementation. *IEEE Transactions on VLSI Systems*, 24(2):479–492, 2016.
- [73] E. Kasapaki and J. Sparsø. Argo: A Time-Elastic Time-Division-Multiplexed NOC using Asynchronous Routers. In *Proc. IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 45–52. IEEE Computer Society Press, 2014.
- [74] Evangelia Kasapaki. An EDA tool for the timing analysis, optimization and timing validation of asynchronous circuits. Master's thesis, Computer Science Department, University of Crete, Greece, Heraklion, Crete, Greece, April 2008.
- [75] Evangelia Kasapaki and Jens Sparsø. The Argo NOC: Combining TDM and GALS. In *European conference on circuit theory and design (ECTD)*, pages 1–4, 2015.
- [76] Joep Kessels, Torsten Kramer, Gerrit den Besten, Ad Peeters, and Volker 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.

- [77] Joep Kessels, Torsten Kramer, Ad Peeters, and Volker Timm. DESCAL: a design experiment for a smart card application consuming low energy. In Rene van Leuken, Reinder Nouta, and Alexander de Graaf, editors, *European Low Power Initiative for Electronic System Design*, pages 247–262. Delft Institute of Microelectronics and Submicron Technology, July 2000.
- [78] David J. Kinniment. *Synchronization and Arbitration in Digital Systems*. Wiley, 2008.
- [79] Zvi Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, 1978.
- [80] Alex Kondratyev, Jordi Cortadella, Michael Kishinevsky, Luciano Lavagno, and Alexander Yakovlev. Logic decomposition of speed-independent circuits. *Proceedings of the IEEE*, 87(2):347–362, February 1999.
- [81] I Kotleas, D.R. Humphreys, R.B. Sørensen, E. Kasapaki, F. Brandner, and J. Sparsø. A Loosely Synchronizing Asynchronous Router for TDM-Scheduled NOCs. In *Proc. IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 151–158, 2014.
- [82] Andrew Lines, May 2019. Private communication.
- [83] Andrew Lines, Prasad Joshi, Ruokun Liu, Steve McCoy, Jonathan Tse, Yi Hsin Weng, and Mike Davies. Loihi asynchronous neuromorphic research chip. In *Proc. IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 32–33, 2018.
- [84] J. Liu. *Arithmetic and control components for an asynchronous micro-processor*. PhD thesis, Department of Computer Science, University of Manchester, 1997.
- [85] S. Lubkin. Asynchronous signals in digital computers. *Mathematical Tables and Other Aids to Computation*, 6(40):238–241, 1952.
- [86] Adrian Mardari, Zuzana Jelčicová, and Jens Sparsø. Design and FPGA-implementation of Asynchronous Circuits Using Two-phase Handshaking. In *Proc. IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 1–10. IEEE Computer Society Press, 2019.
- [87] 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, 1997.
- [88] Alain J. Martin. The probe: An addition to communication primitives. *Information Processing Letters*, 20(3):125–130, 1985. Erratum: IPL 21(2):107, 1985.

- [89] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.
- [90] Alain J. Martin. Formal program transformations for VLSI circuit synthesis. In Edsger W. Dijkstra, editor, *Formal Development of Programs and Proofs*, UT Year of Programming Series, pages 59–80. Addison-Wesley, 1989.
- [91] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Advanced Research in VLSI: Proceedings of the Sixth MIT Conference*, pages 263–278. MIT Press, 1990.
- [92] Alain J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*. Addison Wesley, 1990. UT Year of Programming Institute on Concurrent Programming.
- [93] Alain J. Martin. Asynchronous datapaths and the design of an asynchronous adder. *Formal Methods in System Design*, 1(1):119–137, July 1992.
- [94] Alain J. Martin, Steven M. Burns, T. K. Lee, Drazen Borkovic, and Pieter J. Hazewindus. The First Asynchronous Microprocessor: The Test Results. *Computer Architecture News*, 17(4):95–98, 1989.
- [95] Peggy McGee and Steve Nowick. DES (Discrete Event System) Analyzer: A performance analysis and timing verification tool for concurrent digital systems, 2003. <http://www1.cs.columbia.edu/~nowick/asynctools> Accessed: February 2018.
- [96] Peggy B. McGee and Steven M. Nowick. An efficient algorithm for time separation of events in concurrent systems. In *Proc. IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, pages 180–187. IEEE, 2007.
- [97] Carver A. ”Mead and Lynn A.” Conway. *Introduction to VLSI Systems*. Addison-Wesley, 1980.
- [98] D. Messerschmitt. Synchronization in digital system design. *IEEE Journal on Selected Areas in Communications*, 8(8):1404–1419, 1990.
- [99] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
- [100] Charles E. Molnar. Excerpts from two seminars on metastability presented at Hewlett-Packard in february 1992. On line. Available at: https://www.cse.wustl.edu/history/molnar_c/.

- [101] Charles E. Molnar, Ian W. Jones, Bill Coates, and Jon 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.
- [102] Charles E. Molnar, Ian W. Jones, William S. Coates, Jon K. Lexau, Scott M. Fairbanks, and Ivan E. Sutherland. Two FIFO ring performance experiments. *Proceedings of the IEEE*, 87(2):297–307, February 1999.
- [103] David 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.
- [104] David 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.
- [105] Robert Mullins and Simon Moore. Demystifying data-driven and pauseable clocking schemes. In *Proc. IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 175–185, March 2007.
- [106] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.
- [107] Chris J. Myers. *Asynchronous Circuit Design*. John Wiley & Sons, July 2001. ISBN: 0-471-41543-X.
- [108] Christian 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.
- [109] Christian Dalsgaard Nielsen, Jørgen Staunstrup, and Simon 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*, pages ??–?? North-Holland, Amsterdam, 1991.
- [110] 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.
- [111] Lars S. Nielsen and Jens Sparsø. A Low-power Asynchronous Datapath for a FIR Filter Bank. In *Proc. IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 197–207. IEEE Computer Society Press, 1996.

- [112] Lars S. Nielsen and Jens Sparsø. An 85 μW Asynchronous Filter-Bank for a Digital Hearing Aid. In *Proc. IEEE International Solid State circuits Conference*, pages 108–109, 1998.
- [113] Lars S. Nielsen and Jens 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.
- [114] Lars Skovby Nielsen. *Low-power Asynchronous VLSI Design*. PhD thesis, Department of Information Technology, Technical University of Denmark, 1997. IT-TR:1997-12.
- [115] C. Niessen, C.H. van Berkel, M. Rem, and R.W.J.J. Saeijs. VLSI programming and silicon compilation. In *Proc. International Conf. Computer Design (ICCD)*, pages 150–166. IEEE Computer Society Press, 1988.
- [116] David C. Noice. *A Two-Phase Clocking Discipline for Digital Integrated Circuits*. PhD thesis, Department of Electrical Engineering, Stanford University, February 1983.
- [117] 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.
- [118] Steven M. Nowick, Kenneth Y. Yun, and Peter 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.
- [119] N.Weste and K. Esraghian. *Principles of CMOS VLSI Design – A systems Perspective, 2nd edition*. Addison-Wesley, 1993.
- [120] International Standards Organization. LOTOS — a formal description technique based on the temporal ordering of observational behaviour. ISO IS 8807, 1989.
- [121] I. Miro Panades and A. Greiner. Bi-synchronous FIFO for synchronous circuit communication well suited for network-on-chip in gals architectures. In *Proc. IEEE/ACM Intl. Symposium on Networks-on-Chip (NOCS)*, pages 83–92, 2007.
- [122] 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. IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 32–42, 1998.

- [123] Nigel C. Paver. *The Design and Implementation of an Asynchronous Microprocessor*. PhD thesis, Department of Computer Science, University of Manchester, 1994.
- [124] Miroslav Pechoucek. Anomalous response times of input synchronizers. *IEEE Transactions on Computers*, C-25(2):133–139, Feb 1976.
- [125] Michael 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).
- [126] Ad Peeters, Frank te Beest, Mark de Wit, and Willem Mallon. Click elements: An implementation style for data-driven compilation. In *Proc. Intl. Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 3–14, 2010.
- [127] Ad M. G. Peeters. *Single-Rail Handshake Circuits*. PhD thesis, Eindhoven University of Technology, June 1996.
- [128] J. L. Peterson. Petri nets. *Computing Surveys*, 9(3):223–252, September 1977.
- [129] C. V. Ramamoorthy and Garry S. Ho. Performance evaluation of asynchronous concurrent systems using petri nets. *IEEE Transactions on Software Engineering*, 6(5):440–449, 1980.
- [130] 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.
- [131] M. Roncken. Defect-oriented testability for asynchronous ICs. *Proceedings of the IEEE*, 87(2):363–375, February 1999.
- [132] M. Roncken, S. M. Gilla, H. Park, N. Jamadagni, C. Cowan, and I. Sutherland. Naturalized communication and testing. In *Proc. IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 77–84. IEEE Computer Society Press, 2015.
- [133] S. M. Nowick and M. B. Josephs and C. H. (Kees) van Berkel (editors). Special Issue on Asynchronous Circuits and Systems. *Proceedings of the IEEE*, 87(2), February 1999.
- [134] Arash Saifhashemi and Peter A. Beerel. High level modeling of channel-based asynchronous circuits using verilog. *Concurrent Systems Engineering Series*, 63:275–288, 2005.

- [135] Arash Saifhashemi and Peter A. Beerel. System VerilogCSP: Modeling digital asynchronous circuits using systemverilog interfaces. *Concurrent Systems Engineering Series*, 68:287–302, 2011.
- [136] Charles L. Seitz. System timing. In Carver A. Mead and Lynn A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.
- [137] M. Singh and SM Nowick. MOUSETRAP: Ultra-high-speed transition-signaling asynchronous pipelines. In *Intl. Conference on Computer Design (ICCD)*, pages 9–17. IEEE Computer Society Press, 2001.
- [138] M. Singh and SM Nowick. MOUSETRAP: High-speed transition-signaling asynchronous pipelines. *IEEE Transactions on VLSI Systems*, 15(6):684–698, 2007.
- [139] N.P. Singh. A design methodology for self-timed systems. Master’s thesis, Laboratory for Computer Science, MIT, 1981. MIT/LCS/TR-258.
- [140] Danil Sokolov, Ivan Poliakov, and Alex Yakovlev. Asynchronous data path models. In *Proc. Intl. Conference on Application of Concurrency To System Design (ACSD)*, pages 197–210. IEEE, 2007.
- [141] Danil Sokolov, Ivan Poliakov, and Alex Yakovlev. Analysis of static data flow structures. *Fundamenta Informaticae*, 88(4):581–610, 2008.
- [142] J. Sparsø and S. Furber, editors. *Principles of asynchronous circuit design – A systems perspective*. Kluwer Academic Publishers, 2001.
- [143] 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.
- [144] Jens Sparsø, Christian D. Nielsen, Lars S. Nielsen, and Jørgen 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.
- [145] Jens Sparsø and Jørgen Staunstrup. Delay-insensitive multi-ring structures. *INTEGRATION, the VLSI Journal*, 15(3):313–340, October 1993.
- [146] Leon Stok. *Architectural Synthesis and Optimization of Digital Systems*. PhD thesis, Eindhoven University of Technology, 1991.

- [147] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.
- [148] Synopsys, Inc. *Synopsys VSS Family Core Programs Manual*, 1997.
- [149] Paul Teehan, Mark Greenstreet, and Guy Lemieux. A survey and taxonomy of gals design styles. *IEEE Design and Test of Computers*, 24(5):418–28, 418–428, 2007.
- [150] Roger L. Traylor. Self-timed data pipeline apparatus using asynchronous stages having toggle flip-flops, January 1995. U.S. Patent 5,386,585.
- [151] S. H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley-Interscience, John Wiley & Sons, Inc., New York, 1969.
- [152] C. H. van Berkel. Beware the isochronic fork. *INTEGRATION, the VLSI journal*, 13(3):103–128, 1992.
- [153] C. H. van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.
- [154] C. H. van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, and Frits Schalij. Asynchronous Circuits for Low Power: a DCC Error Corrector. *IEEE Design & Test*, 11(2):22–32, 1994.
- [155] 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.
- [156] 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.
- [157] Kees van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, and Frits 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.
- [158] Kees van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, Frits Schalij, and Rik 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.

- [159] Hans van Gageldonk, Daniel Baumann, Kees van Berkel, Daniel Gloor, Ad Peeters, and Gerhard Stegmann. An asynchronous low-power 80C51 microcontroller. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 96–107, 1998.
- [160] Peter Vanbekbergen. *Synthesis of Asynchronous Control Circuits from Graph-Theoretic Specifications*. PhD thesis, Catholic University of Leuven, September 1993.
- [161] V. I. Varshavsky, M. A. Kishinevsky, V. B. Marakhovsky, V. A. Peschan-sky, L. Y. Rosenblum, A. R. Taubin, and B. S. Tzirlin. *Self-timed Control of Concurrent Processes*. Kluwer Academic Publisher, 1990. V.I.Varshavsky Ed., (Russian edition: 1986).
- [162] T. Verhoeff. Delay-insensitive codes - an overview. *Distributed Comput-ing*, 3(1):1–8, 1988.
- [163] P. Viviet and M. Renaudin. CHP2VHDL, a CHP to VHDL transla-tor - Towards asynchronous-design simulation. In L. Lavagno and M.B. Josephs, editors, *Handouts from the ACiD-WG Workshop on Specifica-tion models and languages and technology effects of asynchronous design*. Dipartimento di Elettronica, Politecnico de Torino, Italy, January 1998.
- [164] J. Wang. *Timed petri nets: Theory and Application*, volume DEDS 9. Kluwer Academic Publishers, 1998.
- [165] P. Wielage, J.E. Marinissen, M. Altheimer, and C. Wouters. Design and DfT of a high-speed area-efficient embedded asynchronous FIFO. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 853–858, 2007.
- [166] 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.
- [167] Ted Williams, Niteen Patkar, and Gene 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.
- [168] Ted E. Williams. *Self-Timed Rings and their Application to Division*. PhD thesis, Department of Electrical Engineering and Computer Sci-ence, Stanford University, 1991. CSL-TR-91-482.
- [169] Ted E. Williams. Analyzing and improving latency and throughput in self-timed rings and pipelines. In *Tau-92: 1992 Workshop on Timing Is-sues in the Specification and Synthesis of Digital Systems*. ACM/SIGDA, March 1992.

- [170] Ted E. Williams. Performance of iterative computation in self-timed rings. *Journal of VLSI Signal Processing*, 7(1-2):17–31, 1994.
- [171] Chantal Ykman-Couvreur, Bill Lin, and Hugo de Man. Assassin: A synthesis system for asynchronous control circuits. Technical report, IMEC, September 1994. User and Tutorial manual.
- [172] Kenneth Y. Yun and David 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.
- [173] Jun Zhou, David Kinniment, Gordon Russell, and Alex Yakovlev. A robust synchronizer. In *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)*, pages 2 pp.–, March 2006.

Index

- acknowledgment (or indication), 15
- actual case latency, 78
- addition (ripple-carry), 77
- and-or-invert (AOI) gates, 112
- arbiter, 34, 190
- arbitration, 89, 141, 190
- asymmetric delay, 56, 139
- asynchronous advantages, 1
- atomic complex gate, 104, 113

- Balsa, 204
- barrier, 180, 183
- better than worst-case, 167
- bit-serial, 162
- bubble, 30
- bubble limited, 57
- bundled-data, 9
- burst mode, 96
 - input burst, 96
 - output burst, 96

- C-element, 14, 64, 102
 - asymmetric, 64, 110
 - generalized, 110, 113, 115
 - implementation, 15
 - specification, 15, 102
- Caltech, 214
- capture-pass latch, 19
- CCS (calculus of communicating systems), 204
- channel, 5, 30
 - probe, 203
 - receive, 203
 - send, 203
- channel type
 - biput, 196
 - nonput, 195
 - pull, 10, 195
 - push, 10, 195
- CHP (communicating hardware processes), 204
- circuit templates:
 - for statement, 43
 - if statement, 42
 - while statement, 44
- classification
 - delay-insensitive (DI), 25
 - quasi delay-insensitive (QDI), 25
 - self-timed, 26
 - speed-independent (SI), 25
- click element, 175
 - alternative implementation, 176
- clock
 - stretchable, 158
- closed circuit, 23
- codeword (dual-rail), 11
 - empty, 12
 - intermediate, 12
 - valid, 12
- compatible states, 95
- complete state coding (CSC), 98
- completion
 - detection, 21, 22
 - indication, 75
 - strong, 75
 - weak, 75
- completion detector, 73
- completion indication, 78
- complex gates, 114
- Concurrent processes, 203
- concurrent statements, 203

- consistent state assignment, 98
- control limited, 58
- control logic for transition signaling, 20
- control-data-flow graphs, 42
- CSP (communicating sequential processes), 203
- cycle time
 - of a handshake, 129
 - of a ring, 57
- data encoding
 - bundled-data, 9
 - dual-rail, 11
 - m-of-n, 14
 - one-hot (or 1-of-n), 13
 - single-rail, 10
- data limited, 57
- data validity scheme (4-phase bundled-data)
 - broad, 196
 - early, 196
 - extended early, 196
 - late, 196
- data-flow abstraction, 6
- DCVSL, 83
- deadlock, 30
- delay assumptions, 23
- delay insensitive minterm synthesis (DIMS), 80
- delay matching, 11
- delay model
 - fixed delay, 93
 - inertial delay, 93
 - inertial delay, delay time, 93
 - inertial delay, reject time, 93
 - min-max delay, 93
 - transport delay, 93
 - unbounded delay, 93
- delay selection, 79
- delay-insensitive (DI), 11, 17, 25
 - codes, 12
- demultiplexer (DEMUX), 34, 187
- differential logic, 83
- DIMS, 80, 81
- dual-rail carry signals, 78
- dual-rail encoding, 11
- dummy environment, 97
- dynamic wavelength, 57
- empty word, 11, 12, 29, 30
- environment, 94
- escapement system, 159
- event, 10
- event graph (EG), 127
- excitation region, 107
- excited gate/variable, 23
- Fibonacci circuit, 38, 182, 191
- FIFO, 17
- finite state machine (using a ring), 37
- firing (of a gate), 24
- for statement, 43
- fork, 34, 185
- forward latency, 55
- FPGA prototyping, 191
 - Fibonacci circuit, 191
 - GCD circuit, 191
- function block, 33, 73, 74
 - bundled-data, 18, 78
 - bundled-data (“speculative completion”), 79
 - dual-rail, 22
 - dual-rail (DIMS), 80
 - dual-rail (Martin’s adder), 84
 - dual-rail (null convention logic), 82
 - dual-rail (transistor level CMOS), 83
 - hybrid, 86
 - strongly indicating, 75
 - weakly indicating, 75
- fundamental mode, 92, 94
- generalized C-element, 113, 115
- generate (carry), 77
- globally-asynchronous locally-synchronous (GALS) systems, 155

- greatest common divisor (GCD), 44, 191, 211
- guarded command, 207
- guarded repetition, 207
- handshake channel, 195
 - biput, 196
 - nonput, 195, 209
 - pull, 10, 195, 209
 - push, 10, 195, 209
- handshake circuit, 208
 - 2-place ripple FIFO, 211
 - 2-place shift register, 209
 - greatest common divisor (GCD), 212
- handshake component
 - arbiter, 34, 89
 - bar, 211
 - conditional send, 67
 - demultiplexer, 34, 65, 211
 - DEMUX with latched outputs, 68
 - do, 211
 - fork, 34, 63, 211, 213
 - join, 34, 63, 210
 - latch, 29, 32, 61, 184
 - 2-phase bundled-data, 19, 184
 - 4-phase bundled-data, 18, 117
 - 4-phase dual-rail, 21
 - merge, 34, 63
 - multiplexer, 34, 65, 119, 211
 - mutex, 34
 - passivator, 210
 - repeater, 210
 - sequencer, 210
 - transferer, 210
 - variable, 210
- handshake expansion, 214
- handshake latch, 29, 184
- handshake protocol, 5, 9
 - 2-phase bundled-data, 9
 - 2-phase dual-rail, 13
 - 4-phase bundled-data, 9
 - 4-phase bundled-data, 197
 - 4-phase dual-rail, 11
 - non-return-to-zero (NRZ), 10
 - return-to-zero (RTZ), 10
- handshaking, 5
- hazard
 - dynamic-01, 93
 - dynamic-10, 93, 105
 - static-0, 93
 - static-1, 93, 104
- Huffmann, David A., 94
- hysteresis, 22, 77
- if statement, 42
- IFIR filter bank, 45
- indication (or acknowledgement)
 - dependency graphs, 85
 - distribution of valid/empty indication, 84
 - of completion, 78
 - strong, 75
 - weak, 75
- indication (or acknowledgment), 15
- initial marking, 127
- initial state, 111
- initialization, 30, 111, 182
 - barrier, 180
- input free choice, 98
- input-output mode, 92, 94
- intermediate codeword, 12
- isochronic fork, 26
- iterative computation (using a ring), 37
- join, 34, 185
- kill (carry), 77
- latch (see also: handshake comp.), 18
- latch controller, 117
 - click elements, 175
 - fully-decoupled, 200
 - Loihi, 176
 - mousetrap, 174
 - normally opaque, 201
 - normally transparent, 201
 - phase-decoupled click, 179

- semi-decoupled, 200
 - simple/un-decoupled, 199
- latency, 55
 - actual case, 78
- link (see channel), 5, 30
- liveness, 98
- logic decomposition, 104
- logic thresholds, 27
- Loihi, 176
- LOTOS, 204
- m-of-n threshold gates with hysteresis, 82
- marked graph (MG), 127
- matched delay, 11, 78
- memory, 69
 - read/write, 70, 71
 - read/write/read-before-write, 70
 - read/write/write-before-read, 71
- merge, 34, 186
- mesochronous, 161
 - communication link, 164
- mesochronous link
 - self-timed at receivers input (STARI), 165
 - using a dual-clock FIFO, 164
 - using an asynchronous FIFO, 165
- metastability, 89, 141, 142
 - filter, 151
 - halfway, 144
 - mean time between failure (MTBF), 146
 - measuring τ and Δ , 149
 - oscillating, 144
 - probability of, 144
 - synchronization, 151
 - time-safe systems, 153
 - value-safe systems, 153
 - values for τ and Δ , 148
- metastability filter, 150
- metastability detector, 158
- micropipelines, 19, 173
- microprocessors
 - asynchronous MIPS, 46
 - asynchronous MIPS R3000, 214
- minterm, 22, 80
- monotonic cover constraint, 107, 109, 114
- mousetrap, 174
- Muller C-element, 15
- Muller model of a closed circuit, 23
- Muller pipeline/distributor, 16
- Muller, David, 94
- multi-clock systems, 155
- multiplexer (MUX), 34, 119
- multiplexor (MUX), 187
- mutual exclusion, 64, 88, 150, 190
 - handshake component (MUTEX), 34, 88
 - RGD mutex, 190
- NCL adder, 83
- non-return-to-zero (NRZ), 10
- NULL, 11
- Null Convention Logic (NCL), 82
- OCCAM, 204
- occupancy (or static spread), 57
- one-hot encoding, 13
- operator reduction, 215
- peephole optimization, 67, 68, 188
 - fused components, 188
- performance
 - analysis and optimization, 49
- performance parameters:
 - cycle time of a ring, 57
 - dynamic wavelength, 57
 - forward latency, 55
 - latency, 55
 - period, 56
 - reverse latency, 56
 - throughput, 57
- period, 56
- periodic, 162
- persistence, 98
- Petri net, 96, 125
 - 1-bounded, 98
 - choice, 127

- confusion, 127
 - controlled choice, 99
 - firing, 96
 - fork, 98
 - initial marking, 127
 - input free choice, 98
 - join, 98
 - liveness, 98
 - marked graph, 127
 - merge, 98
 - places, 96
 - state machine, 127, 128
 - timed, 125
 - timed-place (TPPN), 125
 - timed-transition (TTPN), 125
 - token, 96
 - transition, 96
- Petrify, 113
- phase-decoupled click component
 - arbiter, 190
 - delay element, 185
 - DEMUX, 187
 - fork, 185
 - function block, 185
 - fused components, 188
 - handshake latch, 184
 - join, 185
 - merge, 186
 - mutual exclusion, 190
 - MUX, 187
- phase-decoupled click components, 184
- phase-decoupled handshake latches, 181
- phase-decoupled handshaking, 179
- pipeline, 3, 30
 - 2-phase bundled data, 171
 - 2-phase bundled-data, 19
 - 4-phase bundled-data, 18
 - 4-phase dual-rail, 20
- place, 96
- plesiochronous, 162
- precharged CMOS circuitry, 198
- primitive flow table, 95
- probe, 203, 205
- process decomposition, 214
- production rule expansion, 214
- propagate (carry), 77
- pull channel, 10, 195
- push channel, 10, 195
- quasi delay-insensitive (QDI), 25
- quiescent region, 107
- Razor, 167
- re-shuffling signal transitions, 112, 122
- read pointer, 157
- read-after-write data hazard, 46
- receive, 203, 205
- reduced flow table, 95
- register
 - locking, 46
- rendezvous, 205
- reset function, 107
- return-to-zero (RTZ), 9, 10
- reverse latency, 56
- RGD mutex, 190
- ring, 30
 - finite state machine, 37
 - iterative computation, 37
- ripple FIFO, 17
- self-timed, 26
- self-timed at receivers input (STARI), 165
- semantics-preserving transformations, 214
- send, 203, 205
- set function, 107
- Set-Reset implementation, 106
- shared resource, 88
- shift register
 - with parallel load, 52
- signal transition, 10
- signal transition graph (STG), 96
- silicon compiler, 204
- single input change, 94
- single-rail, 10
- spacer, 11
- speculative completion, 79
- speed-independent (SI), 23–25, 93

- spread token semantics, 39
- stable gate/variable, 23
- standard C-element, 116
 - implementation, 106
- state graph, 95
- state machine (SM), 127, 128
- static data flow
 - 2-phase bundled-data, 177
 - two stage ring, 177
- static data-flow structure, 5, 29
 - for, if, and while constructs, 42
 - greatest common divisor (GCD), 44
 - IFIR filter bank, 45
 - MIPS microprocessor, 46
 - Read/write memory, 70
 - read/write/read-before-write memory, 70
 - read/write/write-before-read, 71
 - simple example, 35
 - vector multiplier, 46
- static spread (or occupancy), 57, 199
- static type checking, 198
- stretchable clock, 158
- stuck-at fault model, 27
- synchronization, 141, 151
 - in handshake interface, 155
 - using dual-clock FIFO, 157
 - using dual-ported memory, 156
- synchronous message passing, 203
- syntax-directed compilation, 208
- Tangram, 204
- Tangram examples:
 - 2-place ripple FIFO, 207
 - 2-place shift register, 206
 - GCD using guarded repetition, 208
 - GCD using while and if statements, 208
- taxonomy of timing organizations, 161
 - asynchronous, 162
 - mesochronous, 161
 - periodic, 162
 - plesiochronous, 162
 - synchronous, 161
- technology mapping, 114
- test, 27
 - I_{DDQ} testing, 28
 - halting of circuit, 28
 - isochronic forks, 28
 - short and open faults, 28
 - stuck-at faults, 27
 - toggle test, 28
 - untestable stuck-at faults, 28
- throughput, 50, 57
- time separation
 - between signal transitions, 129
- time-safe systems, 153
- timed Petri net, 125
- timed-place Petri net (TPPN), 125
- timed-transition Petri net (TTPN), 125
- token, 5, 30, 96
 - spread token semantics, 39
- TPPN, timed-place Petri net, 125
- transition, 96
- transparent to handshaking, 5, 23, 34, 74
- TTPN, timed-transition Petri net, 125
- unique entry constraint, 107, 109
- valid codeword, 12
- valid data, 11, 29
- valid token, 30
- value-safe clocking
 - escapement system, 159
 - with metastability, 158
 - without metastability, 159
- value-safe systems, 153
- vector multiplier, 46
- Verilog, 205
- VHDL, 204
- VLSI programming, 208
- VSTGL (Visual STG Lab), 113
- wave, 16
 - crest, 17
 - trough, 17
- while statement, 44

WorkCraft, 113, 114
write pointer, 157
write-back, 46



Jens Sparsø is a Professor at the Technical University of Denmark (DTU) in the department for Applied Mathematics and Computer Science, where he has taught courses on digital electronics, VLSI-design, digital systems, computer architecture and asynchronous circuits. His research field is hardware platforms for embedded systems and he has special interests in asynchronous circuits, reconfigurable hardware, application-specific multi-core processors and networks-on-chip.

This book is an introduction to the design of asynchronous circuits. It is an updated and significantly extended version of an eight-chapter tutorial that first appeared as Part I in the book "Principles of asynchronous circuit design - A systems perspective," edited by Sparsø and Furber (2001); a book that has become a standard reference on the topic.

The extensions include improved coverage of data-flow components, a new chapter on two-phase bundled-data circuits, a new chapter on metastability, arbitration, and synchronization, and a new chapter on performance analysis using timed Petri nets. With these extensions, the text provides more complete coverage of the field and is now made available as a stand-alone book. The book is a beginner's text, and the amount of formal notation is deliberately kept at a minimum, using instead plain English and graphical illustrations to explain the underlying intuition and reasoning behind the concepts and methods covered.

The book targets senior undergraduate and graduate students in Electrical and Computer Engineering and industrial designers with a background in conventional (clocked) digital design who wish to gain an understanding of asynchronous circuit design. The book aims to enable its readers to design asynchronous control and data processing circuits of small and medium complexity, to read the literature, and to decide where/whether to use asynchronous circuits in new designs.

