

Self-Timed Control of Concurrent Processes

Mathematics and Its Applications (*Soviet Series*)

Managing Editor:

M. HAZEWINKEL

Centre for Mathematics and Computer Science, Amsterdam, The Netherlands

Editorial Board:

A. A. KIRILLOV, *MGU, Moscow, U.S.S.R.*

Yu. I. MANIN, *Steklov Institute of Mathematics, Moscow, U.S.S.R.*

N. N. MOISEEV, *Computing Centre, Academy of Sciences, Moscow, U.S.S.R.*

S. P. NOVIKOV, *Landau Institute of Theoretical Physics, Moscow, U.S.S.R.*

M. C. POLYVANOV, *Steklov Institute of Mathematics, Moscow, U.S.S.R.*

Yu. A. ROZANOV, *Steklov Institute of Mathematics, Moscow, U.S.S.R.*

Self-Timed Control of Concurrent Processes

**The Design of Aperiodic Logical Circuits
in Computers and Discrete Systems**

edited by

Victor I. Varshavsky

*Computing Science Department, Leningrad Electrical Engineering Institute,
Leningrad, U.S.S.R.*

with contributions by

Mikhail A. Kishinevsky, Vyatcheslav B. Marakhovsky,
Valerij A. Peschansky, Leonid Ya. Rosenblum, Alexandre R. Taubin,
Boris S. Tzirlin, and Victor I. Varshavsky



KLUWER ACADEMIC PUBLISHERS
DORDRECHT / BOSTON / LONDON

Library of Congress Cataloging in Publication Data

Avtomatnoe upravlenie asinkhronnymi protsessami v ÈVM i diskretnykh sistemakh. English

Self-timed control of concurrent processes : models and principles for designing aperiodic logical circuits in computers and discrete systems / edited by Victor I. Varshavsky ; with contributions by Mikhail A. Kishinevsky ... [et al. ; translated from the Russian by Alexandre V. Yakovlev].

p. cm. -- (Mathematics and its applications. Soviet Series : v. 52)

Translation of Avtomatnoe upravlenie asinkhronnymi protsessami v ÈVM i diskretnykh sistemakh.

Includes bibliographical references.

ISBN-13:978-94-010-6705-8

1. Computer architecture. 2. Logic design. 3. Discrete-time systems. I. Varshavskii, V. I. (Viktor Il'ich) II. Kishinevskii, M. A. (Mikhail Aleksandrovich) III. Yakovlev, Alexandre V. IV. Title. V. Series: Mathematics and its applications (Kluwer Academic Publishers). Soviet series : 52.

QA76.9.A73A9813 1990

004.2'2--dc20

89-24541

ISBN-13: 978-94-010-6705-8 e-ISBN-13: 978-94-009-0487-3

DOI: 10.1007/978-94-009-0487-3

Published by Kluwer Academic Publishers,
P.O. Box 17, 3300 AA Dordrecht, The Netherlands.

Kluwer Academic Publishers incorporates
the publishing programmes of
D. Reidel, Martinus Nijhoff, Dr W. Junk and MTP Press.

Sold and distributed in the U.S.A. and Canada
by Kluwer Academic Publishers,
101 Philip Drive, Norwell, MA 02061, U.S.A.

In all other countries, sold and distributed
by Kluwer Academic Publishers Group,
P.O. Box 322, 3300 AH Dordrecht, The Netherlands.

Printed on acid-free paper

This is the translation of the original work

Автоматное управление асинхронными процессами в ЭВМ и дискретных системах
Published by Nauka Publishers, Moscow, © 1986.

Translated from the Russian by Alexandre V. Yakovlev.

All Rights Reserved

This English edition © 1990 by Kluwer Academic Publishers
Softcover reprint of the hardcover 1st edition 1990

No part of the material protected by this copyright notice may be reproduced or utilized in any form or by any means, electronic or mechanical including photocopying, recording, or by any information storage and retrieval system, without written permission from the copyright owner.

SERIES EDITOR'S PREFACE

'Et moi, ..., si j'avait su comment en revenir,
je n'y serais point allé.'

Jules Verne

The series is divergent; therefore we may be
able to do something with it.

O. Heaviside

One service mathematics has rendered the
human race. It has put common sense back
where it belongs, on the topmost shelf next
to the dusty canister labelled 'discarded non-
sense'.

Eric T. Bell

Mathematics is a tool for thought. A highly necessary tool in a world where both feedback and nonlinearities abound. Similarly, all kinds of parts of mathematics serve as tools for other parts and for other sciences.

Applying a simple rewriting rule to the quote on the right above one finds such statements as: 'One service topology has rendered mathematical physics ...'; 'One service logic has rendered computer science ...'; 'One service category theory has rendered mathematics ...'. All arguably true. And all statements obtainable this way form part of the *raison d'être* of this series.

This series, *Mathematics and Its Applications*, started in 1977. Now that over one hundred volumes have appeared it seems opportune to reexamine its scope. At the time I wrote

"Growing specialization and diversification have brought a host of monographs and textbooks on increasingly specialized topics. However, the 'tree' of knowledge of mathematics and related fields does not grow only by putting forth new branches. It also happens, quite often in fact, that branches which were thought to be completely disparate are suddenly seen to be related. Further, the kind and level of sophistication of mathematics applied in various sciences has changed drastically in recent years: measure theory is used (non-trivially) in regional and theoretical economics; algebraic geometry interacts with physics; the Minkowsky lemma, coding theory and the structure of water meet one another in packing and covering theory; quantum fields, crystal defects and mathematical programming profit from homotopy theory; Lie algebras are relevant to filtering; and prediction and electrical engineering can use Stein spaces. And in addition to this there are such new emerging subdisciplines as 'experimental mathematics', 'CFD', 'completely integrable systems', 'chaos, synergetics and large-scale order', which are almost impossible to fit into the existing classification schemes. They draw upon widely different sections of mathematics."

By and large, all this still applies today. It is still true that at first sight mathematics seems rather fragmented and that to find, see, and exploit the deeper underlying interrelations more effort is needed and so are books that can help mathematicians and scientists do so. Accordingly MIA will continue to try to make such books available.

If anything, the description I gave in 1977 is now an understatement. To the examples of interaction areas one should add string theory where Riemann surfaces, algebraic geometry, modular functions, knots, quantum field theory, Kac-Moody algebras, monstrous moonshine (and more) all come together. And to the examples of things which can be usefully applied let me add the topic 'finite geometry'; a combination of words which sounds like it might not even exist, let alone be applicable. And yet it is being applied: to statistics via designs, to radar/sonar detection arrays (via finite projective planes), and to bus connections of VLSI chips (via difference sets). There seems to be no part of (so-called pure) mathematics that is not in immediate danger of being applied. And accordingly, the applied mathematician needs to be aware of much more. Besides analysis and numerics, the traditional workhorses, he may need all kinds of combinatorics, algebra, probability and so on.

In addition, the applied scientist needs to cope increasingly with the nonlinear world and the extra mathematical sophistication that this requires. For that is where the rewards are. Linear models are honest and a bit sad and depressing: proportional efforts and results. It is in the non

linear world that infinitesimal inputs may result in macroscopic outputs (or vice versa). To appreciate what I am hinting at: if electronics were linear we would have no fun with transistors and computers; we would have no TV; in fact you would not be reading these lines.

There is also no safety in ignoring such outlandish things as nonstandard analysis, superspace and anticommuting integration, p -adic and ultrametric space. All three have applications in both electrical engineering and physics. Once, complex numbers were equally outlandish, but they frequently proved the shortest path between 'real' results. Similarly, the first two topics named have already provided a number of 'wormhole' paths. There is no telling where all this is leading - fortunately.

Thus the original scope of the series, which for various (sound) reasons now comprises five sub-series: white (Japan), yellow (China), red (USSR), blue (Eastern Europe), and green (everything else), still applies. It has been enlarged a bit to include books treating of the tools from one subdiscipline which are used in others. Thus the series still aims at books dealing with:

- a central concept which plays an important role in several different mathematical and/or scientific specialization areas;
- new applications of the results and ideas from one area of scientific endeavour into another;
- influences which the results, problems and concepts of one field of enquiry have, and have had, on the development of another.

Time is a problem, as many people have noticed; timing even more so. In spite of that, and in spite of a large-scale preoccupation with dynamical phenomena in many parts of mathematics, 'timing' as a serious research topic is a recent one. Concurrently (sic), time has explicitly entered the worlds of logic and automata.

Mathematics and computer science have been grouped together (by Von Weisäcker) as the structure sciences with mathematics dealing with static structures and computer science with dynamic ones. I am inclined to dispute the latter half of this idea, but it is not to be denied that the particular set of time and timing problems discussed in this volume come from the world of computers, more precisely they are mathematical problems connected with the design of asynchronous or self-timed chips (speed independent circuits, or circuits (whose functioning is) independent of element delays).

With the advent of very large complex circuits, even before multiprocessor boards and distributed systems, self-timing has become an important issue that needs and merits serious investigation both at the level of actual efficient design and at the level of a systematic underlying mathematical theory (which, so far, does not yet exist).

These problems of design are new in character and they differ in principle from the more traditional ones. As already noted, a fully fledged systematic theory does not yet exist (but there do exist several heuristics). The present, definitely systematically inclined volume, will certainly help significantly in bringing it about that this rather disturbing hole in our mathematical/computer science expertise is taken care of. It brings and describes a set of serious and fascinating challenges to the world of mathematics.

The shortest path between two truths in the real domain passes through the complex domain.

J. Hadamard

La physique ne nous donne pas seulement l'occasion de résoudre des problèmes ... elle nous fait pressentir la solution.

H. Poincaré

Never lend books, for no one ever returns them; the only books I have in my library are books that other folk have lent me.

Anatole France

The function of an expert is not to be more right than other people, but to be wrong for more sophisticated reasons.

David Butler

CONTENTS

Preface to the English Edition	xi
From the Editor	xv
Chapter 1 Introduction	1
Chapter 2 Asynchronous processes and their interpretation	12
2.1 Asynchronous processes	12
2.1.1 Definition	12
2.1.2 Some subclasses	17
2.1.3 Reposition	20
2.1.4 Structured situations	22
2.1.5 An asynchronous process as a metamodel	24
2.2 Petri nets	25
2.2.1 Model description	25
2.2.2 Some classes	27
2.2.3 Interpretation	28
2.3 Signal graphs	30
2.4 The Muller model	33
2.5 Parallel asynchronous flow charts	36
2.6 Asynchronous state machines	39
2.7 Reference notations	41
Chapter 3 Self-synchronizing codes	43
3.1 Preliminary definitions	43
3.2 Direct-transition codes	48
3.3 Two-phase codes	51
3.4 Double-rail code	52
3.5 Code with identifier	53
3.6 Optimally balanced code	57
3.7 On the code redundancy	59
3.8 Differential encoding	61
3.9 Reference notations	63
Chapter 4 Aperiodic circuits	64
4.1 Two-phase implementation of finite state machine	65
4.1.1 Matched implementation	68
4.2 Completion indicators and checkers	69

4.3	Synthesis of combinatorial circuits	74
4.3.1	Indicatability	77
4.3.2	Standard implementations	85
4.3.2.1	Minimum form implementation	85
4.3.2.2	Orthogonal form implementation	86
4.3.2.3	Hysteresis flip-flop-based implementation	87
4.3.2.4	Implementation based on “collective responsibility”	87
4.4	Aperiodic flip-flops	89
4.4.1	Further discussion of flip-flop designs	91
4.4.1.1	RS-flip-flops	91
4.4.1.2	D-flip-flops	94
4.4.1.3	T-flip-flops	96
4.5	Canonical aperiodic implementations of finite state machines	98
4.5.1	Implementation with delay flip-flops	98
4.5.2	Implementation using flip-flops with separated inputs	100
4.5.3	Implementation with complementing flip-flops	102
4.6	Implementation with multiple phase signals	106
4.7	Implementation with direct transitions	109
4.8	On the definition of an aperiodic state machine	111
4.9	Reference notations	112
Chapter 5	Circuit modelling of control flow	114
5.1	The modelling of Petri nets	115
5.1.1	Event-based modelling	115
5.1.2	Condition-based modelling	119
5.2	The modelling of parallel asynchronous flow charts	124
5.2.1	Implementation of standard fragments	124
5.2.2	A multiple use circuit	128
5.2.3	A loop control circuit	130
5.2.4	Using an arbiter	132
5.2.5	Guard-based implementation	137
5.3	Functional completeness and synthesis of semi-modular circuits	140
5.3.1	Formulation of the problem	141
5.3.2	Some properties of semi-modular circuits	142
5.3.3	Perfect implementation	144
5.3.4	Simple circuits	149
5.3.5	The implementation of distributive and totally sequential circuits	159
5.4	Synthesis of semi-modular circuits in limited bases	165

5.5	Modelling pipeline processes	175
5.5.1	Properties of modelling pipeline circuits	177
5.5.1.1	Pipelinization of parallel fragments	182
5.5.1.2	Pipelinization of a conditional branch	183
5.5.1.3	Transformation of a loop	184
5.5.1.4	Pipelinization for multiply-used sections	185
5.6	Reference notations	187
Chapter 6	Composition of asynchronous processes and circuits	189
6.1	Composition of asynchronous processes	189
6.1.1	Reinstated process	189
6.1.2	Process reduction	190
6.1.3	Process composition	193
6.2	Composition of aperiodic circuits	196
6.2.1	The Muller theorem	197
6.2.2	The generalization of the Muller theorem	198
6.3	Algebra of asynchronous circuits	203
6.3.1	Operations on circuits	204
6.3.2	Laws and properties	207
6.3.3	Circuit transformations	209
6.3.4	Homological algebras of circuits	211
6.4	Reference notations	214
Chapter 7	The matching of asynchronous processes and interface organization	215
7.1	Matched asynchronous processes	216
7.2	Protocol	217
7.3	The matching asynchronous process	219
7.4	The T2 interface	222
7.4.1	General notations	222
7.4.2	Communication protocol	225
7.4.3	Implementation	228
7.5	Asynchronous interface organization	231
7.5.1	Using the code with identifier	233
7.5.2	Using the optimally-balanced code	237
7.5.2.1	Half-byte data transfer	237
7.5.2.2	Byte data transfer	238
7.5.2.3	Using non-balanced representation	239
7.6	Reference notations	242
Chapter 8	Analysis of asynchronous circuits and processes	243
8.1	The reachability analysis	244
8.2	The classification analysis	252

8.3	The set of operational states	258
8.4	The effect of non-zero wire delays	264
8.5	Circuit Petri nets	273
8.6	On the complexity of analysis algorithms	278
8.7	Reference notations	279
Chapter 9	Anomalous behaviour of logical circuits and the arbitration problem	281
9.1	Arbiters	284
9.2	Oscillatory anomaly	286
9.3	Meta-stability anomaly	288
9.4	Designing correctly-operating arbiters	292
9.5	“Bounded” arbiters and safe inertial delays	300
9.6	Reference notations	307
Chapter 10	Fault diagnosis and self-repair in aperiodic circuits	309
10.1	Totally self-checking combinational circuits	311
10.2	Totally self-checking sequential machines	313
10.3	Fault detection in autonomous circuits	314
10.4	Self-repair organization for aperiodic circuits	321
10.5	Reference notations	328
Chapter 11	Typical examples of aperiodic design modules	329
11.1	The JK-flip-flop	329
11.2	Registers	331
11.3	Pipeline registers	336
11.3.1	Non-dense registers	336
11.3.2	Semi-dense pipeline register	339
11.3.3	Dense pipeline registers	340
11.3.4	One-byte dense pipeline register	344
11.3.5	Pipeline register with parallel read-write and the stack	345
11.3.6	Reversible pipeline registers	349
11.4	Converting single-rail signals into double-rail ones	352
11.4.1	Parallel register with single-rail inputs	352
11.4.2	Input and output heads of pipeline registers	353
11.5	Counters	355
11.6	Reference notations	363
Editor's Epilogue		364
References		370
Index		393

PREFACE TO THE ENGLISH EDITION

“Time present and time past
Are both perhaps present in time future
And time future contained in time past.
If all time is eternally present
All time is unredeemable.
What might have been is an abstraction
Remaining a perpetual possibility
Only in a world speculation”.

T.S. Eliot

The book as it is presented to the English-reading audience contains results that had been obtained by its authors up to 1983. The Rusian edition was published in 1986, and hence a legitimate question may arise of how the authors consider the fact that the English edition is appearing in 1989. To give a plausible answer, it is expedient to turn our attention briefly to history.

The roots of the theory of self-timed speed-independent circuits go back as far as early 50s when D.E. Muller and his colleagues at the University of Illinois developed a fundamental algebraic theory which may be regarded as an excellent example of originality, perfection and completeness. Perhaps, the only complaint related to its completeness was concerned with a fundamental problem which was left open, the problem of the synthesis of self-timed circuits in physically existing, constrained, bases of functional elements. Due to this fact, as well as of course the lack of an adequate “social request”, self-timed circuits had, for nearly fifteen years, been put off the agenda as a n exotic of theoretical subject.

To be frank we should note that, in the meantime, the synthesis problem was solved by G. Zemanek, and his solution was reported at the 1962 IFAC International Meeting on Theory of Relay Networks and Finite Automata held in Moscow. However, the complexity of the proposed solution, and what is more important, the complexity of the formal presentation, meant that Zemanek's results were almost unnoticed. I attended his talk but, to my great shame, had clearly understood “what it was all about ” only ten years later. The fate of the later publication of Armstrong, Friedman and Menon (1969) was similar. (Also see [145].)

The beginning of the 70s was marked as the birth of the second wave of works on self-timed systems. Here we should first of all mention the J. Dennis' group of the MAC Project at the Massachusetts Institute of Technology, and the group from the Aerospace Automation Centre at Toulouse.

What was the most specific feature in this period of the evolution of the subject? To my mind, the major goal, both at MIT and at Toulouse, was to create a general theory. Time has shown that this goal, though laudable and justifiable, was

rather premature in view of the already existing results of Muller and an obvious lack of extensive design practice. These two groups were, however, very productive and obtained many interesting and useful results, even though a general theory of self-timed circuits has not, thus far, emerged.

When we started, or better say, joined the pursuit in this field in the early 70s we were, to our great shame, absolutely unaware of Muller's results. Nor did we know anything about the work at MIT and Toulouse. It was only two years later that Professor E. Moore drew our attention to these developments. On the other hand, it now seems that our lack of awareness later stood us in good stead.

What were the main rationale and goals of our early work? Launching the design project for a special-purpose computer, required to operate at high speed, we were rather surprised to find that the majority of standard IC packages (particularly in the case of shift-registers) were totally inoperable for some undesirable gate delay combinations. (It should also be borne in mind that wire delays were ignored at that time.) Gate delays varied a great deal even for such technologically well-developed ICs as those produced by Texas Instruments. The delay variations presented in TI's Market Catalogue were of 5-20 ns for SN9703N and 20-70 ns for SN49803N.

Thus, aiming, on the one hand, at ensuring that circuit behaviour was independent of poor delay ratios and, on the other hand, at gaining the highest performance from the operation based on actual delays, rather than those given in the IC specifications, we were led to the idea of organizing an indication of completion of a transition process. The early collection of several examples of circuits designed in this way was greatly extended. It still keeps growing and thus far constitutes rather a substantial library backed up by more than one hundred patent application certificates.

The opportunity to design circuits with the process completion indication capability at the logical level had evidently opened up the way for further development of the self-timing principle. This had, however, raised a fundamental question about the possibility of organizing a completion indication for an arbitrary circuit. In my report "Dead-beat automata with self-timing" at the 1974 IFAC Symposium on Discrete Systems, I had shown that any finite-state machine can be implemented with the indication of transition process completion in all its internal elements for the AND-OR-NOT logical basis (without constraints on element complexity).

This approach had, by and large, determined the long term strategy and methodology of our research and development work. This can be summarized as designing modules with the completion indication capability at various complexity levels and modular synthesis of composite structures based on formal specifications of joint module functioning. The first stage results were shown in 1976 in the

Russian edition of the book entitled "Aperiodic Automata", while the results of the second stage are contained in the present volume.

In the meantime, what was going on in the West from the viewpoint of a "man from the East"?

The progress that was quite natural for the development of a research field continued until the beginning of the last decade. Some research teams were leaving the area, some were coming in. (In this short forward I shall not pursue the task of giving a review of work carried out in the field of self-timing.) However, at the time one could sense that the results accumulated over the years, would be extremely useful for developing new IC design disciplines making an effective use of novel technological opportunities. An obvious "social order" had thus emerged that was explicitly announced in 1979 at the MIT Workshop on Self-Time Systems chaired by J. Dennis. Dozens of new researchers have put their efforts into the area during the last decade. A fairly complete survey of the state of the art was presented by C. Barney in the 1985, December 9, issue of Electronics Magazine. An important role in this process belongs to Charles Seitz of the California Institute of Technology. This role, as we see it, however, is not totally definitive.

The major contribution of C. Seitz was in several important conference reports and his Chapter 7 in the famous Mead-Conway text "Introduction to VLSI Systems". They were directed towards bringing the self-timed design discipline, at a proper level of significance, to a substantially wider audience of designers and manufacturers. It would appear from the "outside" that he became a "Father and Creator" of self-timing in the eyes of some young American researchers because they read a rather limited selection of references. Of course, there would be no problem in that if it were not that a somewhat "easy-to-design" approach displayed by Seitz created an illusion of extreme transparency of a self-timed design process. As a result, many circuits that have been presented in the literature as self-timed ones appear to be delay sensitive after checking them with software tools based on the analysis methods presented in this book.

Everyone who wishes to build good circuits should bear in mind that the design of self-timed circuits is, in principal, different from the conventional design fashions advocated for synchronous and asynchronous circuits. Design errors related to temporal behaviour cannot be detected by traditional simulation systems and their manifestation during the circuit maintenance stage is, therefore, unpredictable.

As we may see from publications, the research work on self-timing carried out in the West can be divided into two co-existing, but hardly interacting, streams. One encapsulates brilliant, highly abstract mathematical models, almost ignoring all engineering problems, while the other gives us purely technical contributions that, so far as is practicable, keep away from theory.

Wise men say that the truth never lies in the middle, since if it did, it would be easily attainable. However, in spite of this, the book serves as an attempt to find exactly such a compromise, even though it still lacks any presentation of actual design experience.

Leningrad
August 1988

Victor Varshavsky.

The clock hand kept on ticking away and ignoring the figures which it approached, touched, left behind and then again reached and again touched. It was absolutely indifferent to any goal, point or segment. Should it however having measured another sixty seconds stop at least for a moment or give us a slight wink: here, it said, something ended. But from the way how it was hurriedly stepping over a chosen point - it stepped in the same fashion over any other point, not marked with a figure - one could see that all that mass of figures and sections were totally nothing to it. They were brought from the outer world to mark the hand's path while it kept on moving on and on, without noticing any of them ...

Thomas Mann

FROM THE EDITOR

The fortunes of scientific ideas are very similar to those of human beings. Not everyone is given a chance to go through a drama of events. Most human lives sink in everyday routine. There are, however, some magic moments, or the so-called "star hour", in everyone's own life when the personality and his or her inimitable features, together with the surrounding circumstances, become most decisive. It is highly important not to miss such an instant of time. This is, in many ways, true for the ideas that underlie this book.

First attempts to overcome the discrepancy between the notion of a finite state machine as a formal object representing a sequence of atomic events and the notion of a discrete device being the *dynamic system* which unfolds such events in real physical time were made nearly three decades ago. One such attempt was made by D.E. Muller in his theory of "*speed-independent circuits*" (more precisely, *the circuits whose behaviour is independent of element delays*). For years this theory remained an "ugly duckling" in a series of multiple attempts to modernize the classical approach to the design of discrete (digital) computer and control circuits. Many researchers contributed to the theory which however did not progress rapidly. It had by no means any technological support nor any "social demand". Most of the statements of his ideas seemed to be given no proper credit. In fact, had it been necessary to explore the nature of intrasystem time and the mechanisms which "generate" such time when all the questions related to the temporal behaviour of a circuit could have been easily removed by introducing a systemwide clock? Had it also been so necessary to devise complicated theoretical notations and, moreover, to complicate real practical designs, only with the aim of getting rid of such a trivial unit as a clock pulse

generator? Generally, from the theoretical viewpoint, it had most likely been sensible, but from the practical viewpoint the need for such explorations was, indeed, rather vague. Nevertheless, in the course of time, the technological advances were becoming so obvious as to be able to make the situation more favourable for the above mentioned theory. Two main trends promoted its progress. One of them was related to the development of the micro-level, i.e. the basic underlying technology and the circuitry, and the other was concerned with the achievements at the macro-level, i.e. at the system organization level. As a consequence of these two trends, the interconnection delays became relatively high compared with the delays of the active components. What is more, the whole temporal behaviour of some of these components becomes increasingly indeterminate. At the same time the growth of complexity lays rather strict and sometimes infeasible requirements upon the centralized synchronization mechanism. An adequate assessment of such an evolution of technological influence on potential design strategies was manifested at the closing ceremony of the *MIT Workshop on Self-Timing* by Professor Jack Dennis who said: "... *I sense a general feeling that self-timed principles could have an important, if not essential, role in the design of very large scale integrated circuits*". Now VLSI designers are becoming more and more convinced that this statement is true.

In 1976 we published the book "*Aperiodic Automata*" [6], the first and, until now, the only volume dedicated entirely to theory and practice of *delay-insensitive circuit design*. The material presented in it had pointed the way to how to design and implement such circuits. Here a fundamental question arises: why then have these sort of devices still not been fabricated and marketed? The answer to this question is almost trivial: because most manufacturers have not yet experienced real "hunger" to change over to the new circuitry rationale.

Over the last few years, the situation has, in principle, changed. There are several functioning demonstrator boards based on aperiodic principles. First samples of LSIs containing aperiodic pipeline registers are now manufactured. Meanwhile the March 1984 issue of the "*Computer Design*" Magazine reports that TRW and Mostek are already on the market with their FIFO register LSIs. The Carnegie-Mellon University, together with Washington University and DEC have designed a self-timed interface prototype, called TRIMOSBUS (which, however, has proved to be not totally delay-insensitive). Most American universities followed the famous Mead and Conway text and included the sections on *self-timed logic design* in their hardware design courses. In the Soviet Union, at least two colleges began to give logic design courses involving *aperiodic circuits*. A tutorial workshop on *aperiodic automata* has also been held for industry. The *self-timed principle* is currently being incorporated into a series of future designs.

The sharp increase of in the designer's interest in *aperiodic circuitry* can also be explained by the established fact that aperiodic circuits are *totally self-checking* with respect to *stuck-at faults* in logical elements. At the intuitive level this fact had been mentioned even in the "Aperiodic Automata" book, but the strict proof had emerged later. Bearing in mind that the reliability and the self-checking issues are increasingly becoming fundamental for the further development of computer organization discipline, the establishing of such a close relationship between these two issues has significantly raised the interest of a wider audience in the subject.

This book presents a "snapshot" of the current status of work as seen in the group at the end of 1983. Of course, since then, new results have been obtained. Beginning with D.E. Muller's work, all publications on aperiodic automata or speed-independent, or self-timed, circuits have been on the assumption that the delay in an element is with reference to its output, and that the wire delays are negligibly small. In branched wires, the delays are supposed to be concentrated before the branching point. This assumption is obviously incorrect. When the integration scale changes, the role of wire delays that occur have a serious effect on the behaviour of circuits. In addition, the assumption that a stuck-at fault in a logical element is equivalent to the creation of an infinite delay at the element's output is a somewhat rough approximation to what really takes place in a VLSI circuit. From this point of view, there arises a problem of designing circuits whose behaviour is independent of *transistor* and *wire delays*, rather than just of *gate delays*, since gates are the interconnections of transistors and wires. Now, a positive solution for this problem has recently been found (for a fairly large class of circuits), but these results are not reported in this book.

Another important problem in the development of the subject is the decomposition of aperiodic circuits. This problem is closely related to the use of an intuitive and informal set of rules when a topological layout increasingly has a major effect upon a decomposition strategy. It becomes crucial when the principles of object-modular design ("layout-placement plans") oriented towards mask libraries are used. In this case, the "decomposition" is more ~~akin~~ to the interface between ready-to-use subcircuits according to a given composite specification than to the partitioning of a circuit into subcircuits. The corresponding minimization criteria for this procedure are the number of interconnections between subcircuits and their length, and the major constraint is to keep the behaviour independent of delay values with respect to a given (required) level of abstraction, e.g. component, gate or transistor and wire delays. First results in this area indicate that the aperiodic principle opens up some promising opportunities but, on the other hand, obviously requires a more active theoretical attack.

On behalf of the authors, I would like to express the hope that this book will be useful, not only for researchers, but for computer designers and digital hardware

engineers, and will promote the aperiodic circuitry in its march into industrial projects in the future.

I would also refrain from establishing a "formal mapping" between the authors and the chapters of the text. The drafts of all the chapters have been revised several times, thus we bear a collective responsibility for any mistakes and flaws that still remain.

We would like to acknowledge the referees of this book N.Ya. Matyukhin and D.A. Pospelov, and especially the Russian text editor S.V. Petrov for his great effort in making the book clear and easy to read. We also thank our colleagues whose names are not listed among the authors, but who have actively participated in obtaining the results presented in this book.

Leningrad, August 1984

V. Varshavsky

CHAPTER 1

INTRODUCTION

And so time does not exist per se, on its own.
Objects themselves create a sense of what has
gone with centuries,
What is going on now and what awaits us in the
future

Lucretius Carus

Modern control systems include an important subclass of systems which coordinate the processes with a discrete state set. Although the term “system state” is intuitively clear to the reader, in the subsequent discussion it can be, and will be, refined as may be required, depending upon the particular syntax and semantics of a system model description language. A typical example of a system with a finite number of states is the data exchange control in computer networks. This, however, does not exclude the presence of continuous, or analogue, variables. For example, the tool displacement in a metal-cutting machine can be characterized by coordinates of the cutting edge of the tool, by speeds of coordinate drives, by components of forces on the tool, etc. However, such variables as turn-on and turn-off of driving motors, the status of the reduction gear, the presence of information about the next operation modes and many other variables constitute the system's discrete state. A sequence of such states realizes the discrete control of the machining process. Similar examples can be found in various fields, for example, in control of the starting and stopping of a power plant, or in control of motions and displacements of robot arms, to mention but a few. Essential characteristics of such control systems can be summarized as follows:

1. *Compliance*, or *matching*, implies that processes (and their component subprocesses) should have explicit phases during which the process state changes (phase transition) are carried out. The control system should, in each phase, be notified about the end of the phase, which in turn activates the subsequent phase transition.
2. *Parallelism* and *concurrency* imply the possibility of parallel execution and simultaneous phase transitions in several subprocesses.
3. *Asynchrony* implies the absence of any timing bounds on phase transitions whose lengths are subject to many uncontrolled factors.

The term *transition* will further be used as an abbreviation of “*phase transition*”.

Parallelism and asynchrony are natural characteristics of practically all technical systems. They can be seen both in control systems and in controlled plants. *Variations of transition times* in elements and units of control systems and communication links are caused not only by differences in values of technological parameters but also by changes in the physical operating conditions such as temperature, pressure etc. Asynchrony requires that a system should be invariant to transition time variations except, of course, for the cases where the transition time is essential for the logic of the control process.

Compliance is a less obvious characteristic of a system's behaviour. It stems from the requirement of functional determinism in a system, which is absolutely necessary for realizing the desired operation of a controlled plant. It represents *causal semantics* of interaction between the plant and the control system. The fact that the control system is a *compliant (matched) parallel asynchronous system*, whose behaviour is defined by its interaction with the controlled plant and the *environment*, enables us to consider the pair "controlled plant - control system" as a unit without being afraid of missing some qualitative aspects of the interaction. Since the system has a finite set of states, then it is quite natural to refer to a *finite state machine* as a model for such a system.

However, the difficulty in using the classical model of a finite automaton is attributable to the fact that this model ignores the concept of real physical time. Thus, every time when the problem of physical implementation of a finite state machine is encountered, there arises the need for essential changes in the original model – a physical mechanism representing a finite state machine will always be a dynamic system with all the characteristics of the latter. As a rule, modern technology seeks for the solutions to overcome these problems through using external synchronization facilities.

There are two main types of *synchronization mechanisms*. If transition times in elements and units are bounded and their upper bounds are known, then the influence of physical time is bridled by the recognition of the system's state after the completion of all activated transitions. A new cycle of phase changes is initiated only after the transitions in the state recognition mechanisms have been accomplished. There are, however, the techniques which do not disturb the common principle of synchronization. Such synchronization seems attractive because it can easily be implemented physically. On the other hand, if there are some variations in transition times and they are not properly taken into account, it will result in an "under-utilisation" of the speed capacity of a system. In addition, if the real transition time exceeds an allowed upper bound, then incorrect operation of the whole system may follow. The latter effect, often called a *parametric fault* is one of the most common sources of failure in a system. It is well-known to computer engineers; in

case of a malfunction they first try to avoid it again by simply lowering the frequency of the clock generator.

In cases where the transition times in certain units exceed the transition times in other units of the system, or when the time variations are so high that it is not effective to use the first type of synchronization, the recognition of a new state is realized using a periodical probing of asynchronous signals by clock signals. A typical example of such a synchronization strategy is the interrupt handling system. One may imagine that increasing the frequency of probing (clock) signals and the speed of logic elements would lead to as exact a synchronization as desired. But these expectations disappear as soon as the *arbitration effect* is detected and carefully analysed. This effect is concerned with the *anomalous behaviour* of a circuit which arises every time when the clock signal is close enough to the edge of a probed asynchronous signal. In such a conflict situation the *arbiter* should decide between two alternatives: whether the clock signal has been before or after the change of the probed signal. Today, the impossibility of implementing absolutely *reliable arbiters*, *synchronizers* and *inertial delays* without using analogue circuits (say, comparators) may be regarded (see Chapter 9) as an established fact. The greater the probing frequency becomes, the higher is the probability of the occurrence of *arbitration conditions*. Moreover, with increasing speeds of logical elements the *metastability* duration becomes relatively longer which makes the anomaly even more damaging.

Thus, in spite of the widespread use of clocked systems in modern technology, problems for further increase in speed in such systems are apparent. They get even worse with the use of submicron technology where the major signal transfer delays are mainly caused by wires. The variations of delay values in wires may be hundreds of times greater than the delays in transistor junctions. Many leading researchers now agree that *the use of the external synchronization principle in the next generations of very large scale and high speed integrated circuits is very problematic.* (See, for example, [207] and [291].) In sub-micron integrated circuits the dimensions of transistor junctions are approaching several thousand atomic radii, and the interaction regions are becoming comparable to the wave length of interaction signals. One can thus speak of the so-called *equichronic regions*, i.e. the regions within which the components function in the same “clock beat”. Of more importance, then, are the problems of matching (synchronizing) the operation of objects situated in different equichronic regions of the chip, though the formulation of such problems may today be assessed as speculation. With the increase of system complexity, the problems occurring at the microlevel begin to be mimicked at the system level.

Bearing in mind that the term “clocked”, or “synchronous”, normally implies a set of concrete technical solutions, it is rational to restrict ourselves within the bounds of the questions related to the organization of the temporal behaviour of complex

systems and to the control of parallel asynchronous processes. Here, we shall be interested in various abstraction levels, beginning with transistors and wires and progressing to rather sophisticated modules, for which the model of functioning can be regarded as a process with a discrete state set. It would be suitable to consider a system at all abstraction levels from a unified methodological viewpoint.

First of all, what we imply by the term “time” has to be defined more exactly. The encyclopaedia of physics defines it in the following way: “Time is a set of relations expressing the coordination of states (phenomena), their sequence and duration”. From this definition, it follows that there is not other way to express time than through the relationships between observed events. External synchronization is a process of coordination between events in a system and those external to the system, which are expressed by the state changes of an external clock. Events taking place in an external clock do not, in principle, have any causal relation to the events in the system and in the environment interacting with the system: “After this does not imply because of this”⁽¹⁾. This fact is likely to be a source of those methodological problems which arise in the design of external synchronization systems. On the other hand, the change of states of the system in real physical time gives rise to the relationships between the events in the system and those in the environment. This set of relationships appears to be the system time for the given specific system, and the environment. The quantization of time by the events taking place in the system and the environment rather than an external clock leads us to the concept of *self-timing* or *self-synchronization*. The attractiveness of this concept, central to the book, essentially depends on the possibility of using it in engineering applications, particularly in the design of matched parallel asynchronous systems.

As mentioned above, a matched parallel asynchronous system is a collection of interacting elements (units, subprocesses). An event in such a system is a *transition*, a change of the system state. For a unified representation of the behaviour of such systems at various abstraction levels we need appropriate description tools that are able to define a set of relationships between events, thereby establishing a system time.

Attempts to formalize a system of relationships for the definition of time were made as far back as in the 4th century B.C. by Aristotle, but perhaps the first rigorous system was proposed by J.N. Findlay in 1941, in the article “Time: a treatment of some puzzles”. However, nearly twenty years elapsed before works formalizing the relationships between events in systems emerged. In 1959, D.E. Muller and W.S. Bartkey proposed a model called a *state-transition diagram*, or *Muller diagram*, for the description of asynchronous circuits (Muller circuits). In

⁽¹⁾ In principle the “clock” may be associated with the environment which is external to the system. But it is the separation of the “clock” and the environment that distinguishes a synchronous state machine from an asynchronous one in classical automata theory.

1962, C.A. Petri proposed another model, later called *Petri nets*, which allows the description of flows of discrete events independent of the duration of the occurrence of these events. Both these models have advantages and limitations. However, due to somewhat magical reasons, Petri nets won tremendous popularity and became a rapidly developing area of knowledge, while the language of Muller diagrams is far less popular than it deserves.

When first examining the operation of a system in Chapter 2, it is natural to introduce a certain metamodel with the possibility of various modelling and semantic interpretations. Such a model, in which the relationships between the system states reflect only the causal nature of their ordering, is called an *asynchronous process*. The realization of various logical dependencies is here defined both as internal semantics of the system and as environmental factors external to the system. Taking into account the external factors at this stage is as unnecessary as it is impossible. It is, however, important that the environment can also be considered as an asynchronous process. As far as the system is concerned, every other system with which it has interaction can be regarded as a component of the environment. The language of asynchronous processes allows, already at this level of abstraction, the solution of fundamental problems of process and system composition. When a pair of processes has to be matched, we implement an additional matching process that can be considered as a formal model of the interaction protocol between the system and the environment.

The first step towards the refinement of the semantics of system processes is reduced to the interpretation of an asynchronous process in terms of a lower abstraction level language. Petri nets, Muller diagrams, parallel asynchronous flow charts, and finite state machines are used here as such languages. Each of these reflects a particular behavioural characteristic that makes them all useful in practice.

For both solving the interpretation problem and that of the composition of processes, we introduce the idea of *structuring the states* of original processes. The term “structuring”, originating from the Latin “*structura*”, here implies the refinement of a state such that a state is represented as a vector with binary or integer components. It does not, however, exclude the splitting of a state into some characteristic components, for example, the input, output and internal ones. Such a refinement allows for an explicit representation of events associated with state changes in the asynchronous process.

Our ultimate goal is the construction of a composition of (controlled) subprocesses and a control system. First, we ignore the control system as such and consider only the coordination of transitions in subprocesses. In structured subprocesses, two types of variables can be distinguished: *data variables* that characterize informational, energetic or material flows, and *phase, or control, variables* that initiate and indicate process transitions. For the sake of clarity, we

restrict ourselves to considering only binary phase variables. They will be used as the starting point in the structuring of the states of a process associated with the control system. Furthermore, we assume that a transition initiated by an input phase variable terminates with the output phase variable repeating the value of the input one. From this, it follows that the subprocesses can be modelled by a delay with respect to corresponding phase variables. The value of such a delay is arbitrary – it is exposed to uncontrolled variations. The only restriction is that the delay must be finite. If they are, then a system description constructed in this way reflects the mutual coordination of phases of subprocesses independently of the execution times of these subprocesses. Hence, such a description contains sufficient information for the design of a control system but leaves the question of the design techniques open.

In turning from control process definitions to control system descriptions, first of all, we should tackle the problem of encoding the phase variables of subprocesses and the states of the entire system. Code representations, or simply codes, that meet the demands necessary for organizing the control, will be studied in Chapter 3 and are called *self-synchronizing codes*. Since binary encoding of variables has been accepted, we shall consider, as control system elements, the functional elements whose input-output functioning is expressed by Boolean functions, or finite state machine equations. We postulate that a *circuit built of functional elements, models the control system* which controls concurrent asynchronous processes that are given in one of the above mentioned languages, if

- a) the description is homomorphic to the original transition diagram of the circuit in the sense that the order of the changes of variables common to both the original and the expanded transition diagrams remains the same, and
- b) each phase signal is associated with exactly one output of a circuit element.

Since the transition diagram for a circuit is intended to describe the ordering of transitions that are allowed for any lengths of these transitions, this diagram will not change if between the output of every element of the modelling circuit and the wires connecting this output to the inputs of other elements, an arbitrary delay of a finite value is inserted. As has already been noted, a subprocess represents a delay with respect to phase signals, and thus the modelling circuit, with the inserted delays, guarantees the required coordination of phases of subprocesses that are modelled by these delays.

Our interest is then addressed to *the theory of circuits whose behaviour*, in the above sense, *does not depend on the values of element delays, also called aperiodic circuits, Muller circuits, or speed-independent circuits*. The classical theory of such circuits, founded by D.E. Muller and W.S. Bartkey, actually left unsolved the questions relating to the synthesis of circuits of elements of a limited functional basis and those relating to the interaction between such a circuit and the environment.

These limitations were overcome in the theory of aperiodic automata, which was a substantial extension of the theory of Muller circuits.

The Muller diagrams were primarily included as a tool for analysing circuits for their independence of element delays. On the other hand, a set of Boolean functions that satisfy the delay-independence, can be derived from a *semi-modular Muller diagram*. This procedure, however, does not solve the synthesis problem because it may lead to the necessity of using the elements for which – when using existing technologies – the initial prerequisites of Muller theory are not satisfied. For example, it may occur that the delay cannot be reduced to the output of an element. For certain elements there just may not exist a race-free implementation. However, as will be shown in Chapter 5, for any Muller diagram with k variables, there exists a semi-modular expanded Muller diagram with $2k$ variables, for which the original diagram is a projection of the expanded diagram on the set of original variables. The expanded diagram can then be realized by AND-OR-NOT elements.

The above approach opens up the way to *implement an arbitrary Muller diagram by a circuit that models it* in an AND-OR-NOT basis. Dually, this result is also applicable to the basis OR-AND-NOT. Note that here we have been dealing with so-called unconstrained bases. The technological requirements, of course, restrict the power of this method. Thus, there arises the question of minimal circuit bases. Such bases are: for distributive Muller diagrams – 2-input NAND elements, or dually 2-input NOR, and for semi-modular Muller diagrams – 2-input NAND plus 2-input NOR. The proofs of *functional completeness* are constructive in the sense that they supply useful synthesis methods.

The class of *safe and persistent Petri nets* (see Chapter 2) also allows a direct transformation of a specification to modelling circuits which are independent of element delays. For these purposes, in Chapter 5, we present a series of standard, or canonical, circuit fragments in which each place of Petri net is modelled by a flip-flop, and each transition is modelled by a wire. A *marking* in a net is modelled by the combination of the states of these flip-flops, and the occurrence of a transition is represented by the sequence 1-0-1 or 0-1-0 in the corresponding wire. Although the circuit obtained is redundant, it is – which is crucial for our purposes – homomorphic to the original Petri net. The circuit basis for constructing such circuits consists of AND-OR-NOT or OR-AND-NOT gates, and the miniml basis fits 2-input NAND or 2-input NOR, as in the case of implementing *distributive Muller diagrams*. This means that the language of semi-modular Muller diagrams has a higher descriptive power than that of safe persistent Petri nets. In any case, we have not been able to provide a method for non-distributive and semi-modular Muller diagrams to be expressed through safe and persistent Petri nets.

The language of *parallel asynchronous flow charts* presented in Chapter 5 describes concurrent functioning of subprocesses associated with its operators. In

fact, for the design of a speed-independent circuit, modelling a parallel asynchronous flow chart, it is sufficient to implement a set of circuits modelling fragments of a flow chart. The set of fragments proposed by J.B. Dennis encompassed: bifurcator, merger, synchronizer, conditional branch and loop. This is, however, insufficient for our implementation strategy. It is necessary to include in the above list, a special block providing the isolation of phase signals of the subprocesses represented by multiply used operators (say, all addition operations are performed by the same adder subprocess). Consequently, such a block is called a *multiple use circuit*.

Realization of a set of standard blocks is considered for *two-phase disciplines* in a control system. One of the phases may be called a *working phase*, and the other phase is used for reinitializing a block and, thus, is called an *idle phase*. Two possible organizations of control can be considered with respect to such a discipline. In the first one, all the blocks perform the working phase and, thereafter, in the same order, they proceed through the idle phase, returning to the initial states. In the second organization, every block performs both phases before the control activity is “handed” over to subsequent blocks. Neither of these organizational modes can be given preference with respect to their costs since the choice of discipline always depends on the concrete requirements of the control system.

An alternative to the distributed control circuit, which we obtain from the association of flow chart fragments with standard blocks, is an *asynchronous state machine*. It can be suitable for realizing complex fragments of an algorithm where there are no parallel sections. We assume that any transition of a state machine is initiated by the setting of the input phase signal, and after the transition termination the corresponding output phase signal changes. Furthermore, if the operation of the state machine is correct with respect to any delays of the elements of the implementation, the machine is called *aperiodic*. For any given finite state machine, an aperiodic equivalent can be constructed. For this, the following two techniques are proposed:

- a) a *canonical procedure* for synthesizing an automaton with memory using *D*-flip-flops or *T*-flip-flops and with a special circuitry for the indication of transient process completion (see Chapter 4);
- b) a *procedure for representing an aperiodic machine as a composition of Muller circuits* (see Chapter 5) that implement transition diagrams for fixed values of the input variables; Here the interaction with the environment is provided by breaking certain wires and closing them, throughout the environment.

The organization of the interaction with the environment is very similar to the interconnection of semi-modular circuits using the so-called *Muller theorem*.

In some cases, the nature of the interaction is such that there arises a need for *arbitration facilities* that must be used to resolve *conflicts*. Resolving conflicts may

cause substantial delays in the overall functioning of a system. Thus, for example, any pre-ordering in a set of sub-processes wishing to acquire a common resource, with no regard to the variations of the sub-process durations, will not succeed because it either allows arbitration, or else may introduce a long halt in a certain sub-process when, according to the pre-ordering, one of the sub-processes is given the right to acquire the resource, but no longer needs it. The presence of conflicts is equivalent to the violation of Petri net persistence or semi-modularity in Muller diagrams, and causes an anomalous functioning of corresponding circuits. Chapter 9 presents a discussion of the *arbitration phenomenon* with respect to dynamic characteristics of circuit elements. It is shown that using only logical elements, it is impossible to implement an absolutely safe arbiter that will be free of a *metastability* or an *oscillatory anomaly*. On the other hand, an obvious condition for a mutual exclusion flip-flop to leave the anomalous state is a change in the difference between the voltages on its two poles beyond a certain threshold value. Hence, the termination of the anomaly can be indicated by an analogue comparator connected to the flip-flop output poles. Speed-independent circuits can thus be implemented to control processes with conflicts. Therefore, the class of situations and processes modelled by control circuits can be substantially expanded. Here, however, there arises the serious problem of detecting such situations. Furthermore, the problem of analysis is not only connected with the task of detecting the possibility of conflicts. The point is that although the above mentioned synthesis methods provide ways to design speed-independent circuits, they, as many universal methods, are excellent for all cases, but may turn out to be highly redundant in each particular case. This is convincingly confirmed by the century-old experience of the evolution of technology. Concrete circuits would, in most cases, be devised with considerable heuristic effort. However, the circuits designed in that way require verification of their speed-independence and of their functional adherence to the original specification. Although the common indications of speed-independence are known from Muller's work, their direct verification using transition diagrams appears to be very difficult. The complexity of this task grows exponentially with the number of elements in the circuit. Analysis methods presented in Chapter 8 provide an acceptable time (with the aid of a computer) to check circuits consisting of up to hundreds of elements.

Speed-independent circuits have another very useful feature. They are totally *self-checking with respect to stuck-at faults*. A *stuck-at fault* of an element is equivalent to the incorporation of an infinitely long delay. This means that if an element liable to switch in accordance with the functional logic of a circuit cannot do so because of a stuck-at fault in it, then the whole circuit stops. This fact is important from two points of view. Firstly, speed-independent circuits are safe in case of stuck-at faults in the sense that their operation ceases. This is crucial in many applications. Secondly, it has been proved rigorously (see Chapter 10) that such

circuits are totally self-checking, and thus also self-testing, with respect to stuck-at faults of elements. The presence of a stuck-at fault can be detected through temporal redundancy – using some knowledge of the physical parameters of a circuit, we can always define a critical time during which an activation of a given signal in a circuit should occur. The absence of such an activation during the critical time interval indicates the presence of a fault in the circuit. This allows us to localize the fault to within some standard building block, without using special external testing, by means of a simple internal control circuit. The opportunity to receive a signal of a localized fault allows the organization of circuit *self-repair*. It is especially attractive for circuits with a high degree of homogeneity such as, of course, memory where a standby cell or a numeric bit-slice section can be used as a spare unit for the whole memory. As an illustrative and instructive example, we present a circuit for a binary counter that compensates one stuck-at fault at the expense of one spare bit unit. We should, however, note that our approach cannot be applied with the same effect when a fault occurs during the switching time of the corresponding element, because such a fault may result in an arbitration condition, or shift the system to a state which has not been included in the system specification. Problems associated with such faults require further research.

All the problems which we have touched on here, and discussed further in the main body of the text, have a wider application range than just in the design of control systems. The aperiodic automata themselves can be interpreted as asynchronous processes, and this opens up wide opportunities for modular synthesis of self-timed computing and control circuits of high dimensions where both the standard units of computers and those of multi-computer systems can be used as building blocks. In this case, one can generalize the concept to a hierarchy of modular descriptions.

A highly important class of devices for which the methods and techniques presented here appear to be effective are *interfaces* (see Chapter 7). The class of interfaces is rather large – from the interface between equichronic regions in VLSI circuits to the interfaces between chips and boards, as well as interfaces between computers and multi-processor systems. Typical of such systems are:

- 1) the necessity of controlling communication channels with uncontrolled delays,
- 2) skews in bundled parallel wires,
- 3) the interaction between entities whose response time is unpredictable.

In addition, as far as the organization of self-repair is concerned, the interfaces are interesting because of the homogeneity of the communication channels and corresponding receiver-transmitter devices. Examples of *self-synchronizing interfaces* that perform the correct communication independently of delays in

channels and in the elements of the control hardware, prove themselves to be a good illustration of the effectiveness of the proposed approaches.

An essential resource for increasing performance of interfaces and some data processing systems is *process pipelining*. The organization of an *asynchronous pipeline* is most effective. The design of asynchronous pipeline structures (see Chapter 11) inevitably requires the synthesis of matched parallel asynchronous systems and control circuits with speed-independent features. Corresponding modelling circuits also illustrate the application of the results obtained.

CHAPTER 2

ASYNCHRONOUS PROCESSES AND THEIR INTERPRETATION

We know that time can be expanded
It always does depend upon
What sort of content we demanded
To fill it with just by our own

S. Marshak.

At present, the designers of discrete systems, such as computers, information processing systems and data sampled control systems, are provided with a number of *models*. These allow the dynamics of the operation of the systems to be defined with adequate regard to potential parallelism and concurrency of their activity, and asynchrony of the interaction of their sub-components. *Mathematical model*, according to the Mathematical Encyclopaedia, is “an approximate description of a class of phenomena from the real world expressed in terms of mathematical symbols”. Each of these models reflects some aspects of system operation and, thus, cannot be universally used throughout all design stages and applications. Nevertheless, certain characteristics that are common to such models allow us to propose a methodologically convenient *meta-model* that may form a base for other, particular or, better say, *object models*. The term “meta-model” can be viewed as a model that is used for studying the models of another class, as is usually implied when notions are defined by adding the prefix “meta”. The mechanics of generating an object model from a metamodel is defined by the *interpretation* – “showing the meaning (semantics) of mathematical expressions (symbols, formulae, etc.) – of key concepts of the target model through those of the meta-model”. This is also a quotation from the Mathematical Encyclopaedia.

The concept of an asynchronous process, which is fundamental in this chapter, will be used as such a meta-model.

2.1 Asynchronous process

2.1.1. DEFINITION.

The term “asynchronous process” will be used throughout this book, both for specifying the behaviour of systems, and for solving problems of analysis and implementation of systems. To pick out one or other aspect of modelling, we shall,

in the following, distinguish between the terms “descriptive asynchronous process” and “asynchronous process”.

DEFINITION 2.1. A *descriptive asynchronous process DP* is a quadruple $\langle S, \mathcal{F}, \mathcal{I}, \mathcal{R} \rangle$ in which

S is a non-empty set of *situations*;

\mathcal{F} is a binary relation defined on the set $\mathcal{P}(S)$ of subsets of situations, which assigns to a subset of situations $\alpha \in \mathcal{P}(S)$, another subset of situations $\beta \in \mathcal{P}(S)$ such that $\alpha \neq \beta$;

\mathcal{I} is a set of *initiators* $\mathcal{I} \subseteq \mathcal{P}(S)$ such that

- 1) for any $i \in \mathcal{I}$ there exists an α such that $i \mathcal{F} \alpha$;
- 2) if $i \mathcal{F} \alpha$ and $\alpha \notin \mathcal{I}$ then from $\alpha \mathcal{F} \beta$ it follows that $\beta \notin \mathcal{I}$;
- 3) for any $i \in \mathcal{I}$ there is no $\alpha, \alpha \notin \mathcal{I}$ such that $\alpha \mathcal{F} i$;

\mathcal{R} is a set of *resultants* $\mathcal{R} \subseteq \mathcal{P}(S)$ such that

- 1) for any $r \in \mathcal{R}$ if $r \mathcal{F} \alpha$ then $\alpha \in \mathcal{R}$;
- 2) for any $r \in \mathcal{R}$ there exists an α such that $\alpha \mathcal{F} r$.

If $\mathcal{I} = \mathcal{R} = \emptyset$ for some asynchronous process, then such a process is called *autonomous*.

Let us comment on Definition 2.1. The words “situation” and “process” should be understood in the general sense. “Situation (from Latin *situs* – position) is a combination of conditions and circumstances which create position”. (GSE, 2nd edition, vol. 39, p.182). The concretization of the term “situation” depends on the interpretation of the concept of asynchronous process and will be given below. It should be pointed out that at any observation moment, an asynchronous process can be in one, and only one, situation.

“Process (from Latin *processus* – movement forward) is:

- 1) sequential changes in some object or phenomenon through which certain objective regularities are expressed ... ;
- 2) a set of consecutive actions directed to the achievement of a certain result ...” (GSE, 2nd edition, vol. 35, p.177).

In our case, a process describes the dynamics of the change of situations. The term “process” is specified by adding such adjectives as “*descriptive*” and “*asynchronous*”. The first specification underlines the fact that the model is primarily intended for compact description of a modelled system – as opposed to the other model defined below. The other model called “asynchronous process” (it does not have the attribute “descriptive”) is mainly oriented towards analysing the system properties and solving problems of its hardware implementation. The specification “asynchronous” shows that the category of time is not formally involved in the definition. The linkage with time is introduced by the relation \mathcal{F} : if $\alpha \mathcal{F} \beta$ then the transition time is not specified. It may be arbitrary but finite.

The relation \mathcal{F} introduced here is a sequence relation and may be understood as a “logical necessity” (this term is taken from modal logic). In other words, the notation $\alpha \mathcal{F} \beta$ means that α is a set of causes and β is a set of possible effects. Relation \mathcal{F} defined on the set of subsets of situations emphasizes the non-determinism inherent in a (descriptive) asynchronous process. Such an “approximativity” of a process specification is, first of all, concerned with the absence of exact knowledge about the fact: in which of the situations of set α the process is, and to which of the situations of set β should it move.

The *initiators*, which activate a process, constitute a subset of situations. The purpose of the initiators must be worked out from the process semantics. The set of restrictions on the choice of the initiators introduced in Definition 2.1 demands only that an initiator cannot be just a consequence of a subset of situations; it must always be a cause for some of them.

Resultants are subsets of final situations. Their choice is also made on the basis of the process semantics. The restrictions in Definition 2.1 demand that a resultant cannot be just a cause but must be a consequence of some of the causes. Through defining resultants, we have emphasized that if a situation from set S is declared as a resultant, then any situation that “follows it” must also be a resultant.

EXAMPLE 2.1. Let an asynchronous process be given in the following form

$$\begin{aligned} S &= \{s_1, s_2, s_3, s_4, s_5\}; & \{s_1, s_2\} &\mathcal{F} \{s_3, s_4, s_5\}; \\ \{s_1, s_2, s_3, s_4\} &\mathcal{F} \{s_5\}; & I &= \{i_1\}, i_1 = \{s_1, s_2\}; \\ R &= \{r_1\}, & r_1 &= \{s_5\}. \end{aligned}$$

In this example, according to Definition 2.1, the relation \mathcal{F} is defined on the set of subsets of situations. Since the process at the moment of observation must be in one of the situations, a question arises about establishing the relation between the situations, i.e. about specifying \mathcal{F} in a more exact form. Such a concretization may be achieved by introducing M defined on S . The record s_jMs_k is interpreted as “logical possibility” of the transition from s_j to s_k . The record s_jMs_p is then interpreted as a “logical impossibility” of the transition from s_j to s_p , i.e. s_j is not the cause of s_p . For some s_j, s_k, s_t it is possible that both s_jMs_k and s_jMs_t are true, and hence in this sense, relation M reflects a certain character of non-determinism in a process. If for some s_j there exists a single s_k such that s_jMs_k then the logical possibility coincides with the logical necessity of the transition from s_j to s_k .

If a process is defined by relation M then it is natural to define the sets of initiators I and resultants R as subsets of S ($I \subset S, R \subset S$).

The relation M may be represented as a directed graph. Its vertices are associated with situations. An arc connects s_j with s_k if s_jMs_k . The situations s_l and s_m for which $s_l - Ms_m$ and $s_m - Ms_l$ are not connected by arcs.

EXAMPLE 2.1. (continued). Two possible concretizations of the descriptive asynchronous process of Example 2.1 are shown in Fig. 2.1. Fig. 2.1(a) corresponds to the relation M_1 : $s_1M_1s_3, s_1M_1s_4, s_1M_1s_5, s_3M_1s_5, s_4M_1s_5$, and Fig. 2.1(b) corresponds to the relation M_2 : $s_1M_2s_3, s_1M_2s_5, s_2M_2s_4, s_2M_2s_5, s_3M_2s_5, s_4M_2s_5$.

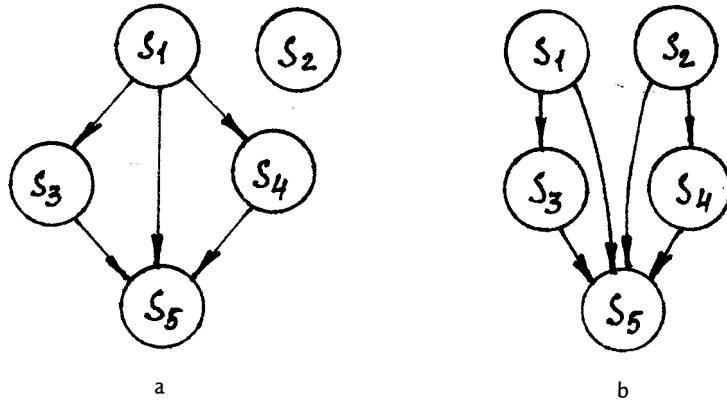


Figure 2.1 (a) and (b). Two asynchronous processes represented by oriented graphs.

Note that if $\alpha \not\propto \beta$ then for a certain pair $s_j, s_k \in \alpha \cup \beta$ such that s_jMs_k one of the following cases may hold

- 1) $s_j, s_k \in \alpha$,
- 2) $s_j \in \alpha, s_k \in \beta$,
- 3) $s_j, s_k \in \beta$.

The relation M is transitive but not reflexive because $\alpha \not\propto \alpha$ was not assumed in Definition 2.1. As a matter of fact, M is not symmetric, either. Thus, for Fig. 2.1(a) $s_1M_1s_5$, but $s_5 \neg M_1s_1$. We note that s_jMs_k does not necessarily imply that the process immediately reaches s_k from situation s_j .

Let, for example, $s_jMs_k, s_jMs_l, s_lMs_k$. Then the relation M for the given fragment can be represented as shown in Fig. 2.2(a). The study of the semantics of the process may show that the process reaches situation s_k from situation s_j only through s_l , although s_j is the cause for both s_k and s_l . Therefore, there may arise some interest in further refinement of the process definition given on a fixed set of

situations. Such a refinement can be achieved by introducing the relation of immediate sequence of situations, which will be denoted by F . Using the above restriction for the fragment in Fig. 2.2(a), the relation F is defined as follows: s_jFs_l , s_lFs_k . The corresponding graph is shown in Fig. 2.2(b). If we also assume that s_j may be immediately followed by s_k then the graph for relation F will be identical to that of relation M (Fig. 2.2(a)).

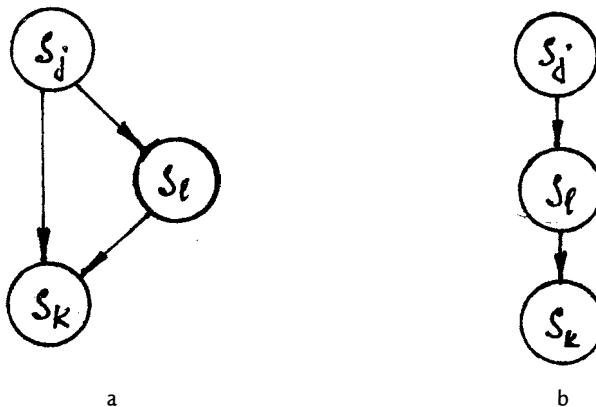


Figure 2.2 (a) and (b). Two fragments of asynchronous processes.

Note that M can be expressed as a closure of F . The denotation $s_iF^n s_k$ means that there are $n - 1$ intermediate situations s_a, s_b, \dots, s_w for which $s_iFs_a, s_aFs_b, \dots, s_wFs_k$ hold, i.e. there exists a path of length n including arcs from s_i to s_k .

Thus, from the definition of a descriptive asynchronous process, we can now turn to another definition that will be used in most cases, if Definition 2.1 is not especially indicated.

DEFINITION 2.2. An *asynchronous process AP* is a quadruple $\langle S, F, I, R \rangle$ in which

S is a non-empty set of *situations*;

F is a *relation of immediate sequence* (may also be termed “*the immediately followed by relation*”) defined on S , $F \subseteq S \times S$;

I is a set of *initiators*, $I \subseteq S$, i.e. such situations for which if $iFs_k, i \in I$, $s_k \notin I$ then s_kFs_l implies that $s_l \notin I$;

R is a set of *resultants*, $R \subseteq S$, i.e. such situations for which if $rFs, r \in R$ then $s \in R$.

(In contrast to Definition 2.1, other letters are used here for the designation of the relation and subsets of initiators and resultants, in order to avoid confusion.)

The comments related to Definition 2.1 may, with slight modifications, be applied to Definition 2.2.

2.1.2. SOME SUBCLASSES.

Let there be given a sequence (possibly infinite) of situations $s(1) \dots s(i)s(i+1) \dots$ where $s(i)$ is the element in i -th place of the sequence. A set of situation sequences can be formed for an AP, in which for any i we have $s(i)Fs(i+1)$ and in which no sequence is a subsequence of another one. All such sequences are called acceptable. Every acceptable sequence of situations describes a possible path of the changes of process situations (a *trace* of AP) and corresponds to a particular realization of the AP.

It is clear from the above, that it is always possible to build the relation M corresponding to the relation F . Indeed, if for the situations s_i, s_j there exists a trace from s_i to s_j then s_iMs_j .

Let an AP be given, such that

- (1) for any $s \in SR$ there exists an $r \in R$, such that sMr ;
- (2) for any $s \in SV$ there exists an $i \in I$, such that iMs ;
- (3) there exist no situations s_i and s_j , such that

$$s_i \notin R, s_j \notin R, s_iMs_j \text{ and } s_jMs_i.$$

DEFINITION 2.3. An asynchronous process satisfying the conditions (1) – (3) is called *effective*.

Thus, all traces in an effective process lead from initiators to resultants (conditions (1) and (3)) and each trace leading to a resultant must go out from one of the initiators (conditions (1) and (2)). An effective AP may still be non-determinate as it is possible to reach different resultants from the same initiator. It does not, however, contain directed loops formed by the situations which are not in R .

EXAMPLE 2.2. Consider the AP $P = \langle S, F, I, R \rangle$ where $S = \{s_1, \dots, s_6\}$, for which the relation F is shown in Fig. 2.3. If for this AP $I = \{s_1, s_4\}$, and $R = \{s_5\}$, then AP is not effective because the condition (1) does not hold. If we assume $I = \{s_1\}$, and $R = \{s_2, s_3, s_5, s_6\}$, then (2) is not satisfied. Then, if we take $I = \{s_1, s_4\}$ and $R = \{s_5, s_6\}$ then (3) will not hold. Note that if we assume $I = \{s_1, s_4\}$ and $R = \{s_3, s_5, s_6\}$ then the restriction of Definition 2.2 is no longer satisfied, because, in spite of s_3Fs_2 , we have $s_3 \in R$ while $s_2 \notin R$.

The following definition yields an effective AP:

$$I = \{s_1, s_4\}, R = \{s_2, s_3, s_5, s_6\}.$$

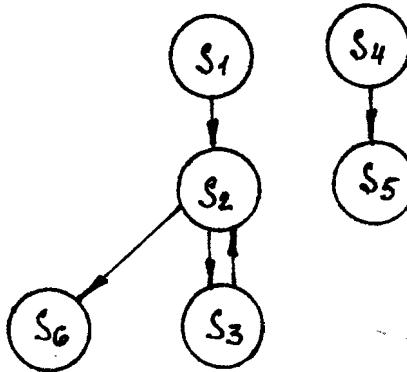


Figure 2.3. An asynchronous process.

For certain subsets of the situation set S a relation E can be defined in the following way:

- (1) for $s_i, s_j \in S$ $s_i Es_j$, if $s_i Ms_j$ and $s_j Ms_i$;
- (2) for any $s \in S$ $s Es$.

The condition (1) provides the symmetry and transitivity of E . The latter follows from the transitivity of M . The condition (2) provides the reflexivity of E . Hence, E is the *equivalence relation*.

The relation E allows us to construct a factor-set of the set S , i.e. to partition the set S into equivalence classes within a partition $\pi = (S_1, S_2, \dots, S_p)$. For equivalence classes, we define the relation F of immediate sequence ("immediately followed relation"). All the acceptable sequences of equivalence classes defined by relation F are finite (as opposed to those for F , defined on the set of situations, which may be infinite). In acceptable sequences of classes, we can pick out some initial and final elements. Corresponding classes will be called *initial* and *final equivalence classes*.

On the basis of the concepts introduced, we can formulate the following statements.

1. If some situation s is an initiator and $s \in S(j)$, where $S(j)$ is an equivalence class in the j -th place of some acceptable sequence, then all situations of the classes $S(1), \dots, S(j)$ in this sequence are initiators.

2. If some situation s is a resultant and $s \in S(j)$, then all situations of the classes $S(j), S(j + 1), \dots, S(q)$ where $S(q)$ is the final class, are resultants.
3. In an effective AP any initial class consists only of initiators and any final class consists only of resultants.
4. In an effective AP any equivalence class of situations that are not resultants consists of one situation.

DEFINITION 2.4. If, in an effective asynchronous process, every acceptable sequence of classes leads from an initial class to one, and one only, final class, then such a process is called a *controlled process*.

Thus, in a controlled AP the degree of non-determinism is substantially restricted. Any trace from a given initiator leads to a unique final class.

EXAMPLE 2.3. Consider an AP with a set of situations $S = \{s_1, s_2, \dots, s_{10}\}$ and relation F given in Fig. 2.4(a). The corresponding relation F for equivalence classes is shown in Fig. 2.4(b).

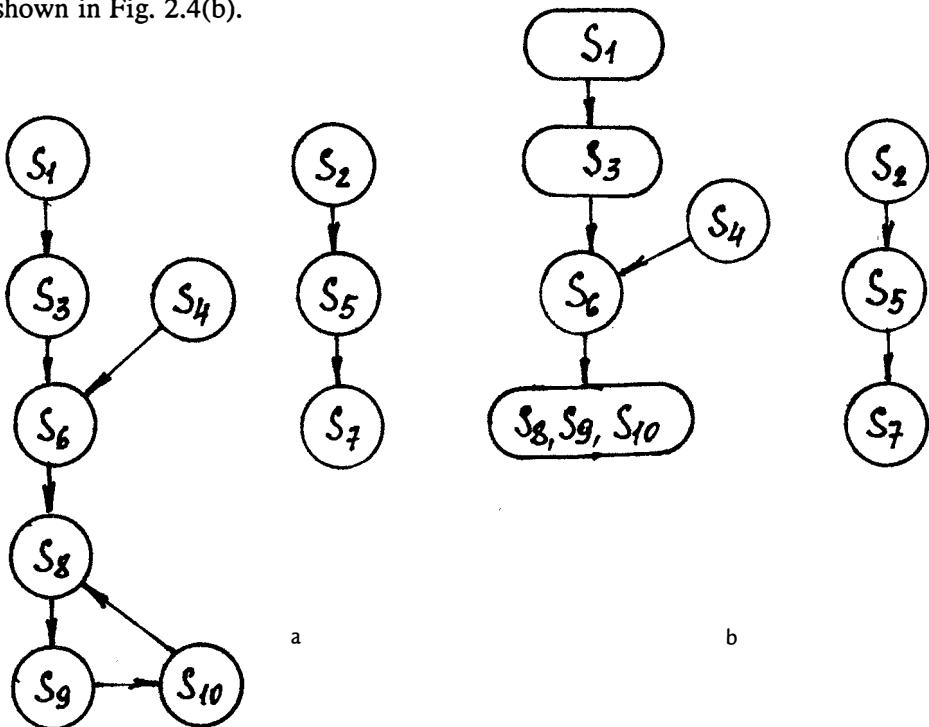


Figure 2.4 (a) and (b). An asynchronous process (a), and the result of partitioning into equivalence classes (b).

If $I = \{s_1, s_2, s_3, s_4\}$ and $R = \{s_7, s_8, s_9, s_{10}\}$, then it is easy to see that this process is controlled. Extending F with pair s_2Fs_4 makes the given AP uncontrolled but still effective.

Below, we introduce a concept that will be useful when considering fragments of APs.

DEFINITION 2.5. If situations s_i and s_k of an asynchronous process are interconnected by the relation $s_iMs_k (s_iF^n s_k)$, then a fragment of the process containing all traces leading from s_i to s_k is called the *transition* $s_i - s_k$.

In the sequel, we shall also use the following class of APs. Let, in an effective AP:

- (1) for any $i \in I$ and $s \in S$ from iFs it follows that $s \notin I$;
- (2) for any $s \in S$ and $r \in R$ from sFr it follows that $s \notin R$.

In other words, it is impossible to reach another initiator (resultant) from an initiator (resultant), i.e. every trace includes exactly one initiator and one resultant.

DEFINITION 2.6. An asynchronous process satisfying the above condition is called *simple*.

An example of a simple AP with $I = \{s_1, s_2\}$ and $R = \{s_5\}$ is shown in Fig. 2.1(b), whereas Fig. 2.3 presents an example of a non-simple AP.

DEFINITION 2.7. *The protocol of a simple asynchronous process* is defined by the relation $Q \subseteq I \times R$.

The protocol of a simple AP can be considered as a simple asynchronous process in which for every pair (i, r) , $i \in I$, $r \in R$, iQr , i.e. each initiator is immediately followed by a resultant. Hence, the set of situations in a protocol of a simple AP consists of only initiators and resultants. In the transformation of a simple AP to its protocol the set of traces that lead from i to r through intermediate situations is replaced by one arc for every pair $(i, r) \in M$. A protocol is a convenient form of external description of an asynchronous process.

2.1.3. REPOSITION.

Within the framework of Definitions 2.1 to 2.4 we have not defined a mechanism of the transition from resultants to initiators. The description of such a mechanism is, however, necessary for achieving the effect of re-initialization for an AP. Such a

mechanism will be defined through a concept of the reposition of an asynchronous process.

DEFINITION 2.8. *The reposition of an asynchronous process $P = \langle S, F, I, R \rangle$ is an asynchronous process $P' = \langle S', F', I', R' \rangle$ such that $S' = I \cup R \cup S^*$, $I' \subseteq R$, $R' \subseteq I$.*

The situations of set S' may contain only those from the original AP which are initiators and resultants, and in addition some situations from set S^* such that $S^* \cap S = \emptyset$; thus, the relation F' defines the traces of the transitions from the elements of $I' \subseteq R$ to the elements of $R' \subseteq I$, perhaps through a number of additional situations in S^* .

Furthermore, if $I' = R$ and $R' = I$, then the reposition is called *complete*. If $F' = \emptyset$, then no reposition exists. In the remaining cases the reposition is *partial*.

The union of an AP and its complete reposition forms an autonomous process. If a process P has a complete reposition, then such an autonomous process is defined by $P^a = \langle S^a, F^a \rangle$, $S^a = S \cup S^*$, $F^a = F \cup F'$.

To formulate the concept of a pipeline process, we should introduce some classes of AP repositions.

DEFINITION 2.9. The complete reposition of a simple and non-simple AP are called *trivial* and *non-trivial*, respectively.

As can be seen from Definitions 2.9 and 2.6, a trivial reposition differs from a non-trivial one in that the latter may have, as its initiator, the resultants of an original AP, which are not necessarily contained in a final equivalence class.

Let R^f denote a union of final equivalence classes of situations of a certain AP, and let S^t be a set of situations belonging to the trace t .

Let a controlled AP $P = \langle S, F, I, R \rangle$ be given, such that

- (1) every final equivalence class of P consists of one resultant r_j ($r_j \in R^f$);
- (2) process P together with its non-trivial reposition forms an autonomous process $P^a = \langle S^a, F^a \rangle$, on the set of situations S^a of which a function $g : S^a \rightarrow R^f$ is defined, then, in the corresponding autonomous process, it follows from $i \in S^t$ that there exists $s \in S^t$ such that $iM^a s$ and $g(s) = r$.

DEFINITION 2.10. An autonomous asynchronous process is called a *pipeline AP* if it satisfies the above conditions (1) and (2) and has the fragments of traces of the form ... $i_1 \dots i_2 \dots s_1 \dots s_2 \dots$ where $g(s_1) = r_1$ and $g(s_2) = r_2$.

The introduction of the concept of a pipeline AP allows us to study the pipeline principle of information processing which is based on the idea of an AP reinitiation before the AP has reached a resultant from a final equivalence class. To illustrate the pipeline principle, we use the following analogy. Let us imagine a chute with balls that can roll along. When the first ball has started, another one can be put on the higher end of the chute, then a third one etc. While rolling, a ball can reach the preceding one, or roll some distance behind, but the preceding ball cannot be overrun. When the preceding ball has been reached, the following ball cannot roll faster than it. In what follows, we shall give various examples of pipeline processes.

2.1.4. STRUCTURED SITUATIONS

For the description of processes, and for the definition of their interactions, it is often necessary to structure situations. This *structuring* may be done in various ways, depending on the problem to be solved. One of these ways is the partitioning of situations into components (events); to every component we assign a predicate P_i which assumes the value 1, if the corresponding logical condition is true, and is equal to 0 in the opposite case. A situation is represented in such structuring by a binary vector whose dimension is equal to the number of semantically defined components. The number of “1”’s in the vector corresponds to the number of true predicates.

We emphasize that attempts to use AP’s in various applications should be based on a semantic interpretation and structuring of situations. Consider the following example.

EXAMPLE 2.4. There is a horizontal band transporter (conveyer belt) (Fig. 2.5) carrying parts of two types – heavy and light. The transporter is activated by a starter. There are scales in zone A of the transporter. The scales give an indication of a heavy part but are insensitive to a light part. Light parts are transported, without machining, to the end of the belt, where they fall into hopper C. When a heavy part appears in zone A, the transporter stops and an arm of a manipulator, which is in its initial position, grips the part. Then the drive of the manipulator carries the heavy part to zone B, and, at the same time the transporter band is restarted. In zone B, after a signal from a corresponding indicator, the grip is released and the heavy part falls on to the table. Then the drive of the manipulator returns it to its initial state. A new heavy part is then selected and the process is repeated.

We can separate nine components (from the point of view of the process control organization some of them may be redundant), to which the following values of predicates correspond:

- $P_1 = 1$ – the transporter band is moving,
 $P_2 = 1$ – a heavy object in zone A,
 $P_3 = 1$ – the transporter band is stopped,
 $P_4 = 1$ – the manipulator arm is in the initial position,
 $P_5 = 1$ – the manipulator arm is holding a heavy object,
 $P_6 = 1$ – the manipulator drive is working towards zone B,
 $P_7 = 1$ – the manipulator arm is in zone B,
 $P_8 = 1$ – the grip on the heavy object is released,
 $P_9 = 1$ – the manipulator drive works towards the initial position.

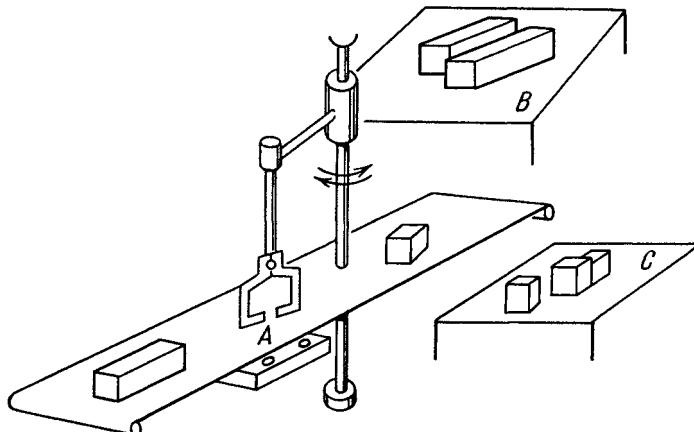


Figure 2.5. A tape transporter as an object for modelling in an asynchronous process language.

Here the positions are represented by a binary vector (P_1, P_2, \dots, P_9). From 2^9 possible values only seven are the situations of the process. They are (components which are not mentioned are equal to 0):

- | | |
|-----------------------------------|-----------------------------|
| $s_1: P_1 = P_4 = P_8 = 1,$ | $s_5: P_3 = P_6 = P_7 = 1,$ |
| $s_2: P_1 = P_2 = P_4 = P_8 = 1,$ | $s_6: P_1 = P_7 = P_8 = 1,$ |
| $s_3: P_2 = P_3 = P_4 = P_5 = 1,$ | $s_7: P_1 = P_8 = P_9 = 1.$ |
| $s_4: P_3 = P_5 = P_6 = 1.$ | |

The description of the process of selecting, gripping and carrying a heavy part from zone A to zone B involves situations $s_1 - s_6$. Relation F defines the sequence $s_1 \dots s_6$. It is natural to have s_1 as an initiator, hence $I = \{s_1\}$, while s_6 is a resultant, and thus $R = \{s_6\}$. The sequence $s_6s_7s_1$ is generated by the reposition of the above process with $I' = \{s_6\}$, $R' = \{s_1\}$, $S^* = \{s_7\}$.

In some cases the structuring of situations implies the picking out of input or output components. These components are associated with those events in a modelled system which take place at the system's inputs and outputs. When specifying both input and output components, a situations s_i is expressed by an ordered triple $s_i = (x_j, y_k, z_l)$ where x_j is the value of the input component, $x_j \in X$, $1 \leq j \leq p$, y_k is the value of the output component, $y_k \in Y$, $1 \leq k \leq q$, and z_l is the value of a component which is neither an input nor an output component, $z_l \in Z$, $1 \leq l \leq n$.

2.1.5 AN ASYNCHRONOUS PROCESS AS A METAMODEL

An asynchronous process is essentially a general model of the operational semantics of concurrently functioning asynchronous systems. This model defines acceptable control sequences for some objectives of the systems. Each of such sequences is associated with a trace in an AP. In this sense, it is possible to consider the AP as a model of the control structure of the system. An asynchronous process may also be a tool for studying rather general characteristics and regularities of parallel systems that are abstract enough to suppress some particular forms of semantics. Therefore, in such circumstances, an AP can be considered to be a non-interpreted model. On the other hand, an AP can be understood as a metamodel from which widely used dynamic models can be derived, including those discussed below. This derivation is supposed to be done with the use of an interpretive mechanism that may consist of two stages. The first stage – we call it *model interpretation* – deals with the idea that the key concepts of an AP, i.e. the situations, the followed by relation, the initiators and resultants, are associated with those of particular, or target, models. This association system is based on the introduction of some additional constraints on the AP.

The model interpretations can themselves remain non-interpreted models in the sense that they still do not express the semantics of conditions and events in a modelled system. However, if an AP or its model interpretations are further assigned with some labelling that will denote certain signal symbols, operators, attributes or predicates from a given set using semantic considerations, then, such models will become interpreted. This is the second stage, the *semantic interpretation* of an AP; it should comply with the actual application and depends on the specific character of

problems to be solved. A particular case of semantic interpretation, with the components of structured situations being assigned to actual values of predicates, was considered in Example 2.4.

2.2 Petri nets

2.2.1 MODEL DESCRIPTION

In this section, we consider a model proposed by C.A. Petri for specifying the flows of *events*. The event concept can, in this case, imply that an event is represented as a change of the value of a certain component of a situation of an AP. The relationship between events is expressed by a number of *conditions* each of which may have two values, viz. “The condition is true”, and “The condition is false”. According to Petri, an event may occur if all the conditions, on which its occurrence depends, are true. Furthermore, if the event has occurred, then the values of some conditions are exchanged. In this model, the relationship between conditions and events is represented by means of a bipartite graph called a Petri net, which has two types of vertices, *circles* and *bars*. Circles (or *places*) correspond to conditions, and bars (or *transitions*) correspond to events. Circles and bars are connected by *arcs*. Arcs from circles can be directed only to bars, and from bars only to circles. By an arc going from a circle to a bar, we specify the relation between the occurrence condition for an event (the input condition) and the event itself. If an arc goes from a bar to a circle, then it shows which condition (called the output condition) will be true as a result of the occurrence of the event. A condition may, simultaneously, be both an input and an output condition for some event. The fact that a condition is true is expressed by the presence of *markers* (called *tokens*) in the corresponding circle. In this case, it is said that the condition is marked.

A Petri net can be formally defined as follows.

DEFINITION 2.11. *Petri net* is a quintuple $N = \langle P, T, M_0, H, F \rangle$ where

$P = \{p_1, p_2, \dots, p_n\}$ is a finite non-empty set of *conditions*,

$T = \{t_1, t_2, \dots, t_n\}$ is a finite non-empty set of *events*,

$F : P \times T \rightarrow \{0, 1\}$ and

$H : T \times P \rightarrow \{0, 1\}$ are *incidence functions*;

$M_0 : P \rightarrow \{0, 1, 2, \dots\}$ is the *initial marking* function.

A *marking* of a Petri net M is a function $M : P \rightarrow \{0, 1, 2, \dots\}$.

A circle corresponding to condition p_i in a Petri net with marking M contains $M(p_i)$ tokens.

We define

$$\begin{aligned} I(p) &= \{t \in T \mid H(t, p) = 1\}, \\ O(p) &= \{t \in T \mid F(p, t) = 1\}, \\ I(t) &= \{p \in P \mid F(p, t) = 1\}, \\ O(t) &= \{p \in P \mid H(t, p) = 1\}. \end{aligned}$$

DEFINITION 2.12. An event t_i is called *enabled* by a marking M , if $M(p_j) \geq 1$ for each $p_j \in I(t_i)$. The set of events $T' \subseteq T$ is *jointly enabled* by a marking M , if $M(p_j) \geq 1$ and $M(p_j) - \sum_{t_i \in T'} (F(p_j, t_i) - H(t_i, p_j)) \geq 0$ for each $p_j \in \bigcup_{t_i \in T'} I(t_i)$.

The firing (occurrence) of an event t_i is a change of the marking M , by which it is enabled, to the marking M' according to the following rule:

$$M'(p_j) = M(p_j) - F(p_j, t_i) + H(t_i, p_j), \quad p_j \in P.$$

We say that the marking M' follows the marking M and denote this by $M \xrightarrow{t_i} M'$. The firing of a set of events T' is a change of the marking M , by which T' is jointly enabled, to the marking M' according to the following rule:

$$M'(p_j) = M(p_j) - \sum_{t_i \in T'} F(p_j, t_i) + \sum_{t_i \in T'} H(t_i, p_j), \quad p_j \in P.$$

We say that the marking M' follows the marking M with respect to the event set T' and denote this by $M \xrightarrow{T'} M'$.

The execution of a Petri net consists of transitions from one marking to another through the firing of enabled events. Let us make some comments relating to the above definition.

If the input conditions of an event are true (all input places of an event contain tokens), then the event is *enabled*. Otherwise it is *disabled*. The firing of an enabled event is defined in such a way that from each of the places corresponding to the event's input conditions one token is removed, and one token is added to each place corresponding to the event's output conditions. A disabled event cannot fire. The sequence of markings in the net, thus reflects a certain kind of process dynamics as the flow of firing events “moves” in the direction given by the arcs.

EXAMPLE 2.5. An example of a Petri net is shown in Fig. 2.6. The given initial marking makes events a and b be enabled, but after the firing of one of them, say b , the other one becomes disabled.

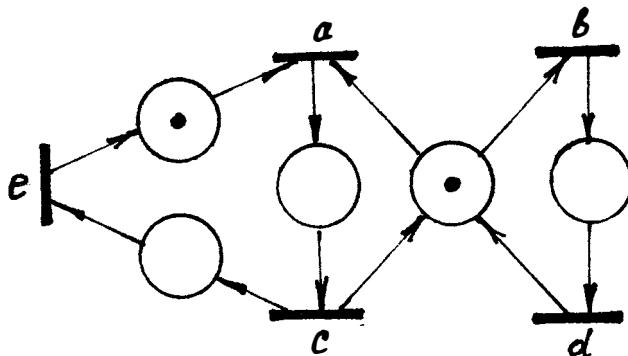


Figure 2.6. A Petri net.

The rules of Petri net execution do not, however, define the time during which an enabled event fires. This language thus responds to the requirements introduced earlier: It allows the real asynchrony and unpredictability of event firing times to be expressed.

DEFINITION 2.13. A marking, for which none of the events of a Petri net n is enabled, is called a *deadlock*.

DEFINITION 2.14. A marking M^* is called *reachable* in a Petri net from marking M , if there exists a sequence of markings from M to M^* that is executed by a series of event firings.

Let $R(N)$ denote the set of all markings in a Petri net N that are reachable from the initial marking M_0 (including M_0).

An event t of a Petri net N is called *live*, if from a marking M in the set $R(N)$, a marking M' can be reached by which the event t is enabled. A Petri net is called *live*, if each of its events is live.

2.2.2. SOME CLASSES

The following definitions allow us to introduce some classes of Petri nets.

DEFINITION 2.15. A marking M of a Petri net N is called a *conflict marking*, if there exist sets T_1 and T_2 , $T_1 \subseteq T$ and $T_2 \subseteq T$, $T_1 \neq T_2$, such that they are jointly enabled by M and for which T_2 is not a set of jointly enabled events by M' ; $M \xrightarrow{T_1} M'$.

The Petri net which does not contain conflict markings in $R(N)$ is called *persistent*.

Hence, a Petri net is persistent for a given initial marking, if each of its enabled events can become disabled only as a result of its firing (not as a result of firing of another event).

DEFINITION 2.16. A Petri net is called *safe*, iff $R(N)$ contains markings such that for all $p_i \in P$ $M(p_i) \leq 1$.

It is obvious that the safeness of a Petri net pertains to the simplest form of the interpretation of conditions in the net (“true” and “false”).

2.2.3. INTERPRETATION

A Petri net is a model interpretation of an AP. The initial marking M_0 and all markings that are reachable from M_0 , i.e. $M \in R(N)$, are *situations of the net*. The relation F defines, for any possible marking M , all the markings which may immediately follow M . It is obvious that it is possible, as in Section 2.1.2, to define on the set $R(N)$, the equivalence relation E , and thereby obtain the followed by relation F for equivalence classes (classes of equivalent markings). If, for some net N , there exists only one equivalence class, then the corresponding asynchronous process is autonomous. If there are several (more than one) classes of markings, then it is possible to distinguish the following subsets of markings, viz. *initiator markings* (always containing the initial equivalence class) and *resultant markings* (always containing at least one final class).

EXAMPLE 2.6. Consider an example of a Petri net shown in Fig. 2.7.

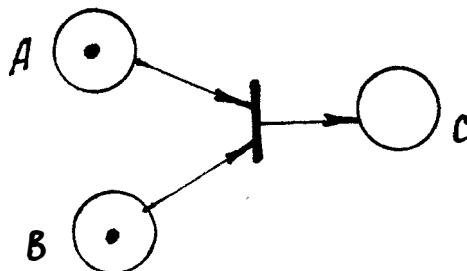


Figure 2.7. Another example of a Petri net.

If we denote a marking of the net by an enumeration of conditions containing tokens, then the process P corresponding to the net can be defined as follows:

$$P : S = \{AB, C\}, F = \{(AB, C)\},$$

$$I = \{AB\}, R = \{C\}.$$

Comments.

1. A Petri net with a finite number of conditions $|P|$ and events $|T|$ may have an infinite set of markings, as for some p_j , $M(p_j)$ may become unbounded. Such unbounded nets will not be considered here.
2. The functioning of real objects cannot often be described by a single Petri net, but rather by some set of nets $\{N_i\}$, such that all the nets in $\{N_i\}$ have the same graph structure and differ from each other only in their initial markings.

EXAMPLE 2.7. The Petri net of Fig. 2.8(a) defines the linear ordering of three events. This net is safe with respect to those markings in which the total number of tokens in conditions does not exceed 1. This fact implies that the net models the process of solving one task. The solving of the next task can be initiated by the process of trivial reposition (through event r). The union of the process with its reposition yields a live Petri net.

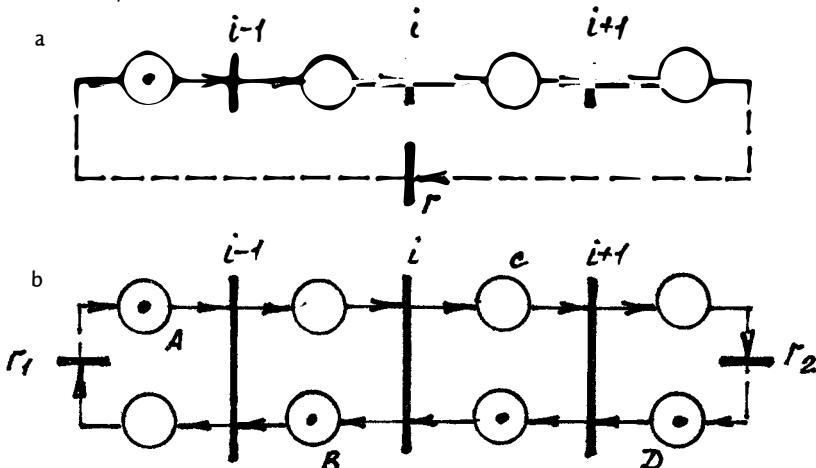


Figure 2.8 (a) and (b). Petri nets modelling a totally sequential process (a), and its pipeline analogue (b).

In the corresponding *pipelined net* (Fig. 2.8(b)), i.e. the net which realizes a pipeline process according to Definition 2.10, the i -th event is enabled if the $(i - 1)$ -th and the $(i + 1)$ -th events have occurred. This condition implies that the output conditions of the i -th event do not contain tokens that guarantee the safeness of the net when the i -th event occurs.

Note that any of the acceptable markings that contain a token in the upper left condition may be considered as an initial marking. The process of non-trivial reposition for this net is executed through events r_1 and r_2 . The overall autonomous

net provides that all the acceptable markings can be reached from each other during the net execution.

In the net of Fig. 2.8(b), at most two events can fire concurrently (excluding r_1 and r_2). If we associate some sort of assignment of a corresponding unit to a particular operation with the presence of a token in the left input condition, and also associate the firing of the event with the execution of the operation in the unit, then we deduce that at most three commitments to operation can be accommodated within such a net, with at most two of them being at the execution stage.

In the general case, a linear pipeline net with n events may concurrently enable at most $\lceil n/2 \rceil$ events, where $\lceil a \rceil$ denotes the integer which is upper nearest a .

2.3 Signal graphs

We continue the classification of Petri nets (Section 2.2.2) and introduce the following definition.

DEFINITION 2.17. A *marked graph* is a Petri net in which each condition has exactly one input event and one output event.

Traditionally, a marked graph is depicted as an ordinary directed graph with arcs marked with tokens. The correspondence between fragments of a Petri net and a marked graph is shown in Fig. 2.9. The presence of at least one token at each of the incoming arcs of a vertex means that the vertex is enabled (sometimes called excited). After an unpredictable, but finite, time interval, an enabled vertex fires. The firing consists in removing one token from each of the incoming arcs of the vertex and adding one token to each of the outgoing arcs of the vertex.

A vertex that has more than one incoming arc, e.g. fragments 4 and 6 in Fig. 2.9 is called a *synchronizer*. A vertex that has more than one outgoing arc, e.g. fragments 5 and 6, is called a *bifurcator*.

Thus, the dynamics of the functioning of a marked graph is described by the change of its markings that are generated by the above firing rule, and a given initial marking.

A marked graph is called *live*, if each of its vertices is either enabled, or may be enabled, in some firing sequence. The notion of a safe marked graph, which is analogous to Definition 2.16, is given below.

DEFINITION 2.18. A marked graph is called *safe* if 1) its marking is live, 2) none of its arcs contains more than one token, 3) there exists no firing sequence leading to two or more tokens in any one of the arcs.

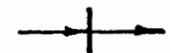
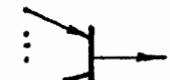
No.	Petri Net Fragment	Marked Graph Fragment
1		
2		
3		
4		
5		
6		
7		—
8		—
9		—

Figure 2.9. Correspondence between Petri nets and marked (monochromatic) graphs. If a Petri net contains some fragment of the type 7 - 9, it cannot be represented as a marked graph.

It is clear that a marked graph, being a subclass of Petri nets, is also a model interpretation of an AP. A situation in a graph is its current marking. The relation F

is defined by the rule of marking change which is the same as the firing rule. Initiators and resultants are associated according to Definition 2.1.

The descriptive power of marked graphs is lower than that of Petri nets. Marked graphs are dual to the so-called *finite state machine nets* in which each event (transition) has exactly one input condition and one output condition. The latter can model conflicts (in the sense of Definition 2.15) because they may have conditions with several outputs, but they cannot represent concurrent processes because they lack the capability to synchronize (bifurcate output) conditions for events. By contrast, marked graphs are powerful at specifying concurrency but cannot describe conflicts and alternatives in processes.

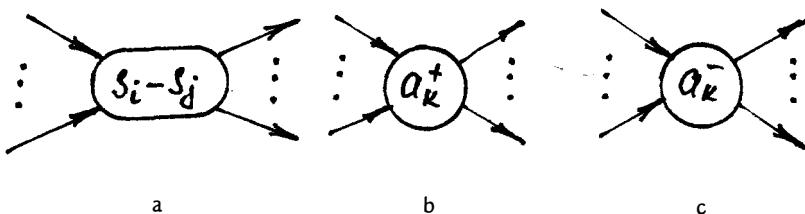


Figure 2.10 (a) - (c). Three types of vertices in signal graphs.

A semantic interpretation of a marked graph can be introduced, for example, in the following way. We associate vertices in a graph with certain transition notations. Let us take advantage of the notations presented in Fig. 2.10. When a vertex, labelled as shown in Fig. 2.10(a), is enabled, then the transition $s_i - s_j$ is initiated. When s_j is reached, the vertex fires. Any intermediate situation of the transition cannot change the marking. The situations of the transitions of the form $s_i - s_j$ may be structured in a usual way and assigned to some binary encoding. The firing of a vertex of the form presented in Fig. 2.10(b) (Fig. 2.10(c)) corresponds to the change of the value from 0 to 1 (from 1 to 0) of some component a_k .

DEFINITION 2.19. A *signal graph* is a semantic interpretation of a marked graph such that it assigns vertices of the graph to transitions of types $s_i - s_j$, a_k^+ , a_k^- .

Thus, a signal graph has an advantage allowing us to reduce the description of a certain class of APs – a complete fragment of an AP can be expressed in a succinct way by using a single transition between an initiator and a resultant. The classification of signal graphs inherits the properties of that of marked graphs. Examples of using signal graphs for the definition of the operation of circuits and devices are presented at length in the following chapters.

2.4 The Muller model

Consider an asynchronous process with the following special features.

1. Situations from S are structured in such a way that they are represented as *combinations* (a_1, \dots, a_n) of values of *binary variables* z_i , $1 < i < n$.
2. The relation F is represented by a directed graph whose vertices correspond to situations. If $s_i F s_j$, then the vertices s_i and s_j are connected by an arc directed from s_i to s_j . Moreover, if in s_i a component z_k is equal to a_i , and in s_j , z_k is equal to a_j , and $a_i \neq a_j$, $a_i, a_j \in \{0, 1\}$, then the value a_i of the component z_k is marked with an asterisk (*) in the combination s_i . Components whose values are marked with asterisks are called *excited*. Other components are called *idle*.
3. Initiators and resultants are assigned in accordance with Definition 2.2.

DEFINITION 2.20. A model and semantic interpretation of an asynchronous process that satisfy conditions 1 – 3 is called a *transition diagram*, or, fully, a *state-transition diagram*.

DEFINITION 2.21. A situation s_i of a transition diagram is called *conflict if*

- 1) there exists a component z_k whose value is marked with an asterisk in s_i ;
- 2) there exists a situation s_j such that $s_i F s_j$, the values of component z_k in s_i and s_j are equal, and the value z_k is not marked with an asterisk in s_j .

DEFINITION 2.22. A transition diagram is called *semi-modular*, if it does not contain conflict situations.

Note that Definitions 2.21 and 2.22 are analogous to Definition 2.15 related to a persistent Petri net.

EXAMPLE 2.8. A transition diagram that corresponds to a certain autonomous AP is shown in Fig. 2.11(a). It does not contain conflict situations and is, thus, semi-modular. We should also point out that the given transition diagram can be represented in the form of a signal graph, as shown in Fig. 2.11(b).

DEFINITION 2.23. A transition diagram is called *controlled* if it corresponds to a controlled asynchronous process.

Subclasses of controlled transition diagrams are semi-modular, *distributive* and *totally sequential* diagrams, denoted by U, D and K, respectively. The class of controlled transition diagrams is denoted by W.

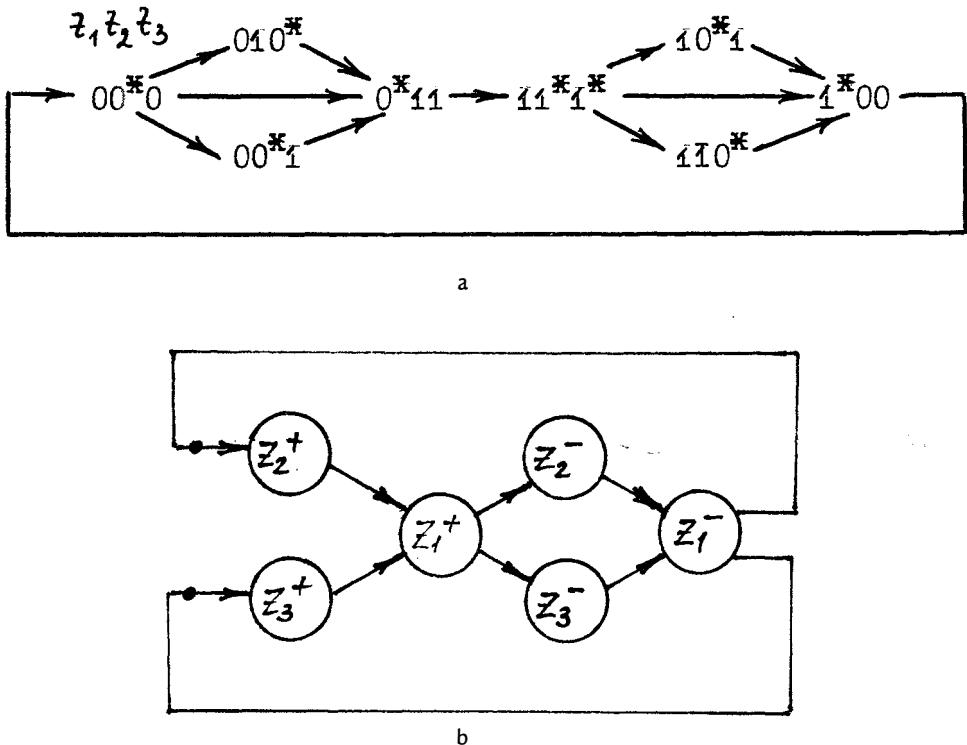


Figure 2.11 (a) and (b). (a) A state-transition diagram and (b) the corresponding signal graph.

The definition of semi-modular transition diagrams has just been given. However, an alternative definition can be outlined from which the reason for using the term “semi-modularity” becomes clearer. In doing so, we first introduce the notion of a *cumulative state* which is a vector whose i -th component is the number of changes of the value of component z_i on the trace leading from a given initiator. Cumulative states may belong to a partially ordered set that allows us to define the above mentioned subclasses of diagrams in the following way.

DEFINITION 2.24. A *semi-modular (distributive, totally sequential) transition diagram* is a diagram for which the set of cumulative states forms a semi-modular (distributive, totally ordered) lattice.

The obvious conclusion for the classes holds: $K \subset D \subset U \subset W$.

DEFINITION 2.25. The *Muller model* of an asynchronous process is a system of Boolean equations $z_i = f_i(z_1, \dots, z_i, \dots, z_n)$, $i = 1, 2, \dots, n$.

In some cases the Muller model of a process can be augmented by giving a combination of initial values of variables. Such a model is called the initialized Muller model. If the model corresponds to a non-autonomous AP those initial values stand for the AP initiators.

The relationship between a transition diagram and a system of Boolean equations can be established by using a set of rules that demonstrate the way of deriving equations from a truth table built from a given transition diagram. These rules are:

- 1) the left hand side of the truth table contains all possible binary combinations of values of diagram variables, given, for example, in the increasing order of their decimal equivalents; those combinations that are used in the diagram should be marked with asterisks at excited variables;
- 2) in the right hand side of the table, opposite to every combination that is used in the diagram, we write a combination in which all variables, whose values are not marked with an asterisk in the corresponding left combination, have the same values as in the left hand side, and those, whose values are marked in the left hand side, have inverse values;
- 3) the combinations in the left hand side which are not used in the transition diagram are associated with “don't care” values in the right hand side at appropriate rows, (traditionally “don't care” values can be given 0 or 1 values in an arbitrary way when a corresponding logical function is minimized).

A system of Boolean equations can then be derived from a table by using any of the known minimization algorithms.

EXAMPLE 2.8 (continued). From the transition diagram shown in Fig. 2.11(a), we can build a truth table, given in Table 2.1.

z_1	z_2	z_3	z_1	z_2	z_3
0	0^*	0^*	0	1	1
1^*	0	0	0	0	0
0	1	0^*	0	1	1
1	1^*	0	1	0	0
0	0^*	1	0	1	1
1	0	1^*	1	0	0
0^*	1	1	1	1	1
1	1^*	1^*	1	0	0

Table 2.1

The table is then converted into the following system of equations:

$$z_1 = z_2 z_3 \vee (z_2 \vee z_3) z_1, \quad z_2 = \bar{z}_1, \quad z_3 = \bar{z}_1.$$

The Muller model is an effective technique for defining APs in as much detail as a set of inherent functions of logical elements of a corresponding circuit allows.

DEFINITION 2.26. In the Muller model for a circuit, the vector $\alpha = \alpha_1, \dots, \alpha_n$, $\alpha_i \in \{0, 1\}$ of the values of all the variables z_1, \dots, z_n is called *the state of the circuit*. A variable z_i is called *excited* in a state α if $\alpha_i \neq f(\alpha)$, and is called *idle* (sometimes called in *equilibrium*) in the opposite case.

We consider each variable in the Muller model for a circuit to be the output of an element in the modelled circuit. We assume that the element operates in the following way. When it is excited, after some time interval, it moves to the idle state with the consequence that the *output value is changed*. If, in the excited state, the set of its input values changes in such a way that the excitement conditions are removed, then the element returns to the idle state without a change of its output value.

From state α the model may go to any state β such that it differs from α only in the values of variables that are excited in α . Then it is said that α *immediately precedes* β , or β *immediately follows* α (is immediately reachable from α). The relation of immediate reachability in the Muller model will be denoted by $\alpha \rightarrow \beta$. If $\alpha \rightarrow \beta$ and α differs from β only in the value of one variable, say, z_i , then we shall denote that as $\alpha \xrightarrow{z_i} \beta$, or $\alpha \xrightarrow{1} \beta$. In this case, we shall say that α and β are adjacent with respect to z_i .

2.5 Parallel asynchronous flow charts

Flow charts are widely used for specifying sequential algorithms. To be able to define asynchronous processes within the same specification strategy, flow charts must be able to express concurrency and asynchrony in an adequate way. Our objective, here, is to take into consideration a modified version of flow charts which is called a *parallel asynchronous flow chart* (PAFC).

PAFCs are such flow charts that may be represented as directed graphs with five types of vertices: operator, conditional branch, merger, bifurcator and synchronizer. The symbolic notation of vertices is given in Fig. 2.12.

A vertex of *operator type* (Fig. 2.12(a)) stands for the execution of an action A_i . This type of vertices has always one incoming, and one outgoing, arc.

A *conditional branch* vertex (Fig. 2.12(b)) has one incoming and two outgoing arcs. The direction of flow from this vertex depends upon a decision made as a result of computation of some predicate q corresponding to the vertex. One of the output arcs stands for the value $q = 1$, and the other stands for the alternative $\bar{q} = 1$.

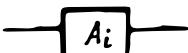
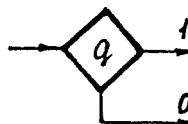
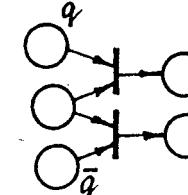
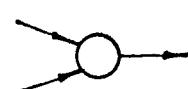
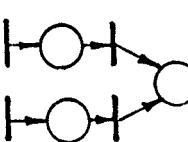
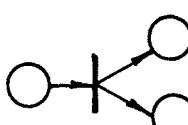
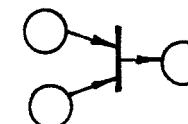
No.	PAFC Vertex Type	Vertex Name	Corresponding Petri Net Fragment
1		Operator	
2		Conditional Branch	
3		Merger	
4		Bifurcator	
5		Synchronizer	

Figure 2.12 (a) - (e). Five types of vertices in parallel asynchronous flow charts and corresponding fragments of Petri nets.

A *merger* vertex (Fig. 2.12(c)) allows us to realize the merging of alternative input flows into one output flow. In other words, it is used for graphic representation of a transition to some fragment of the algorithm from different points. Vertices of that type may have two or more input arcs and exactly one output arc. The design of a PAFC should guarantee that exactly one input may be simultaneously

active with flow in the merger, in order to avoid clashes. A PAFC satisfying that condition is called safe, by analogy with a Petri net.

A *bifurcator* vertex (Fig. 2.12(d)) realizes the activation of flows in a number of parallel fragments. These fragments are executed concurrently. Vertices of that type should always have two or more output arcs with a single input arc. The same semantics in other parallel programming notations is represented by such operators as *parbegin* and *fork*.

A *synchronizer* vertex (Fig. 2.12(e)) realizes a so-called *rendez-vous mechanism* between two or more parallel sections of PAFC which may be activated concurrently. The synchronization is achieved only upon the termination of executions in all input sections. Connected to such vertices may be exactly one output, and two or more input, arcs. The same sort of semantics in other parallel programming notations is expressed by, for example, *parend* or *join* primitives.

In order to save the total number of vertices, we may also allow the use of pairwise combination of some types of vertices, for example, a bifurcator and a synchronizer, to form a single vertex.

Using such vertices as operator and conditional branch, we already assume the presence of a semantic interpretation of the AP to be described. However, for a more precise model interpretation it is necessary to introduce a mechanism for marking input and output arcs for each vertex with tokens, in much the same way as it is in a marked graph. Thus, a current situation in an AP described by a PAFC is given by its marking.

The changing of a marking in a PAFC is executed according to the following rules:

1. If an input arc of an operator or a conditional branch has a token on it, then the given operator or branch vertex may be executed. Upon execution, (or firing) of the operator or the branch, the token is removed from the input arc and the corresponding output arc is marked with a token.
2. For the remaining vertices to fire, it is necessary: a) for a merger vertex – one of its input arcs must have a token, and the firing results in the appearance of a token on its output arc, b) for a synchronizer vertex – all of its input arcs must have tokens, and the result of firing is the same as for a merger vertex, and c) for a bifurcator – the input arc must be marked with a token, and as a result of firing, all the output arcs become marked with tokens.
3. We also assume that the removal of tokens from input arc(s) and adding tokens to output arc(s) are indivisible operations.

Thus, the execution of an AP is simulated by changes of markings on the arcs of PAFC. It is obvious that a PAFC may be interpreted by a Petri net composed of fragments corresponding to the vertices of a PAFC (Fig. 2.12) and subjected to the same semantic interpretation as the PAFC.

In a PAFC there is usually a vertex with no input arcs – the begin operator – and a vertex with no output arcs – the end operator. Then, the initiator of a process given by a PAFC is the marking with a single token on the output arc of the begin operator, and the resultant is the marking with a token on the arc leading to the end operator. Connecting the begin and the end operators by an arc directed from the end to the begin, we obtain an autonomous AP. Examples of using PAFCs are presented in subsequent chapters.

2.6 Asynchronous state machines

DEFINITION 2.27. A finite (deterministic) state machine is a quintuple $\langle Y, X, Z, \lambda, \delta \rangle$ where

- $Y = \{y_1, \dots, y_m\}$ is a finite set of *internal states*,
- $X = \{x_1, \dots, x_n\}$ is a finite set of *input symbols* (input alphabet),
- $Z = \{z_1, \dots, z_r\}$ is a finite set of *output symbols* (output alphabet),
- $\lambda : X \times Y \rightarrow Y$ is a *function of state transitions* (or the *transition function*),
- $\delta : X \times Y \rightarrow Z$ is a *function of output responses* (or the *output function*).

One of the features by which finite state machines are classified is an approach to introduce the notion of time into the above abstract model. A finite state machine, whose temporal behaviour is conducted by the environment which is responsible for arranging the time intervals during which the state of the inputs remains unchanged, is usually defined as asynchronous. The following definition makes the issue more exact.

DEFINITION 2.28. A finite (deterministic) state machine, which, after every change of an input signal reaches a stable state (sometimes termed a “steady state response”), is called an *asynchronous state machine*, i.e. for every pair (x, y) , $x \in X, y \in Y$, $\lambda(x, \lambda(x, y)) = \lambda(x, y)$ holds.

This definition may look incorrect, because a stable state that can be reached in an asynchronous machine from a sequence of unstable states, will contradict the above statement. However, if we consider the definition of an asynchronous machine that does not contain unstable states, then such a contradiction will not take place. Unstable states are those that may arise at subsequent synthesis stages. They are introduced by the designer as an extension of an original alphabet Y , thereby

making Definition 2.28 inapplicable. In this sense, we should certainly distinguish between the notion of a state machine model and its implementation.

Since, here, we are not pursuing the goal of considering in detail all the problems involved in abstract and structural synthesis of asynchronous state machines (these problems, though well studied, are far from being solved), we only hint at the following.

As is known from automata theory, during the transformation of an abstract model of an asynchronous machine into an implementation, there is need for certain concretization of a mechanism of interaction between machine and environment. Furthermore, the requirements on allowed sequences of input symbols, the methods for encoding input and output symbols, as well as internal states, in such a way that the possibility of functional hazards and races are excluded, and the assumption of a particular character of delays in elements and in communication lines, must also be elaborated in detail.

To interpret an asynchronous machine in terms of an AP, we represent a situation as a full state of the machine $s_i = (x_j^a, y_k^a, z_l^a)$, where x_j^a , y_k^a , and z_l^a are an input symbol, an output symbol and an internal state, respectively. Note that the input, the output and the internal state do not necessarily coincide with their respective counterpart components of situations in the AP representation of the state machine – in an AP, the mappings $X \times Y \times Z \rightarrow Z$ and $X \times Y \times Z \rightarrow Y$ are realized, rather than functions δ and λ , given in Definition 2.27. Let $x_j^a = (x_j, x'_j)$ and $y_k^a = (y_k, y'_k)$. Then, the input component of the process interpretation of a state machine may, for example, be x_j , and the output component may be y_k . The component z_l is then written in the form $z_l = (z_l^a, x'_h, y'_k)$. Thus, the input and output components may correspond to only a subset of the inputs and outputs. The interpretation of an initiator and a resultant for an asynchronous machine coincides with their interpretation for an AP of the general form. The relation F is defined by the transition and output functions as well as by the rules which define the reposition. The latter are usually expressed by phrases of the type “The following input symbol can be applied only after the machine has reached a stable state”, or “It is allowed to change the input combination only to the adjacent one”.

It is important to note that the proposed model of an AP allows us to describe the operation of an asynchronous state machine on different levels of refinement. On the level of abstract synthesis, both the normal table of transitions, and outputs, and that of the error functions may serve as protocols of simple APs. Intermediate situations arise on the level of encoding input and output symbols and on the state assignment level.

2.7 Reference notations

The notion of an asynchronous process was first proposed in [9]. Somewhat close to that are the formalisms given in [75], [113], [253] and [259]. They are directed towards theoretical programming topics.

The concept of equivalence classes is, in essence, adopted from [120], with some more detail given in Chapter 5.

The model, which was later called Petri net, was first proposed by C.A. Petri [285]. That model became a subject for deep and extensive investigation, and, by now, has obtained the status of an independent research field in computer science. A detailed survey of the major achievements in Petri nets theory was reported in [284], which was followed by [144]. The advantage of these survey works, which also contain vast bibliographies, is their informal style of presentation of the latest results in the area. There are also several other publications which discuss the problems of Petri nets in a thorough way. These are [275] and [292]. A considerable amount of research effort has been applied in this area within the framework of the MAC Project at MIT [83], [224], [225], [238], [239], [243], [260], [263], [264], [269], [278], [279] and [280] where this research was in association with novel computer architectures and further steps on automata theory. Many interesting results are obtained in Bonn (FRG) where Petri nets have been studied in a more formal and abstract way. These results are well presented in [284]. The geography of Petri nets expands rapidly, involving more and more mathematicians and computer scientists. A survey of French research on Petri nets and adjacent topics is given in [299]. Petri net studies in the Soviet Union are reported, for example, in [99], [101], [132] and [164]. So far, three books have been published, exclusively on the subject of Petri nets [102], [283] and [295].

The fundamental research on marked graphs was reported in [260]. The concept of a signal graph, introduced here, differs from the interpretation given in [247], but has been partly inspired by the switching table, proposed some time ago, by M.A. Gavrilov [71].

The modelling approach described in Section 2.4 was proposed by D.E. Muller and his colleagues [199], [229], [270–273]. It was also given, at length, in [120] and is based on lattice theory, for which we recommend [47] and [81].

Parallel asynchronous flow charts, as they are presented in this text, were also given in [53], [56], [62], and [307], and they are a modification of the well-known flow chart notation [45], [46] and [108] that is oriented towards an adequate description of asynchronous processes.

Various aspects of the theory of state machines and sequential networks can be found in books [7], [48], [50], [54], [71], [72], [74], [78], [80], [84], [86–91], [100], [106], [108], [109], [117], [120], [121], [137], [143], [166], [169], [170], [173], [187] and [208], while the subject of asynchronous machines is of

considerable interest in [5], [6], [10], [82], [107], [149], [153] and [191], as well as in [105].

Readers especially interested in other potential model and semantic interpretations of asynchronous processes are recommended to refer to the surveys [76], [102], [103], [125], [198], and other works [83], [96], [104], [167], [189], [224], [246] and [288].

CHAPTER 3

SELF-SYNCHRONIZING CODES

A famous publishing magnate once had a talk with a philosopher:
‘I would be glad if I had also taken in “Time”’,
said the magnate.
‘But I am afraid, sir, that time can never be
taken in’, was the answer.

Paraphrase of Ilya Varshavsky.

This chapter presents a discussion of signal representation problems that arise in the design of both functional circuits and data transfer facilities, whose behaviour is required to be tolerant of variations in temporal parameters of logical elements and communication channels.

In the preceding chapter, the idea of an asynchronous process was introduced as the basis for a number of dynamic models which are used throughout the text for specifying the behaviour of asynchronous concurrent algorithms and devices inherent in computers and discrete control systems.

As one may have gathered from the introduction, the major goal of the book is to demonstrate how the self-timed style vehicle can be used in approaching the design of the hardware support of parallel asynchronous processes. Although most of the topics studied are fairly specific, because they are accommodated within the design fashion declared, the material of the book is organized in the order which is quite typical for volumes on automata theory and its applications. Such an order implies the following traditional chain of stages: modelling (algorithm of functioning) – abstract synthesis (encoding of states and signals) – structured synthesis (constructing an implementation in a given functional element basis) etc. The step involving the encoding of states (situations) and signals is of prime importance, since that represents a bridge between the abstract and structural synthesis stages. This chapter claims to be a systematic review of encoding techniques for signals transmitted along parallel data lines with a particular emphasis on totally asynchronous design requirements.

3.1 Preliminary definitions

We define some notions that will be needed for further discussion.

Consider a transition $s_i - s_k$ (see Definition 2.5) in an AP with structured situations which are encoded with sets of values of the binary variables z_1, \dots, z_n in

the set Z . A set of values of such variables, which is an n -tuple of 0's and 1's, will further be called a *combination*.

Let a combination $a = a_1 \dots a_n$ be assigned for s_i , and a combination $b = b_1 \dots b_n$ be assigned for s_k .

We denote the constituent of unity of the combination a as $\omega(a)$, i.e.

$$\omega(a) = \bigwedge_{i=1}^n z_i^{a_i}$$

where $z_i^{a_i} = z_i$ if $a_i = 1$, and $z_i^{a_i} = \bar{z}_i$ if $a_i = 0$.

Let us build an n -tuple of the form $\alpha_1 \dots \alpha_n$ such that

$$\alpha_j = \begin{cases} 0, & \text{if } a_j = b_j = 0 \\ 1, & \text{if } a_j = b_j = 1 \\ -, & \text{if } a_j \neq b_j, \quad 1 \leq j \leq n. \end{cases}$$

Let that n -tuple contain k dashes (if $k = 0$ then $a = b$). If we assign dashes to all possible configurations of 0's and 1's (by 2^k ways), we shall obtain a set of combinations which we shall call a *sub-cube* (a, b) of transition $a-b$. The usage of this term is due to the geometrical interpretation of a transition. The sub-cube of transition $a-b$ contains all the situations in the transition from a to b inclusive. The sub-cube without combinations a and b will be called the *internal sub-cube* and denoted as $[a, b]$.

The sub-cube (a, b) of transition $a-b$ can also be associated with the *transition constant term*

$$\omega(a, b) = \bigwedge_{j=1}^n \beta_j$$

where

$$\beta_j = \begin{cases} \bar{z}_j, & \text{if } \alpha_j = 0 \\ z_j, & \text{if } \alpha_j = 1 \\ 1, & \text{if } \alpha_j = -, \quad 1 \leq j \leq n. \end{cases}$$

Thus, the transition constant term contains only those variables which do not change in the given transition.

Furthermore, we build an n -tuple of the form $\gamma_1 \dots \gamma_n$ such that

$$\gamma_j = \begin{cases} 0, & \text{if } a_j = 1, b_j = 0 \\ 1, & \text{if } a_j = 0, b_j = 1 \\ -, & \text{if } a_j = b_j, \quad 1 \leq j \leq n. \end{cases}$$

and associate it with the *transition variation term*

$$\varepsilon(a, b) = \bigwedge_{j=1}^n \rho_j$$

where

$$\rho_j = \begin{cases} \bar{z}_j, & \text{if } \gamma_j = 0 \\ z_j, & \text{if } \gamma_j = 1 \\ 1, & \text{if } \gamma_j = -, \quad 1 \leq j \leq n. \end{cases}$$

Thus, the transition variation term contains only those variables which change in the given transition.

DEFINITION 3.1. A transition $a-b$ is called *regular* if, in this transition, each variable $z_j, 1 \leq j \leq n$, changes its value at most once.

For every combination t_k in a regular transition such that $t_k \neq a$ and $t_k \neq b$, $t_k \in [a, b]$ holds.

DEFINITION 3.2. Combinations a and b are called *adjacent* (w.r.t. z_j or \bar{z}_j), if $\varepsilon(a, b) = z_j$ or $\varepsilon(a, b) = \bar{z}_j$ holds for transition $a-b$. Such a transition will also be called an *adjacent transition*.

DEFINITION 3.3. Combinations a and b are called *comparable*, if the transition variation term $\varepsilon(a, b)$ has either all its variables in the direct form (without inversion bars) or all variables in the complemented form, i.e. either $a < b$, or $a > b$, respectively. A transition between comparable combinations will also be called a *comparable transition*.

We denote by $N^-(b)$ ($N^+(b)$) a set of combinations c that are adjacent to b and such that $c \in [a, b]$, and if $c \in N^-(b)$ ($c \in N^+(b)$), then $c < b$ ($c > b$). It is obvious

that the cardinality of $N^-(b)$ for the transition $a-b$ is equal to the number of variables which enter in the variation term $\varepsilon(a, b)$ without inversion bars, and the cardinality of $N^+(b)$ for $a-b$ is equal to the number of variables entering $\varepsilon(a, b)$ in the complemented form.

DEFINITION 3.4. A combination which is comparable with all combinations from a given set is called a *spacer*.

If the set is given on 2^n combinations, then the spacers are combinations denoted as i and e which consist of only 0's and 1's, respectively.

DEFINITION 3.5. A Boolean function $f(z_1, \dots, z_n)$ is called

- a) *isotonous (antitonous) in z_j* , if $f(z_j = 1) \geq f(z_j = 0)$ ($f(z_j = 1) \leq f(z_j = 0)$);
- b) *isotonous (antitonous)*, if it is isotonous (antitonous) in each of its variables;
- c) *monotonous*, if it is either isotonous or antitonous;
- d) *isotonous (antitonous) in transition $a-b$* , if it is isotonous (antitonous) in all the variables which enter in the transition variation term $\varepsilon(a, b)$.

The notion of “isotonous function” is synonymous with the notion of “positive function”, and that of “antitonous function” has the same meaning as “negative function”. The term “monotonous function”⁽¹⁾ is thus used here to stand for functions that are either isotonous or antitonous in all their variables. In the literature on discrete mathematics, the term “monotonous function” is often used in a more restricted sense, i.e. for the function which is isotonous in all its variables.

We should also note there that if $f(z_1, \dots, z_n)$ is isotonous in z_j , then it can be represented in the form⁽²⁾

$$f(z_1, \dots, z_n) = z_j f(z_j = 1) \vee f(z_j = 0)$$

where

$$f(z_j = c) = f(z_1, \dots, z_{j-1}, c, z_{j+1}, \dots, z_n).$$

Hence, if f is isotonous, then for any comparable transition $a-b$ ($a < b$) $f(a) \leq f(b)$ holds.

A similar representation can be obtained for antitonous functions.

(1) One may also meet the term “unate function”, as, for example, in [120].

(2) Here and in the sequel, we omit the conjunction operator sign in Boolean expressions, by using an attached symbol notation.

EXAMPLE 3.1. The combinations $i = 00000$ (spacer) and $r = 10011$ correspond to terms $\omega(i) = \bar{z}_1 \bar{z}_2 \bar{z}_3 \bar{z}_4 \bar{z}_5$ and $\omega(r) = z_1 \bar{z}_2 \bar{z}_3 z_4 z_5$. The sub-cube (i, r) of the transition $i \rightarrow r$ corresponds to the transition constant term $\omega(i, r) = \bar{z}_2 \bar{z}_3$ and transition variation term $\varepsilon(i, r) = z_1 z_4 z_5$. The combinations are comparable, i.e. $i < r$, but not adjacent. The set $O^-(r)$ of combinations adjacent to r in the transition $i \rightarrow r$ consists of 00011, 10001 and 10010.

EXAMPLE 3.2. The function $f = z_1 \bar{z}_2 \vee z_2 \bar{z}_3$ is isotonic in z_1 , antitonic in z_3 and not monotonic in z_2 .

We now turn our attention to the definition of self-synchronizing codes. The notion of such a code is fundamental for specifying the classes of encoding systems which are used in the state assignment for aperiodic machines and the representation of signals in asynchronous systems with parallel data transfers.

Consider an alphabet X in which every symbol ξ is associated with a subset of symbols $X(\xi) \subseteq X$ that may appear after ξ . The subset $X(\xi)$ may, in the general case, include all symbols of the alphabet ($X(\xi) = X$). The least possible number of symbols in $X(\xi)$ is certainly one.

The subject matter will be a *code system* Z whose cardinality (the number of code combinations in the system) is not less than the cardinality of the alphabet X (the number of symbols in X), and every symbol is assigned to a unique (not coinciding with another symbol's code) code combination. The encoding method presumes that for a given ξ each symbol α that may follow ξ , i.e. $\alpha \in X(\xi)$, is assigned to a particular code combination $a \in Z$. Such an encoding satisfies the following requirements:

- 1) the set of symbols $X(\xi)$ corresponds to the set of code combinations $Z(\xi) \subseteq Z$ with the same cardinality as that of $X(\xi)$;
- 2) each symbol may, in the general case, correspond to several code combinations but the decoding of a combination is unique only if the combination preceding the decoded one is known.

Let $z(a)$ denote a set of combinations in the code system Z that may appear after a . If the symbol α is encoded with a , then $Z(a) = Z(\alpha)$.

DEFINITION 3.6. A transition $a \rightarrow b$ in the code system Z is called *allowed*, if the following conditions are satisfied:

- 1) $a \in Z, b \in Z(a)$;
- 2) $a \neq b$;
- 3) $a \rightarrow b$ is a regular transition;
- 4) for any $t \in [a, b] t \notin Z(a)$.

DEFINITION 3.7. A code system, in which every possible transition is allowed, is called a *self-synchronizing code* (SSC).

As will be shown below, the completion of an allowed transition $a-b$ can be indicated by registering the resultant combination b without any notion of transition realization time. This feature lies at the heart of the title given to the codes being discussed – self-synchronizing. The existence of SSCs will be proved constructively in the following sections.

3.2 Direct-transition codes

Let us consider a case, when

- a) for every symbol ξ we explicitly define a set $X(\xi)$ of symbols that may appear after ξ ;
- b) one and the same symbol may not appear in two consecutive moments of time, i.e. $\xi \notin X(\xi)$;
- c) every symbol is associated with a unique code combination.

DEFINITION 3.8. A code system satisfying the above conditions is called a *direct-transition code*.

For this code, a particular encoding discipline will be presented that is based on the following requirements.

1. According to Condition 4 in Definition 3.6, for any combination $x \in Z(a) \setminus b$, there should exist an encoding variable such that, in the sub-cube (a, b) , and, hence, in both a and b , it assumes one value, say 1, and in combination x , it assumes the opposite value (0). Such a variable gives rise to, what is called in automata theory, a π -partition of the following form

$$\pi = \{\overline{a.b}, \bar{x}\} \quad (3.1)$$

(Recall that π -partition on a subset S_π of a set S is a set of subsets (blocks) in S_π such that their pairwise intersections are empty, and the union of all the blocks constitutes S_π . If, for some two-block π -partition, the states of one block are assigned the 0-value (1-value) of the encoding variable y_k , while the states of the other block are assigned with the 1-value (0-value) of that variable, then it is said that the variable y_k generates this π -partition).

2. If combination a may be followed only by combination b , then the sub-cube (a, b) may be arbitrary, i.e. the realization of the transition $a-b$ puts no restrictions on the encoding.

3. All the symbols in X must be encoded by different combinations. This implies that, for any pair of symbols, α and β , $\alpha, \beta \in X$, there must be found an *encoding variable* that assumes different values in a and b which correspond to these symbols, i.e. the variable which gives rise to the π -partition of the form

$$\pi = \{\bar{a}, \bar{b}\} \quad (3.2)$$

EXAMPLE 3.3. Let $X = \{\alpha, \beta, \sigma, \delta, \varepsilon, \phi\}$, $X(\alpha) = \{\beta, \sigma, \delta\}$, $X(\beta) = \{\delta, \varepsilon\}$, $X(\sigma) = \{\delta, \phi\}$, $X(\delta) = \{\phi\}$, $X(\varepsilon) = \{\alpha\}$, $X(\phi) = \{\alpha, \varepsilon\}$. The result of the encoding produces combinations a, b, c, d, e, f that form a code system Z and correspond to symbols $\alpha, \beta, \sigma, \delta, \varepsilon, \phi$. Then $Z(a) = \{b, c, d\}$, $Z(b) = \{d, e\}$, $Z(c) = \{d, f\}$, $Z(d) = \{f\}$, $Z(e) = \{a\}$, $Z(f) = \{a, e\}$.

We build a table of π -partitions of the type (3.1). For the example considered, such a table is formed by the first twelve rows of Table 3.1. In this table, columns are marked with symbols in alphabet X and rows correspond to π -partitions.

π_i	α	β	σ	δ	ε	ϕ
$\pi_1 = \{\bar{a}, \bar{b}\}$	1	1	0	—	—	—
$\pi_2 = \{\bar{a}, \bar{b}, \bar{d}\}$	1	1	—	0	—	—
$\pi_3 = \{\bar{a}, \bar{c}, \bar{b}\}$	1	0	1	—	—	—
$\pi_4 = \{\bar{a}, \bar{c}, \bar{d}\}$	1	—	1	0	—	—
$\pi_5 = \{\bar{a}, \bar{d}, \bar{b}\}$	1	0	—	1	—	—
$\pi_6 = \{\bar{a}, \bar{d}, \bar{c}\}$	1	—	0	1	—	—
$\pi_7 = \{\bar{b}, \bar{e}, \bar{d}\}$	—	1	—	0	1	—
$\pi_8 = \{\bar{b}, \bar{d}, \bar{e}\}$	—	1	—	1	0	—
$\pi_9 = \{\bar{c}, \bar{d}, \bar{f}\}$	—	—	1	1	—	1
$\pi_{10} = \{\bar{c}, \bar{f}, \bar{d}\}$	—	—	1	0	—	1
$\pi_{11} = \{\bar{a}, \bar{f}, \bar{e}\}$	1	—	—	—	0	1
$\pi_{12} = \{\bar{e}, \bar{f}, \bar{a}\}$	0	—	—	—	1	1
$\pi_{13} = \{\bar{b}, \bar{f}, \bar{e}\}$	—	1	—	—	—	0
$\pi_{14} = \{\bar{c}, \bar{e}\}$	—	—	1	—	0	—

Table 3.1

A table entry is equal to 1, if the symbol marking the column is a member of the first block of the partition, and is equal to 0, if such a symbol enters in the second block.

An entry is marked with a dash (-), if the symbol does not belong to either of the blocks. It is also necessary to include partitions of the form (3.2) into the table of π -partitions. It is, however, a consequence of the existence of the partition $\{\bar{a}, \bar{b}, \bar{c}\}$ that the same variable gives rise to the partitions $\{\bar{a}, \bar{c}\}$ and $\{\bar{b}, \bar{c}\}$. Thus, it is easy to prove that only two partitions of the form (3.2), viz. π_{13} and π_{14} , should be added to Table 3.1.

Further steps towards a solution to the problem of encoding symbols by combinations with minimum code length are performed using a method based on the search of so-called largest groups of intersections.

DEFINITION 3.9. Two combinations $a = a_1 \dots a_n$ and $b = b_1 \dots b_n$ whose elements a_i and b_i ($1 \leq i < n$) can assume values from $\{0, 1, -\}$ form an *intersection* if one of the following conditions is satisfied:

- either $a_i = b_i$ for all i such that $a_i, b_i \in \{0, 1\}$,
- or $a_i \neq b_i$ for all such i .

We shall say that a string of 0's and 1's with a dimension equal to the cardinality of the alphabet gives rise to a partition π_i , if it forms an intersection with a row π_i of the table of π -partitions.

Because the rows corresponding to π -partitions are not defined in some positions (some entries are “dashed”) a single string of 0's and 1's may give rise to a number of partitions. Let us assign dashes in the table rows to 0's and 1's in all possible ways. In the set of rows obtained, we may find pairs of rows that form an intersection. From every such pair, we leave one row in the list, and delete the other. We continue the deletion process until there are no more rows forming intersections. Then we take each of the remaining rows and compare it with the rows of the table of partitions. We record which partitions the given row generates.

The following step involves solving a *covering problem*: to choose the minimum number of rows generating all the partitions.

Using the procedure described, we construct Table 3.2 for Example 3.3. The numbers of partitions generated by each row are written next to that row. In finding the minimum number of rows generating all partitions, one may refer to any of the known methods for solving the covering problem⁽¹⁾. In our example, the minimum number of rows is three. These rows are marked with “ \checkmark ”. The choice of these rows corresponds to the following encoding: $a = 111, b = 110, c = 011, d = 101, e = 000, f = 010$. Each code combination is formed by a corresponding column in the “reduced” Table 3.2 obtained by deleting all the rows marked with “ \checkmark ”.

(1) The problems of estimating the computational complexity of the covering problem are outside the subject matter of this text.

α	β	σ	δ	ε	ϕ	π_i
1	1	0	0	0	0	1, 2, 12, 13
v	1	1	0	1	0	1, 6, 8, 10, 12, 13
1	1	0	0	1	0	1, 2, 7, 13, 14
1	1	0	1	1	0	1, 6, 10, 13, 14
1	1	0	0	0	1	1, 2, 9, 11
1	1	0	1	0	1	1, 6, 8, 11, 14
1	1	0	0	1	1	1, 2, 7, 9, 14
1	1	0	1	1	1	1, 6, 14
1	1	1	0	0	0	2, 4, 12, 13, 14
1	1	1	0	1	0	2, 4, 7, 13
v	1	1	1	0	1	2, 4, 10, 11, 14
1	1	1	0	1	1	2, 4, 7, 10
1	0	1	0	0	0	3, 4, 12
v	1	0	1	1	0	3, 5, 7, 9, 12, 14
1	0	1	0	1	0	3, 4, 8, 14
1	0	1	1	1	0	3, 5, 9
1	0	1	0	0	1	3, 4, 10, 11, 13
1	0	1	1	0	1	3, 5, 7, 11, 13, 14
1	0	1	0	1	1	3, 4, 8, 10, 13, 14
1	0	1	1	1	1	3, 5, 13
1	0	0	1	0	0	3, 6, 7, 10, 11
1	0	0	1	1	0	5, 6, 10, 13
1	0	0	1	0	1	5, 6, 7, 11, 13
1	0	0	1	1	1	5, 6, 13, 14
1	0	0	0	1	1	8, 9, 13, 14
1	1	1	1	0	0	8, 9, 12, 13, 14
1	0	0	0	1	0	8, 14
1	1	1	1	0	1	8, 11, 14
1	0	0	0	0	1,	9, 11, 13
1	1	1	1	1	0	9, 13

Table 3.2

As a conclusion to this section, we note that for the code considered, any possible transition is performed in one step – without fixing the intermediate (empty) combinations. This feature is not characteristic of all SSCs.

3.3 Two-phase codes

The code systems which are considered in this, and the subsequent sections, are used if the set of symbols to be encoded is split into two subsets – S and L . The subset S

contains a single, *empty*, symbol, while the subset L contains the remaining symbols, which we shall call *working* symbols. For any $\lambda \in L$, $X(\lambda) = S = \{\sigma\}$ and $X(\sigma) = X \setminus S$, i.e. after the empty symbol, any working symbol may appear, and after any working symbol, only the empty symbol may do so. Such a constraint, though being not too important for direct-transition codes, is accepted for the convenience of message interpretation – working symbols carry the transmitted data, whereas the empty symbol stands for a delimiter, or pause, between messages. The reception of a working code combination initiates the active phase in a corresponding device, whereas the reception of a delimiter (*spacer*) initiates the passive or idle phase which implies the preparation of the device for the reception of the next working code combination. Due to that discipline, the SSCs which will be discussed further, are given a common title – two-phase codes.

In the following three code systems, every symbol is assigned to one code combination in such a way that the empty symbol is encoded by a spacer, and the working symbols are encoded by incomparable code combinations.

3.4 Double-rail code

In this code system, we use $2^n + 1$ code combinations of the length $2n$. 2^n of these combinations are pairwise incomparable. The working symbols of the alphabet X with the cardinality $|X|$ are assigned numbers from 0 to $|X| - 2$. Binary representations of the symbol numbers are then re-encoded to combinations of a double-rail code in such a way that the i -th bit of the binary number is associated with two bits of the double-rail code: z_{2i-1} and z_{2i} . If the i -th bit of the binary number has the value a_i , then the double-rail representation is $z_{2i-1} = a_i$ and $z_{2i} = \bar{a}_i$. The empty symbol corresponds to one of the spacers.

An example of encoding the alphabet $X = \{\alpha, \beta, \gamma, \delta, \sigma\}$ with the empty symbol σ is presented in Table 3.3. The symbol σ is assigned to the spacer i .

Symbol	Binary Code	Double-rail code			
α	00	0	1	0	1
β	01	0	1	1	0
γ	10	1	0	0	1
δ	11	1	0	1	0
σ	–	0	0	0	0

Table 3.3

THEOREM 3.1. *A double-rail code is a self-synchronizing code, if each possible transition is a regular one.*

Proof. We demonstrate that each possible transition in the double-rail code is allowed, i.e. that all the conditions in Definition 3.6 are satisfied. Conditions 1 and 2 are already true from the definition of a double-rail code. The regularity of each transition guarantees Condition 3. Furthermore, since the working combinations of a double-rail code are pairwise incomparable, then for any pair of combinations a and b we can find a variable giving rise to the partition $\pi = \{\overline{a}, \overline{i}, \overline{b}\}$. Hence $b \notin (a, i)$, and Condition 4 is also satisfied. *Q.e.d.*

We abbreviate “double-rail code” to DRC. Besides, for the sake of convenience in further discussions, we agree upon the following. If we use DRC with the all-one spacer e , then the j -th bit of the DRC will be designated by the pair (x_j, \check{x}_j) . Otherwise, i.e. using the all-zero spacer, the j -th bit will be designated by the pair (x_j, \hat{x}_j) . Thus, the pair (x_j, \check{x}_j) is defined on the set of values $\{10, 01, 11\}$, while the pair (x_j, \hat{x}_j) is defined on $\{10, 01, 00\}$. In such definitions, the combinations 10 and 01 are assigned to the values 1 and 0 (respectively) of the encoded bit of a symbol. The combination 11 or 00 is associated with the transient (intermediate) value of the bit.

3.5 Code with identifier

In this code system, we use binary combinations consisting of n *basic* and k *additional* bits, $k = \lceil \log_2(n + 1) \rceil$; we remember that the notation $\lceil \cdot \rceil$ is used here, and in the following, to designate the closest upper integer.

The basic bits are used for encoding *working symbols*, while additional bits are used as an *identifier*. The empty symbol is encoded by one of the spacers. For the sake of definiteness, we shall use, as a spacer, the combination i consisting of all, both basic and additional, bits equal to zero.

We divide the code combinations represented by basic bits into $n + 1$ classes in such a way that codes with an equal number w of ones, $0 \leq w \leq n$, enter in the same class (w -*class*). In the same way, we divide the identifier codes represented by additional bits, thus forming v -*classes*, $0 \leq v \leq k$.

For the combinations which encode working symbols, we require that the following three conditions hold, simultaneously.

1. Each code combination in v -class is associated with a unique w -class.
2. If several identifiers are assigned to one w -class, then these identifiers must be incomparable.
3. If two code combinations c and d belong to different w -classes, have comparable identifiers and $w(c) > w(d)$, then $v(c) < v(d)$, where $w(c)$ and $w(d)$ are the numbers of 1's in the basic bits, and $v(c)$ and $v(d)$ are those in the identifiers of combinations c and d , respectively.

A code system satisfying the above requirements will be called a *code with identifier* (CI).

We prove the existence of a CI for the case in which each w -class is assigned exactly one identifier combination.

It is, therefore, unnecessary to check Condition 2 in this case. Condition 1 is always satisfied because the number of different w -classes is $n + 1$, and the number of different identifiers is $2^k \geq (n + 1)$ for the accepted $k = \lceil \log_2 (n + 1) \rceil$.

Condition 3 can be satisfied, for example, for the following assignment of identifiers. The class with $w = 0$, which consists of a single code consisting of n 0's, is assigned the identifier with $v = k$. Each of the following $C_k^{k-1} = k$ w -classes with $w = 1, 2, \dots, k$ will be assigned one of the identifiers in the class $v = k - 1$. Each of the following C_k^{k-2} w -classes with $w = k + 1, \dots, k + C_k^{k-2}$ will be assigned one of the identifiers in the class $v = k - 2$ etc.

If $\log_2(n + 1)$ is an integer, then the class with $w = n$ will be assigned the identifier with $v = 0$, and all code combinations in additional bits are thus reserved. Otherwise, the encoding will be completed with an identifier with $v > 0$ and some identifiers will be vacant. Then it will be possible to assign several identifiers to one w -class, thereby providing that CI can be used for encoding more than 2^n symbols.

THEOREM 3.2. *A code with identifier is a self-synchronizing code, if each possible transition is a regular one.*

Proof. As in Theorem 3.1, it is necessary to demonstrate that Condition 4 in Definition 3.6 is satisfied for a CI. Assume the contrary, that in the CI there exists a working combination c that belongs to the sub-cube (i, d) of the transition $i-d$.

Consider four cases.

1. $w(c) > w(d)$, or $v(c) > v(d)$. It is clear that, in this case, $c \notin (i, d)$ because in any part of an intermediate combination there must be more 1's than in the same part of the working combination to which the transition leads.
2. $w(c) \leq w(d)$, $v(c) < v(d)$. This case may occur because of Condition 3 of the definition of a CI only if the identifiers of combinations c and d are incomparable. This means that the combinations c and d should, thus, also be

incomparable. Consequently, the combination c cannot be intermediate in the transition $i-d$.

3. $w(c) = w(d)$, $v(c) = v(d)$. In this case, either $c = d$ which implies $c \notin [i, d]$ or $c \neq d$ and the values of the basic bits in c and d belong to the same w -class. Since the combinations in the same w -class are incomparable, we find that $c \notin (i, d)$.

4. $w(c) < w(d)$, $v(c) = v(d)$. In this case, the values of the additional bits in c and d are different according to Condition 1 of the CI definition and they belong to the same v -class. However, since the combinations in the same v -class are incomparable we have $c \notin (i, d)$.

Since there are no other cases, the assumption always leads to the contradiction. *Q.e.d.*

EXAMPLE 3.4. Consider a CI for the case $n = 8$, $k = \lceil \log_2(8 + 1) \rceil = 4$. This CI is represented by 12 bits x_1-x_{12} where x_1-x_8 are the basic bits and x_9-x_{12} are the additional bits. The representation of the CI which satisfies Conditions 1 to 3 is shown in the nine rows of the left part of Table 3.4. The leftmost column contains different w -classes, and the other columns contain the values of corresponding identifiers. These nine rows allow the encoding of $2^8 = 256$ working symbols which are represented by their basic bits in the positional notation.⁽¹⁾

w	x_9	x_{10}	x_{11}	x_{12}	w	x_9	x_{10}	x_{11}	x_{12}
0	1	1	1	1	1	1	1	1	0
1	0	1	1	1	2	0	1	1	0
2	1	0	1	1	3	1	0	1	0
3	0	0	1	1	4	0	0	1	0
4	1	1	0	1	5	1	1	0	0
5	0	1	0	1	6	0	1	0	0
6	1	0	0	1	7	1	0	0	0
7	0	0	0	1					
8	0	0	0	0					

Table 3.4

Since $\log_2(n + 1)$ is, in this case, not an integer, some of the values of identifiers remain unused (only 9 of 16 are used). The use of seven vacant values may yield a number of extra working combinations in the CI. However, one should note that

(1) The *positional binary notation* is formed by the code combinations with the ordered distribution of bit weights: $2^0, 2^1, \dots, 2^{n-1}$ where n is the number of bits.

these, auxiliary, working combinations are not represented in their basic bits in the positional binary notation.

When constructing a CI we face two problems. The first one is to find an encoding that guarantees obtaining a maximal number of extra code combinations. The second one is to find an encoding that provides (however, at the cost of informational capacity) the maximum simplicity of the conversion of extra code combinations in CI into the positional binary code, if such a conversion is necessary.

For solving the first problem, there exists an encoding procedure that is simple enough. However, we restrain ourselves from considering it here because it leads to a rather complicated decoding function. For a 12-bit CI, the maximal number of extra working combinations is 462, and the total number of working combinations will be $256 + 462 = 718$. The corresponding variant of encoding is given in Table 3.5 where the first nine rows (left hand part) represent the compulsory combinations and the other seven rows (right hand part) represent extra combinations.

w	x_9	x_{10}	x_{11}	x_{12}	w	x_9	x_{10}	x_{11}	x_{12}
0	1	1	1	1	3	1	1	1	0
1	0	1	1	1	4	1	0	1	0
2	1	0	1	1	4	1	0	0	1
3	1	1	0	1	4	0	1	1	0
4	1	1	0	0	4	0	1	0	1
5	0	0	1	0	4	0	0	1	1
6	0	1	0	0	5	0	0	0	1
7	1	0	0	0					
8	0	0	0	0					

Table 3.5

A variant of encoding having a simple arithmetic function of conversion into the positional binary notation is illustrated in Table 3.4. The lower seven rows in this table cover 254 extra combinations, thus making the total number of 510 working combinations in this CI. For representing 510 binary numbers with decimal equivalents from 0 to 509, we need 9 bits of the positional notation a_1-a_9 . The conversion of the latter into CI combinations is carried out in the following way:

- numbers from 0 to 255, for which $a_9 = 0$, are represented in x_1-x_{12} in such a way that $x_i \neq a_i$, $1 \leq i \leq 8$, and the identifiers x_9-x_{12} correspond to the first 9 rows in Table 3.4; These are compulsory combinations;
- numbers from 256 to 509, for which $a_9 = 1$, are given in x_1-x_{12} in such a way that the values of x_1-x_8 are equal to the values of the bits in the number

$$N = a_1 2^0 + a_2 2^1 + \dots + a_8 2^7 + 1,$$

and the identifiers x_9-x_{12} correspond to the lower 7 rows of Table 3.4; these are extra combinations.

The inverse conversion of combinations of CI to the positional binary notation is obvious.

3.6 Optimally balanced code

In this section, we discuss another self-synchronizing code that has, as will be shown later, the least redundancy among SSCs.

Coding systems in which working symbols are encoded by the combinations which have the same given number of 1's (*weight*) are called *balanced*.

In a set of binary combinations with s bits we can pick out a maximum of C_s^k combinations, each of which has k 1's, $0 \leq k \leq s$. Supposed we need to encode N different working symbols. Then the length s of an encoding word needed for their representation by a balanced code with a given weight k should be such that

$$C_s^k \geq N.$$

Recall that $C_s^{\lfloor s/2 \rfloor} = C_s^{[s/2]}$. therefore, we shall use the denotation

$$C_s^{s/2} = C_s^{\lfloor s/2 \rfloor} = C_s^{[s/2]}.$$

Furthermore, for a fixed s

$$C_s^{s/2} = \max_k C_s^k, \quad 0 \leq k \leq s \tag{3.3}$$

For encoding N symbols we choose a length m of a code combination such that the following must hold

$$C_{m-1}^{(m-1)/2} < N \leq C_m^{m/2} \tag{3.4}$$

A balanced code with a weight $k = m/2$ where m satisfies (3.4) will be called an *optimally balanced code* (OBC).

According to (3.4) for $N = 2^8 = 256$ (representing a byte) we shall need an OBC with $m = 11$ where code combinations will contain 5 (or 6) 1's because

$$C_{11}^5 = 462 > 256 > C_{10}^5 = 252.$$

With such an OBC it is possible to encode from 253 to 462 working symbols. Its name, the OBC, comes from the following theorem.

THEOREM 3.3. *An optimally balanced code has a minimal length over all code systems that use the combinations which are incomparable to each other for coding working symbols.*

Proof. 1. For balanced codes, the result follows from (3.3).

2. Consider the case of non-balanced codes, i.e. a set of code combinations in a system that contains combinations with a varying number of 1's. Assume that x combinations with a weight $[m/2] - p$, $0 < p \leq [m/2]$, are used as encoding combinations. Denote $u = [m/2]$ and $v =]m/2[$. Each of these combinations has $v + p$ adjacent combinations with the weight $u - p + 1$ (we shall say: has $v + p$ connections with combinations having the weight $u - p + 1$; x combinations with weight $u - p$ have $x(v + p)$ such connections.

On the other hand, each of the combinations with the weight $u - p + 1$ has $u - p + 1$ adjacent combinations with the weight $u - p$; hence, y combinations with the weight $u - p + 1$ will have $y(u - p + 1)$ connections with combinations of weight $u - p$. Thus, x combinations with weight $u - p$ are connected with no less than x^* combinations of weight $u - p + 1$. x^* is defined by the expression

$$x(v + p) = x^*(u - p + 1)$$

from which

$$x^* = x \frac{v + p}{u - p + 1}$$

Bearing in mind that $v \geq u$, we have allowed for values of p that $x^* > x$ holds. This means that instead of using x combinations with the weight $u - p$, we can use a greater number of code combinations (x^*) with the weight $u - p + 1$. In the same way, we may show that x^* combinations with the weight $u - p + 1$ can be replaced with x^{**} ($x^{**} > x^*$) combinations of weight $u - p + 2$ etc. Thus, it is obvious that a highest number of code combinations can be achieved if their weight will be equal to $[m/2]$.

Assume now that x combinations, with the weight $]m/2[+ p$, $0 \leq p \leq]m/2[$, are used for encoding. Applying the same reasoning to them, we may deduce that, since $v \geq u$, then for allowed p $x^* > x$ will hold.

From the latter, we may also see that the number of code combinations will be largest if they all have the weight $]m/2[$. Since, for an even m , $]m/2[= [m/2] = m/2$, then for this case the minimal is a balanced encoding with $k = m/2$.

For an odd m , a non-balanced encoding will also take place when the weights u and v are used together. However, it can be seen that instead of using x combinations of weight u we can use, equally well, the number of combinations of weight v , and vice versa (this follows from calculating the number of connections between combinations). Thus, the balanced encoding with either the weight u or v will, at least, not be worse than a non-balanced encoding by code combinations of weights u and v . *Q.e.d.*

THEOREM 3.4. *A balanced code (including OBC) is a self-synchronizing code if every transition is regular.*

The proof of the theorem follows from the incomparability of any pair of working combinations in a balanced code.

3.7 On the code redundancy

Consider the redundancies of a DRC, a CI and an OBC and compare them for the case when the information interchange system is arranged to transmit any positional binary n -tuple. For this we consider SSCs of lengths t (for DRC), q (for CI) and r (for OBC).

The *redundancy of a code system* is commonly expressed by the equation

$$R = 1 - \frac{\log_2 N_0}{\log_2 N}$$

where N_0 is the number of combinations used in the coding system and N is the number of possible binary combinations.

Thus for DRC $t = 2n$,

$$R_{DRC} = 1 - \frac{\log_2 2^n}{\log_2 2^{2n}} = \frac{2n-n}{2n} = 0.5$$

Values of t and R_{DRC} for $2 \leq n \leq 16$ are given in columns 2 and 3 in Table 3.6.

For CI $q = n + \lceil \log_2(n+1) \rceil$,

$$R_{CI} = 1 - \frac{\log_2 2^n}{\log_2 2^{n + \lceil \log_2(n+1) \rceil}} = \frac{\lceil \log_2(n+1) \rceil}{n + \lceil \log_2(n+1) \rceil} = \frac{q-n}{q}$$

Values of q and R_{CI} for $2 \leq n \leq 16$ are given in columns 4 and 5 of Table 3.6.

n	t	R_{DRC}	q	R_{CI}	r	R_{OBC}
1	2	3	4	5	6	7
2	4		4	0.500	4	0.500
3	6		5	0.400	5	0.400
4	8		7	0.429	6	0.333
5	10		8	0.375	7	0.286
6	12		9	0.333	8	0.250
7	14		10	0.300	10	0.300
8	16		12	0.333	11	0.273
9	18	0.5	13	0.308	12	0.250
10	20		14	0.286	13	0.231
11	22		15	0.267	14	0.214
12	24		16	0.250	15	0.200
13	26		17	0.235	16	0.188
14	28		18	0.222	17	0.176
15	30		19	0.211	18	0.167
16	32		21	0.238	19	0.158

Table 3.6

From (3.4) it follows that the length r , when an OBC is used, for a fixed n should satisfy

$$C_{r-1}^{(r-1)/2} < 2^n \leq C_r^{r/2}$$

or

$$\log_2 C_{r-1}^{(r-1)/2} < n \leq \log_2 C_r^{r/2} \quad (3.5)$$

For $2 \leq n \leq 16$, the values which satisfy (3.5) are presented in column 6 in Table 3.6.

It is obvious that for OBC

$$R_{OBC} = 1 - \frac{\log_2 2^n}{\log_2 2^r} = \frac{r-n}{r} \quad (3.6)$$

The corresponding values of R_{OBC} are given in column 7 in Table 3.6.

For large n an adequate approximation can be obtained using the Stirling formula

$$r! = \sqrt{(2\pi)} r^{r+0.5} e^{-r}$$

from which

$$C_{r-1}^{(r-1)/2} \approx 2^r/\sqrt{2\pi(r-1)}, \quad C_r^{r/2} \approx 2^{r+1}/\sqrt{2\pi r} \quad (3.7)$$

From (3.6) using (3.5), we have

$$\frac{\log_2 \sqrt{2\pi(r-1)}}{r} < R_{OBC} \leq \frac{\log_2 \sqrt{2\pi r} - 1}{r}$$

Substituting (3.7) into (3.5) gives

$$r - \log_2 \sqrt{2\pi(r-1)} < n \leq r - \log_2 \sqrt{2\pi r} + 1 \quad (3.8)$$

It can be seen from Table 3.6 that for $2 \leq n \leq 16$, we have $r \leq q \leq t$ and $R_{OBC} \leq R_{CI} \leq R_{DRC}$. ($r = q = t$ only when $n = 2$, and, in addition, $r = q$ when $n = 3$ or 7 .)

From Theorem 3.3 it follows that $r \leq q$ and $R_{OBC} \leq R_{CI}$ for any n . The inequalities (3.8) define the relationship between r and n in an implicit way. It can be seen from Table 3.6 that for $2 \leq n \leq 16$

$$r \leq n + [\log_2 n].$$

Thus, “the extra cost” of using an SSC for representing n -tuples will not exceed (for OBC) $\log_2 n$.

3.8 Differential encoding

A disadvantage of the above three code systems – DRC, CI and OBC – is that the application of working combinations must alternate with that of a spacer. When using SSC for data transmission in communication channels with considerable wire delays, it is desirable to exclude the transmission of combinations corresponding to empty symbols. Such an exclusion of a spacer will be possible, if we use the so-called *differential encoding*. (In the original text it is called “*encoding by changes*” - A.Y.) In this case, the data transmitted in the channel consists of the values of signal changes, rather than of signal values themselves. This method is often used in serial data transmission. The absence (presence) of a change in the signal value is interpreted as transmission of a bit with value 0 (1).

We address ourselves towards using the idea in parallel data transmission. In the transition from one combination to another, an independent change of signal values in parallel lines is allowed. Consider the transmission of an s -bit code combination on s lines using a differential encoding. Assume that, at a given

moment, a combination $a_1 = a_{11} \dots a_{s1}$ is established on the lines, and we need to transmit a working combination $r = r_1 \dots r_s$. In order to do so, we apply, on the lines, the next combination $a_2 = a_{12} \dots a_{s2}$ such that $a_{i2} = a_{i1} \oplus r_i$, $1 \leq i \leq s$, where \oplus is the modulo 2 sum sign. The reconstruction of the combination r in the receiver, which has a special unit, called a receiver converter, is performed by using the expression $r_i = a_{i1} \oplus a_{i2}$, $1 \leq i \leq s$. From this, one may deduce that the receiver should have the ability to store the previous combination a_1 . After the reconstruction of r , the outputs of the receiver converter, denoted by y_i , become such that $y_i = r_i$ for $1 \leq i \leq s$. After the acceptance of that value, and the preparation for the reception of the next combination, the receiver substitutes the combination a_2 for a_1 in its memory. This results in the establishment of a spacer, say e , on the outputs y_i . Thus, any transition $a_k - a_l$ on the input lines of the sender causes a sequence of two sub-transitions $e - r - e$ (a two-phase discipline with a spacer) on the outputs y_i of the receiver converter.

One should note the following.

1. The working combinations denoted by r which are converted into a differential notation can be represented in DRC, CI or OBC notation
2. Any code combination subject to the transmission may be associated with a number of differential code values. Thus, it is essential that the previously received value is stored to elicit the present working symbol.
3. Generally speaking, the differential notation is not an SSC.⁽¹⁾ However, it can be re-configured to an SSC by increasing the length of the code combinations (e.g. by including in a set of encoding variables those variables which model the memory facilities). This makes the differential notation suitable for designing aperiodic circuits.

The central point in this chapter is the notion of a self-synchronizing code (SSC) defined by Definition 3.7. As soon as we have assumed that, during the process of changing one code value by another, different variables change their values with different speeds, we immediately arrive at the fact that among all code systems, only SSCs allow the indication of the completion of a transmission independently of its duration. It is, thus, a fundamental advantage of SSCs that they help to organize a speed-independent process of data transfer and processing. In the subsequent chapters, namely Chapters 4, 5 and 7, the SSCs of different types (direct transition codes, double-rail codes, codes with identifiers and optimally balanced codes) are used for encoding signals in aperiodic circuits and interfaces.

⁽¹⁾ SSC's can obviously be generalized to contain the differential encoding; however, for the sake of clarity, we avoided that in this text.

3.9 Reference notations

The possibility of using codes with equal weights (balanced codes) in the design of circuits with unbounded delays was first shown in [195]. Unfortunately, this work did not indicate that the simplest sub-class of balanced codes is that of double-rail codes which are extremely suitable for implementation. A double-rail representation was studied in [193] and, further used in [5] and in [173], where the corresponding codes were called auto-synchronous. Similar code systems have been used in works on technical diagnostics (see Chapter 7 for details). Codes with identifiers are also called Berger codes [200].

The result of Theorem 3.3, or more strictly, the fact that the maximal number of incomparable code combinations can be achieved in an optimally balanced code was, most probably, first reported in [122]; the proof was based on [73]. The attention of the authors was drawn to this by J.L. Sagalovich.

The concept of double-rail logic was developed in [94] and [95]. However, the utilization of both transient states (rather than one such state) for diagnostic purposes was an obstacle to the use of this concept in hardware implementation and fault detection in asynchronous circuits.

We followed the traditional way in introducing the redundancy of a code system – see, for example, [73].

The direct-transition encoding (Section 3.6) utilizes a modification of the Tracey method [304], also described in [123]. Results on the construction of completely separating code systems can also be found in [150].

The material in this chapter has been partly published in [8], [123] and [129].

CHAPTER 4

APERIODIC CIRCUITS

‘What is the distance between Haleb and Damascus?’ they asked a stranger.

He said: ‘Twelve ... days journey - six there and six back.’

Abu'l-Faraj (1226 - 1286)

In this chapter our discussion turns to a number of approaches to the design of circuits whose operation always remains correct independent of the delays in their elements. It has already become a tradition that such circuits, both combinational and sequential, are called *aperiodic* or *self-timed* circuits.

Particular emphasis, in this chapter, is laid on questions related to the implementation of a finite state machine model; other models of self-timed circuits will be considered in later chapters.

The interaction between a state machine and the environment is organized through a “request-acknowledge” discipline, and the implementation of the machine, i.e. the circuit is such that the correctness of the circuit behaviour is completely insensitive to the values of gate delays. The delay-insensitivity requirement does not allow the use of the results of classical automata theory in the design of such a circuit. This is because of one obvious reason – if the upper bound of the delays is unknown, then there will be no sense in applying the structural models of synchronous or asynchronous state machines. An aperiodic circuit originally defined by the model of a finite state machine must operate in a specified way under arbitrary ratios of gate delay values. It is no longer necessary to establish any upper bound on these values, i.e. it is sufficient to assume them to be finite. This characteristic is the major advantage of aperiodic circuits over their traditional counterparts. The self-timed design way allows us to build circuits that operate under real gate delays and, thereby, to make maximum use of their speed capacity, which none of the other approaches are able to provide.

It is often the case that the term “aperiodic implementation of a finite state machine” is replaced by a shorter, and hence less precise, term “*aperiodic state machine*” or “*aperiodic automaton*”, which will be defined in the final section of this chapter. This definition requires that the two main features of such a machine are:

- 1) the “request-acknowledge” interaction with the environment, and
- 2) independence of the implementation behaviour from gate delays.

These requirements are likely to reflect the structural rather than the abstract model of the machine. They are akin to those of the Huffman model of an asynchronous

machine: a new state transition is allowed only if the machine has reached the stable state. It is assumed that the aperiodic machine is able to indicate the moment when the transition is completed, thereby, giving a special sign to the environment.

4.1 Two-phase implementation of finite state machine

Certain restrictions should be imposed upon the model of a state machine. They are:

1. The operation of a machine is performed in two phases that will be called the *working phase* and the *idle phase*.
2. The symbols in the input alphabet X and the output alphabet Y of a machine are encoded by the working SSC combinations, (see Section 3.3), X^i , $1 \leq i \leq n$, and Y^j , $1 \leq j \leq m$, which belong to sets X' and Y' , respectively. It is assumed that, apart from the SSC combinations in X' (Y'), the code also includes an additional combination, the so-called spacer, $s_X(s_Y)$ that corresponds to the void symbol of the input (output) alphabet.
3. The *reposition* of an operational phase is done by the environment through replacing the input spacer s_X with a working combination from X' (the initiation of the working phase), or the other way round, by changing the working combination from X' with the input spacer s_X (the initiation of the idle phase). Thus, the allowed sequences of input combinations can be expressed in the form $s_X X^1 s_X X^2 \dots$.
4. The set of internal states Z consists of two subsets – Z_1 (*working states*) and Z_2 (*idle states*), $Z_1 \cup Z_2 = Z$.
5. The *state transition function* takes the following form

$$\lambda : \{s_X\} \times Z_1 \rightarrow Z_2, \quad X' \times Z_2 \rightarrow Z_1.$$

6. The *output function* is as follows

$$\delta : \{s_X\} \times Z_2 \rightarrow \{s_Y\}, \quad X' \times Z_1 \rightarrow Y'.$$

From these restrictions, we can deduce that the allowed output sequences take the form $s_Y Y' s_Y Y^2 \dots$

7. The *environment* interacting with the machine is organized in the same way, i.e. all of the above items from 1 to 6 will hold for it, except that its input and output alphabets are Y and X , respectively. The appearance of a spacer s_Y at the input is followed by the production of a working combination from X' , and after the receipt of a working combination from Y' , the spacer s_X is generated.
8. The *assumption of the character of delays*:
 - a) the values of gate delays are unpredictable; the only constraint imposed on them is that they are finite;
 - b) the gate delays can be both inertial and perfect;

- c) the delays of non-branching wires are referred to the gate delays of those elements to whose output they are connected (i.e. if the gate delay is equal to D and the wire delay is d , then, referring the latter to the former, results in the total gate delay of $D + d$ and the wire delay can thus be assumed to be zero). Hence, the delays of such wires can be assumed to be arbitrary, whereas the delays of after-the-branch wires, being referred to the inputs of the elements to which they are connected, must be such that the spread in their values is less than the minimum delay of these elements;
- d) the AND assemblies of AND-OR-NOT elements are non-delaying, i.e. all internal delays of these elements are referred to their output inverter.

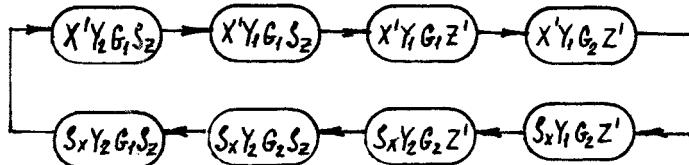
9. An *additional requirement for the implementation is that the usage of special (built-in) scaled delay elements is not allowed.*

DEFINITION 4.1. The implementation of a finite state machine which satisfies the restrictions from 1 to 9 is called the *two-phase implementation*.

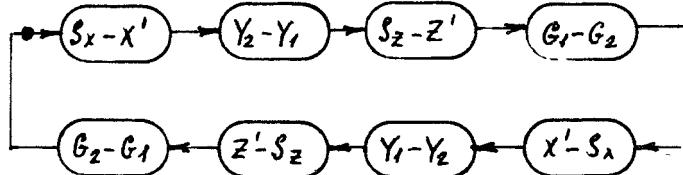
The joint operation of the two-phase implementation of a state machine and the environment can be characterized as follows. Let, at the beginning, the machine be in the idle state from set Z_2 and produce the spacer s_Y at its output. Feeding the machine with a working combination from set X' (request) makes the machine go to the working state from Z_1 , and produce an output combination from set Y' (response). This causes a transition in the environment that will result in the application of the spacer s_X (request removal) at the machine's inputs. The substitution of the spacer s_X for a working combination from X' forces the machine to return to the idle state from Z_2 and results in the appearance of spacer s_Y (response removal) at the machine's outputs. The change of an output working combination to spacer s_Y leads to a transition in the environment which can further establish a combination from X' (a new request), and the process will recur.

A set of working and idle states of a machine that represents the environment will be denoted as G_1 and G_2 , respectively. The situations of an asynchronous process representing the joint operation of the machine and the environment are defined by quadruples of the following components:

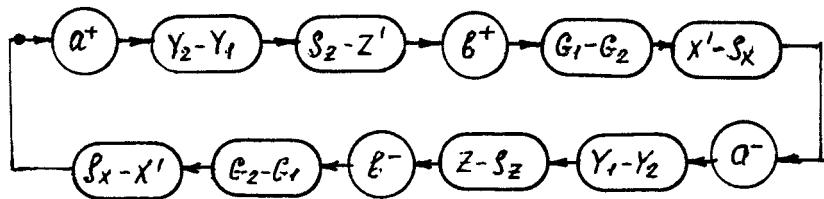
- 1) the input combination of the machine (the output combination of the environment) from the set $\{s_X\} \cup X'$,
- 2) the state of the machine from $Z_1 \cup Z_2$,
- 3) the state of the environment from $G_1 \cup G_2$,
- 4) the output combination of the machine (the environment's input combination) from $\{s_Y\} \cup Y'$.



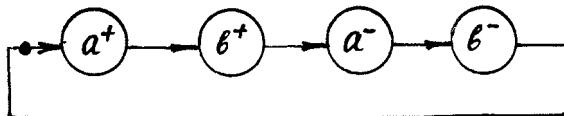
a



b



c



d

Figure 4.1 (a) - (d). Description of the interaction between a machine and the environment: (a) a structured asynchronous process, (b) a signal graph, for the case of two-phase implementation, (c) a signal graph, (d) a signal graph, at the control signal level, for the case of matched implementation

The corresponding autonomous (sequential) asynchronous process for the system "machine-environment" is shown in Fig. 4.1(a), whereas, in the signal graph notation, it is shown in Fig. 4.1(b), which shows that the change of the environment's output combination leads to the change of the machine's output combination etc.

We should make an additional refinement relating to the need for storing the previous state code during the idle phase. In order to operate correctly, the machine must have a two-stage memory. During the working phase, one of the registers (stages) produces the internal state code for which the other stage (buffer) must store the previous state codes. The combination of states of both registers is encapsulated in the machine's working state belonging to the set Z_1 . During the idle phase, the state code of the first register is rewritten into the buffer, and the register is either loaded with a spacer, or is held in the same state as in the working phase. Such a combination of states of both registers now forms the idle code belonging to the set Z_2 .

Code combinations belonging to $Z_1 \cup Z_2$ should be self-synchronizing. The reader will find examples in Section 4.5.

4.1.1. MATCHED IMPLEMENTATION

The two-phase model can be further refined in the following way.

A special binary signal, called phase signal (denoted by a) is picked out in order to mark every initiation of operational phases, i.e. events relating to the change of spacer s_X to a combination in X' and vice versa. A special type of device that indicates the moment of the transition completion (the change of combination class) will be called an *indicator*. It would be quite natural to place the indicator into the environment. In order to initiate a transition in the environment, the machine should also have its own indicator. The major role of this indicator is to produce the signal b which will mark the moment of substitution of a working combination from Y' for the spacer S_Y and vice versa.

DEFINITION 4.2. The two-phase implementation of a finite state machine with explicit binary phase signals, the request signal and the acknowledge signal, is called the *matched* (or compliant) implementation.

When $a = 1$ (request) the working phase begins. The end (transition process completion) of the phase is manifested by the machine's indicator signal. This signal, $b = 1$ (acknowledge) initiates the transition in the environment after which $a = 0$ (request removal). The latter initiates the transition in the machine which goes to the idle phase $b = 0$ (acknowledgement removal). Then the whole process repeats.

Using signal graphs, we can represent the joint operation of the matched implementation and the environment as it is shown in Fig. 4.1(c). We also note that the changes of phase signals are always performed strictly after the changes of internal states, and the changes (if any) of the output values, in both the machine and

the environment. Bearing that rule in mind, we can obtain a more succinct signal graph description, just at the phase signal level as shown in Fig. 4.1(d).

Recall that according to the accepted denotation a^+ (request) and b^+ (acknowledge) correspond to the 0–1 transition of a and b respectively, and similarly, a^- (request removal) and b^- (acknowledgement removal) correspond to the 1–0 transition of a and b .

4.2 Completion indicators and checkers

Later, in this text, a variety of indicators will be used in different aperiodic implementations. The *indicator* is the most important accessory in any matched implementation. The arrival of a new signal value at its output indicates that a corresponding transition phase has been completed. Most of the material in this chapter is related to the methodology for the design of aperiodic circuits with a special emphasis on indicators.

We distinguish a sub-class of indicators, called checkers, which are intended to indicate that a code combination belongs either to an SSC or to spacers. Thus a *checker* is a device having, say, s inputs x_1, \dots, x_s and one output y . When a certain combination of the SSC is applied to its inputs, the output signal must be equal to one value, say, 0, while applying the spacer should result in producing the opposite value (1) of the output signal.

The simplest case of a checker design can be one which is based on the assumption that each of two classes consists of only one combination. Let one of the classes consist of the spacer i , the all-zero code value, and the other class consists of the spacer e , the all-one code value. Such an indicator will be called a *spacer checker*. Its functioning is described by the signal graph shown in Fig. 4.2(a). Signal y goes from 0 to 1 (from 1 to 0) only after the spacer e (i) is applied to its inputs. Signal y remains in its previous state for any intermediate value on its inputs. The behaviour of the spacer checker satisfies the following expression

$$y = \bigwedge_{j=1}^s x_j \vee y (\bigvee_{j=1}^s x_j). \quad (4.1)$$

This function is implemented by the so-called *hysteresis flip-flop* (Γ -*flip-flop*) built of an AND-OR-NOT gate and an inverter as shown in Fig. 4.2(b). The symbolic designation for a Γ -flip-flop is given in Fig. 4.2(c). The joint behaviour of the Γ -flip-flop and the environment whose model is a group of s inverters (see Fig. 4.2(d), for $s = 2$) can be represented by the signal graph shown in Fig. 4.2(e), which is, in general, semantically equivalent to that of Fig. 4.2(a).

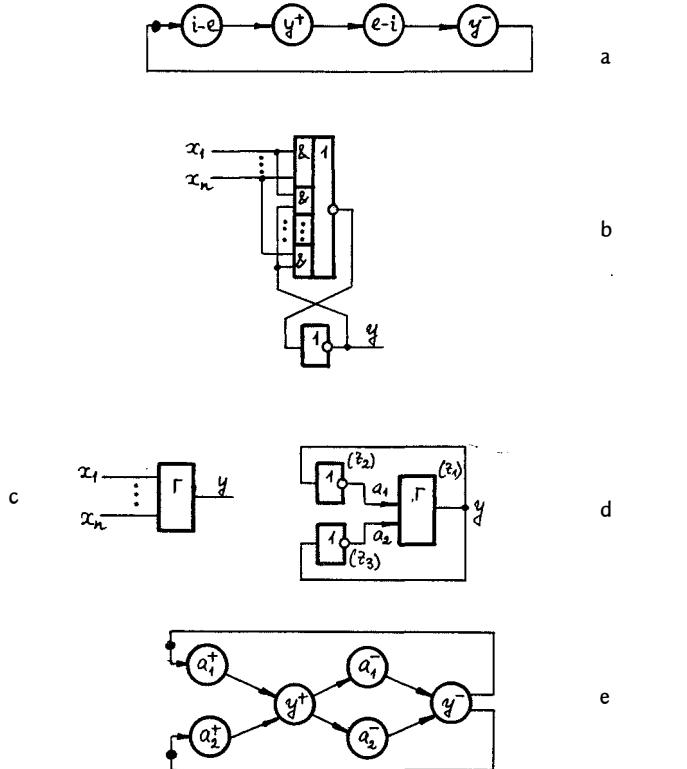


Figure 4.2 (a) - (e). A checker for single-rail signals: (a) a signal graph; (b) implementation in the form of a Γ -flip-flop; (c) a Γ -flip-flop symbol; illustration of the interaction between a Γ -flip-flop and the environment by a circuit (d), and a signal graph (e).

Assume that coding combinations are represented by the double-rail code (DRC) $x_1 \dot{x}_1 \dots x_s \dot{x}_s$ (the denotation \dot{x}_j stands for the DRC with the all-one spacer e as has been agreed in Section 3.4). Then it is clear that the behaviour of the *DRC checker* can be defined by the following equation

$$y = \bigwedge_{j=1}^s x_j \dot{x}_j \vee y \left(\bigvee_{j=1}^s x_j \dot{x}_j \right). \quad (4.2)$$

The implementation of the DRC checker (Fig. 4.3(a)) is a modification of the Γ -flip-flop; the symbolic notation for it is shown in Fig. 4.3(b). In the case of $s = 1$ (one-bit double-rail code signal) the equation (4.2) degenerates into $y = x_1 \dot{x}_1$ and thus can be implemented as an AND gate.

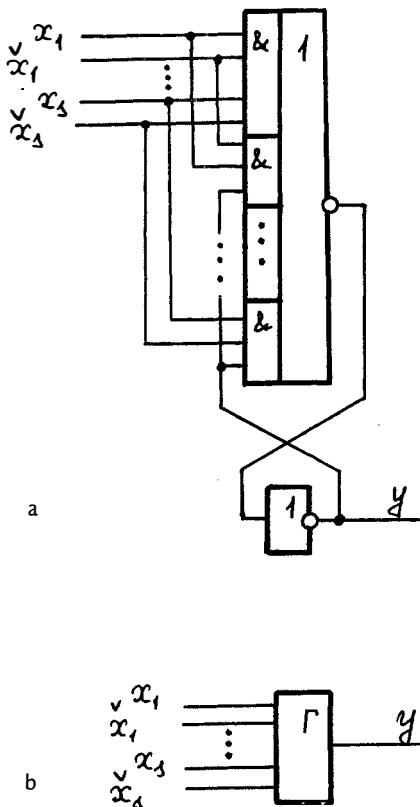


Figure 4.3 (a) and (b). (a) a double-rail code checker ; (b) its symbol.

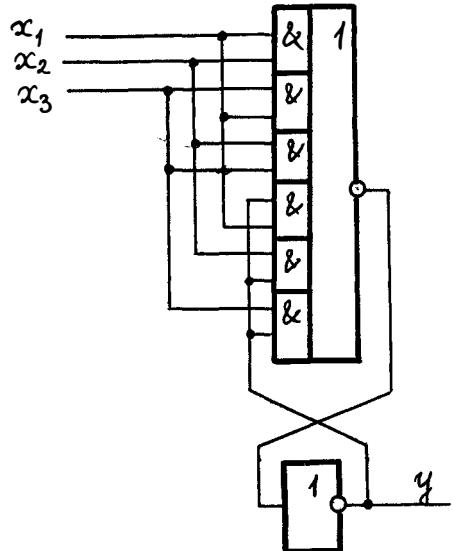


Figure 4.4. A checker of an optimally balanced code for $s = 3$.

A *checker for the optimally balanced code* (OBC) is defined by the equation

$$y = \text{maj}(x_1, \dots, x_s) \vee y \left(\bigvee_{j=1}^s x_j \right) \quad (4.3)$$

where $\text{maj}(x_1, \dots, x_s)$ is a majority function consisting of $C_s^{s/2}$ terms of the rank $[s/2]$. The implementation of the OBC checker for $s = 3$ is given in Fig. 4.4.

A *checker for the code with identifier* (CI) is most difficult to implement. In order to recognize that a combination belongs to the CI it is required:

- 1) to determine a w -class of the combination,
- 2) to check the correspondence of the identifier to that class,

3) to indicate that the CI combination belongs either to the compulsory set, or to the auxiliary set, of combinations, and then to produce the checker output signal.

The part of the checker which determines the w -class does actually recognize the value k of the number of 0's in the basic bit positions x_1, \dots, x_n . Using the value of k it is possible to obtain the corresponding w -class: $w = n - k$. For the case $n = 8$, a two-stage circuit can be used for that task. The first stage consists of two identical sub-circuits Σ and \mathcal{H} with the inputs x_1-x_4 and x_5-x_8 and outputs e_1-e_4 and h_1-h_4 , respectively.

Sub-circuit Σ finds the number of 0's in the bit positions x_1-x_4 of the CI combination. Applying the all-one spacer e on its input results in establishing $e_1 = \dots = e_4 = 0$ on the outputs. If exactly one of the inputs x_1-x_4 is equal to 0, then $e_1 = 1$. Similarly, if exactly two (three or four) inputs are equal to 0, then $e_1 = e_2 = 1$ ($e_1 = e_2 = e_3 = 1$ or $e_1 = \dots = e_4 = 1$). The system of Boolean functions for circuit Σ is as follows:

$$\begin{aligned} e_1 &= \overline{x_1 x_2 x_3 x_4}, \\ e_2 &= \overline{\overline{x_1 x_2 x_3} \vee \overline{x_1 x_2 x_4} \vee \overline{x_1 x_3 x_4} \vee \overline{x_2 x_3 x_4}}, \\ e_3 &= \overline{x_1 x_2 \vee x_1 x_3 \vee x_1 x_4 \vee x_2 x_3 \vee x_2 x_4 \vee x_3 x_4}, \\ e_4 &= \overline{x_1 \vee x_2 \vee x_3 \vee x_4}. \end{aligned}$$

As the sub-circuit \mathcal{H} operates in the same as Σ the corresponding system of functions h_1-h_4 for \mathcal{H} can be derived by the straightforward substitution of x_j , $1 \leq j \leq 4$, by x_{j+4} .

The second stage realizes a system of functions u_1-u_8 in the following form

$$\begin{aligned} u_1 &= \overline{e_1 \vee h_1}, \quad u_2 = \overline{e_2 \vee h_2 \vee e_1 h_1}, \\ u_3 &= \overline{e_3 \vee h_3 \vee e_1 h_1 h_2 \vee e_1 e_2 h_1}, \\ u_4 &= \overline{e_4 \vee h_4 \vee e_1 e_2 e_3 h_1 \vee e_1 e_2 h_1 h_2 \vee e_1 h_1 h_2 h_3}, \\ u_5 &= \overline{e_1 e_2 e_3 e_4 h_1 \vee e_1 e_2 e_3 h_1 h_2 \vee e_1 e_2 h_1 h_2 h_3 \vee e_1 h_1 h_2 h_3 h_4}, \\ u_6 &= \overline{e_2 e_3 e_4 h_2 \vee e_2 e_3 h_2 h_3 \vee e_2 h_2 h_3 h_4}, \\ u_7 &= \overline{e_3 e_4 h_3 \vee e_3 h_3 h_4}, \quad u_8 = \overline{e_4 h_4}. \end{aligned} \tag{4.4}$$

This system contains redundant variables. They are necessary for the correct implementation of aperiodic combinational circuits (see Section 4.3).

The circuit checking the correspondence between identifiers and w -classes (again for $n = 8$) consists of four stages.

The first stage implements the system

$$\begin{aligned}
 g_1 &= \overline{x_9 \vee x_{10} \vee x_{11} \vee x_{12}}, \\
 g_2 &= \overline{u_2x_9 \vee u_3x_{10} \vee u_4x_9x_{10} \vee u_5x_{11} \vee u_6x_9x_{11} \vee u_7x_{10}x_{11} \vee u_8x_9x_{10}x_{11}}, \\
 g_3 &= \overline{u_1 \vee u_2x_{10} \vee u_2x_{11} \vee u_3x_{11} \vee u_4x_{11} \vee u_8x_{10}x_{11}}, \\
 g_4 &= \overline{u_3x_9x_{11} \vee u_5x_9x_{11} \vee u_4x_{10}x_{11} \vee u_7x_9x_{10} \vee x_{12}}, \\
 g_5 &= \overline{u_7u_8x_9x_{10}x_{11}}, \quad g_6 = \overline{u_1u_2u_3u_4u_5u_6}.
 \end{aligned} \tag{4.5}$$

The second stage produces the signals a_1 , a_2 and a_3 in the form

$$\begin{aligned}
 \bar{a}_1 &= \overline{g_1g_2g_4g_5 \vee g_2g_3g_5g_6}, \\
 \bar{a}_2 &= \overline{g_3g_4g_5g_6}, \quad \bar{a}_3 = \overline{g_1 \vee g_2 \vee g_3 \vee g_5 \vee g_6}.
 \end{aligned}$$

The two remaining stages present the spacer checker which is defined by the following equation

$$\bar{y} = \bar{a}_1\bar{a}_2\bar{a}_3 \vee \bar{y}(\bar{a}_1\bar{a}_2 \vee \bar{a}_3)$$

(this can be obtained from (4.1) using $a_1a_2 = 0$).

After applying the spacer e ($x_1 = \dots = x_{12} = 1$) the outputs of elements are as follows

$$e_i = h_i = g_l = 0 \quad (1 \leq i \leq 4, 1 \leq l \leq 6)$$

$$u_j = \bar{a}_k = 1 \quad (1 \leq i \leq 8, 1 \leq k \leq 3)$$

When applying the compulsory (auxiliary) combination, and after completion of the transition process in the checker, we have $\bar{a}_1 = 0$ ($\bar{a}_2 = 0$), $\bar{a}_3 = 0$ and then $\bar{y} = 0$.

In designing multi-input checkers, we should take into account that, for technologically available ICs, the number of inputs of AND and OR (AND-OR) elements are always limited. The practical technique to obviate this problem is to build a pyramid circuit of checkers with a limited number of inputs.

STATEMENT 4.1. *Any pyramid network of spacer checkers is itself a spacer checker.*

The proof is trivial: the output of the next stage checker cannot change until the states of all the checkers in the previous stage have changed.

This statement shows how to construct checkers of elements with a limited number of inputs, provided that the following conditions hold:

- 1) a set of checked inputs can be partitioned into sub-sets each of which is checkable with a checker built of elements with a given limit on the number of inputs, and
- 2) the checker of the outputs of the checkers built in this way is a checker of single-rail signals.

The best practical results are obtained by using another technique, the so-called *parallel compression*, which is discussed further in Example 4.2.

The above designs of checkers are fundamental in the design of indicators. Other recommendations on the implementation of indicators become clear in the following discussion.

4.3 Synthesis of combinational circuits

One of the crucial steps in the structured synthesis of two-phase implementations is the design of combinational circuits, by which, basically, we shall mean circuits realizing systems of Boolean functions, perhaps with some “local” feedback. We presume that such circuits satisfy Definition 4.1 with the restriction that the set Z of internal states (see item 4) is empty and the state transition function (item 5) is not defined, whereas the output function (item 6) becomes $\{s_X\} \rightarrow \{s_Y\}$, $X' \rightarrow Y'$.

These circuits fall into a category of *aperiodic circuits* because their behaviour is insensitive to the gate delay values. In the following sections of Chapter 4, we shall deal with *aperiodic combinational circuits based on the two-phase implementation strategy*, and, unless it has been noted otherwise, we shall discuss only these circuits.

The problem of how to synthesize an aperiodic combinational circuit can be formulated in the following way.

For a given system of Boolean functions

$$\phi_q = \phi_q(x_1, \dots, x_h), \quad 1 \leq q \leq g$$

an aperiodic combinational circuit can be constructed. It is obvious that such a circuit is likely to be redundant with respect to a given system because of the conventions of

Chapter 3. These are on the necessity of using code redundancy for input and output variables and representing combinations by an SSC. We assume that the aperiodic circuit will have n inputs $X = \{x_1, \dots, x_n\}$, $n \geq h$, and m outputs $Y = \{y_1, \dots, y_m\}$, $m \geq g$, and it will correspond to a *system of inherent functions* (SIF)

$$y_j = f_j(x_1, \dots, x_m), \quad 1 \leq j \leq m$$

or, more briefly, $Y = F(X)$. As usual, by an SIF, we mean a system written as a super-position of functions realized by all elements of an asynchronous circuit (for brevity, referred to as a *circuit*).

Let combinations of the sets \tilde{X} and \tilde{Y} (the sets of all binary combinations at the inputs and outputs of the circuit, respectively) be encoded with an SSC (see Chapter 3) in such a way that in each of them we can pick out two classes of combinations A and B , and K and L , respectively, such that $A \cup B \subset \tilde{X}$ and $K \cup L \subset \tilde{Y}$. Recall Definition 3.6 where the notion of an allowed transition was introduced. The allowed transition $a-b$ ($b-a$) between input combinations $a \in A$ and $b \in B$ initiates the process in a circuit which terminates with the transition $k-l$ ($l-k$) where $k \in K$ and $l \in L$ are output combinations. It is natural, from the interconnection viewpoint, to demand that the output transitions are, also, allowed transitions. The requirement of allowedness for both input and output transitions guarantees the absence of any non-allowed transition, and, essentially, generalizes the common notion of *functional hazards*. We also demand that aperiodic circuits are free from *logical hazards* (related to potential glitches (malfunctions) at the circuit's outputs), which is concerned with the accepted assumption about the character of element delays and wire delays.

Note that in an aperiodic circuit, with every element realizing an isotonus (antitonous) function with respect to allowed input transitions, no logical hazards can arise if, for every element, only an allowed transition is ever provided on the element's inputs.

When synthesizing aperiodic circuits, certain problems that we should tackle are demonstrated in the following example.

EXAMPLE 4.1. We show a construction of an aperiodic circuit implementing the carry function of a one-bit adder

$$P = xy \vee yp \vee xp \tag{4.6}$$

where x and y are the bits to be added (summands), and p is the carry signal from the lower bit-sum position. Input and output variables are represented by an SSC, and the two-phase discipline, with spacer i , is used. We also require that only the NAND gate basis is available.

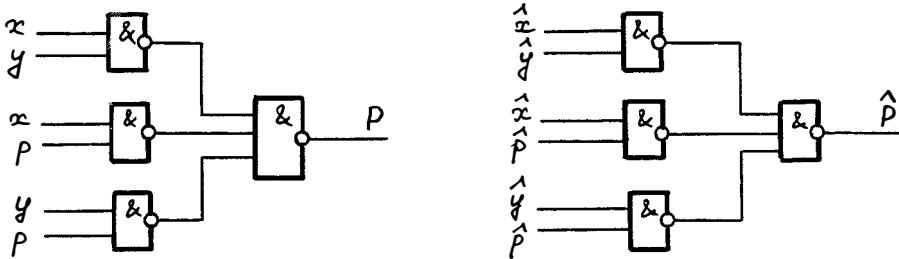


Figure 4.5. Incorrect implementation of the adder's carry function.

One of the possible solutions under the given restrictions is shown in Fig. 4.5. The circuit realizes the functions

$$P = \overline{\overline{xy} \cdot \overline{xp} \cdot \overline{yp}}, \quad \hat{P} = \overline{\overline{\hat{x}\hat{y}} \cdot \overline{\hat{x}\hat{p}} \cdot \overline{\hat{y}\hat{p}}} \quad (4.7)$$

(see Section 3.4 for conventions regarding the denotations for double-rail codes).

When the all-zero spacer $i(x = \hat{x} = y = \hat{y} = p = \hat{p} = 0)$ is applied to inputs, the output values of the elements of the first (with respect to inputs) and the second layers become equal to 1 and 0, respectively. The transition of the form “spacer-working combination” will cause a 0–1 transition of either P or \hat{P} . However, the operation of the circuit in Fig. 4.5 may be incorrect. Indeed, assume that the working combination 101010 is applied to the inputs, i.e. $x = y = p = 1$ and $\hat{x} = \hat{y} = \hat{p} = 0$, and that is followed by the 0–1 transition of P with \hat{P} remaining 0.

Let us also assume that the delay of the chain which consists of elements \overline{xy} , P and the environment is less than the delay of either of the elements \overline{xp} and \overline{yp} . These assumptions are quite justifiable because we do not impose any limitations on the values of delays in the aperiodic implementation. Thus, after $\overline{xy} = 0$ and $p = 1$ the environment may “conclude” that the transition process in the circuit has already terminated and the spacer i may be applied to the circuit's inputs.

If the transition to the spacer will change the value of input x first, i.e. $x = \hat{x} = 0$ (recall that $y = p = 1$ but the output of \overline{yp} is delayed until it becomes 0), \overline{xy} and p will become 1 and 0, respectively. According to outputs P and \hat{P} , this

will imply the completion of the idle phase. However, since $y = p = 1$, the output of \bar{y}_p may generate 0 that will force P to become 1 before a new working combination is applied to inputs. This will obviously be incorrect. To operate correctly, the circuit should be able to observe the state changes not only of its outputs, but also of the inputs and all outputs of intermediate elements.

It is clear, that in the general case, the detection of incorrectness of such a kind is a rather difficult problem. Thus, it seems necessary to search for a formal technique to check that a combinational circuit belongs to the class of aperiodic circuits. The following subsection presents an outline of such a formalism.

4.3.1. INDICATABILITY

The restrictions of Definition 4.1 appear to be insufficient for the correct construction of logical circuits that realize systems of Boolean functions. For more careful examination, we introduce the notion of indicatability which is formulated as follows.

DEFINITION 4.3. Let an allowed transition $a-b$ where $a \in A$ and $b \in B$ be applied to the inputs of a combinational circuit with SIF $Y=F(X)$. Let this transition initiate an allowed transition $k-l$ of the circuit outputs where $k \in K$ and $l \in L$. If, for any $t \in [a, b]$, the condition $F(t) \neq l$ holds, then we shall say that *the inputs X are indicated to the outputs Y for the transition a-b* (or in a briefer form, the *circuit indicates the transition a-b*). Generally, if the circuit's inputs are indicated to the outputs for all allowed transitions, we shall say that the *circuit is indicatable*, or, in other words, the *inputs X are indicated to the outputs Y*.

The idea of “*circuit indicatability*” stems from the principle that an output combination which is a member of the set L can appear as a result of a transition process in a circuit only after an input combination, a member of the set B , has been applied at the circuit inputs. Although the concept of indicatability sheds some light on the problem of circuit correctness, it is still unable to establish an explicit relationship between that property and an SIF defining the circuit. A special form of indicatability will first be examined to fully establish this relationship.

DEFINITION 4.4. Let an adjacent transition $a-b$ where $a \in N^-(b)$, $\varepsilon(a, b) = x_i$, $1 \leq i \leq n$ (or $a \in N^+(b)$, $\varepsilon(a, b) = \bar{x}_i$) be applied to the inputs of a combinational circuit with SIF $Y=F(X)$. Let this transition initiate an allowed transition $k-l$ of the circuit outputs such that the variation term $\varepsilon(k, l)$ contains y_r or \bar{y}_r , $1 \leq r \leq m$. We

shall say that, in this case, y_r translates x_l in the transition $a-b$ and denote it as $y_r = T(x_l/a-b)$.

The idea of “*translatability*” stems from the principle that a change of the value of some input signal in a circuit should always result in a change of the value of some output signal.

DEFINITION 4.5. If, for every allowed adjacent transition a^i-b where $a^i \in N^-(b)$, $\epsilon(a, b) = x_j$, $1 \leq j \leq n$ (or $a^i \in N^+(b)$, $\epsilon(a, b) = \bar{x}_j$) applied to the inputs of a combinational circuit with SIF $Y = F(X)$ there exists a variable y_r , $1 \leq r \leq m$, such that $y_r = T(x_j/a^i-b)$, then we shall say that the *inputs X are translated to the outputs Y in one step from the combination b* and denote it as $Y = T_s(X/b)$.

Further examination of indicatability is made through a series of assertions.

Recall that the variation term $\epsilon(a, b)$ for the transition $a-b$ consists of those variables in either direct (x_j) or complemented (\bar{x}_j) form which change in the given transition either from 0 to 1 or from 1 to 0, respectively. Variables that appear in the variation term in direct form will be called *increasing* variables in the transition $a-b$, and those which appear in complemented form will be called *decreasing* in $a-b$.

LEMMA 4.1. *The necessary condition for a circuit to be indicatable is that all functions in the SIF of the circuit are isotonous (antitonous) in increasing variables and antitonous (isotonous) in decreasing variables in all allowed input transitions.*

Proof. Assume, to the contrary, that there is an allowed transition $a-b$ and a function $f_i(X) \in F(X)$ such that the following cases are satisfied:

- 1) $f_i(X)$ is not monotonous in $a-b$,
- 2) $f_i(X)$ is antitonous (isotonous) in both increasing and decreasing variables in $a-b$,
- 3) there exists a set of increasing (decreasing) variables in some of which $f_i(X)$ is isotonous in $a-b$, and in the other variables $f_i(X)$ is antitonous in $a-b$.

Now we discuss these in turn.

1) If $f_i(X)$ is not monotonous, then according to Definition 3.5, there exists a variable x_v in which $f_i(X)$ is neither isotonous nor antitonous in transition $a-b$. Hence, by item d) of Definition 3.5, neither $f_i^\omega(x_v = 0) \geq f_i^\omega(x_v = 1)$ nor $f_i^\omega(x_v = 0) \leq f_i^\omega(x_v = 1)$ holds for the function $f_i^\omega(X)$ which can be obtained from $f_i(X)$ through fixing the values of those variables that appear in $\omega(a, b)$. Note that both $f_i^\omega(x_v = 0)$

and $f_i^\omega(x_v = 1)$ depend on other variables, appearing in $\varepsilon(a, b)$. The above means that in the transition $a-b$ there exists such a sequence of intermediate combinations in which $f_i(X)$ will change its value more than once. Therefore, we may conclude that the output transition $k-l$ will not be regular (see Definition 3.1).

2) In this case there exists a pair of variables x_j and x_k such that they both appear in $\varepsilon(a, b)$: x_j is increasing and x_k is decreasing in $a-b$, and both $f_i^\omega(x_j = 0) \geq f_i^\omega(x_j = 1)$ and $f_i^\omega(x_k = 0) \geq f_i^\omega(x_k = 1)$ hold. The transition $a-b$ for the pair x_j, x_k appears to be a 01–01 transition and there exists such a sequence of intermediate combinations in $[a, b]$ on which the function $f_i(X)$ changes its value more than once, i.e. the output transition $k-l$ will not be regular again.

3) There is a pair of increasing (or decreasing) variables x_j, x_k for the transition $a-b$ such that both $f_i^\omega(x_j = 0) \geq f_i^\omega(x_j = 1)$ and $f_i^\omega(x_k = 0) \geq f_i^\omega(x_k = 1)$ hold. As in the previous cases, there will exist a sequence of intermediate combinations in $[a, b]$ such that $f_i(X)$ will change its value more than once, and, hence, the transition $k-l$ will not be regular.

Thus, in all of the above cases, neither the conditions of Definition 3.6 of an allowed transition (item 3) nor those of Definition 4.2 are fulfilled which leads to contradiction. *Q.e.d.*

COROLLARY. *If every allowed input transition $a-b$ which is performed through comparable combinations is followed by the output transition $k-l$ and $a, k \in \{i, e\}$, then for a circuit to be indicatable it is necessary that*

- 1) *all the functions in SIF are isotonous (antitonous) in allowed input transitions,*
- 2) *the output transitions which are caused by allowed input transitions are performed through comparable combinations.*

LEMMA 4.2. *The necessary condition for the transitions $s-b$ ($b-s$), $b \in B$, of the two-phase discipline with spacer s to be allowed, is that the class B consists of pairwise incomparable combinations.*

Proof. For the two-phase discipline with spacer s , the latter may be followed by any combination from the set B , say, by b^k or b^l . Let $s = i$ (i is the all-zero spacer). Then, because of Definition 3.3, both transitions $i-b^k$ and $i-b^l$ must be comparable: $i < b^k$ and $i < b^l$ (i is the least value in the lattice of combinations). Assume that b^k and b^l are comparable. Due to symmetry, let $b^k < b^l$. Then $i < b^k < b^l$ and hence $b^k \in [i, b^l]$. This will contradict item 4 of Definition 3.6. Similar reasoning is

applicable for the case of the reverse transition, $b-i$, for $s-e$ and for the general case $s \neq i, e$. *Q.e.d.*

DEFINITION 4.6. The *partial derivative* $\partial f(X)/\partial x_i$, or what is also called the *Boolean difference of a function with respect to a variable x_i* is defined by the expression

$$\partial f(X)/\partial x_i = f(x_i = 1) \oplus f(x_i = 0)$$

where \oplus is the modulo 2 sum sign (exclusive-OR operator).

If the function $f(X)$ is isotonic, then the Boolean derivative can be given in the form

$$\partial f(X)/\partial x_i = \tilde{f}(x_i = 0) \tilde{f}(x_i = 1). \quad (4.8)$$

A corresponding expression can be obtained for the case where $f(X)$ is antitonic in x_i .

To obtain necessary and sufficient conditions for a circuit to be indicatable, we must first examine the translatability conditions.

For the sake of brevity, we assume, in the sequel, the following denotations.

By \tilde{x}_j we denote either x_j or \bar{x}_j , i.e. $\tilde{x}_j \in \{x_j, \bar{x}_j\}$, also for either $\varepsilon(a, b) = x_j$ or $\varepsilon(a, b) = \bar{x}_j$, we shall write $\varepsilon(a, b) = \tilde{x}_j$. Then, due to the symmetry of the Boolean derivative $\partial f(X)/\partial x_j$, which stems from $\partial f(X)/\partial \bar{x}_j = \partial f(X)/\partial x_j$, we may provide proofs, only for $\varepsilon(a, b) = x_j$. The case for $\varepsilon(a, b) = \bar{x}_j$ will thus, follow immediately.

STATEMENT 4.2. *The output y_r , $1 \leq r \leq m$, translates the input x_j , $1 \leq j \leq n$, in an adjacent transition $a-b$ such that $\varepsilon(a, b) = \tilde{x}_j$, if and only if the combination of values of variables defined by the term $\omega(a, b)$ is a root of the equation $\partial f_r(X)/\partial x_i = 1$.*

Proof. Let the conditions of Definition 4.4 be satisfied. For example, if $\varepsilon(a, b) = x_l$ then let the variation term include y_r . Hence, there exists a variable y_r such that $y_r(a) \oplus y_r(b) = 1$ for $a_l \neq b_l$, i.e. $a_l = \bar{b}_l$. Then $y_r(x_l = 0) \oplus y_r(x_l = 1) = 1$. It follows from Definition 4.6 that the above equality holds if and only if the combination of values of variables in a subset X defined by $\omega(a, b)$ is a root of the equation $\partial f_r(X)/\partial x_l = 1$. *Q.e.d.*

The next statement follows directly from the above statement and Definition 4.5.

STATEMENT 4.3. *The inputs X are translated to the outputs Y in one step from the combination b if and only if for every allowed adjacent transition $a^i - b$ the combination of values of variables defined by the term $\omega(a^i, b)$ is a root of the corresponding (j-th) equation.*

$$\bigvee_{r=1}^m \frac{\partial f_r(X)}{\partial x_j} = 1 \quad (4.9)$$

where \tilde{x}_j is a variable that constitutes the variation term $\epsilon(a^i, b)$, and m is the number of outputs in the circuit.

If all functions $f_r(X)$ are isotonous in x_j , then according to (4.8), equation(4.9) will assume the form

$$\bigvee_{r=1}^m \bar{f}_r(x_j = 0) f_r(x_j = 1) = 1 \quad (4.10)$$

This simplifies the utilization of Statements 4.1 and 4.2. The corresponding expression is easily constructed for the case in which $f(X)$ is antitonous in x_j .

EXAMPLE 4.2. Let the SIF for a circuit be

$$f_1(X) = x_1 x_2, f_2(X) = x_1 \vee x_2$$

The set $N^-(b^1)$ for the combination $b^1 = 11$ consists of combinations $a^1 = 01$ and $a^2 = 10$, and the set $N^+(b^1)$ is empty. For $b^2 = 00$, the set $N^+(b^2)$ consists of a^1 and a^2 , and $N^-(b^2) = \emptyset$.

The derivatives of functions $f_1(X)$ and $f_2(X)$ with respect to x_1 are obtained by

$$\frac{\partial f_1(X)}{\partial x_1} = f_1(x_1 = 0) \oplus f_1(x_1 = 1) = x_2, \quad \frac{\partial f_2(X)}{\partial x_1} = \tilde{x}_2.$$

In the same way, the derivatives with respect to x_2 can be obtained. Then it is easy to establish that $y_1 = T_s(X/b^1)$ if transitions from $N^-(b^1)$ to b^1 are allowed, and $y_2 = T_s(X/b^2)$ if transitions from $N^+(b^2)$ to $\{a^1, a^2\}$ are allowed.

The following theorem makes it possible to consider translatability as a special case of indicatability.

THEOREM 4.1. *Let a circuit have the two-phase discipline with a spacer $s \in \{i, e\}$ where all transitions are adjacent, i.e. for every $b \in B$ $\varepsilon(s, b) = \tilde{x}_j$, $1 \leq j \leq n$. Then the circuit is indicatable if and only if it satisfies the following three conditions:*

- 1) *all the functions in the SIF are isotonous or antitonous in allowed transitions,*
- 2) *all combinations in the class L are pairwise incomparable,*
- 3) *for any allowed transition there exists an output y_r , $1 \leq r \leq m$, which translates either x_j or \bar{x}_j .*

Proof. Without loss of generality, we consider the case in which all the functions in the SIF of the circuit are isotonous in allowed transitions and $\varepsilon(s, b) = x_j$.

a) *Necessity.* Necessity of condition 1) follows from Lemma 4.1. Necessity of condition 2) follows from the fact that the output combination $k = F(s)$ plays the role of a spacer in the two-phase output discipline and, hence, from Lemma 4.2. Necessity of condition 3) follows from Definition 4.3: for any allowed transition $s-b$ ($b-s$), $F(s) \neq F(b)$, and hence, there exists at least one variable y_r such that $y_r = T(x_j/s-b)$ for $\varepsilon(a, b) = \tilde{x}_j$.

b) *Sufficiency.* The allowedness of the input discipline follows immediately from the pre-condition of the theorem. The requirement $F(t) \neq l$ is fulfilled because of the adjacency, thus $[s, b] = \emptyset$. The output transitions are allowed because of the following reasons. If any allowed transition $s-b$ ($b-s$) there exists y_r such that $y_r = T(x_j/s-b)$, then $k \neq l$ ($F(s) \neq F(b)$). The regularity and comparability of the transition $k-l$ follows from the fact that every function in the SIF is isotonous. It then remains to be shown that for every $h \in [k, l]$ we have $h \in Y(\{k\} \vee \{l\})$. From the isotonicity of the functions, we see that $k < h < l$, and because all pairs of combinations in L are incomparable, we have $h \notin L$. All the conditions of Definition 4.3 are thus satisfied. *Q.e.d.*

The result of Theorem 4.1 can be generalized to the case of arbitrary allowed transitions, by the following theorem.

THEOREM 4.2. *Let a circuit have the two-phase discipline with a spacer $s \in \{i, e\}$. Then the circuit is indicatable if and only if it satisfies the following four conditions:*

- 1) *all the functions of the SIF are isotonous or antitonous in allowed transitions,*
- 2) *all combinations in the class B are pairwise incomparable,*
- 3) *all combinations in the class L are pairwise incomparable,*
- 4) *the inputs X are translated to the outputs Y in one step from combinations b and s.*

Proof.

a) Necessity of condition 1) follows from Lemma 4.1. Conditions 2) and 3) follow – as in the proof of Theorem 4.1 – from Lemma 4.2. Necessity of condition 4) follows from Definition 4.3: $F(t) \neq l$ for all $t \in (s, b)$, $t \neq b$, s including all t that are adjacent to b . Hence, $F(t) \neq F(b)$. According to Definition 4.5 we have $Y = T_s(X/b)$.

b) *Sufficiency.* The allowedness of the input discipline follows from the pre-condition of the theorem. The requirement $F(t) \neq l$ is fulfilled because for every $t \in [s, b]$ such that $t \neq b$, there exists a combination d such that d is adjacent to b and $d \in [t, b]$. Due to Definition 4.5 $F(d) \neq F(b)$, and, furthermore, we have $F(t) \neq F(b)$, $F(t) \neq l$, because $F(t) \leq F(d) \leq F(b)$ (without loss of generality, we also assumed that the functions are isotonic). The output transitions are proved to be allowed in the same way as in Theorem 4.1. Thus, all the conditions of Definition 4.3 are satisfied. *Q.e.d.*

It should be pointed out, that the indicatability in the transition $s-b$ does not guarantee the indicatability in the transition $b-s$. From Definition 4.3 and Theorem 4.2, we may immediately deduce that the composition of two indicatable circuits in series (sequential composition) also yields an indicatable circuit.

EXAMPLE 4.1. (continued). The circuit which implements the carry function (4.7) in the form

$$\begin{aligned} d &= \overline{x y p \check{x} \check{y} \check{p}}, \\ P &= \overline{\check{x} \check{y} \check{p}} \vee \overline{\check{x} \check{y} p} \vee \overline{x y \check{p}} \vee \overline{x \check{y} \check{p}}, \\ \hat{P} &= \overline{\check{x} y p} \vee \overline{x \check{y} p} \vee \overline{x y \check{p}} \vee \overline{x y p}. \end{aligned} \quad (4.11)$$

has three double rail inputs and three outputs and is indicatable. In fact, when the spacer e is applied to the inputs, we have $P = \hat{P} = d = 0$, and when any working combination is applied such that $x \neq \check{x}$, $y \neq \check{y}$, and $p \neq \check{p}$, we have $P \neq \hat{P}$ and $d = 1$.

The circuit's inputs are indicated to the output for all allowed transitions in the two-phase discipline with spacer e . Indeed, all the functions of the system (4.11) are antitonus, and both input and output combinations are represented in DRC and OBC, respectively, and hence, are incomparable in pairs. Then the inputs are translated, in one step, from the working combinations. For example, for the transition 010110 – 111111 (the order of value corresponds to the vector

$x \dot{x} y \dot{y} p \dot{p}$) the disjunction of three derivatives of the form $\partial d / \partial \dot{x} \vee \partial P / \partial \dot{x} \vee \partial \dot{P} / \partial \dot{x}$ becomes equal to 1 in the sub-combination $x y \dot{y} p \dot{p} = 10110$, as well as the disjunctions of derivatives with respect to all other five variables in corresponding sub-combinations. Thus, all four conditions of Theorem 4.2 are fulfilled, and the circuit with SIF (4.11) is indicatable.

A fundamental design technique for constructing checkers is shown below.

EXAMPLE 4.2. The inputs of a combinational circuit consisting of one AND-gate (or OR-gate) with the inherent function $y_1 = x_1 x_2 \dots x_n$ ($y_2 = x_1 \vee x_2 \vee \dots \vee x_n$), which is isotonicous, are indicated to the output for the transition $i-e$ ($e-i$), but are not indicated for the inverse transition $e-i(i-e)$. This circuit is not indicatable because the condition 4) of Theorem 4.2 is not satisfied. However, the circuit shown in Fig. 4.6 consists of a pair of AND-gate and OR-gate, the so-called “parallel compression” circuit is indicatable because its inputs x_1, x_2, \dots, x_n are indicated to the outputs y_1, y_2 for the discipline $i-e-i-\dots$.

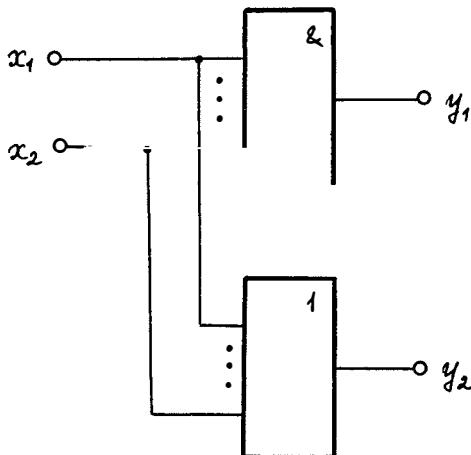


Figure 4.6. A parallel compression circuit.

Indeed,

$$\partial y_1 / \partial x_i = x_1 x_2 \dots x_{i-1} x_{i+1} \dots x_n,$$

$$\partial y_2 / \partial x_i = \bar{x}_1 \bar{x}_2 \dots \bar{x}_{i-1} \bar{x}_{i+1} \dots \bar{x}_n$$

and hence $y_1 = T_s(X/e)$, $y_2 = T_s(X/i)$ and thus all the conditions of Theorem 4.2 are satisfied.

It is easily seen that the multi-input AND-gate and OR-gate (or, alternatively, NAND and NOR gates) in the above circuit can be replaced by multi-layered implementations of elements with the same functions but having fewer inputs, for example, of two-input gates. We suggest that the reader may prove, as an exercise, that the sub-circuits of the CI checker defined by the SIFs (4.4) and (4.5) are indicatable. (*Hint.* We recommend using Theorem 4.2 as has been done in Example 4.1.)

STATEMENT 4.4. *A two-phase implementation of a combinational circuit is aperiodic if and only if it is indicatable.*

Proof.

Necessity. Assume to the contrary, that the two-phase implementation of the combinational circuit is not indicatable. This implies that there exists a combination $t \in [a, b]$ such that $F(t) = l$. But in this case, the transition to l occurs before some element of the circuit switches, i.e. before the completion of the input transition from $a \in A$ to $b \in B$, and this contradicts the two-phase implementation condition.

Sufficiency. Again, assume the contrary, i.e. that although the circuit is indicatable, the two-phase discipline is not always satisfied, in that the change of output signals may occur before the completion of the transition process in one of the circuit elements. Then a combination $t \notin B$ can still be applied to the inputs, while at the output, a combination $l \in L$ has already appeared. This contradicts the definition, Defintion 4.3, of indicatability.

Statement 4.4 is a central point of this section, It provides a justification for using the techniques developed in subsection 4.2.1 to check whether a combinational circuit is, or is not, aperiodic.

4.3.2. STANDARD IMPLEMENTATIONS

Since the procedure of checking that a circuit is indicatable is quite difficult, it is sensible to consider some *canonical techniques* that allow the construction of an implementation from a system of functions. Such an implementation will, from now on, be called *standard*. Methodologically, the synthesis process in this case becomes rather straightforward which, certainly requires a sacrifice, in the form of a higher circuit redundancy.

4.3.2.1. *Minimum Form Implementation*

If an SIF is defined by minimum forms (disjunctive normal form (DNF) or conjunctive normal form (CNF)), then these forms can be associated with certain

minimum representations of functions in DRC. These can be obtained by replacing each variable in complemented form \bar{x}_j by \hat{x} , or by \check{x} , in both the direct function and its complement. To preserve the indicatability, we can require that all the outputs of circuit elements should be represented in DRC. This requirement brings us to the *two-channel*, the so-called *crossed implementation*, in which for each element with output α , there exists an element with the dual inherent function associated with the output $\hat{\alpha}$ ($\check{\alpha}$). As a consequence the pair $(\alpha, \tilde{\alpha})$ where $\tilde{\alpha}$ is either $\hat{\alpha}$ or $\check{\alpha}$, represents two-rail variable. Thus, in this case, the standard implementation has twice the number of inputs and outputs.

EXAMPLE 4.1 (continued). The above technique is illustrated by converting the function (4.6) into the form

$$P = xy \vee yp \vee xp, \quad \hat{P} = \hat{x}\hat{y} \vee \hat{y}\hat{p} \vee \hat{x}\hat{p} \quad (4.12)$$

This pair can be implemented in the form

$$P = \overline{\overline{xy} \cdot \overline{yp} \cdot \overline{xp}}, \quad \hat{P} = \overline{\overline{\hat{x}} \vee \overline{\hat{y}}} \vee \overline{\overline{\hat{y}} \vee \overline{\hat{p}}} \vee \overline{\overline{\hat{x}} \vee \overline{\hat{p}}} \quad (4.13)$$

The circuit obtained is aperiodic if its inputs are $x, \hat{x}, y, \hat{y}, p, \hat{p}$ and its outputs are P, \hat{P} and all the six outputs of NAND and NOR gates whose inherent functions are written under the top bar in (4.13). The overall number of inputs and outputs is 14, the number of elements is 8. Moreover, the matched implementation must also include a 14-input indicator.

4.3.2.2. Orthogonal Form Implementation

Abandoning the representation of an SIF by minimum forms, we may turn to the *orthogonal representation* in which, for any two terms β_i and β_j , $i \neq j$, the condition $\beta_i \beta_j = 0$ always holds. It is obvious that such a representation always exists: ultimately, it will be a canonical DNF.

EXAMPLE 4.1 (continued). Rewrite the equations (4.12) in the form

$$P = xy \vee x\hat{y}p \vee \hat{x}yp \quad \hat{P} = \hat{x}\hat{y} \vee x\hat{y}\hat{p} \vee \hat{x}y\hat{p} \quad (4.14)$$

In the two-layered implementation of (4.14) using NAND gates, the outputs of the elements \overline{xy} , $\overline{x\hat{y}p}$, $\overline{\hat{x}yp}$, $\overline{\hat{x}\hat{y}}$, $\overline{x\hat{y}\hat{p}}$ and $\overline{\hat{x}y\hat{p}}$ of the first layer, are indicated to the outputs P and \hat{P} because, due to orthogonality, only one gate of the first layer

switches in each of the two channels. However, the inputs of the circuit are indicated to the outputs P and \hat{P} . Hence, all six inputs of the circuit together with P and \hat{P} should be regarded as the outputs of the circuit. The matched implementation version must also include an 8-input indicator.

4.3.2.3. Hysteresis-Flip-Flop-Based Implementation

This approach is based on the idea of integrating the SIF implementation with indicators that are, in this case, imbedded in functional elements. Such an integration requires the use of elements of AND-OR-NOT type, as well as inverters which are organized in flip-flops that are somewhat similar to the hysteresis type.

If Boolean functions of the form x_1x_2 , $x_1 \vee x_2$ and $x_1 \oplus x_2$ are represented in the form of automata equations

$$\begin{aligned} f_{\wedge} &= x_1 \dot{x}_1 x_2 \dot{x}_2 \vee f_{\wedge} (x_1 x_2 \vee x_1 \dot{x}_1 \vee x_2 \dot{x}_2), \\ f_{\vee} &= x_1 \dot{x}_1 x_2 \dot{x}_2 \vee f_{\vee} (x_1 \vee x_2), \\ f_{\oplus} &= x_1 \dot{x}_1 x_2 \dot{x}_2 \vee f_{\oplus} (x_1 \dot{x}_2 \vee \dot{x}_1 x_2) \end{aligned} \quad (4.15)$$

respectively, then it is clearly seen that applying the spacer e to the input, results in setting the value of each function in (4.15) to 1 and when the DRC is applied, each value either remains the same (1) or changes to 0.

Using the basic set of \wedge , \vee and \oplus it is possible to obtain a two-channel crossed implementation by elements of the Γ -flip-flop type with inherent functions of the form (4.15) in the one channel, and with the dual functions in the other channel. In such an implementation, all the inputs are indicated to the outputs. The matched implementation must also include an indicator whose only inputs are the outputs of the crossed implementation.

4.3.2.4. Implementation based on “Collective Responsibility”

Consider another technique for constructing circuits. As we can see, the circuit can be non-indicatable because it is not known how many elements in a given layer will change their values in the working phase. The following idea is fundamental for the technique. We implement the “forced” transition of all elements of a layer in one channel to the same state. Such a transition is forced by the feedback signal from the elements of the next layer.

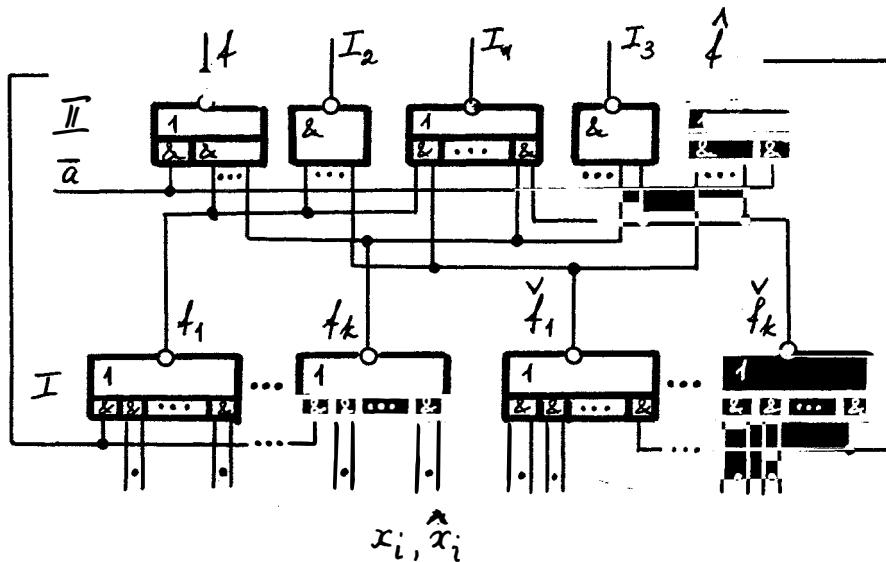


Figure 4.7. An idea of implementation by the “collective responsibility” method.

The idea is illustrated in Fig. 4.7. When the spacer i is applied to the inputs, and the phase signal \bar{a} is set to 1, the outputs of all the elements of the first layer become equal to the all-one spacer value e , and those of the second layer will assume the all-zero spacer i . When a working combination in DRC is applied and $\bar{a} = 0$, the output of one or more elements in one of the two channels of the first layer become equal to 0. Then the output of an element in the second layer of the same channel will become equal to 1. Thereby, all the outputs of the first layer elements in this channel will be equal to 0. The outputs of the first layer elements are indicated to the outputs of the circuit with the SIF

$$I_1 = \overline{f_1 \hat{f}_1 \vee \dots \vee f_k \hat{f}_k}, \quad I_2 = \overline{f_1 \hat{f}_1 \dots f_k \hat{f}_k}$$

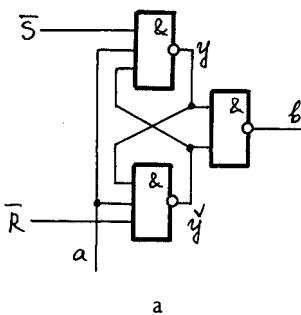
At the end of the working phase $I_1 = I_2 = 1$, and at the end of the idle phase $I_1 = I_2 = 0$. In the matched version of the implementation both outputs f , \hat{f} , I_1 , I_2 and all input variables in DRC are used as inputs of an indicator.

The techniques, discussed above, for standard implementations of Boolean functions within the class of aperiodic combinational circuits, certainly do not cover all the potential approaches. In conclusion of this section, we note that, due to the redundancy aspects related to self-synchronizing codes, there must exist an aperiodic analogue of a synchronous circuit with h inputs and q outputs such that it will have $h + \log_2 h$ inputs and $q + \log_2 q$ outputs.

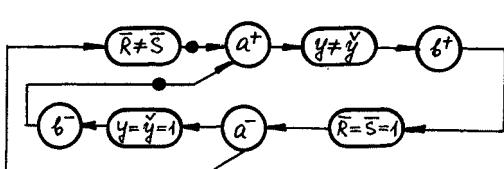
4.4 Aperiodic flip-flops

Before we proceed to the consideration of aperiodic implementations of sequential circuits (with memory), we should first discuss ways of constructing memory devices for the state machines which, due to the assumption that built-in delays must not be used, may incorporate flip-flops of various types (usually, RS-, D- and T-types).

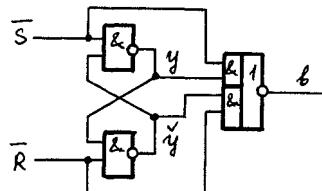
Two fundamental ideas of how to indicate the completion of transitions in RS-flip-flops are shown in Fig. 4.8.



a



b



c

Figure 4.8 (a) - (c). The implementation of the indication of process completion in flip-flops: (a) using a phase signal; (b) the interaction with the environment; (c) Muller-based indication.

An ordinary asynchronous *RS-flip-flop* shown in Fig. 4.8(a), is built of *two NAND gates*. It has an additional *phase signal a* and an *indicator b* which is simply a 2-input NAND gate.

In the initial state $a = 0$, $y = \bar{y} = 1$, $b = 0$. If the inputs are such that $R \neq S$ and $a = 1$, the working phase gets started. The first to change is one of the arms of the

flip-flop \hat{y} ⁽¹⁾, after which the indicator b changes to 1. In the idle phase, after $a = 0$, the inputs \bar{R} and \bar{S} may change, and one of the arms of the flip-flop will return to 1 ($y = \hat{y} = 1$), after which b will be equal to 0. The next working phase ($a = 1$) can only be started if $R \neq S$ and $b = 0$. The dynamics of the above operation is illustrated by the signal graph in Fig. 4.8(b).

Vertices marked with $y = \hat{y} = 1$ and $y \neq \hat{y}$ are used to label the transitions of y, \hat{y} of the following kind: $y = \hat{y} = 1$ denotes 01–11 or 10–11 transitions and $y \neq \hat{y}$ denotes 11–01 or 11–10 transitions (the same holds for R, S). In this graph, the concurrency between transitions is introduced to show that as soon as $a = 0$, it is possible to apply new values to the set and reset inputs. In a strictly sequential design (when b^- is followed by the transition $R \neq S$, which is, in its turn, followed by a^+) the critical time allowed by the environment for applying new values of R and S would have increased by the value of the switching delay of one of the flip-flop arms, and that of the indicator b .

The length of each transition phase is $2T$ ⁽²⁾.

As in the case with the circuit of Fig. 4.8(a), another RS-flip-flop (\hat{y}) can be built that uses NOR gates. Its inherent functions are dual to those of the previous flip-flop and the input signals should be replaced by their complements.

The indication of the transition phase completion has been easily realized in the circuit of Fig. 4.8(a), because such a circuit is an example of the matched implementation. The latter can be seen from a simple comparison between Fig. 4.8(b) and Fig. 4.1(c): the sequencing of phase signals in both graphs is the same. We should also note that, in the idle phase, the pair of values satisfying $\bar{R} = \bar{S} = 1$ is regarded as an input spacer, and after a^- the information stored in the flip-flop is to be lost ($y = \hat{y}$). Hence, we can consider the above design as an *example of a degenerate state machine*. However, it will be shown that aperiodic flip-flops can be based on this structure.

While, in the circuit of Fig. 4.8(a), the completion indication is performed in both transition phases by an elementary indicator – NAND gate – another idea for organizing indication consists in *comparing the state of the flip-flop with the values of the set-reset inputs*.

(1) Here, and in the sequel, for the sake of brevity, flip-flops built of the pairs of elements (y, \hat{y}) and (y, \hat{y}) will be denoted by just one letter – \hat{y} and \hat{y} , respectively. At the same time, this denotation allows us to define which type of the transient state is used in the flip-flop – $\langle 1, 1 \rangle$, in the case of \hat{y} , and $\langle 0, 0 \rangle$ in the case of \hat{y} .

(2) Here, and in the sequel, T denotes the average delay of the gate.

Consider the circuits shown in Fig.4.8(c). Here, the indicator with the inherent function $b = \overline{S}y \vee R\bar{y}$ is used. During the working phase (*write phase*), if the set-reset input values are equal to the state of the flip-flop, $\overline{S} = \bar{y}$ and $\overline{R} = y$, then the latter does not switch, and b goes to 1. Otherwise, i.e. if $\overline{S} = y$ and $\overline{R} = \bar{y}$, the flip-flop switches and b goes to 1, while, during the transient state (11) in the flip-flop $b = 0$. In the idle phase (*store phase*), $\overline{S} = \overline{R} = 1$, the previous state is not changed but the indicator changes – at the end of the phase, b becomes equal to 0.

The technique for constructing indicators, which is based on the idea of Fig. 4.8(c), is common to many aperiodic designs presented in further discussions.

4.4.1. FURTHER DISCUSSION OF FLIP-FLOP DESIGNS

4.4.1.1. RS-Flip-Flops

From the “workpieces” shown in Fig. 4.8, we can proceed to the rather more sophisticated aperiodic RS-flip-flops presented in Fig.4.9, in which a special technique, called a “*cutting of the inputs*”, is used. In fact, the inputs are *cut off* immediately after the completion of the working phase. The values R and S are fixed in this phase only for a period of the switching process until the signal \bar{b} (see Fig.4.9(a)) is equal to 0. After that, R and S are allowed to change.

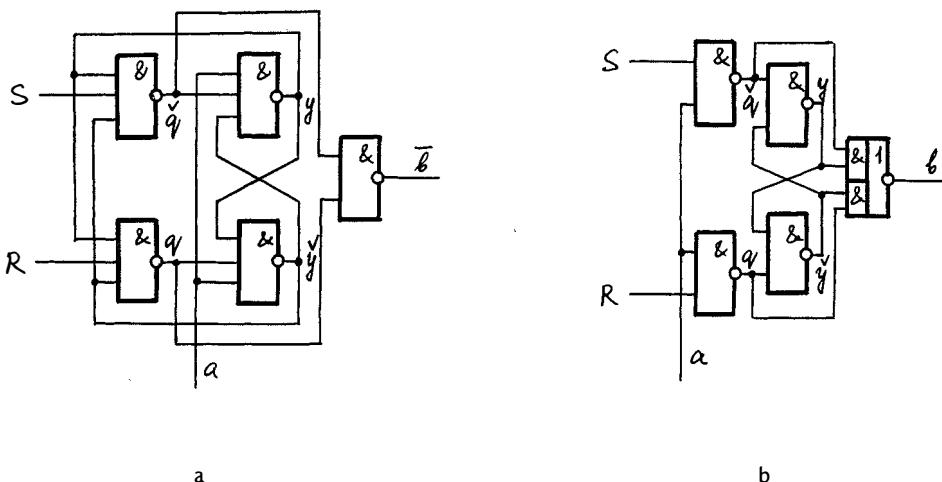


Figure 4.9 (a) and (b). Two aperiodic flip-flops with separated inputs (the NAND-based variant).

In the idle phase, when $a = 0$, the values of R and S must not be such that $RS = 1$, i.e. the spacer must be all-zero. Since it is possible that the input gates change after the switching of the flip-flop \dot{y} , the completion of the transient process must be indicated through the outputs of the gates. The latter may become equal to the complemented input values only after $y = \dot{y} = 1$ and they become equal to 1 ($q = \dot{q} = 1$) only after the completion of the working phase in the flip-flop.

The operating cycle length for the circuit of Fig. 4.9(a) is $6T$ – each phase takes $3T$.

In the idle phase, the information stored in the flip-flop is lost because in that phase $y = \dot{y}$. If we need to store data during the idle phase (except, of course, the transition time of the flip-flop itself), we can use the implementation shown in Fig. 4.9(b). In such a design, instead of the term “idle phase”, we use the “store phase” term. When $a = 0$, the signals R and S do not influence the circuit $q = \dot{q} = 1$, and the flip-flop is in the store phase ($b = 0$). When a goes to 1, the first layer gates complement the input signals, $\dot{q} = \bar{S}$ and $q = \bar{R}$; at the end of the phase $b = 1$. Note that b remains equal to 0, while both $\dot{y} = \dot{q}$ and $y = q$. Thus $b = (\dot{y} \oplus q) (y \oplus \dot{q})$. If we also take into account that the transient state is all-one, then the inherent function of the indicator can be written in the form

$$b = \overline{\dot{q}y} \vee \overline{q\dot{y}}.$$

As has been said, already, in the store phase $a = 0$, $q = \dot{q} = 1$, $y \neq \dot{y}$ and $b = 0$. After a changes to 1, the values of q and \dot{q} also change. If the information previously stored in the flip-flop corresponds to the input signals, then the flip-flop does not switch, and only the indicator's output changes to $b = 1$. At this point, the transition process in the working phase (write phase) terminates. If the input values do not correspond to the flip-flop state, then either $\dot{q}y = 1$ or $q\dot{y} = 1$, and since the flip-flop transient state is all-one, the signal on the indicator's output will change only after the completion of the transient process in the flip-flop. In this design, we require that no change of input signals can take place during the whole write phase, and then, until the end of the write phase, S and R may either keep their values unchanged or both become equal to 0. The data is available from the flip-flop at any time, with the exception of the period of the transition from the store phase to the write phase. The transition from the write phase to the store phase takes $2T$, and the reverse transition requires either $4T$ or $2T$, depending upon whether the flip-flop state does or does not change. Hence, the *overall operating cycle is either $4T$ or $6T$* , and, on the average, it is faster than that of the circuit of Fig. 4.9(a).

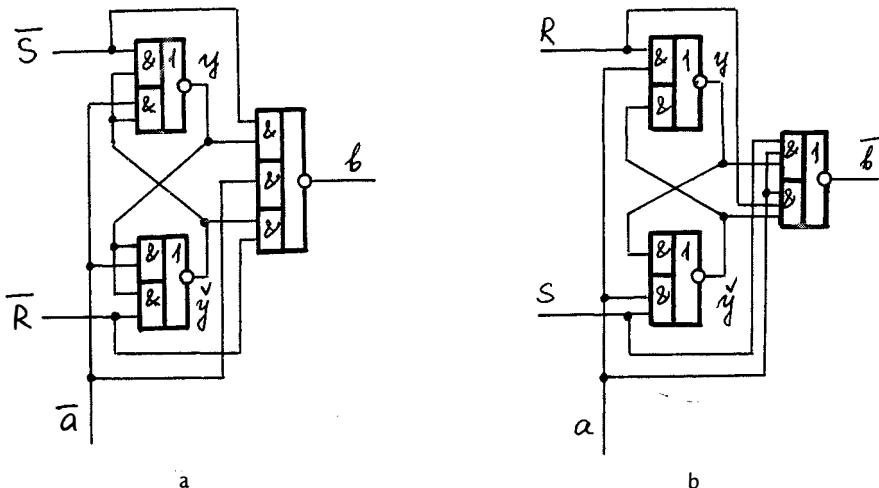


Figure 4.10 (a) and (b). Two aperiodic flip-flops with separated inputs (the AND-OR-NOT-based variant).

The flip-flops shown in Fig. 4.10 are built of AND-OR-NOT gates, and are functionally equivalent to those of Fig. 4.9. Furthermore, they are faster than the latter. In the store phase, when $\bar{a} = 1$, the circuit given in Fig. 4.10(a) is insensitive to inputs R and S , and hence, $b = 0$. In the write phase, after $\bar{a} = 0$, the behaviour of the flip-flop depends on its state and the input values: if its state does comply with the input signals, then it stays in the previous state, and the 0–1 transition of b manifests the termination of the transient process in the write phase. Otherwise, b is kept equal to 0 until the flip-flop changes its state to the required value.

It should be noted that during the transition through the transient, all-one, state, the indicator's output b remains equal to 0. As soon as the phase is over, it changes to 1.

The time of the write phase is either $3T$ or T . After the phase signal changes from 0 to 1, the flip-flop state does not alter, but it allows the feedback signals to work, and thereby, to suppress the effect of input signals on the flip-flop state. Simultaneously, the indicator changes its output value. Hence, the delay is just T .

Thus, the operating cycle of the flip-flop shown in Fig. 4.10(a) is $2T$ or $4T$.

Consider the flip-flop shown in Fig. 4.10(b). In its store phase, when $a = 0$, the flip-flop does not respond to input signals $\bar{b} = 1$. If the flip-flop state does comply with the input values, then after a changes from 0 to 1, the output \bar{b} goes to 0. Otherwise, the flip-flop switches through the transient state, all-zero, and \bar{b} remains equal to 1 until one of the two conditions is satisfied: either $Sy = 1$ or

$R\check{y} = 1$. The phase thus ends with $\bar{b} = 0$.

The duration of the transition from the store phase to the write phase is either T or $3T$. The 1–0 transition of the phase signal a results in cutting off the input signals and changing the flip-flop indicator state, thereby completing the store phase whose length is T .

4.4.1.2. D-Flip-Flops

A *D*-flip-flop imposes a single-cycle delay of the input signal, and, when using component flip-flops with idling, should consist of a master flip-flop and a slave flip-flop. When the master flip-flop is reset (idled), the information written in it in the previous phase has to be rewritten in the slave flip-flop.

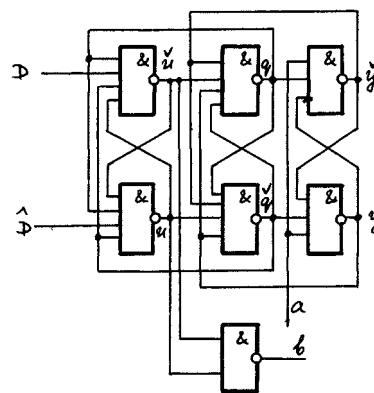
The circuit shown in Fig. 4.11(a) is built using a three flip-flop structure. In the idle phase, when $a = 0$, the master flip-flop \check{y} is reset to the both-ones state, the flip-flop \check{q} is in the working state, and the flip-flop \check{u} is also reset to both-ones. In this state, the inputs do not have any effect on the circuit state, and the indicator state is $b = 0$. When a goes to 1, the flip-flop \check{y} changes to the working state, \check{q} is reset to both-ones and, thereby, allows the information on the inputs to be written into \check{u} . After that b goes to 1. Therefore, the working phase takes $4T$ as does the idle phase. Indeed, when $a = 0$, the first to reset is \check{y} , then \check{q} switches to the working state, and then \check{u} is also reset. This ends with $b = 0$.

The operating cycle of the circuit shown in Fig. 4.11(a) is $8T$. We should note that while the flip-flop \check{u} is reset to both-ones, any changes in the double-rail signal \hat{D} are allowed. There are no limitations whatever on the time period of this signal being in the transient state. Moreover, in this case the feedback connections in the flip-flop \check{u} can be totally ignored.

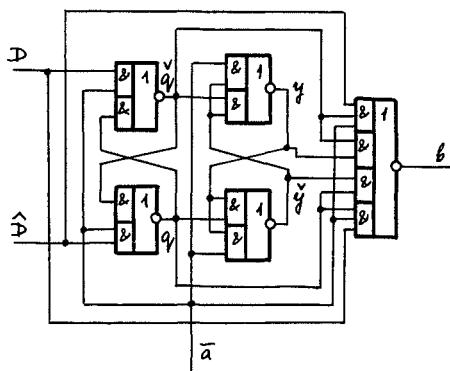
Using flip-flops without resetting (or idling), we can construct a *D*-flip-flop with two component flip-flops. Such a design is shown in Fig. 4.11(b). In the write phase, when $\bar{a} = 0$, the slave flip-flop \check{q} is cut off from the double-rail input \hat{D} , and the information is rewritten from the slave to the master flip-flop. The latter, together with the part of the indicator to which the outputs of the master flip-flop are connected, is, basically, built on the circuit of the type shown in Fig. 4.10(a). At the end of the write phase in the master flip-flop, $b = 1$. The duration of the write phase is either T or $3T$.

In the store phase, the master flip-flop is cut off from the slave flip-flop, and the latter can be loaded with data from the outside. The write operation in the slave flip-flop, which is based on the circuit shown in Fig. 4.10(b), is terminated by the switching of the indicator from 1 to 0. The transient process time in this phase is also

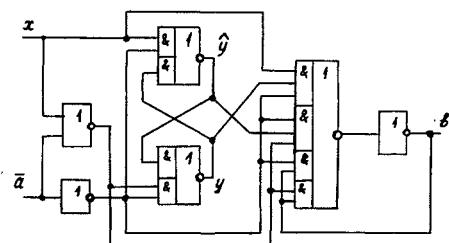
either T or $3T$. Thus, the overall operating cycle of the circuit is made to be either $2T$ or $6T$ (cf., $8T$ for the flip-flop presented in Fig. 4.11(a)).



a



b



c

Figure 4.11 (a) - (c). Aperiodic latches (delay-flip-flops): with double-rail input – (a) the NAND-based and (b) the AND-OR-NOT versions; (c) with single-rail input.

D-flip-flops, shown in Fig. 4.11(a) and (b), both have double-rail input signals. Sometimes it is convenient to use a design with *single-rail data signals*. An example of such a design is shown in Fig. 4.11(c). It is built around a flip-flop of the type shown in Fig. 4.10(b). The NOR element is introduced to create the second data input, and the indicator has the function

$$b = xy a \vee \bar{x}\hat{y} a \vee (a \wedge \bar{x}) b.$$

As a consequence, such a construction can be used for indicating that the state of y complies with the value of the single-rail input x in the working phase: if $x = y$ then $b = 1$, and if $x \neq y$ then $b = 0$. In the idle phase, when $a = 0$, the information is stored in the flip-flop and the inputs are allowed to change.

4.4.1.3. T-Flip-Flops

In the working phase, a *T-flip-flop* switches to the state which is the complement to that of the previous working phase. The phase signal, in this case, plays the role of the complementing input. Generally speaking, the T-flip-flop can be built around a D-flip-flop such, for example, as the one shown in Fig. 4.11(b), provided that the output signals of the master flip-flop of the latter are arranged as the circuit's inputs, i.e. $D = \check{y}$ and $\hat{D} = y$. It is just because the outputs are fed back in such a manner that we can simplify the entire implementation of the T-flip-flop as compared to its D-prototype.

Thus, in the circuit of Fig. 4.12(a), the indicator's inherent function can be reduced as follows:

$$b = \overline{\bar{a}Dq \vee \bar{a}\hat{D}\hat{q} \vee \check{y}q \vee y\hat{q}} = \overline{\hat{y}q \vee y\hat{q}}$$

Both phases have the same length – 3T, and the whole operating cycle takes 6T.

In the T-flip-flop shown in Fig. 4.12(b), let the phase signal a be equal to 0. Then the flip-flop \check{y} is reset – both of its arms are equal to 1 – and its outputs affect neither the states of the gates p and \check{p} nor the state of the flip-flop \check{q} : $\check{p} = \bar{q} = \check{q}$, $p = \check{q} = q$ and $\bar{b} = 1$. After a changes from 0 to 1, the flip-flop \check{y} is loaded with information taken from the outputs of p and \check{p} . This is followed by $\check{y} = \check{p}$ and $y = p$ that, in its turn, leads to changing the state of \check{q} . When \check{q} becomes such that $q = \check{y}$ and $\check{q} = y$, one of the gates (whose output was equal to 0) will change to 1. As a result, $p = \check{p} = 1$, and the 1–0 transition of the indicator's output will terminate the

transient process. Thus, during its transition to the working state, the flip-flop \bar{y} will reach the state which is complement to that of the previous working phase. The length of the working phase is $5T$. When $a = 0$, the flip-flop \bar{y} is reset to the all-one spacer that makes one of the gates change the state. Finally b goes to 1, and that terminates the phase whose length is $3T$, and, hence, *the entire cycle requires $8T$* .

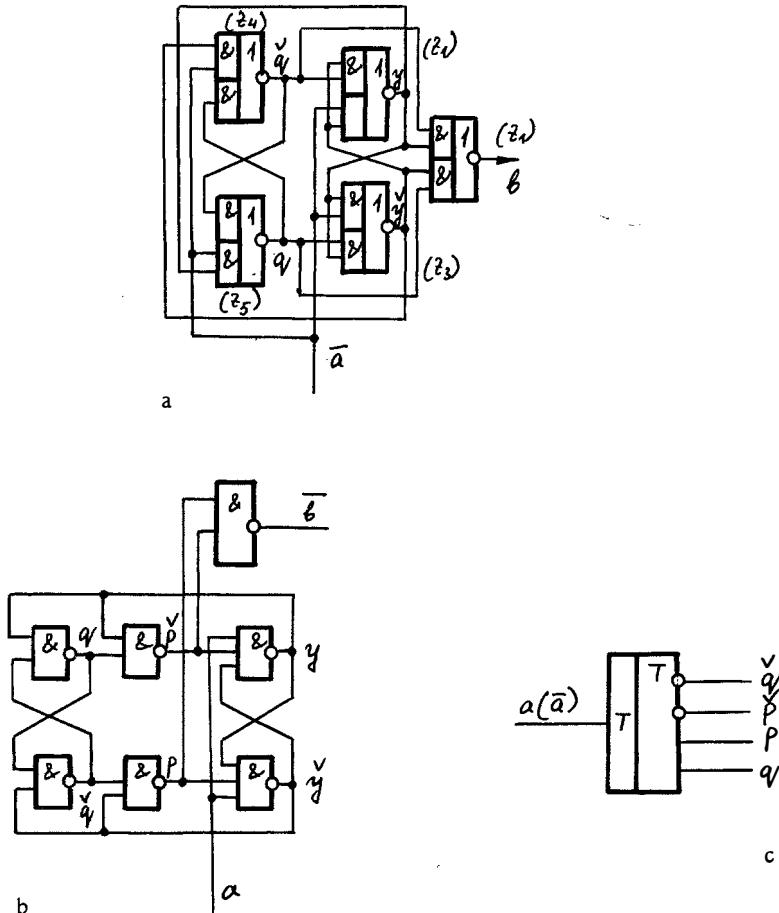


Figure 4.12 (a) - (c). Aperiodic complementing flip-flops: (a) the AND-OR-NOT-based variant; (b) the NAND-based variant (Harvard flip-flop); (c) symbolics.

It should be noted, in conclusion to this section, that the constructions of aperiodic flip-flops described here fall into the category of linear sequential cyclic

circuits and can be formally synthesized by the techniques outlined in Section 5.5. Example 5.8 demonstrate a synthesis procedure for the flip-flop shown in Fig. 4.12(a).

4.5 Canonical aperiodic implementations of finite state machines

In this section we shall discuss a number of universal, canonical, approaches to the design of aperiodic implementations for finite state machines. These approaches allow us to simplify, to a marked extent, the synthesis process, without any special examination of the characteristics of a given machine. However, this is at the cost of more extensive circuitry. As a result, the circuits obtained by such synthesis are usually called *canonical*, or *standard*, *implementations*. In the constructions presented in this section, we generally use non-redundant encoding of the machine states. A machine with s internal states requires at least $s_0 = \lceil \log_2 s \rceil$ memory elements (aperiodic flip-flops). The type of flip-flop does, in fact, determine the type of the canonical implementation. All these implementations are structurally similar. Each of them is composed of a combinational circuit, a register consisting of s_0 memory elements with individual indicators, and a common indicator for the entire machine. Designing combinational circuits and common indicators may use the techniques outlined in Sections 4.3.2 and 4.2, respectively, but it certainly requires maximum attention to be given to possible differences between the transient states indicated by the logic and those of the flip-flops.

4.5.1. IMPLEMENTATION WITH DELAY-FLIP-FLOPS

In the case of using D-flip-flops with individual indicators, the structure of the implementation will be of the form shown in Fig. 4.13.

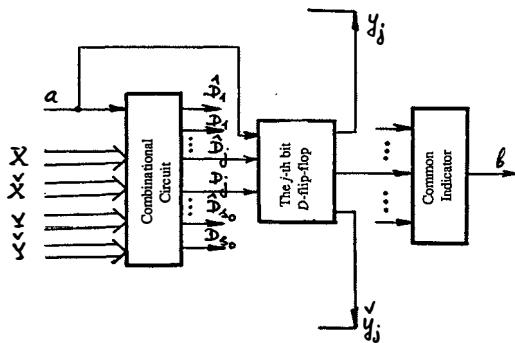


Figure 4.13. Block diagram for an aperiodic implementation of a finite state machine based on D-flip-flops.

It is a composition of a combinational circuit with double-rail inputs and outputs, a register consisting of D-flip-flops, and a common indicator. The phase signal is generally applied to both the combinational circuit and the register. The circuit of Fig. 4.13 can be refined in more detail depending on the implementation strategy that is chosen for the logic and the flip-flops. One such refinement is presented in the following example.

EXAMPLE 4.3. Let a finite state machine be given by the following system of equations

$$\begin{aligned} Y_1 &= x_1 x_2 y_1 y_2 \vee \bar{y}_1 \bar{y}_2 \vee \bar{x}_1 \bar{y}_1 \vee \bar{x}_2 \bar{y}_1, \\ Y_2 &= y_1 y_2 \vee x_1 x_2 y_2 \vee \bar{x}_1 \bar{y}_1 \bar{y}_2 \vee \bar{x}_2 \bar{y}_1 \bar{y}_2 \end{aligned} \quad (4.16)$$

By the designations of Fig. 4.13 $X = \{x_1 x_2\}$ and $Y = \{y_1 y_2\}$. The implementation of a combinational circuit also requires the use of the complements of the above functions. These complements are

$$\begin{aligned} \bar{Y}_1 &= x_1 x_2 \bar{y}_1 y_2 \vee y_1 \bar{y}_2 \vee \bar{x}_1 y_1 \vee \bar{x}_2 y_1, \\ \bar{Y}_2 &= y_1 \bar{y}_2 \vee x_1 x_2 \bar{y}_2 \vee \bar{x}_1 \bar{y}_1 y_2 \vee \bar{x}_2 \bar{y}_1 y_2 \end{aligned} \quad (4.17)$$

Let D-flip-flops of the type shown in Fig. 4.11(a), be used here. Then the canonical implementation based on the structure of Fig. 4.13 will be built as demonstrated in Fig. 4.14. The logic part is realized using the “collective responsibility” principle.

Initially, $a = 0$ and the value of the old state is stored in the flip-flops \check{q}_1 and \check{q}_2 . The 0–1 transition of a opens the working phase, and after the flip-flop transitions, which proceed through the same sequence as has been described in the flip-flop of Fig. 4.11(a), the outputs $Y_1, \hat{Y}_1, Y_2, \hat{Y}_2$ become set to a new state. The end of the phase is manifested by $b_{12} = 1$ and $b = 1$.

When a changes back to 0, the idle phase begins, during which the new state value is reloaded to flip-flops \check{q}_1, \check{q}_2 , and outputs $Y_1, \hat{Y}_1, Y_2, \hat{Y}_2$ are set to the spacer (all-zero). At the end of the phase, $b_{12} = 0$ and $b = 0$.

A stable internal state of the machine shown in Fig. 4.14 is represented by the combination in the balanced code with $4s_0$ components: each j -th bit, $1 \leq j \leq s_0$, is associated with four components $y_j, \check{y}_j, q_j, \check{q}_j$. The working states from the set Z_1 are associated with those combinations in which, for $a = 1, y_j \neq \check{y}_j$ and $q_j = \check{q}_j = 1$.

The idle states from Z_2 are assigned the combinations in which, for $a = 0$, $y_j = \check{y}_j = 1$ and $q_j \neq \check{q}_j$. It should be noted that, in this case, $Z_1 \cap Z_2 = \emptyset$ holds.

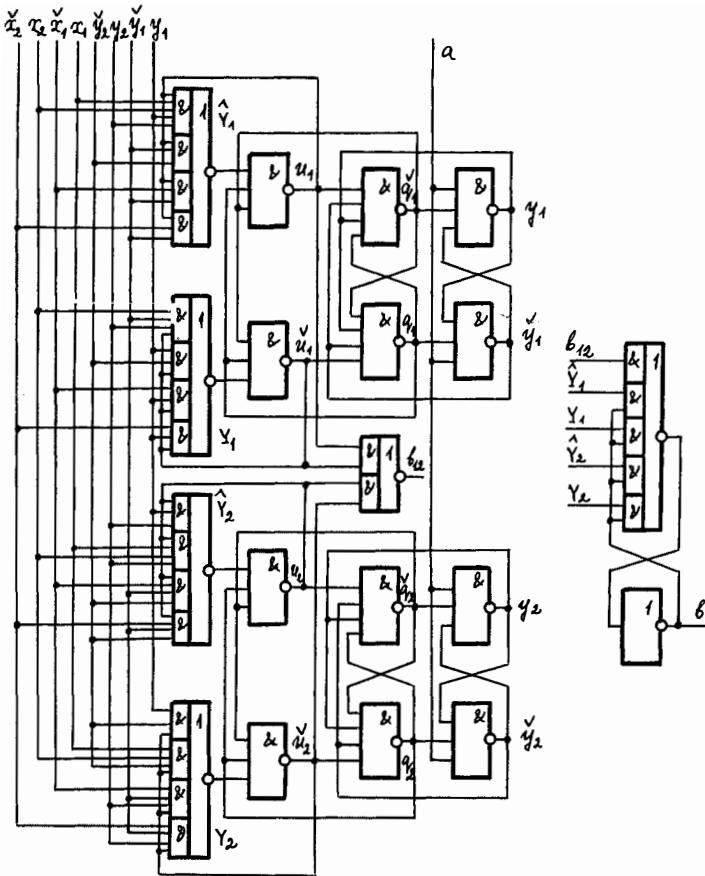


Figure 4.14. Example of the D-flip-flop-based implementation of a state machine.

4.5.2. IMPLEMENTATION USING FLIP-FLOPS WITH SEPARATED INPUTS

A certain modification of the circuit, shown in Fig. 4.10(b), can be used as a memory element in which, together with the master flip-flop \hat{y}_j , the slave flip-flop \check{w}_j will be incorporated. The corresponding implementation is presented in Fig. 4.15.

The operation of the memory element, as well as that of the whole machine, consists of two phases. In the working phase, when $a = 1$ and the excitation functions S_j and R_j have acquired values corresponding to the new state, the flip-flop

\check{w}_j either does not switch, if the values of the excitation functions are already equal to the machine state, or does switch, if they are not. In either case, the phase is completed by the change of the indicator \bar{b}_j from 1 to 0.

In the idle phase, when $a = \bar{S}_j = \bar{R}_j = 1$, the state value is reloaded from flip-flop \check{w}_j to flip-flop \hat{y}_j , and, in the end, $y_j = w_j$, $\hat{y}_j = \check{w}_j$ and $\bar{b}_j = 1$.

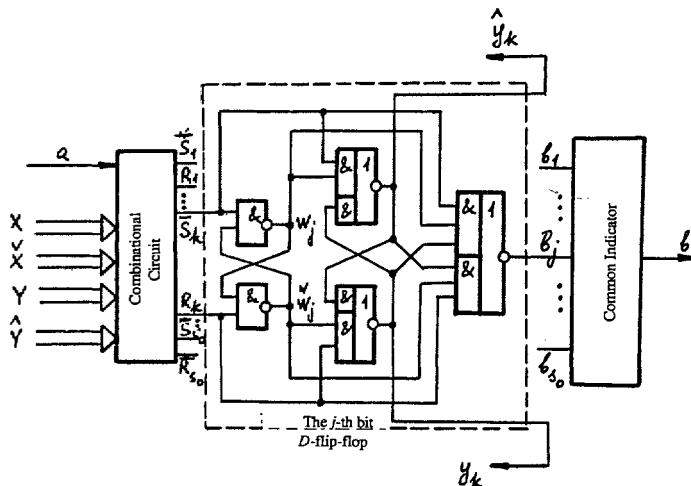


Figure 4.15. Block diagram for an aperiodic implementation of a finite state machine based on flip-flops with separated inputs.

A stable internal state of such a machine is represented by the combination in the double-rail code with $4s_0$ components: for each j -th bit, $1 \leq j \leq s_0$, we assign four components w_j , \check{w}_j , y_j , \hat{y}_j . The working states in Z_1 are associated with those combinations in which, if $S_j \neq R_j$ then $w_j \neq \check{w}_j$ and $y_j \neq \hat{y}_j$. The idle states in Z_2 are assigned such combinations where if $\bar{S}_j = \bar{R}_j = 1$, then $y_j = w_j$ and $\hat{y}_j = \check{w}_j$.

All remaining states are considered intermediate. We should again note that $Z_1 \cap Z_2 = \emptyset$ holds.

It must be pointed out that when deriving expressions for S_j and R_j functions, we can resort to exploiting the fact that the behaviour of RS-flip-flops is defined by equations of the form

$$Y_j = S_j \vee \bar{R}_j y_j, \quad S_j R_j = 0,$$

where Y_j and y_j are the new and the old states of the flip-flop, respectively. From

this fact, we can easily obtain Table 4.1, which shows how to simplify the excitation functions by an appropriate assignment of “don’t care’s” (marked by asterisks).

y_j	Y_j	S_j	R_j
0	0	0	*
1	0	0	1
0	1	1	0
1	1	*	0

Table 4.1

EXAMPLE 4.3 (continued). Making the minimization of truth tables for Y_1 and Y_2 in the system (4.16), with reference to Table 4.1, we obtain

$$\bar{S}_1 = y_1 \vee x_1 y_2, \quad \bar{R}_1 = \bar{y}_1 \vee \bar{x}_1 \bar{y}_2 \vee x_1 x_2 y_2,$$

$$\bar{S}_2 = y_1 \vee y_2 \vee x_1 x_2, \quad \bar{R}_2 = y_1 \vee \bar{y}_2 \vee x_1 x_2$$

4.5.3. IMPLEMENTATION WITH COMPLEMENTING FLIP-FLOPS

Using T-flip-flops as memory elements for the aperiodic implementation of a finite state machine, we should represent the transition functions

$$Y_j = Y_j(x_1, \dots, x_n, y_1, \dots, y_{s_0})$$

in the form

$$Y_j = y_j \oplus F_j(x_1, \dots, x_n, y_1, \dots, y_{j-1}, y_{j+1}, \dots, y_{s_0}).$$

This transformation of the transition functions Y_j to *switching functions* F_j allows us to use the structures shown in Fig. 4.16. The memory elements which should be incorporated in this structure are those presented in Fig. 4.12(b), with the indicators having an extra input, which is connected to the outputs F_j of the combinational circuit. The complementing input of the j -th bit flip-flop is connected to the output \dot{F}_j . Hence, the characteristic function of the indicator \bar{b}_j has the form $\bar{b}_j = \overline{u_j \dot{u}_j F_j}$.

Indeed, at the end of the idle phase (see the description of the operation of the circuit shown in Fig. 4.12(b), in Section 4.4), one of the gates u_j or \dot{u}_j is reset to 0, and then $\bar{b} = 1$. The end of the working phase is marked with $u_j = \dot{u}_j = \dot{F}_j = 1$, after which $\bar{b} = 0$.

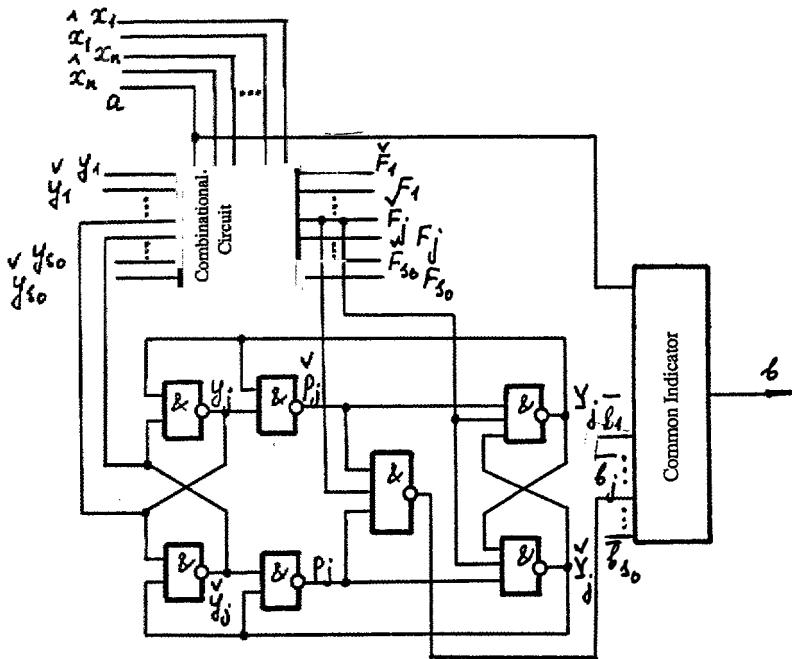


Figure 4.16. Block diagram for an aperiodic implementation of a finite state machine based on complementing flip-flops.

The combinational circuit, whose inputs include the phase signal a , double-rail variables $x_j, \tilde{x}_j, 1 \leq j \leq n$, double-rail internal variables $y_j, \tilde{y}_j, 1 \leq j \leq s_0$, taken from the outputs of flip-flops \tilde{y}_j , which store the values of internal variables during the idle phase, and whose outputs are the double-rail switching functions F_j, \tilde{F}_j , can be implemented using any of the techniques presented in Section 4.3.2.

The common indicator, with inputs $a_j, \bar{b}_j, 1 \leq j \leq s_0$, and the output b , is constructed using the techniques from Section 4.2.

EXAMPLE 4.4. A one-bit converter of the double-rail code into the differential code is built strictly in accordance with a canonical implementation of the type shown in Fig. 4.16. However, it has a degenerate combinational part. The direct input x_j is connected directly to the complementing input of a T-flip-flop, and the complemented input \hat{x}_j is connected to the indicator \bar{b}_j which has an inherent

function of the form $\bar{b}_j = \overline{\hat{x}_j \vee u_j \dot{u}_j}$. The signals y_j are assigned to the differential representation.

In the idle phase $x_j = \hat{x}_j = 0$ and $u_j = \dot{u}_j$. If, during the working phase $x_j = 1$, then the T-flip-flop changes its state, and so does the output signal y_j , but if $x_j = 0$, the T-flip-flop state does not change although $\hat{x}_j = 1$.

In an m -bit converter, the inherent function of the indicator will be

$$\bar{b} = \bigwedge_{j=1}^m \bar{b}_j \vee \bar{b} \left(\bigvee_{j=1}^m \bar{b}_j \right).$$

EXAMPLE 4.3 (continued). By rearranging transition functions (4.16) to the form

$$Y_1 = y_1 \oplus \overline{x_1 x_2 y_2}, \quad Y_2 = y_2 \oplus \overline{y_1 \vee x_1 x_2}$$

we can proceed to the canonical implementation with T-flip-flops of Fig. 4.16. In this case, one of the variants of the combinational circuit will have inherent functions of the form

$$F_1 = \overline{x_1 x_2 y_2}, \quad \dot{F}_1 = \overline{\dot{x}_1 \vee \dot{x}_2 \vee \dot{y}_2}, \quad F_2 = \overline{y_1 \vee x_1 x_2}, \\ \dot{F}_2 = \overline{\dot{y}_1 \dot{x}_1 \vee \dot{y}_1 \dot{x}_2},$$

and the indicator function will be

$$\overline{\bar{b}} = \overline{\bar{b}_1 \bar{b}_2 \bar{a} \vee (\bar{b}_1 \vee \bar{b}_2 \vee \bar{a}) \bar{b}}.$$

If so desired, the canonical implementation can be avoided by merging the functions of complementing flip-flops and those of combinational circuits.

The common indicator for the given example is built up using the equation

$$b = u_1 \dot{u}_1 u_2 \dot{u}_2 a \vee b (\dot{y}_2 u_1 \dot{u}_1 \vee \dot{x}_1 u_1 \dot{u}_1 \vee \dot{x}_2 u_1 \dot{u}_1 \vee \dot{x}_1 \dot{y}_1 u_2 \dot{u}_2 \vee \dot{x}_2 \dot{y}_2 u_2 \dot{u}_2)$$

(4.18)

This requires some clarification. When the phase signal a goes from 1 to 0, the conditions of transient process completion start to depend on the values of the transition functions. If

- (1) $x_1x_2y_2 = y_1 \vee x_1x_2 = 0$, then such a condition is $u_1^{\vee} \dot{u}_1 \vee u_2^{\vee} \dot{u}_2 = 0$, if
 (2) $x_1x_2y_2 = 0$, $y_1 \vee x_1x_2 = 1$, then the condition becomes $u_1^{\vee} \dot{u}_1 = 0$, and if
 (3) $x_1x_2y_2 = y_1 \vee x_1x_2 = 1$, the condition is simply $a = 0$.

The above are true because: in case (1) both flip-flops have the output values of their gates changed, in case (2), only the first flip-flop mimics changes, and in (3) the machine state does not change at all. Hence, when $a = 0$, the phase completion condition is $\overline{x_1x_2y_2} u_1^{\vee} \dot{u}_1 \vee y_1 \vee \overline{x_1x_2} u_1^{\vee} \dot{u}_2$ which is equivalent to the sub-expression in parentheses in the expression (4.18). In the working phase, when $a = 1$, the completion condition is $u_1^{\vee} \dot{u}_1 u_2^{\vee} \dot{u}_2 = 1$. Thus, we obtain (4.18).

After the analysis of the structure shown in Fig. 4.16, the idea may occur to us that the interaction between the machine and the environment can be organized as presented in Fig. 4.17.

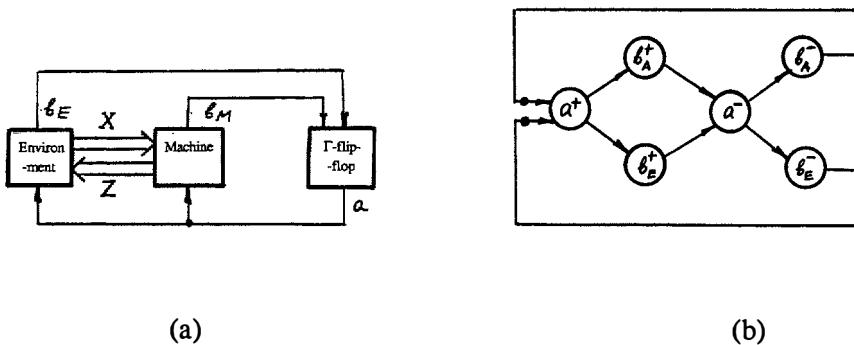


Figure 4.17(a) and (b). An organization of the interaction between a state machine and the environment: (a) block diagram; (b) signal graph.

The design strategy for the above canonical implementations can be summarized in the following steps :

- deriving transition functions and their complements for each of the variables encoding internal states of the machine;
- deriving excitation functions and switching functions;
- analyzing the derived functions and then choosing the most effective technique for the implementation of the excitation (or switching) functions, taking into account the requirement of the matching between

- external input spacers, memory element input spacers and transient states of memory element flip-flops;
- constructing a common indicator.

4.6 Implementation with multiple phase signals

The “machine-environment” model was considered in Section 4.2 within the framework of the assumption that transient processes in the machine and the environment mimic their completion by changing a unique signal – the indication signal in the machine, and the phase signal in the environment. In such a model, the machine and the environment are connected to each other, not only by the data buses, but also by a special pair of control (phase) lines – the request line and the acknowledgement line – thereby supporting the matched implementation discipline.

In this section, we discuss a *more general model* which, apart from the data buses, has more than one pair of control lines. *The discipline of changing signals on these lines is as follows:*

- combinations of signal values on the control buses can be changed only to adjacent combinations,
- the transient process in the machine is initiated by changing the value of the signal on just one of its input control lines,
- the completion of the transient process in the machine is indicated by changing the value of the signal on just one of its output control lines.

The above assumptions imply the following sequencing dynamics of the machine-environment model.

The first step is performed by the environment, which applies a combination at the input data bus and then changes the value on one of the input control lines. This causes the transient process (the second step) in the machine to flow, which involves two serial sub-steps. The first sub-step ends with producing the data at the output data bus. The second sub-step terminates with changing the value of one of its output control signals. Then the environment goes back to the first step: applies a new combination at the input data lines, and then changes the value of one of the input control signals, thereby initiating the next transition in the machine etc.

The structural implementation of this model can be achieved through *decomposing the machine into a control part and an operational part*. The realization of this concept, however, differs from that of V.M. Glushkov. By the *operational machine* we mean a device, capable of converting a sequence of input symbols to a sequence of output symbols, but which also has a pair of control signals and, hence, could be realized as an ordinary matched machine (see 4.1.1). The main function of the *control machine* is to convert the given discipline of changing the values of input control signals into the discipline of changing the values of output control signals that

will be appropriate for the operational machine. In the canonical matched implementation model, the value of the control input always corresponds to one operating phase of the operational machine. In this model, however, one change of signal values on the control lines is associated with both of its operating phases. To indicate the transition on input control lines, if the adjacent discipline is used, we can exploit the technique of incorporating a modulo 2 adder for the input variables. For an adjacent input transition, the adder's output will perform either a 0-1 or a 1-0 transition. In this case, we require that the phase signal of the operational machine will switch twice (0-1-0) on every change of the input value. The solution can be reached if we apply both the input and output signals of the machine to the modulo 2 sum circuit. In fact, in the stable state, let the modulo 2 sum of these two signals be 0. After the change of the input value to the adjacent one, the sum goes to 1, and then, after the next adjacent transition, it will return to 0.

The circuit implementation of the control machine must certainly provide the completion indication of both phases in the operational machine as well as the transition of the control machine to the following state.

An example of a control machine structure satisfying the above requirements is shown in Fig. 4.18, where x_1, x_2, \dots, x_n are input control lines and y_1, y_2, \dots, y_m are output control lines, and flip-flops \dot{T}_i , $1 \leq i \leq n$, have all-one transient states, whereas flip-flops \hat{T}_i and those constituting the transition function circuit have all-zero transient states.

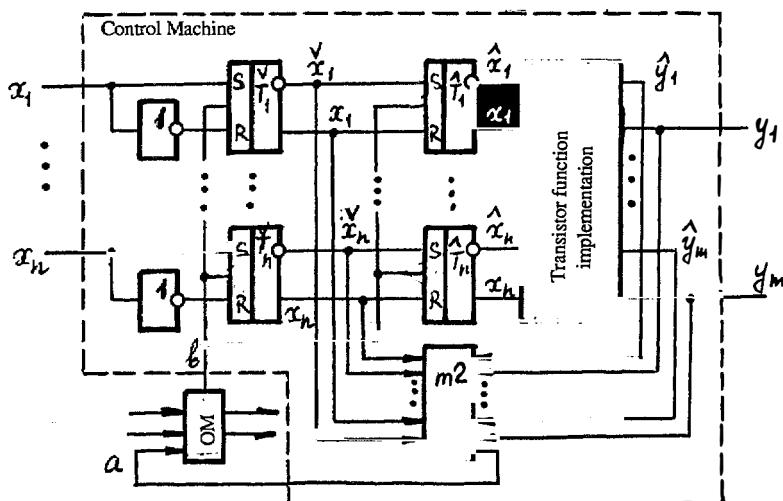


Figure 4.18. Block diagram for the control machine with several control signals. OM – operational machine.

Assume, in the initial state, $a = x_1 \oplus \dots \oplus x_n \oplus y_1 \oplus \dots \oplus y_m = 0$, $a = b = 0$, and flip-flops \check{T}_i are in the states which comply with the values of inputs x_i , $1 \leq i \leq n$, and the operational machine is in the idle state. Changing the input combination, say, in the position of x_j , will result in changing the state of flip-flop \check{T}_j .

The adder of modulo 2, with the output a , may be built in such a way as to make the change in its output value be produced only after \check{T}_i comes to the stable state. As soon as $a = 1$, the operational machine will pass into the working phase and, after the setting of data outputs, b becomes equal to 1. Then the data is reloaded from \check{T}_j to \hat{T}_j , which results in initiating the transition of the control machine to a new state. The transition process consists of changing the state of one of the flip-flops, say, y_k , and the 0–1 transition of y_k indicates the completion of the process in this flip-flop. During the 1–0 transition, the appearance of 0 on the output y_k does not mean that the transient process in this flip-flop has been completed, because the latter may stay in a transient state.

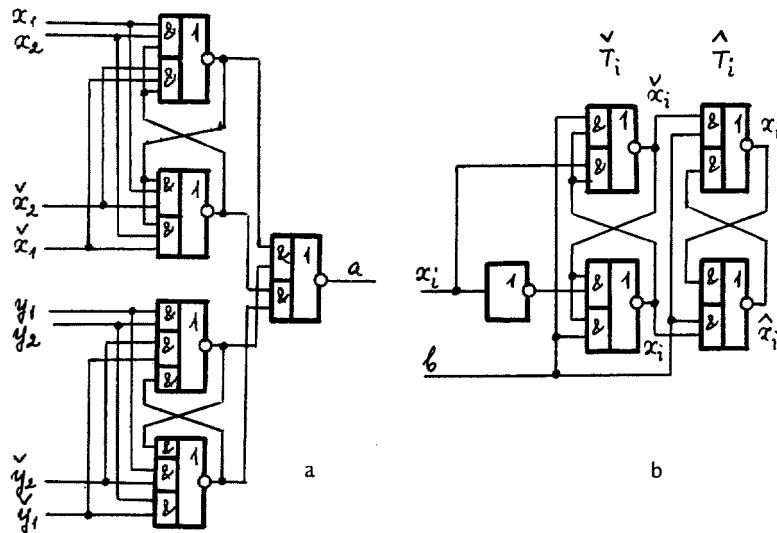


Figure 4.19 (a) and (b). (a) A modulo 2 adder for $n = 2$ and $m = 2$; (b) the implementation of flip-flops \hat{T}_i and \check{T}_i with two control signals.

The 1–0 transition on the output a of the modulo 2 adder is possible only after the setting of the flip-flop to the stable state. Then it is possible to change the data

and control inputs of the machine. However, the loading of new values in flip-flops \hat{T}_i can be done only after the completion of the idle time in the operational machine, i.e. when $b = 0$. After that, the next transition may begin.

The implementation of the modulo 2 adder for $n = 2$ and $m = 2$ and flip-flops \hat{T}_i to \check{T}_i satisfying the requirements indicated, is shown in Fig. 4.19(a) and (b), respectively. An approach to the implementation of the transition functions for the control machine will be demonstrated later, in Section 7.4.3.

4.7 Implementation with direct transitions

In this section, we turn our attention towards the case when *both input and output signals are encoded by the self-synchronizing code with direct transitions* (see Section 3.6). The corresponding structure is shown in Fig. 4.20, which depicts a composition of a register for input variables X with an indicator producing a and a matched implementation with the phase signal a and indicator b . In this structure, the double arrows stand for data buses and the single arrows designate control lines.

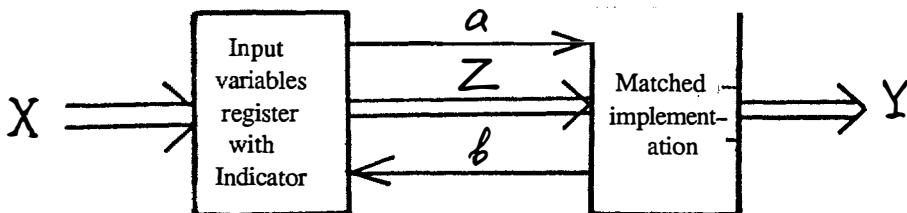


Figure 4.20. Block diagram for an implementation with direct transitions.

Before applying the next input combination, the matched implementation is in its working phase ($a = 1, b = 1$), and the register is in the phase of storing the previous input combination. After changing the input data, a is reset to 0, and the implementation goes to the store phase, during which it will store the output value Y . When b becomes equal to 0, another change of data inputs Z is allowed, and the next input value is loaded into the input register. After this, $a = 1$. Then the implementation passes into the working phase where the internal state is changed and the new output value produced. The end of the phase is marked when $b = 1$, whereupon the circuit is back to its initial state.

A finite state machine can be defined in any way, say, using a transition table (not necessarily normal) or automata equations. For the sake of simplicity, we assume that the machine to be defined is a Moore model machine, i.e. its internal states are, at the same time, its output values. The matched implementation can be built by any of the techniques described in Section 4.5.

In the event that the matched implementation should be interfaced with another one, also having the structure of the type shown in Fig. 4.20, the states of the machine are assigned the combinations of the direct-transition code (see Section 3.6). Given a transition table, we can derive for each state $q_j \in Q$, $1 \leq j \leq m$, a sub-set $Q(q_j)$ that the machine may reach from the state q_j . For the set of states Q and the sub-set $Q(q_j)$, we can make an assignment according to the method presented in 3.6. After that, the matched implementation is constructed as shown in Fig. 4.20.

For example, the register bit-cells can be built using the circuit of Fig. 4.11(c).

Here, the j -th bit-cell of the register consists of a flip-flop \hat{z}_j and an indicator \bar{d}_j . The phase signal, denoted by a in Fig. 4.11(c), allows the register to accept the input data, but, in this case, it should be connected to the indicator output b of the matched implementation. In the write phase, $b = 1$, and in the store phase $b = 0$.

To define an indicator for the completion of a transition $a-b$, we should first form an n -tuple $\delta_1 \delta_1 \dots \delta_n$ such that

$$\delta_j = \begin{cases} d_j, & \text{if } a_j \neq b_j \\ 1, & \text{if } a_j = b_j, \end{cases}$$

and then the term $\delta(a, b) = \bigwedge_{j=1}^n \delta_j$. This term thus contains those d_j which correspond to the variables whose values change in the transition, i.e. to the variables included in the variation term (see Section 3.1). Let d_j be realized by the modulo 2 adder. Then we define a function $f(a, b) = \omega(a) \delta(a, b)$ where $\omega(a)$ was defined in Section 3.1. It is easily seen that after the completion of transition $a-b$ $f(a, b) = 1$.

EXAMPLE 4.5. Let $a = 010$ and $b = 100$. Then $\omega(a) = \bar{x}_1 x_2 \bar{x}_3$, $\delta(a, b) = d_1 d_2$, and $f(a, b) = \bar{x}_1 x_2 \bar{x}_3 d_1 d_2$.

The 1-0 transition of the register indicator a is possible when one of the functions $f(a, x)$ is equal to 1 and $b = 0$, i.e. if

$$u = (\bigvee_{a \in Z} \bigvee_{x \in Z(a)} f(a, x)) \bar{b} = 1.$$

The reverse transition, from 0 to 1, is possible when $b = 1$ and the register loading process is completed (all functions of the form $f(a, x)$ and all d_j are equal to 0), i.e. if

$$w = (\bigvee_{a \in Z} \bigvee_{x \in Z(a)} f(a, x) \vee \bar{b} \vee \bigvee_{j=1}^n d_j) = 0.$$

These two conditions help us to derive the indicator function in the form

$$\check{a} = \overline{\overline{w} \vee \bar{u} a}.$$

4.8 On the definition of an aperiodic state machine

The discussion up to this point has been concerned with aperiodic implementation of a finite state machine, but the notion of an aperiodic machine, as such, has not yet been defined. The following can be regarded as an example of such a definition.

DEFINITION 4.7. Let an allowed transition $a-b$, $a \in A$, $b \in B$, be applied to the inputs of a machine as defined by the Moore model. Let this transition initiate an allowed transition $c-d$ between two internal states of the machine $c \in C$, $d \in D$. If, for every $t \in (a, b)$ and $q \in (c, d]$, the system of functions is such that $\lambda(t, q) \neq d$, then we shall say that the *machine indicates transition a-b*, and if this condition holds for all allowed transitions, we shall call that *machine indicatable*.

This definition may be interpreted as a requirement that the assignment for both input symbols and internal states should be made by means of a self-synchronizing code in such a way as to provide that the next internal state d can only be reached after the application of a new input combination has been completed. Furthermore, during such a transient process, none of the transient states may coincide with one of its possible stable states.

As far as the Mealy model machine is concerned, such a machine can be represented as a composition of the Moore machine and the output converter, which will be a combinational circuit. Thus, the generalization of the indicatability notion to the case of the Mealy model is fairly trivial.

The concept of indicatability is, in some sense, a generalized version of the issue requiring the absence of functional hazards and critical races in input signal changes and internal state transitions, respectively. Adding to these requirements, the absence of logical hazards with respect to arbitrary gate delays (the assumption about the character of delays was given in a more exact way in Section 4.1, item 8), we obtain the following definition.

DEFINITION 4.8. A finite state machine is called *aperiodic (self-timed)*, if:

- 1) it is indicatable,
- 2) there exists an implementation whose behaviour is invariant to the values of gate delays.

The reader may convince him/herself that the constructions presented in this chapter can be described by the model of an aperiodic machine. In particular, the two-phase implementation of a sequential machine is adequate for this model.

Thus, in this chapter, we have presented the material which, it is hoped by the authors, will help the designer familiar with classical automata theory and logic design principles, to carry out the structural synthesis of a finite state machine from its specification (by transition or flow table, or by automata equations), i.e. to construct a circuit which will implement it by means of integrated logic components. Perhaps, the reader is sure that he/she could do this without any prompts, and hence, the authors should not have “entered the lists”. However, this is a somewhat superficial point of view, because the major goal of the entire text, as well as that of this chapter, is to provide some aid in acquiring basic skills in synthesizing not synchronous, or clocked, machines, but essentially asynchronous ones, that operate correctly with arbitrary element delays. It seems quite important that this thesis is repeated as it is a strong motive for revising some classical methods and developing substantially new techniques. A self-timed (aperiodic) circuit to realize a finite state machine is built as a composition of a combinational circuit, flip-flops and indicators. Preserving the indicatability condition ensures the correct operation of the machine with the environment, which must comply with the machine. (For the classical approach, it is “essential” that the question of compliance is ignored in practice.) For that, we need specific techniques for encoding both data and internal states such as those, presented earlier, in Chapter 3.

4.9 Reference notations

An aperiodic state machine, and some of its structural models, have been defined in [6], [52] and [68]. A definition of a matched (compliant) machine was introduced in [145] and specified in more detail in [144].

The notion of a checker is common in engineering diagnostics where it is usually associated with the idea of a built-in error checking device, as can be found in [111].

The model of a Γ -flip-flop, also known as a C-element, from the pioneering work of Muller [272], appears in many works [120], [175] and [274], to name but a few. The implementation of the Γ -flip-flop on an AND-OR-NOT basis has been suggested in [6].

The indicatability concept and an approach to asynchronous circuit indicatability analysis were given in [60]. These use the idea of a Boolean function derivative [58], [137] and [300].

The concept of a canonical implementation is due to M.A. Gavrilov [3] and [70]. The discussion of canonical implementations of combinational aperiodic

circuits and aperiodic state machines built with D- and T-flip-flops, as well as the implementations for the flip-flops themselves, is based on [6], where a thorough analysis of functional capacities depending on the parameters of their elements can be found. One of the implementation techniques for combinational circuits based on Γ -flip-flops is proved in [19].

Techniques for the indication of transient process completion in circuits were inspired by [13] and the idea of D.E. Muller [120].

The implementation strategy based on flip-flops with separated inputs was described in [144]. The strategy based on the direct-transition approach can also be found in [123].

The idea of using a modulo 2 adder for the case of adjacent transitions was inspired by [205].

Earlier discussions on the structural organization of aperiodic machines using balanced and double-rail codes have been outlined in [5], [6], [66], [92], [173], [195] and [236]. Unfortunately, these structural models are far from being elegant. Furthermore, the model in [195] has the following drawback. Because of the awkward assignment, the environment cannot be realized without using built-in delay circuits. Turning from asynchronous to aperiodic circuits helps to overcome a number of conceptual problems inherent in asynchronous sequential machines. This was indicated in [231], [240], [257] and, in more detail, in [144].

Discussion of the issue that aperiodic circuits are free from hazards can be found in [6], [57], [183], [184] and [276].

CHAPTER 5

CIRCUIT MODELLING OF CONTROL FLOW

I realize very clearly what time is until I am asked to explain what it is. I always lose the meaning when I attempt the explanation.⁽¹⁾

Saint Augustine.

In the preceding chapter, we discussed techniques for constructing a self-timed circuit given by the model of a finite state machine. These are certainly not the only possible techniques: for example, we can also specify the behaviour of a synthesized circuit, by means of flow charts, cyclo-diagrams, timing diagrams etc. In Chapter 2, we described a group of behavioural models among which the asynchronous process (AP) was fundamental, and the other models were generated by AP as particular interpretations of the latter. In this chapter, we shall discuss methods for constructing self-timed circuits using the terminology presented in Chapter 2.

In Section 2.3, we saw how the Muller model could be used to describe a circuit by associating a system of Boolean equations with circuit elements whose inherent functions are defined in these equations. Provided that the assumption of the character of the delays is true, and given an initial state, this circuit would model an AP and, hence, may be referred to as a *modelling circuit*.

The use of modelling circuits opens up wide opportunities for applying related techniques for AP analysis. On the other hand, since the modelling of an AP is not the ultimate goal in itself, but rather an intermediate step in solving a more general problem – to construct a device for a given AP – the modelling circuit may be regarded as a basis for the implementation of such a device, or to be more exact, its *control part*.

Let us assume that we have to model the behaviour of a system represented as a composition of n modules of the type B_i , each of which is realized as a matched aperiodic state machine (see Chapter 4), i.e. in addition to some data inputs and outputs, it is also equipped with two control signals, input signal a_i (“request”) and output signal b_i (“acknowledge”). Signal b_i is supposed to be the output of the machine’s indicator. *In its operation that module, is virtually, a delay element function with respect to signals a_i and b_i .* The value of this delay is unknown (unpredictable) and hence, unbound, but assumed to be finite.

Let an AP be given such that it will describe the interaction of modules in a

⁽¹⁾ In the original: “Quid ergo est tempus? Si nemo ex me querat scio; si quaerenti explicare velim, nescio”.

modelled system. Let it, also, belong to the class of controlled APs (see Chapter 2). Its situations are assigned binary n -tuples such that the i -th component of each situation may assume the value of either 1 or 0. The operation of AP consists of changing the values of components in n -tuples. We associate the i -th module of the system with the i -th component in a situation. Here, it is assumed that if the i -th component is equal to 1, then B_i is in the working phase, and if it is equal to 0, this implies that B_i is in the non-working (idle) phase. For such an interpretation of the Muller model, a state-transition diagram (see Section 2.4) can represent a set of sequences of changes in the system module's phases in much the same way as a circuit, realizing an appropriate system of equations, represents the system's behaviour. If the modelling circuit is semi-modular, i.e. if it generates a semi-modular transition diagram, then the module B_i is allowed to be interconnected in series with the i -th element (the output of the i -th element is connected to the input a_i) because such an interconnection *will not alter the original order of switching of the elements*. The latter is guaranteed by the fact that the functioning of a semi-modular circuit does not depend on the values of element delays and this remains true even when an arbitrary delay is introduced with the insertion of module B_i . Thus, any composition of the above kind between a modelling circuit and modules realizing the associated sub-processes will be a realization of a given AP. In some cases, that composition requires that the semantics of an asynchronous process are taken into account, for example, the passing of data flow through the modules. However, the general idea of interconnection between operational modules and associated elements of a modelling circuit would be the same. If an AP is given in the form of a state-transition diagram, the latter can be transformed into a modelling circuit by the technique presented in Section 2.4.

Later in this chapter, we shall see how methods for constructing modelling circuits can be derived for other AP interpreted notations such as Petri nets and parallel asynchronous flow charts.

5.1 The modelling of Petri nets

In this section we consider APs defined by *safe and persistent Petri nets*. The limitation of safeness and persistence is related to the class of processes involved. Firstly, it allows us to interpret a marking of a Petri net in the most adequate terms and, secondly, facilitates obtaining semi-modular modelling circuits.

5.1.1. EVENT-BASED MODELLING

Let an occurrence of a Petri net event be interpreted as a switching of certain element in a modelling circuit, i.e. as a change of signal value of the element's output. The

value of the input of an excited element does not comply with its output value, whereas that of a stable element does. Any event with a single input condition is modelled by a delay element, while an event with several input conditions, which are not shared with other events, can be modelled by a Γ -flip-flop with an appropriate number of inputs, as shown in Fig. 5.1.

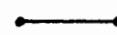
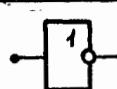
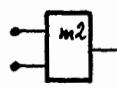
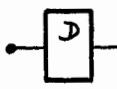
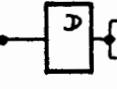
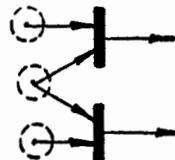
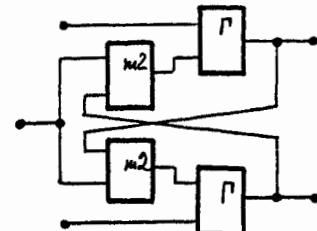
Petri Net Fragment	Fragment's Reference	Modelling Circuit Element
	a	
	b	
	c	
	d	
	e	
	f	
	g	

Figure 5.1 (a) - (g). Correspondence between fragments of a safe, persistent Petri net and circuit elements.

It is quite obvious that those fragments of a Petri net which contain events having input conditions shared with other events, are the most difficult to model adequately in a circuit. To reduce the number of possible types of fragments, we shall restrict ourselves to the modelling of so-called *simple Petri nets*, i.e. nets such that, being safe and persistent, only have events with, at most, one input condition shared with other events. A typical fragment of a simple net with a shared condition is shown in Fig. 5.1(g). Allowing, for this fragment, only those markings that do not violate safeness and persistence of a net, we can depict a state-transition diagram corresponding to this fragment as shown in Fig. 5.2.

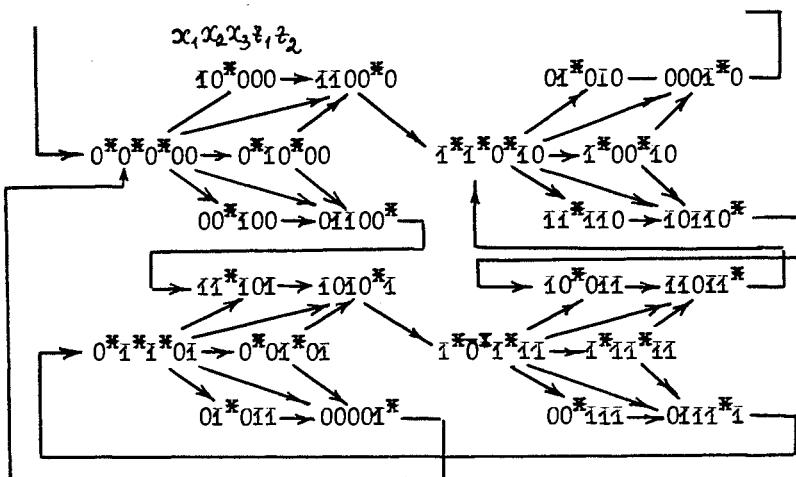


Figure 5.2. State-transition diagram for the fragment (g) in Fig. 5.1.

The first three positions in code combinations in this diagram are associated with the inputs of the implementation, and the other two positions stand for the outputs. Although some of the combinations are labelled with asterisks (*) in the first and third positions, there are no transitions in this diagram that correspond to their simultaneous changing, because such a changing would violate the accepted assumption of persistence for the Petri net of Fig. 5.1(g). Denoting inputs as x_1, x_2, x_3 and outputs as z_1, z_2 in this diagram, we derive

$$\begin{aligned}
 z_1 &= z_1(x_1 \vee z_2 \bar{x}_2 \vee \bar{z}_2 x_2) \vee \bar{z}_2 x_1 x_2 \vee z_2 \bar{x}_1 \bar{x}_2 = \\
 &= z_1(x_1 \vee (z_2 \oplus x_2)) \vee (z_2 \oplus x_2)x_1, \\
 z_2 &= z_2(x_3 \vee (z_1 \oplus x_2)) \vee (z_1 \oplus x_2)x_3
 \end{aligned}$$

by the method presented in Section 2.4. Since these functions are not monotonous in their arguments, their implementation is fraught with problems; for example, it is necessary to use *non-inertial (with zero inertial delay) modulo 2 adders* (see Fig. 5.1(g)).

Let us now turn our attention to the implementation of Petri net conditions. If one of the output conditions of an event is, at the same time, the input condition of another event, the modelling circuit must contain a pair of elements, corresponding to these two events, that are connected to each other in series – either directly, if the condition is false (contains no tokens), or through an inverter, if the condition is true (contains a token). This is illustrated in Fig. 5.1(a) and Fig. 5.1(b). If the output condition is shared between several events, then it may become true as a result of the firing of any of these events.

The safeness of the Petri net then guarantees that at most one of these events can fire. In a related circuit, the switching of any one of the associated elements should always cause the change of the output value of an element realizing the event for which the indicated condition is the input one. The condition, in this case, can be realized by a modulo 2 adder, as shown in Fig. 5.1(c).

Since the implementation of the conditions shown in Fig. 5.1(a) to Fig. 5.1(c) act as interconnection lines between elements, their delays, must satisfy the same constraints as the wire delays specified in Section 4.1. Thus, both circuits of Fig. 5.1(c) and Fig. 5.1(g) must contain non-inertial modulo 2 adders, i.e. the adders with zero delays.

The collection of modules shown that implement fragments of simple Petri nets is intended for an arbitrary discipline of changing the marking of the input conditions of a given fragment. It certainly does not disturb the safeness and persistence conditions. This would allow the isolation of each of these fragments, in any simple Petri net, without special analysis of its operation. Nevertheless, such an analysis can sometimes establish that a particular discipline is satisfied for a given fragment. This may result in the situation when, during its normal operation, a related modelling circuit passes through only a sub-set of all the allowed combinations of input and output values. Taking this fact into account helps to simplify the implementation for that modelling circuit. For example, the module shown in Fig. 5.1(c) may, in some cases, be replaced either by an AND gate or by an OR gate, while for the module shown in Fig. 5.1(g), we may sometimes use either two Γ -flip-flops with one common input or an aperiodic complementing flip-flop.

5.1.2. CONDITION-BASED MODELLING

Assume that we need to construct a modelling circuit in which *a particular type of element switching, say, the 0–1 transition of the element's output value, is associated with an event of a given Petri net*. In this case we must also ensure that the reverse ($1 - 0$) transition of the element's output occurs before the next 0–1 transition becomes enabled again. *The method is based on the concept that for each Petri net condition we assign a flip-flop in the modelling circuit.* The flip-flop is in the “1” state if the associated condition is true, otherwise it must be in the “0” state. (Using the registers as alternatives to some flip-flops, we can model non-safe, but bounded Petri nets.) Thus, the firing of an event in a Petri net is represented as setting to “1” for flip-flops associated with the output conditions of that event, and resetting to “0” for flip-flops related to its input conditions.

We begin our discussion of that method by considering a modelling circuit shown in Fig. 5.3. It has been constructed for the Petri net presented earlier in Fig. 2.8(a), with the exception that reposition through the event r is excluded.

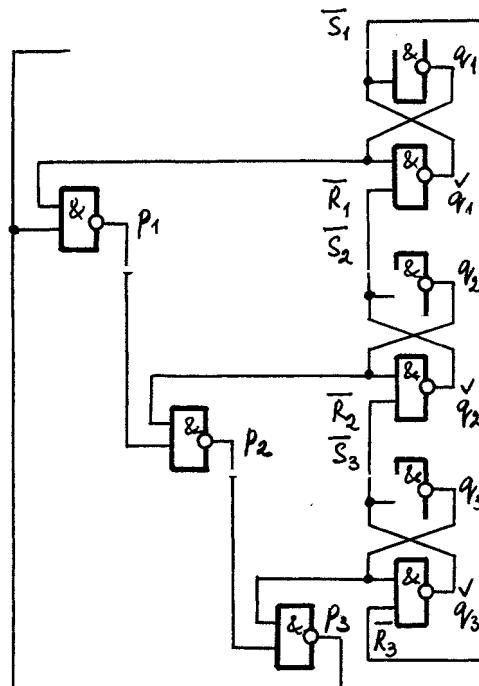


Figure 5.3. Shifter circuit built of David's elements (flip-flop plus NAND gate enclosed within the dashed box) that models the Petri net shown in Fig. 2.8(a).

This circuit consists of three RS-flip-flops \check{q}_j , $j = 1, 2, 3$, realized on NAND gates and three gates p_j such that $p_j = \overline{p_{j-1} q_j}$, $\bar{S}_j = p_{j-1}$, $\bar{R}_j = \check{q}_{j+1}$ where R_j and S_j are the excitation functions of the j -th flip-flop (here, both the addition and subtraction for subscripts are modulo 3 operations).

Assume that the circuit is initially set to the state in which the first flip-flop set to 1, i.e. $q_1 = 1$, $\check{q}_1 = 0$ and the other two flip-flops are both set to 0 ($q_2 = q_3 = 0$, $\check{q}_2 = \check{q}_3 = 1$). It can be shown that with respect to that state, the circuit of Fig. 5.3 is semi-modular. Furthermore, it is totally sequential. In the initial state $p_2 = p_3 = 1$, $p_1 = 0$ and, hence, the only excited element is the gate with the output q_2 . Since all other elements are in the idle state, q_2 remains excited until its output changes. After this change, the next element to be excited is \check{q}_2 . Similarly, \check{q}_2 will go to the idle state only after its output changes. Thus, the flip-flop \check{q}_2 will be set to 1 ($q_2 = 1$, $\check{q}_2 = 0$) thereby making the element \check{q}_1 excited, and when its output changes to 1, q_1 will then go to 0. As soon as q_1 is equal to 0, the element p_1 becomes excited, and after $p_1 = 1$, p_2 will go to 0. This state, with p_2 equal to 0 is analogous to the initial state, the only difference being that the "1" has been shifted from flip-flop \check{q}_1 to flip-flop \check{q}_2 . Due to the uniformity of the circuit, the process of the shifting the "1" from \check{q}_2 to \check{q}_3 , and further from \check{q}_3 back to \check{q}_1 , will be executed in the same way and, therefore, during the full operation cycle every element of the circuit, having once become excited, does not become idle until, and unless, its output changes its value. From this feature, we may deduce that this circuit is semi-modular. Because of our assumption that each occurrence of an event in a Petri net is associated with the shifting of the "1" from the "input condition flip-flop" to the "output condition flip-flop", the event will be regarded as the fired event if the "output condition flip-flop" is set to 1 and the "input condition flip-flop" is reset to 0.

The above description of the operation of a modelling circuit exhibits an obvious difference between the Petri net semantics and that of the associated circuit. The firing rule in Petri nets implies that both the input and the output conditions of a firing event change their state simultaneously, whereas, in the modelling circuit, the corresponding "firing" process is by no means an atomic action because it consists of a pair of separate sub-processes. The first of these amounts to a change in the "output condition flip-flop" and the second is associated with the change in the "input condition flop-flop". Therefore, strictly speaking, the accepted interpretation does not fully preserve the semantics of the original specification of a system.

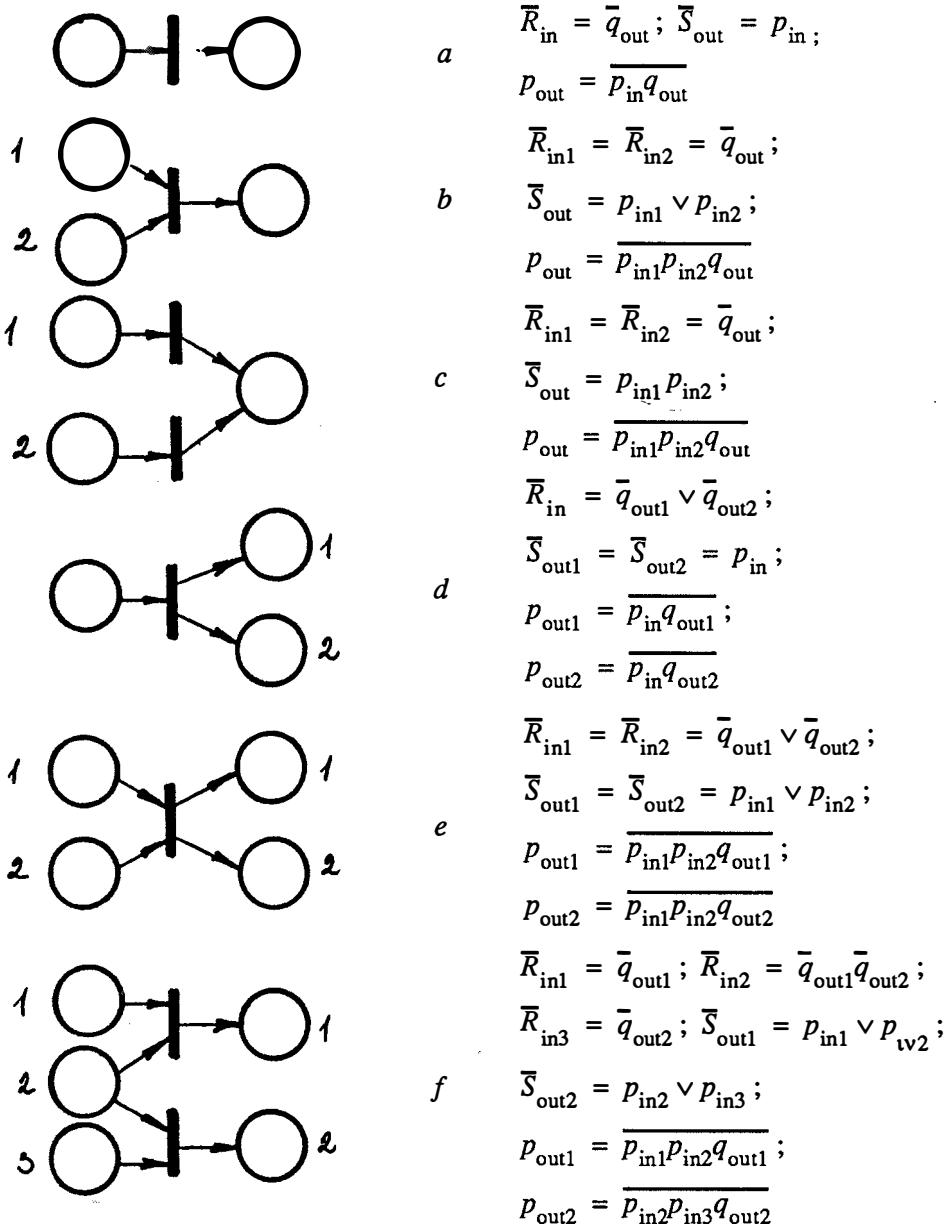


Figure 5.4 (a) - (f). Correspondence between fragments of a persistent, safe Petri net and their implementations by David's elements.

Nevertheless, the ordering relation between firings of different events in a Petri net is preserved in exactly the same form in the modelling circuit.

The circuit shown in Fig. 5.3 realizes a totally sequential process that pertains to the simplest form of connection between two conditions: as soon as a single input condition is true, an event becomes enabled, and, after its firing, a single output condition is set to be true simultaneously with the first condition being reset to false. If an event has multiple input/output conditions, these conditions will be related to RS-flip-flops with multiple inputs. Their excitation functions will thus depend upon the type of Petri net.

Fig. 5.4 (a) - (f) shows several typical fragments of interconnections between conditions that can be found in persistent and safe Petri nets. The fragment given in Fig. 5.4(a) has just been discussed. In Fig. 5.4(c), two events have the same input condition in common. To be compliant with the safeness requirement, these two events must not be enabled together. Bearing this constraint in mind, we obtain a simplified function for setting the output condition flip-flop. This flip-flop is set to 1 if either of the input condition flip-flops is set to 1. After the output condition flip-flop goes to 1, the resetting signal is applied to both of the input condition flip-flops, although it is actually only used in one of them.

Fig. 5.4(b) shows the event which has two input conditions. The implementation of this event requires that the output condition flip-flop is set as soon as both of the input condition flip-flops are set to 1. The reset function for the input condition flip-flops is the same as in the preceding case.

Fig. 5.4(d) shows the situation when the event has two output conditions. Flip-flops for these conditions have the same set function as in the simplest case in Fig. 5.4(a). The resetting of the input condition flip-flop is done as soon as both of the output condition flip-flops are set to 1.

Fig. 5.4(e) can be regarded as a union of the two fragments shown in Fig. 5.4(b) and Fig. 5.4(d). Associated set functions for the output condition flip-flops are the same as those for the circuit corresponding to Fig. 5.4(b), while the reset functions for the input condition flip-flops are the same as those for Fig. 5.4(d). The fragment in Fig. 5.4(f) is identical to the fragment in Fig. 5.1(g). The set and reset functions of the flip-flops related to output and input conditions for this fragment can also be easily derived.

The proposed method for synthesizing modelling circuits imposes the following constraints on the shape of Petri nets representing initial specifications for such synthesis.

1. It is easily seen that circuits of that kind operate correctly if they do not contain loops consisting of less than three flip-flops. In fact, if the circuit has a loop with, say, two flip-flops, a deadlock state can be reached in which both these flip-flops can be set to 1 and there is no way to determine which of them is to be the

first to be reset to 0. This constraint, relevant for a modelling circuit, implies that a corresponding Petri net should not contain closed loops with less than three conditions (events) in series.

2. Petri nets should not contain events such that they have output conditions in common with other events, and, at the same time, having more than one input condition. The reason for that constraint can be outlined using a fragment of the Petri net shown in Fig. 5.5.

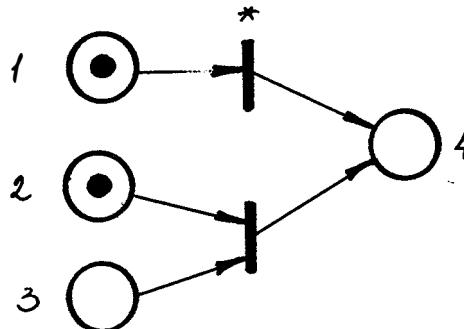


Figure 5.5. Illustration of a constraint on condition-based modelling.

With respect to the given initial marking (conditions 1 and 2 are true, while conditions 3 and 4 are false), perhaps, this fragment is persistent and safe. The event marked with an asterisk (*) is enabled and, after its firing, the net enters the state where conditions 2 and 4 are true. When implementing this fragment in a similar way to the fragment in Fig. 5.4(b), we may notice that the setting of the output condition flip-flop to 1 will cause the resetting to 0 of all flip-flops associated with the input conditions of those events for which this is an output condition. Accordingly, the flip-flop associated with condition 2 will be reset to 0, and, hence, the circuit will no longer be a correct representation of the semantics of the connections between the conditions in a Petri net.

The problems related to compliance with both these constraints can always be obviated by “inserting”, into a Petri net, some dummy conditions and events that do not change the original order of event firings.

We conclude this section by pointing out that although the methods discussed here allow the synthesis of modelling circuits for any safe and persistent Petri net, we should sometimes also refer to the techniques which are oriented to more traditional notations for specifying circuit behaviour. The next section presents a discussion of the synthesis of modelling circuits from specifications given in the more common notation of flow charts, though the latter are introduced in their “parallel and asynchronous” form.

5.2 The modelling of parallel asynchronous flow charts

The idea of a parallel asynchronous flow chart (PAFC) as an interpretation of an AP was introduced in Section 2.5. However, in constructing modelling circuits for PAFCs we shall need an additional refinement. PAFC is a formal specification of AP that does not fully express the semantics of the latter. Such semantics substantially affects the control flow organization and thus must be accounted for in constructing a modelling circuit. If a PAFC describes, say, a system of industrial control, the change of values of data variables is concerned with the change of states of the controlled plant, but does not affect the process (ordering semantics), and the process specification can be made at the control signal level. However, if the PAFC represents a definition of a computation process, then we must take into account both control and data signals. An obvious example is the multiple use of the same operator in different boxes of the PAFC. In this case, the operator may not be re-initialized until the data produced by it, in a given initialization instance, has been accepted and consumed by all appropriate consumer operators. Another reason for bearing the semantics of processing in mind is that a PAFC represents a general form of interaction between modules in a system, but does not reflect a discipline of control of these modules. Aperiodic modules, as has already been noted, usually have the two-phase control discipline: each module can be in one of the two states (phases), the working and non-working (idle) ones, with respect to the control flow. Furthermore, having received a signal indicating the change of a state, the module enters a transient mode, after which the module generates a completion signal, and remains in a waiting mode until the next signal for a state change is activated. In constructing a modelling circuit, we should ensure such a two-phase discipline.

5.2.1. IMPLEMENTATION OF STANDARD FRAGMENTS

Every AP interpreted in terms of PAFC can be reduced to a composition of the so-called standard PAFCs.

Fig. 5.6 shows four *elementary PAFCs*. All of them have one input arc α (the starting point of an algorithm), and one output arc β (the ending point of an algorithm). For PAFCs we can define a composition rule. PAFC B is called a composition of two PAFCs C and D , if B can be obtained from C by substituting D for one of the boxes (in Section 2.5 called operator vertices) in C . For the composition to be unambiguous, we should explicitly define for which box of the PAFC the substitution is made. Thus, to the PAFC obtained, we can apply the composition rule again and again. The PAFCs which may be generated from elementary ones by such a recursive composition are called *well-formed* PAFCs. A

set of well-formed PAFCs is closed with respect to the composition operator, i.e. the composition of a pair of well-formed PAFCs also yields a well-formed PAFC.

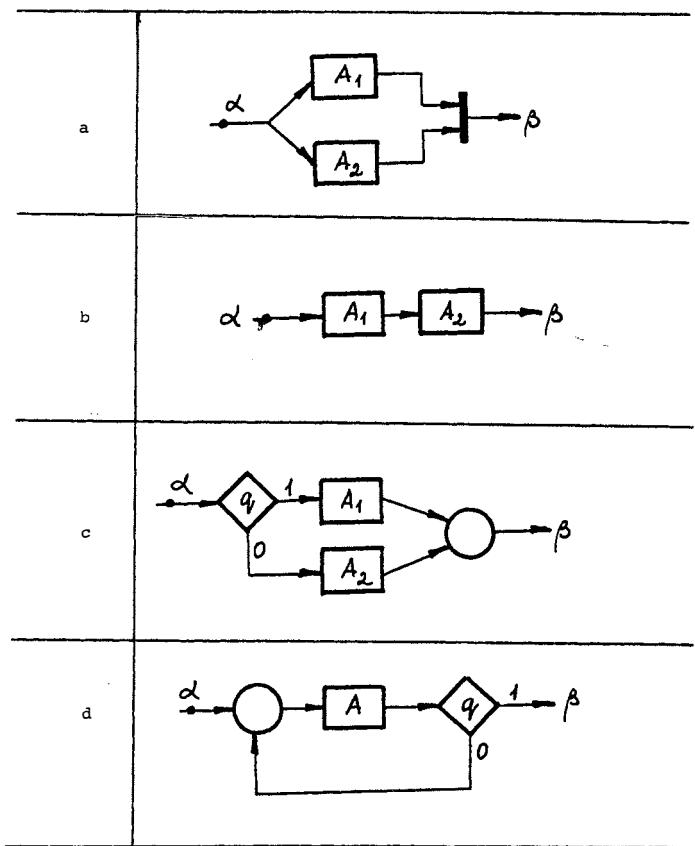


Figure 5.6 (a) - (d). Elementary parallel asynchronous flow charts.

A particular emphasis on this class of PAFCs stems from the fact that a great many PAFCs likely to be met in practice are well-formed. If, on the contrary, a given PAFC is not well-formed (see, for example, Fig. 5.6(a)), it can always be transformed to a well-formed one, which will be algorithmically equivalent (see Fig. 5.6(b)). The transformation is made by inserting some "null" operator vertices, extra variables and conditional branches with checks of the values of these variables. This procedure is quite similar to what is normally done in high level programming when the language does not allow for using loops with two or more entries, or overlapped but not nested loops etc.

We should, however, not conclude that the problem of implementing modelling circuits for non-well-structured PAFC is of no interest. When such an implementation is possible, it may yield more compact circuits than those obtained from the well-formed "equivalents" of original PAFCs. An example of such a design for a PAFC, shown in Fig. 5.7(a), will be examined in more detail.

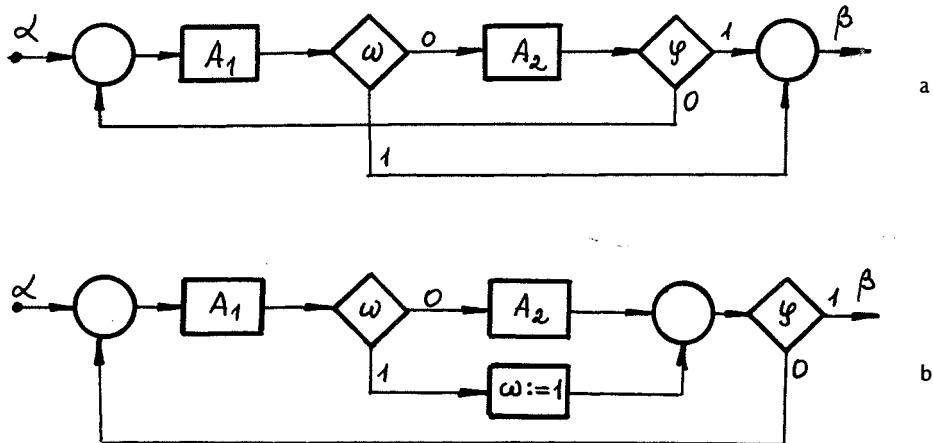


Figure 5.7 (a) and (b). "Non-well-formed" asynchronous flow charts.

Meanwhile, unless another discipline is specially agreed upon, we shall accept that the execution order of processes will be the same for both phases, i.e. after the completion of the first phase of an entire process defined by a PAFC, its second phase will be executed according to the same PAFC. There is a direct analogy between PAFCs and Petri nets which is shown in Fig. 2.13. In this case, the modelling of PAFCs by modelling circuits associated with their fragments will be completely straightforward. Table 5.1 establishes such a correspondence.

PAFC Fragment	Petri Net Fragment
Arc without tokens	Fig. 5.1(a)
Arc with a token	Fig. 5.1(b)
Single-used operator - Fig. 2.8(a)	Fig. 5.1(d)
Conditional branch - Fig. 2.8(b)	Fig. 5.1(g)
Merger - Fig. 2.8(c)	Fig. 5.1(c)
Bifurcator - Fig. 2.8(d)	Fig. 5.1(f)
Synchronizer - Fig. 2.8(e)	Fig. 5.1(e)

Table 5.1

Bringing Table 5.1 and Fig. 5.1 together gives *the implementation of PAFC fragments by corresponding modules of a modelling circuit*. A non-marked (marked) arc of the PAFC is associated with a wire (inverter). A bifurcator is related to a wire branch. A merge vertex stands for an OR element (provided, of course, that both input arcs of the merger may never contain tokens simultaneously). A synchronizer is associated with a Γ -flip-flop. An operator is modelled simply by a delay element being inserted between input and output arcs of the operator vertex. A conditional branch vertex is associated with a switch, whose implementation in a circuit, and symbolic representation are shown in Fig. 5.8 (a) and (b), respectively.

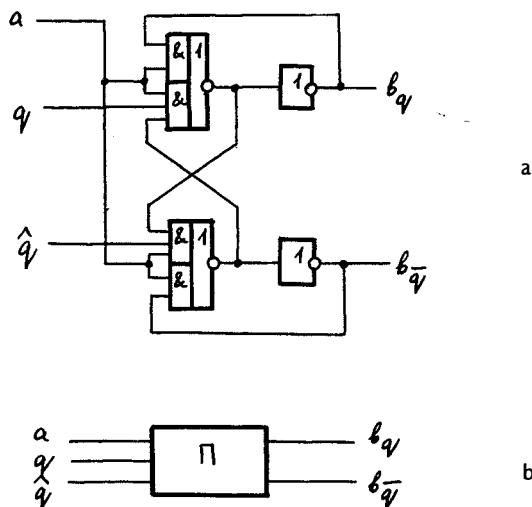


Figure 5.8 (a) and (b). Switch circuit representing (a) a conditional branch, and (b) its symbolics.

The inputs of this module are control signal a and two decision signals q and \hat{q} . In the initial state, $a = 0$. When a becomes equal to 1, the response of the switch is dependent on the value of the decision signal. Return to the initial state is achieved after a changes from 1 back to 0.

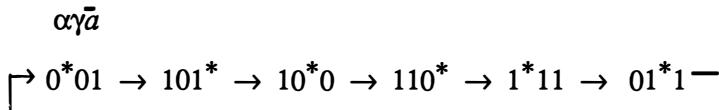
The above implementation technique is applicable for so-called non-repetitive PAFCs, i.e. for the PAFCs in which each operator is used at most once during every operational cycle of the PAFC. However, as has already been stressed, sometimes we need to model multiply-used operators. For this, we should extend our implementation strategy by introducing a special modelling circuit. This circuit will be called a *multiple use circuit*. It should be instituted into the entire modelling circuit, in those places in a PAFC where a particular multiply-used

operator is mentioned. It should, therefore, model the presence of separate copies of the same module. Constructively, there is only one operational module associated with that operator, and this module is controlled by the collection of multiple use circuits, each of which stands for a particular instance of the given operator in the PAFC.

Thus, the design of the modelling circuit for an arbitrary PAFC can be done in a “by-fragment” way in accordance with Table 5.1, with the difference that, for each entry of a multiply-used operator, we assign a copy of the multiple use circuit.

5.2.2. A MULTIPLE USE CIRCUIT

A circuit for modelling the j -th instance of operator A_i in a PAFC has, as its inputs, the output α_{ij} of an entire modelling circuit, and the output b_i of a completion signal of the module B_i which realizes operator A_i . As its outputs, it has the input β_{ij} of the entire modelling circuit, and the input \bar{a}_{ij} of the NAND gate, whose output a_i is used as a phase signal for module B_i . In compliance with the assumptions of the preceding sub-section, the working phase initiated by the change α_{ij} to 1 in the circuit modelling the j -th instance (use) of the i -th operator, should, successively, initiate both operational phases of module B_i ($\bar{a}_{ij} = 0$ and $\bar{a}_{ij} = 1$). In order to distinguish between the states of the circuit when $\alpha_{ij} = 1$ and $\bar{a}_{ij} = 1$ (the first state occurs before the beginning of the working phase B_i , and the other is after the completion of its idle phase), we have to insert an extra internal variable γ_{ij} . In the idle phase, when α_{ij} goes back to 0, the circuit for the j -th instance of the i -th operator must also go back to its initial state without any change of the state of the module B_i . Thus, the circuit operation can be described by the following state-transition diagram



It can be easily seen that the functions for α_{ij} , γ_{ij} and \bar{a}_{ij} obtained from this diagram are non-monotonous and for their implementation we need three flip-flops based on AND-OR-NOT elements.

A better result can be achieved if we construct the multiple use circuit on the basis of a circuit consisting of a pair of RS-flip-flops interconnected to form a loop as shown in Fig. 5.9(a). In this design, a pair of signals z_2 and \dot{z}_2 change their values in the same way as signals α_{ij} and \bar{a}_{ij} indicated in the above diagram. Hence, the implementation shown in Fig. 5.9(a) may be used as a basis for a circuit

modelling the multiple use of an operator. It meets the following requirement: in the j -th initiation of the i -th module only the j -th multiple use circuit should work, while other multiple use circuits must not be activated. The circuit of Fig. 5.9(b) (as well as the circuit obtained directly from the diagram), as such, does not meet the above requirement. Because of this, we have introduced two interlocking connections, shown by dashed lines, thereby preventing signal b_i coming to 1 to cause the change of z_3 to 0 by the inhibiting condition $\alpha_{ij} \cdot z_2 = 0$.

We note that the multiple use circuit, (whose designation is presented in Fig. 5.9(c)), can also be used for the control of whole fragments of PAFCs that may be multiply instantiated. This is provided by the recursive character of the PAFC composition rule.

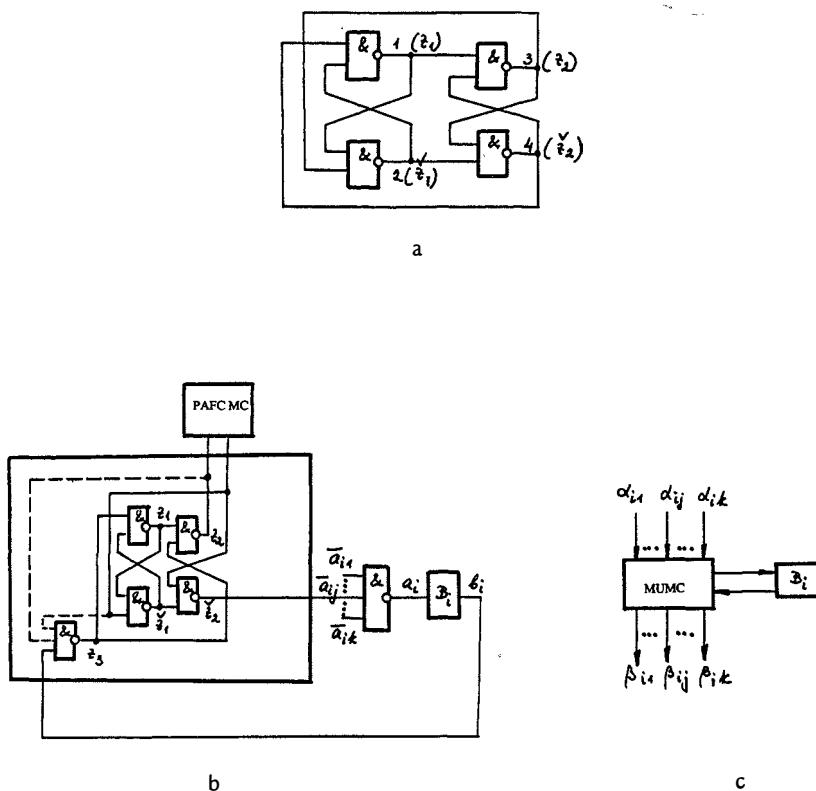


Figure 5.9 (a) - (c). Multiple use circuit: (a) basic construction, (b) implementation and interconnection with a modelling circuit (MC), (c) symbol.

5.2.3. A LOOP CONTROL CIRCUIT

We now turn our attention towards an implementation that, in its working state, can, in much the same way as a multiple use circuit, initiate both phases of a controlled module, and, in its idle phase go back to the initial state without any change of the state of the module. Thus, for constructing a loop control circuit, we may refer to the circuit shown in Fig. 5.9(a).

A *loop control circuit* must have two operation modes that can be expressed in the following statements: “the loop exit condition is true” and “the loop exit condition is false”. In the first mode, the circuit operates as a multiple use circuit (see Fig. 5.9(b)). In the other mode, the circuit reiterates the activation of module B_i (initiates both phases of its operation) until the loop exit condition ϕ is true.

A correctly implemented loop control circuit is demonstrated in Fig. 5.10(a).

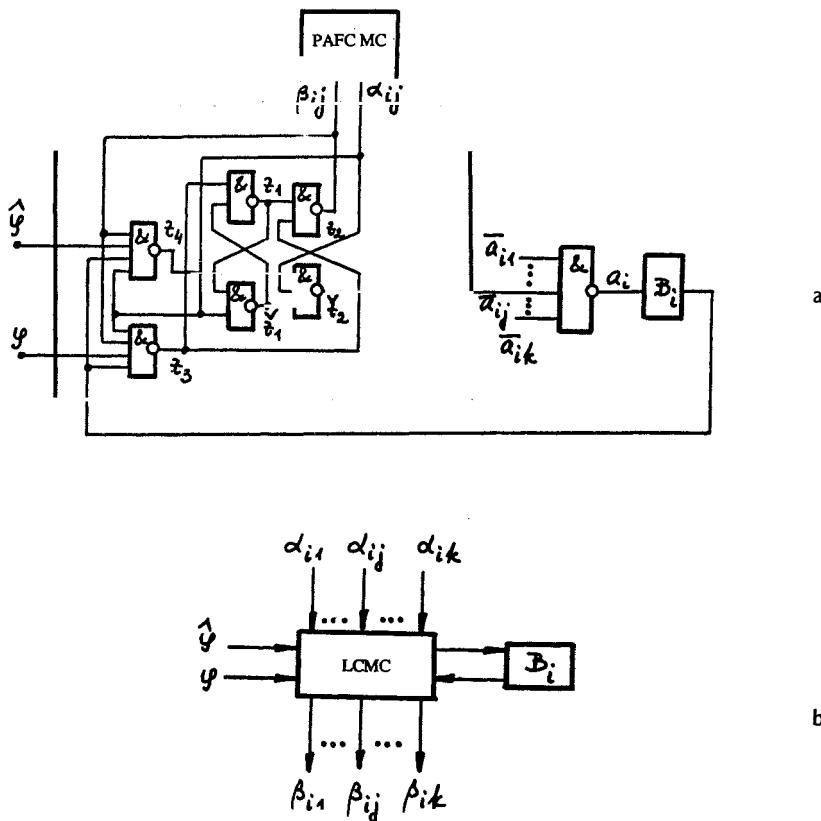


Figure 5.10 (a) and (b). Loop control circuit: (a) implementation and interconnection with a modelling circuit, (b) symbol.

It differs from the circuit in Fig. 5.9(b) in having an additional element z_4 and an extra input ϕ to the element z_3 . To analyse the operation of this circuit we can consider the following three cases:

1. $\phi = 1, \hat{\phi} = 0$. It is clear that $z_4 = 1$, and the operation of the circuits shown in Fig. 5.10(a) and Fig 5.9(b) is identical.
2. $\phi = 0, \hat{\phi} = 1$. Since $z_3 = 1$, the values of outputs z_1, z_2 and z_3 remain fixed for any α_{ij} , and elements $\overset{\vee}{z}_2, a_i$ and z_4 form a ring. Module B_i inserted in this ring cyclically changes its operational phases until the condition $\hat{\phi} = 0$ is true.
3. $\phi = \hat{\phi} = 0$. Here $z_3 = z_4 = 1$, and the circuit remains in a stable state until one of the signals, ϕ or $\hat{\phi}$, goes to 1.

The state-transition diagram for this circuit is presented in Fig. 5.11. It

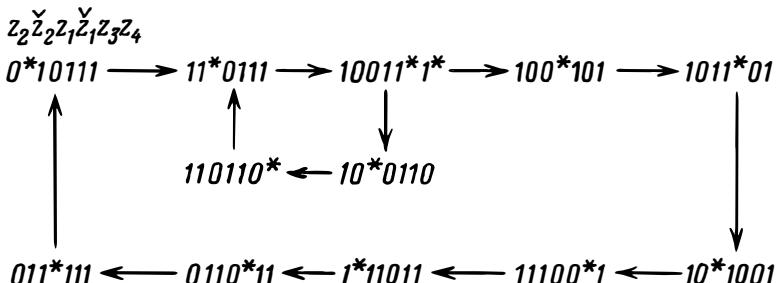


Figure 5.11. State-transition diagram for the loop control circuit.

consists of two main cycles – the “long” cycle, which is executed when $\phi = 1$ ($\hat{\phi} = 0$), and the “short cycle” executed when $\phi = 0$ ($\hat{\phi} = 1$).

The composition of circuits of the type presented in Fig. 5.10(a) for all j , $1 \leq j \leq k$, with element a_i and module B_i is as shown in Fig. 5.10(b).

Making the above modelling circuit a little more complex, we can implement the PAFC shown in Fig. 5.7(a) which specifies the cyclic operation of two modules B_1 and B_2 . As can be seen from this PAFC, the exit from the loop is made possible either after the successive idling of both modules or after the idling of just one of them, namely B_1 . Compared to the circuit of Fig. 5.10(a), the circuit of Fig. 5.12 includes an extra gate z_5 , which generates the initiating signal for the module B_2 .

Another distinctive item is that in order to make it possible to exit from the loop, not only after the completion of B_1 , but also after that of B_2 , we should have inserted an OR-extension in the element z_3 .

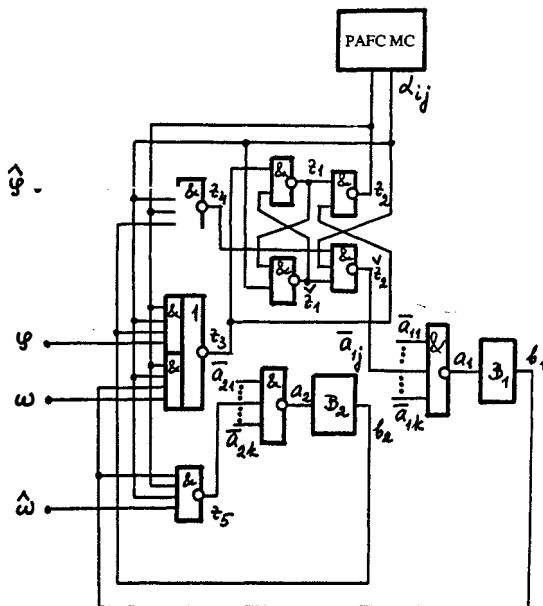


Figure 5.12. Implementation for the “incorrect” loop of Fig. 5.7(a).

The above circuits may be assessed as more or less adequate examples of constructing some modelling circuits by means of modifying the multiple use circuit. Although this technique is certainly not a regular method, its main advantage is the efficiency of circuits obtained with its aid. We should, however, conclude that, in the general case, the problem of finding a set of original modelling circuits is still open.

5.2.4. USING AN ARBITER

An *arbitration situation* is a situation with two or more branches (without loss of generality, we consider the case of two branches) of a PAFC attempting to acquire a common resource (module). In this case, the resource must first be allocated to one (no matter which, from the viewpoint of control flow) of the branches, and after the completion of its use in this branch, it is then allocated to the other branch.

EXAMPLE 5.1. In Fig. 5.13(a), an arbitration situation may arise when the operator A_3 is simultaneously activated in both the left and right branches of the PAFC.

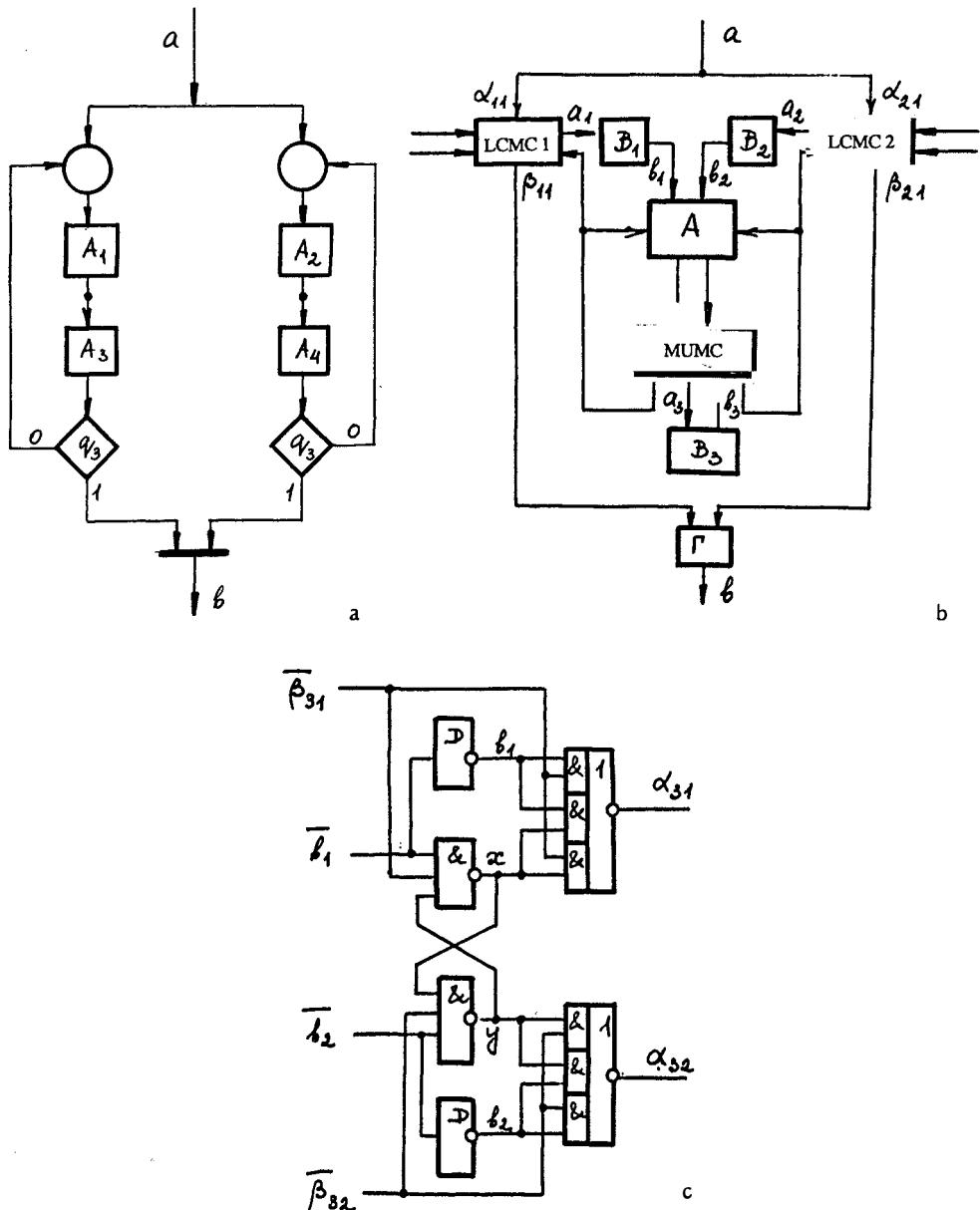


Figure 5.13 (a) - (c). Illustration of the situation leading to (a) the arbitration condition; (b) the corresponding flow chart with arbiter; and (c) implementation for the arbiter with built-in delays.

Existing high level programming tools for the resolution of arbitration conditions or conflicts are based on special *synchro-primitives* like *Dijkstra semaphores* and other constructs.

However, they may hardly be recommended for obviating the difficulties arising in the implementation of these constructs by modelling circuits. At the hardware level, the requirement of atomicity (indivisibility) of semaphores, which is analogous to that of the strict simultaneity of a token transfer (removing tokens from input conditions and adding tokens to output conditions for a firing event) in Petri nets, results, with a non-zero probability, in *anomalous behaviour* of the circuits. This problem is discussed more fully in Chapter 9. Here, we only note that transient processes in arbiters involve *anomalous modes of operation*, and, hence, the duration of transient processes can be substantially, up to two orders of magnitude, greater than those in the normal operation modes. One of the most common techniques to reduce the effect of anomalous modes is to use large enough built-in delays. Here, in contrast to Chapter 9, we use exactly this method.

Fig. 5.13(b) demonstrates a modelling circuit realizing the PAFC of Fig. 5.13(a), containing an arbiter (A), two loop control circuits and a multiple use circuit. The operation of the arbiter can be described as follows. When one of the branches issues a request for the module B_3 (realization of operator A_3), the arbiter replies with the signal α_{31} and does not respond to the other input b_2 until the full operation of B_3 on the request of b_1 is complete, i.e. until the condition $\alpha_{31} = \beta_{31} = \beta_{11} = 1$ becomes true. After this, the arbiter is ready for another request, say, by the signal b_2 . If two requests are issued simultaneously to the arbiter, then the arbiter randomly gives its preference to one of them, and its further operation is as above. We should note that for the period of the transient process of decision between the two inputs in the arbiter, these inputs must be cut off, or locked. The delay elements are used for this. As soon as the module B_3 completes its operation on the request of b_2 , and when b goes to 1, the working phase ends with the modules B_1 , B_2 and B_3 having run through both phases. In the idle phase (after the reposition of AP of Fig. 5.13(a)), all signals return to 0. The implementation of an arbiter with built-in delays is shown in Fig. 5.13(b). If b_1 and/or b_2 go to 1, the flip-flop (x, y) changes, and in a time period determined by the scaled delay D and the delay of element α_{31} or α_{32} , exactly one (not both!) signal, either α_{31} or α_{32} , changes to 1. After the change of the flip-flop, the arbiter does not respond to any changes of the other input (if $x = 0$, then to b_2 , and if $y = 0$, then to b_1). When the module B_3 completes its operation, the multiple use circuit replies with the appropriate signal, either β_{31} or β_{32} , thereby changing the state of the flip-flop and allowing a request at the other input.

The circuits described in the above sub-sections are sufficient for constructing

modelling circuits that implement a particular class of PAFCs. However, the direct translation of a PAFC to a circuit is not always feasible. This is related, primarily, to the fact that a PAFC does not express, in an explicit way, the two-phase discipline of operation of aperiodic modules assigned for PAFC operators. Such problems, and ways of tackling them, are discussed in the following example.

EXAMPLE 5.2. Let a PAFC be given as shown in Fig. 5.14(a).

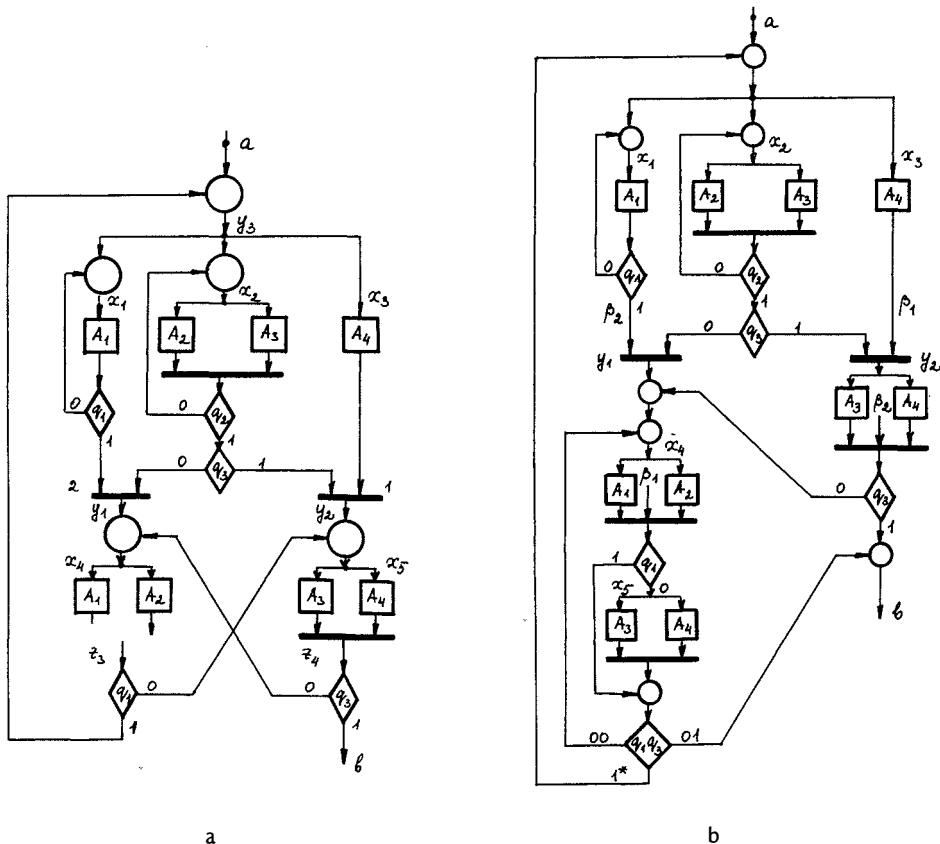


Fig. 5.14 (a) and (b). (a) Example of a parallel asynchronous flow chart, and (b) the result of its transformation to a form suitable for standard implementation.

Analysis shows that before the loop containing x_2 and (A_2A_3) is terminated there is no way of knowing which of the remaining branches of the algorithm will produce the result which will be used later. If it happens that $q_3 = 0$, then the result of operator A_1 will be used, otherwise that of A_4 . In this case, the other branch works “in store”, and the completion of its operation is not indicated at all, which may result in uncertainty in the subsequent operation of the modelling circuit. The PAFC of Fig. 5.14(a) contains nested loops – a fact which is not, as such, an obstacle for the translation into a circuit. The problem emerges from the fact that the loop $x_4 - (A_1A_2) - x_5 - (A_3A_4)$ has two entries at two different points in the chart, viz. y_1 and y_2 , and two exits from the loop, also at two different points b and c and, what is more, these are with respect to two different variables q_1 and q_3 constituting the loop exit conditions. Therefore, as can be seen from the chart, the outer loop has its exit within the body of the above loop, and these loops are not nested. This complicates the problem of constructing a modelling circuit and makes it impossible to use standard circuit modules. However, the idea of using such modules is so attractive that it is quite interesting to see whether it may be applicable to a wider class of PAFCs.

The problem can be tackled through a transformation of an original PAFC to a form appropriate for standard implementation. In so doing, we can transform the PAFC of Fig. 5.14(a) into an algorithmically equivalent one, shown in Fig. 5.14(b), and thus eliminate the above anomaly. This is achieved in the following way.

Fig. 5.15(a) demonstrates a technique for converting a loop-PAFC having an additional entry point, different from the beginning point. We insert an extra path in the multiple use circuit, no matter whether a duplicated fragment of the loop body is a single operator or a composite PAFC (in this case A_2). Fig. 5.15(b) presents a technique for converting a loop-PAFC having an additional exit point within the loop body. The new loop exit condition is constructed for the case when logical variables may be directly applied to a loop control circuit. When the latter is not the case, the circuit is realized in the form of a standard module. The loop exit condition can be implemented by a special logic including an output switch connected to the circuit.

The problem of completion indication for the non-used, or the so-called working “in store”, branches is solved, as shown in Fig. 5.14(b), by two additional synchronizations of signals β_1 and β_2 . An alternative solution can be achieved by using arbiters, but in this case the non-used branches should be synchronized with the output signal of a modelling circuit. The modified PAFC of Fig. 5.14(b) can be further transformed to a modelling circuit containing 4 loop control circuits, 2 two-input (for A_1 and A_2) and 2 three-input (for A_3 and A_4) multiple use circuits, 6 synchronizers (Γ -flip-loops), 4 conditional branches (switches) and 3 merger modules.

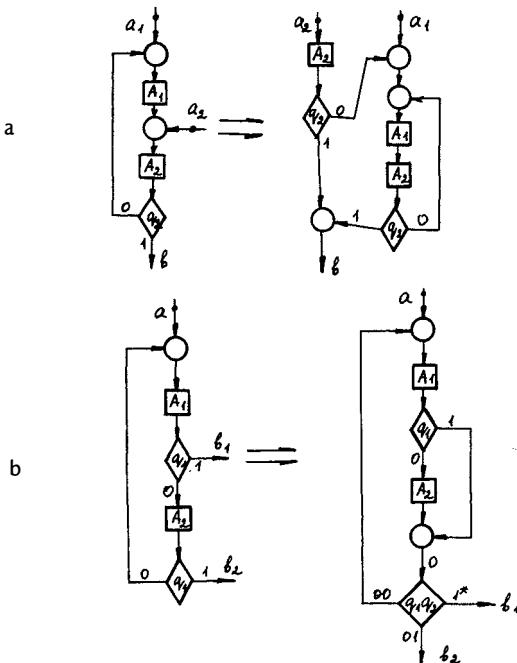


Figure 5.15 (a) - (c). Illustrations for transformations of a flow chart to a form suitable for standard implementation.

5.2.5. GUARD-BASED IMPLEMENTATION

The universal character of the method for constructing modelling circuits based on standard fragments of PAFCs (with potential need for transformation) leads to the assumption that this approach may result in substantial overheads in implementation circuitry. From this point of view, the following approach based on guard functions is generally more economic.

A *guard function*, or simply a *guard*, is a predicate defined on the process states which controls the activation of a given operator. In realizing a guard function within a modelling circuit, we should take into account the two-phase discipline of the operation of modules associated with operators in a PAFC, and make sure that the behaviour of the modelling circuit is independent of the delays of the processes in these modules, as well as of the delays of the elements in the modelling circuit.

The procedure for deriving guard functions from a PAFC involves some informal steps, and, therefore, achieving the desired efficiency largely depends on the designer's skills. The reason is that a PAFC specifies only conditions for the activation of modules, whereas those for the idling may be defined quite freely. We should only ensure that the module is in the idle state before the next activation

condition arrives. This may require analysis as to whether the circuit obtained is semi-modular, and such an analysis is generally concerned with using the methods presented in Chapter 8.

The following example illustrates the procedure of constructing a modelling circuit from guard functions.

EXAMPLE 5.3. Again consider the PAFC shown in Fig. 5.14(a), in which the input arcs of operators are denoted by x_i (bifurcation does not remove this denotation), $1 \leq i \leq 5$. The activation condition for an operator is the disjunction of variables associated with input arcs of the operator. When NAND elements are used for the implementation, it is suitable to assume that the operator is activated if $x_i = 0$. Then for a given PAFC, we obtain

$$a_1 = \overline{x_1 x_4}, \quad a_2 = \overline{x_2 x_4}, \quad a_3 = \overline{x_2 x_5}, \quad a_4 = \overline{x_3 x_5}$$

where a_i denotes the phase signal of module B_i associated with operator A_i . Note that x_i changes from 1 to 0 if and only if the values of all other variables activating this operator are equal to 1.

The input arcs of loops in a PAFC are denoted by y_i , $1 \leq i \leq 3$ (bifurcation does not remove these denotations either). The bottom loop has two exits, one of which is the exit of the entire PAFC denoted by b , and the other is c . The module B_i has an acknowledgement signal denoted by b_i .

Each guard is associated with two conditions, one for the 1–0 transition and the other for the 0–1 transition. The 1–0 transition of x_1 is performed if the variable $y_3 (y_3 = 0)$ is initiated and B_1 is in the idle state. The condition for that transition can be written as $y_3 \vee b_1 = 0$. The condition $y_3 = 0$ must remain unchanged independently of the transitions of other variables in the circuit. If, in the loop activated by $y_3 = 0$, the module B_1 completes the working phase and $q_1 = 0$, the loop reiterates and $y_3 \vee b_1 \bar{q}_1 = 1$ holds. Thus, $x_1 = y_3 \vee x_1 b_1 \vee b_1 \bar{q}_1$, and the corresponding implementation is shown in Fig. 5.16(a). Similarly, x_2 and x_3 are realized in the form

$$x_2 = y_3 \vee x_2 b_2 \vee x_2 b_3 \vee b_2 b_3 \bar{q}_2, \quad x_3 = y_3 \vee x_3 b_4.$$

The guards for x_4 and x_5 differ from the above because they can be driven both by the loop entry signal and by the internal links within the body of the loop.

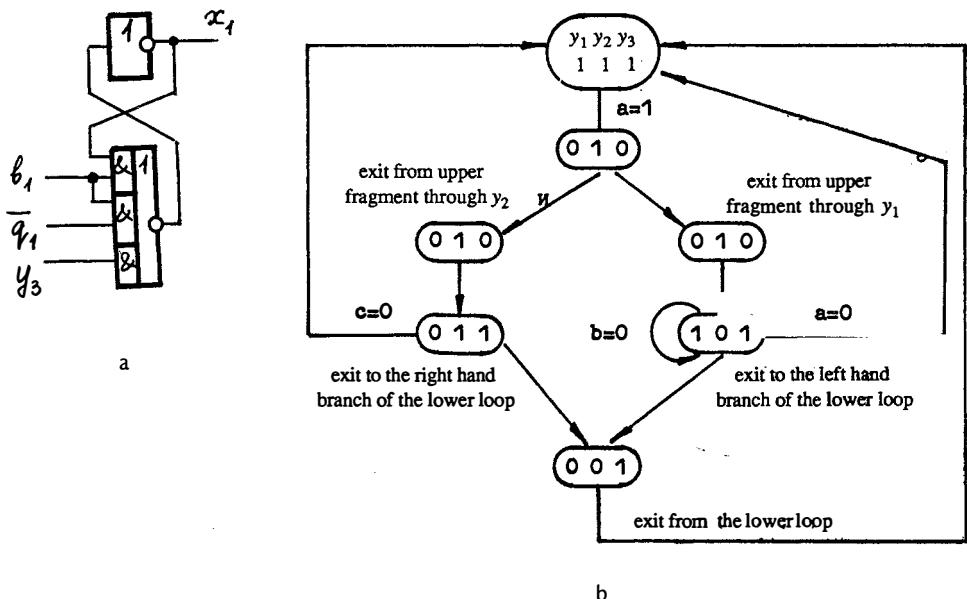


Figure 5.16 (a) and (b). (a) Typical implementation of a guard function, and (b) an approach to the design of a modelling circuit.

Let x_4 and x_5 be realized as

$$x_4 = y_1 \vee x_4 b_1 \vee x_4 b_2 \vee b_3 b_4 \bar{q}_3 ,$$

$$x_5 = y_2 \vee x_5 b_3 \vee x_5 b_4 \vee \bar{b}_1 b_2 \bar{q}_1$$

(by analogy x_1 – x_3), and the peculiarities indicated will be taken into account in implementing y_1 and y_2 . The signalling discipline for variables y_1 , y_2 and y_3 can be illustrated by the transition diagram of Fig. 5.16(b). In the initial state, they are all equal to 1. As soon as a changes to 1, thereby initiating the PAFC, y_3 goes to 0, which activates three parallel branches at the top part of the PAFC. When the condition of activation of the bottom part is true, either y_1 or y_2 is activated, but that will not be enough to execute the bottom loop because the modules involved in it must be idled before their reactivation. The idling can be done only after the completion of the process initiated by the guard y_3 . Hence, the activation condition for y_1 and y_2 will be the completion of the transient processes in the modules of the top part of the PAFC. After arriving either in the state 010 or the state 100, y_3 changes back to 1, and this results in the idling of all the modules and allows the activation of the bottom loop at one of its entry points. If we enter that loop at the

signal point y_1 , then, after completing A_1 and A_2 , the control will either go to y_2 or leave the loop, generating the function c which gives rise to the change of y_1 from 0 to 1. If the control is returned to the point y_2 , then, after the completion of A_3 and A_4 , the modelling circuit either ceases its operation and produces the signal b , or continues to be in the loop. From the above discussion we derive

$$\begin{aligned} y_3 &= \overline{ay_1y_2}, \quad \bar{y}_1 = c\bar{y}_3b_1b_2b_3b_4q_1q_2\bar{q}_3 \vee \bar{y}_2b_3b_4\bar{q}_3 \vee \bar{y}_1c \vee \bar{y}_1a \vee \bar{y}_1\bar{y}_3, \\ \bar{y}_2 &= c\bar{y}_3b_1b_2b_3b_4q_1q_2q_3 \vee \bar{y}_1b_1b_2\bar{q}_1 \vee \bar{y}_2c \vee \bar{y}_2a \vee \bar{y}_2\bar{y}_3. \end{aligned}$$

We end the construction procedure by obtaining the functions

$$c = \overline{y_1b_1b_2q_1}, \quad \bar{b} = y_2b_3b_4q_3 \vee \bar{b}y_1 \vee \bar{b}y_2 \vee \bar{b}y_3 \vee \bar{b}a.$$

Functions y_3 and c , as well as a_1-a_4 are implemented on NANDs, while the remaining functions are built on elements of the type of Γ -flip-flop as shown in Fig. 5.16(a) with complemented variables realized by inverters. It is easily seen that for the example discussed the implementation of the PAFC using guard functions is more economic than that obtained from standard fragments.

5.3 Functional completeness and synthesis of semi-modular circuits

As we saw in Section 2.4, the Muller model can adequately represent the behaviour of asynchronous circuits. There are several *classes of asynchronous circuits* which may be associated with corresponding classes of state-transition diagrams that describe processes in the circuits.

These classes are:

- *speed-independent circuits* that realize controlled diagrams,
- *semi-modular circuits* that realize semi-modular diagrams,
- *distributive circuits* that realize distributive diagrams,
- *totally sequential circuits* that realize totally sequential diagrams.

We attach the same denotations to these classes of circuits as have been introduced for their transition diagram counterparts, viz, W , U , D , K , in their respective order, and, particularly for autonomous diagrams, W^* , U^* , D^* , K^* .

In this section we shall study the implementability issues for the classes of modelling circuits with respect to AND-OR-NOT and NAND logic bases. The proofs are essentially constructive in the sense that they provide rigorous methods for constructing canonical implementations of circuits in these classes.

5.3.1. FORMULATION OF THE PROBLEM

Suppose the circuit which implements a controlled transition diagram is defined by the following *system of Boolean equations*

$$z_i = \phi_i(z_1, \dots, z_i, \dots, z_n), \quad 1 \leq i \leq n, \quad (5.1)$$

where ϕ_i is the inherent function of the i -th logical element z_i of the circuit.

The interconnection of elements z_i in accordance with the system (5.1) where each input to every element is connected to exactly one output, and no two outputs are connected to each other, will operate independently of the delays of the elements.

Assume that a finite set of allowed inherent functions $L = \{f_1, f_2, \dots, f_m\}$ is given. If at least one of the functions ϕ_i in (5.1) does not belong to L , then a problem of decomposing ϕ_i into sub-components f_j such that $f_j \in L$ arises. Such a decomposition may, however, result in a new circuit that will no longer be speed-independent.

DEFINITION 5.1. A *decomposition* of a system of Boolean equations of the form (5.1) into sub-components L is a system of equations in the form

$$\begin{aligned} z_i &= f_{i0}(z_1, \dots, z_i, \dots, z_n, z_{n+1}, \dots, z_{n+k}), \\ z_{n+t} &= f_{it}(z_1, \dots, z_i, \dots, z_n, z_{n+1}, \dots, z_{n+k}) \end{aligned} \quad (5.2)$$

where $f_{i0}, f_{it} \in L$ such that

$$f_{i0}(z_1, \dots, z_i, \dots, z_n, f_{i1}, \dots, f_{ik}) = \phi_i(z_1, \dots, z_i, \dots, z_n).$$

DEFINITION 5.2. Suppose an interconnection of elements in accordance with a system (5.2) is given such that $f_{it} \in L$. The circuit obtained will be called a *correct implementation* if it belongs to the same class as a circuit defined by the original system of the form (5.1).

DEFINITION 5.3. A system of functions $L = \{f_j\}$, $1 \leq j \leq m$, will be called *functionally complete in the class U^* (D^* , K^*)*, if, for any circuit in that class, we can obtain a correct implementation using only elements with inherent functions $f_j \in L$.

The following discussion deals with finding elementary systems of functions that are functionally complete in classes U^* , D^* and K^* .

5.3.2. SOME PROPERTIES OF SEMI-MODULAR CIRCUITS

To solve the stated problem, we need a deeper understanding of the local properties of elements constituting a circuit in the class U^* .

Recall that, according to Definition 2.26, an element z_i is regarded as an *excited element* if $z_i \neq \phi_i(z_1, \dots, z_n)$, whereas it is regarded as an *idle element* (in the equilibrium state) if $z_i = \phi_i(z_1, \dots, z_n)$. In other words, the element is excited if, being subject to a new combination of its input values, it has not yet changed to its implied output state, i.e. the output value does not comply with a new input combination and its inherent function. The transition from excited state to idle state may generally proceed either through (1) *changing the output value of the element* after a finite delay intrinsic to that element, or through (2) *changing the value of the inputs of the excited element* thus restoring the compliance between the input combination and the output value of the element. *It is characteristic for semi-modular circuits that in them only the first way of removing the excitation is allowed for.* Because of this, we may introduce an alternative definition of a semi-modular circuit, which will be equivalent to the lattice-theoretical one (for the sake of brevity, we omit the proof of such equivalence).

DEFINITION 5.4. A circuit is *semi-modular* if each of its elements, having been excited, can reach the idle state only by changing the value of the output.

Therefore, in a semi-modular circuit each element is allowed to make the following state transitions

$$\overrightarrow{1} \rightarrow 1^* \rightarrow 0 \rightarrow 0^* \quad \boxed{-}$$

and there may be no transitions of the form $1^* \rightarrow 1$ and $0^* \rightarrow 0$. It is also implied that the internal implementation of an element is hazard-free.

Let U_n^* denote the class of all semi-modular circuits that contain exactly n elements. For example, U_1^* consists of a circuit shown in Fig. 5.17, the inherent function of a single element in which $z = \bar{z}$.

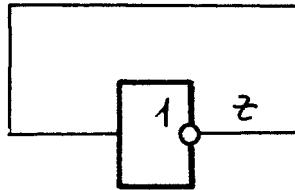


Figure 5.17. The only single-element semi-modular circuit.

THEOREM 5.1. *Let a circuit be given in U_n^* , $n > 1$. The inherent function of every element z_i in this circuit is either isotonous (see Definition 3.1) in z_i or does not depend on z_i .*

Proof. Assume to the contrary that the inherent function of the element z_i of a circuit U_n^* , $n > 1$, is not isotonous in z_i . Then that function can be expressed in the form

$$z_i = z_i x_i \vee \bar{z}_i y_i$$

where x_i and y_i are certain logical functions that do not depend on z_i . There are four cases of combinations of values of x_i and y_i . Let us consider them.

If $x_i = 1$ and $y_i = 1$, then z_i assumes the value 1; if $x_i = 1$ and $y_i = 0$, then z_i is equal to its previous value; if $x_i = 0$ and $y_i = 0$, then z_i will be equal to 0. These three cases show that in the combinations of values of x_i and y_i , the inherent function for z_i is isotonous in z_i . Then if the last, fourth, combination – $x_i = 0$ and $y_i = 1$ – is impossible, then the given assumption leads to contradiction. On the other hand, for that last combination, the element z_i is excited independently of the value of z_i . In this case, any change of x_i or y_i may result in the situation where z_i will become idle without changing its value, and that will contradict Definition 5.4. Hence, if $x_i = 0$ and $y_i = 1$, then both $f x_i$ and y_i must remain unchanged. The latter is possible if either the circuit having entered such states will never leave them or simply $x_i = 1$ and $y_i = 1$ hold for all states. However, this implies that either the circuit states do not form a single equivalence class, thus violating the condition of the theorem, or the inherent function z_i is represented as $z_i = \bar{z}_i$, from which it follows that the circuit belongs to U_1^* , which also contradicts the initial condition that $n > 1$. Therefore, we conclude, that for every element we reach a contradiction because Definition 5.4 must hold for every element in the circuit. *Q.e.d.*

From the above theorem, we may immediately deduce the following.

COROLLARY 5.1. *The inherent function of every element in a circuit in class U_n^* , $n > 1$, can be represented in the form*

$$z_i = z_i \bar{w}_i \vee v_i \quad w_i v_i = 0 \quad (5.3)$$

where w_i and v_i are logical functions that do not depend on z_i .

To prove (5.3) it is sufficient to make the substitution $v_i = y_i$ and $w_i = \bar{x}_i \bar{y}_i$.

5.3.3. PERFECT IMPLEMENTATION

It should be noted that in (5.3) we did not lay any restrictions upon w_i and v_i functions except that they did not depend on z_i . They can, however, be non-monotonous, i.e. contain certain variables z_j both in direct and complemental forms. Meanwhile, the inherent functions of AND-OR-NOT elements are *antitonous in input variables*. Hence, neither isotonous nor non-monotonous functions can be implemented by means of a single AND-OR-NOT element, as required for the implementation of equation (5.3). This implies that in constructing an AND-OR-NOT-based implementation we must have elements with the outputs z_i and \hat{z}_i (or \check{z}_i), i.e. such an implementation must be *double-rail*, or *two-channel*, where the idle state is characterized by either $\hat{z}_i \neq z_i$ or $\check{z}_i \neq z_i$.

Using an inverter to obtain \hat{z}_i (or \check{z}_i) may not guarantee that the resulting circuit will be semi-modular. Indeed, at a certain instance during its operation, there may occur a state where $z_i = \hat{z}_i$ holds (we omit the case of $z_i = \check{z}_i$ due to duality). In that state, the actual value of variable z_i is undefined, and the change of another element z_j is possible, after which the value of z_i may also change and thus result in removing the excitation condition from the input of the inverter realizing \hat{z}_i . From the above follows that if the circuit uses both z_i and \hat{z}_i , then the implementation of the form $\hat{z}_i = \overline{z_i x_i \vee y_i}$ and $z_i = \overline{\hat{z}_i}$ (by an AND-OR-NOT element and an inverter) may be incorrect.

As far as the functioning of an element z_i is concerned all allowed states Z of a circuit in U^* , i.e. the states which the circuit may reach in its operation, can be

divided into *three classes*. The first class is associated with the states in which the element z_i is excited and $z_i = 1$. We denote that class as $R_i, R_i \subset Z$.

The second class, denoted by $S_i, S_i \subset Z$, consists of the states in which z_i is excited and equal to 0.

The third class, denoted by $T_i, T_i \subset Z$, is constituted by the states in which z_i is idle.

THEOREM 5.2. *Let a circuit be given in U_n^* , $n > 1$. If each variable z_j of the circuit is realized in the form (5.3), i.e.*

$$z_j = z_j w_j \vee v_j, \quad w_j v_j = 0,$$

then the states in which $z_i = \hat{z}_i = 0$ belong to the class $T_j, j \neq i$.

Proof. Let the functions w_j and v_j be neither isotonous nor antitonous in z_i . Then if they are represented in the form

$$w_j = w_j(z_1, \dots, z_i, \hat{z}_i, \dots, z_n),$$

$$v_j = v_j(z_1, \dots, z_i, \hat{z}_i, \dots, z_n),$$

where z_i and \hat{z}_i are considered as different variables, then it is implied that the functions in w_j and v_j are isotonous in both z_i and \hat{z}_i . Furthermore, any combination in which $z_i = \hat{z}_i = 0$ is less than a corresponding combination where $z_i \neq \hat{z}_i$, and, hence, according to the definition of an isotonous function

$$w_j^0 = w_j(z_1, \dots, z_i = 0, \hat{z}_i = 0, \dots, z_n) \leq w_j,$$

$$v_j^0 = v_j(z_1, \dots, z_i = 0, \hat{z}_i = 0, \dots, z_n) \leq v_j.$$

In other words, during any transition from a combination in which $z_i \neq \hat{z}_i$ to one in which $z_i = \hat{z}_i = 0$ the values of w_j and v_j may only be reduced. Due to the fact that $w_j v_j = 0$, only the following combinations of values of w_j and v_j are allowed: 10, 01 and 00, which are associated with the stable values of z_j : 0, 1 and z_j , respectively. Hence, functions of the values of w_j and v_j (transitions of the form 10–00 or 01–00) do not force z_j to change. *Q.e.d.*

The following theorem presents a technique for decomposing expressions of the kind of (5.3) that satisfies the conditions of Theorem 5.2.

THEOREM 5.3. *The inherent function of any element z_i in a circuit U_n^* , $n > 1$, can be expressed in the form*

$$z_i = \overline{w_i} \vee \overline{\hat{z}_i} \vee \overline{v_i}, \quad w_i v_i = 0 \quad (5.4)$$

where w_i and v_i are functions that do not depend on z_i .

Proof. Bearing in mind that $w_i v_i = 0$ (5.3) can be transformed as follows

$$z_i = z_i \bar{w}_i \vee v_i = z_i \bar{w}_i \vee v_i \bar{w}_i \vee v_i w_i = z_i \bar{w}_i \vee v_i \bar{w}_i = \overline{w_i} \vee \overline{z_i} \vee \overline{v_i}.$$

It is clear that (5.4) can be rewritten in the form

$$z_i = \overline{w_i} \vee \hat{z}_i, \quad \hat{z}_i = \overline{v_i} \vee z_i, \quad w_i v_i = 0 \quad (5.5)$$

Q.e.d.

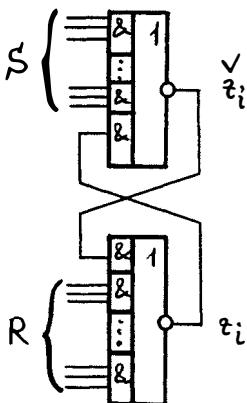


Figure 5.18. Basic construction for perfect implementation. Excitation functions are given in reduced (not minimal!) DNFs.

We draw the reader's attention to the fact that (5.5) are the equations of the symmetric RS-flip-flop (see Fig. 5.18) with excitation functions $S_i = v_i$ and $R_i = w_i$, $S_i R_i = 0$. Note that when the flip-flop of Fig. 5.18 changes its state, then the transient state is $z_i = \hat{z}_i = 0$ whereas the state with $z_i = \hat{z}_i = 1$ does not occur.

Then if each variable in a semi-modular circuit is associated with a flip-flop of that kind, then, by virtue of Theorem 5.3, such an implementation will be correct. Using elements of the AND-OR-NOT type for building flip-flops allows the implementation of the reduced⁽¹⁾ DNFs of flip-flop excitation functions straight on the elements of flip-flop arms.

Theorems 5.2 and 5.3 show the correct way to implement the inherent functions z_i on AND-OR-NOT elements, and furthermore, the state where $z_i = \hat{z}_i = 0$, which occurs during the change of z_i , cannot cause the excitation of $z_j, j \neq i$.

However, in circuits from $U_n^*, n > 1$, it is possible that, at the same time, more than one element can be in an excited state. Let such elements be z_i and z_j . Then it is essential for the correct implementation that excitation of an element may not be removed by a new value of another element that has just changed, i.e. having $z_i = \hat{z}_i$, z_j must stay in the excited state, and vice versa.

The following two lemmata show that in a correct implementation of a circuit in $U_n^*, n > 1$, this condition can be easily provided.

LEMMA 5.1. *If in a circuit from $U_n^*, n > 1$, an element z_i is excited then there is no state where all those elements, on whose values the inherent function of z_i essentially depends, are excited simultaneously.*

Proof. Assume to the contrary that there is a state where elements, on the values of which the inherent function of z_i essentially depends, are simultaneously excited. However, this implies that the inputs of z_i may experience any combinations, including those that agree with the value of the output of z_i . Thus it may change from the excited state to the idle one, and this contradicts Definition 5.4. *Q.e.d.*

LEMMA 5.2. *Let a circuit in $U_n^*, n > 1$, be given such that, in a state α , an element z_i is excited, and its value is equal to 1 (0). Also let the elements $z_j, z_{j+1}, \dots, z_{j+m}$ be in the idle state while the other elements are excited. Then, if the inherent function of z_i is given in the form (5.3), the reduced DNF of $w_i(v_i)$ will contain the term which includes the variables $z_j, z_{j+1}, \dots, z_{j+m}$ and no others.*

Proof. It is sufficient to consider the construction of the reduced DNF of $w_i(v_i)$. It can be seen in the truth table that the value of this function in the state α is equal to 1.

(1) Here, a DNF for a function f is called *reduced* if it consists of all primary implicants [120] of f .

However, due to the semi-modularity of the circuit, the same value is assigned to the function for those combinations in which the elements, excited in state α , have already changed their values. By fixing all these combinations we obtain a term which will contain none of the variables that have been excited in state α . Finally, by fixing and covering, the rank of this term can be made yet lower, but will certainly not be equal to 0, because if it were, the inherent function of z_i would be equal to a constant, and this contradicts the given condition. *Q.e.d.*

From Lemma 5.1 and Theorem 5.2, it follows that the implementation must be based on the representation of z_i in the form (5.5), and from Lemmata 5.1 and 5.2, it follows that the implementation must correspond to the reduced DNF of functions w_i and v_i entering into expression (5.5). If all these conditions are satisfied, the implementation obtained will be called perfect. A *perfect implementation* is correct. From the above considerations, we can formulate the following theorem.

THEOREM 5.4. *The system of inherent functions of AND-OR-NOT elements is functionally complete in the class U_n^* . (It is assumed that a given base is unbounded, i.e. the number of element inputs, fan-in, and the element's loading capacity, fan-out, are not upper-bounded).*

EXAMPLE 5.4. Let a circuit consisting of four elements be given by the diagram shown in Fig. 5.19(a). From this diagram, we derive (see Section 2.4) a system of equations of the form

$$z_1 = \bar{z}_4, \quad z_2 = \bar{z}_4, \quad z_3 = z_3(z_1z_2 \vee \bar{z}_4) \vee \bar{z}_4(z_1 \vee z_2),$$

$$z_4 = z_4(z_1 \vee z_2 \vee z_3) \vee z_1z_2z_3.$$

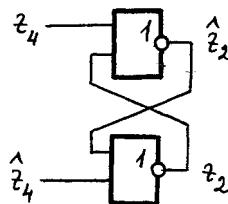
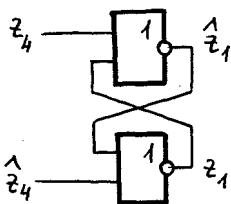
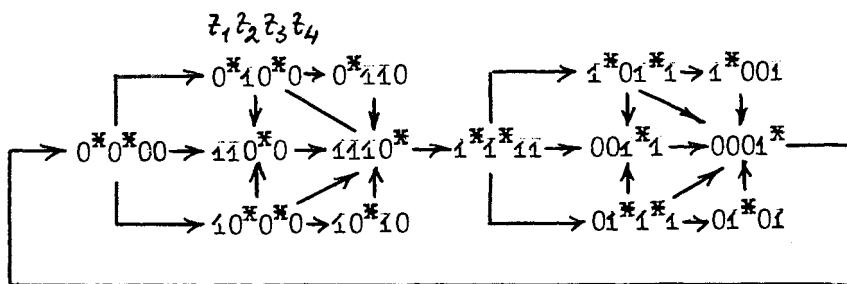
They may be expressed in the form of (5.3) as

$$z_1 = z_1z_2 \vee \bar{z}_4, \quad z_2 = z_2z_4 \vee \bar{z}_4,$$

$$z_3 = z_3 \overline{z_4} \overline{z_1} \vee \overline{z_4} \overline{z_2} \vee \overline{z_4} z_1 \vee \overline{z_4} z_2, \quad z_4 = z_4 \overline{z_1} \overline{z_2} \overline{z_3} \vee z_1 z_2 z_3.$$

The perfect implementation of this system using flip-flops of the type given in Fig. 5.18 is shown in Fig. 5.19(b).

a



b

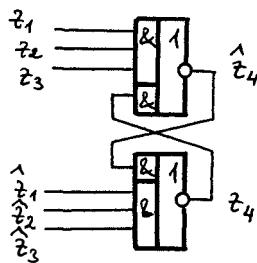
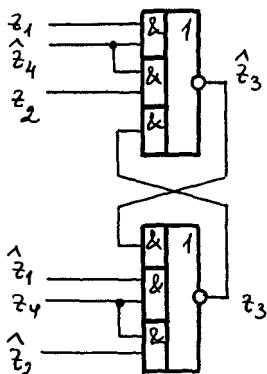


Figure 5.19 (a) and (b). (a) A state transition diagram, and (b) its perfect implementation.

5.3.4. SIMPLE CIRCUITS

In this sub-section we study a process of switching of elements in a semi-modular circuit and define an important class of so-called simple circuits.

The process of circuit operation consists in the changing of values of circuit variables. These changes, or transitions, can be numbered. During the cyclic operation of a circuit, each variable may change its value a certain number of times before the circuit returns to its original state. Any i -th transition of variable z_j will be

denoted by a pair of positive integers (j, i) usually marked with “+” or “−”, associated with transitions of z_j , namely, 0–1 or 1–0, respectively. The i -th transition is assumed to follow the $(i - 1)$ -th excitation and transition of z_j . The excitation may occur for the first time in one, or more, unordered states that can be refined by the following definition.

DEFINITION 5.5. A connected set A of states (the set in which any two states are connected by a path totally belonging to the given set) constituting a sub-set of a set of allowed (working) states of a circuit is called an *excitation region of variable z_j* if z_j is excited in all the states of A and has the same value (either 0^* or 1^*), and the set A is maximal, i.e. for any α and β such that $\alpha \notin A$, $\beta \in A$, and $\alpha \rightarrow \beta$ or $\beta \rightarrow \alpha$, the condition $\alpha_j \neq \beta_j$ holds. The excitation region will be denoted by $A(j, i)$, if the associated excitation causes the i -th transition of variable z_j . A state $\alpha \in A$ will be called a *minimum state of the excitation region*, if for any state $\beta \in A$ ($\beta \neq \alpha$) there exists no sequence of states $\beta \Rightarrow \alpha$ that is totally contained in A . We shall also say that the excitation region has a *lower bound* if it contains only one minimum state.

It is easily seen that each excitation region contains at least one minimum state. On the other hand, later, in Chapter 8, we shall introduce a local characteristic property of distributive circuits that can be formulated as follows.

A semi-modular circuit with respect to an initial state α is distributive with respect to α if and only if each excitation region in the set of its operational states (states that are reachable from the initial state) has a lower bound.

Non-distributive circuits are characterized by the existence of excitation regions that have no lower bounds. Such regions and associated changes (transitions) of variable values are referred to as *detonant* ones, and the number of minimum states in a detonant region is called the *detonancy rank* of that *region*.

The inherent function of each element in a circuit can be expressed in the form of Shannon's expansion

$$f_j(z_1, \dots, z_n) = z_j f_j(z_j = 1) \vee \bar{z}_j f_j(z_j = 0).$$

Functions $f_j(z_j = 0)$ and $\bar{f}_j(z_j = 1)$ are called the *excitation functions of variable z_j* , and denoted by S_j and R_j .

If an excitation region has a lower bound, then the excitation may originate (occur for the first time only) in a single state which is the lower bound of the region. Such an excitation variable z_j is first caused by a single term T of a reduced DNF of its excitation function, either S_j or R_j . During subsequent operation of the

circuit, the variables which enter into the term T can change their values before the z_j manage to change. In this situation, due to the semi-modularity of the circuit, the excitation of variable z_j must be sustained by some other terms of the excitation function. This kind of phenomenon will be referred to as the *excitation function term takeover*, or simply the *term takeover*. It is obvious that the term takeover may occur in a detonant excitation region.

EXAMPLE 5.5. A circuit defined by the system of equations $a = 1$, $b = 1$, $c = a \vee b$ has the state-transition diagram shown in Fig. 5.20.

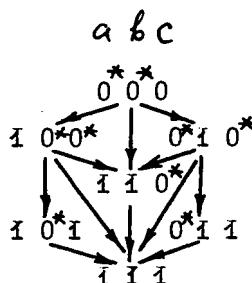


Figure 5.20. State-transition diagram for a circuit with a detonant excitation region.

The region of the first excitation of c consists of three states 100, 010 and 110, thus being detonant because the states 100 and 010 are both minimum for that region (and are not mutually ordered). The excitation function for c is $S_c = a \vee b$. The excitation of c in states 100 and 010 is caused by terms a and b , respectively. It is then sustained in the state 110 by both terms a and b . The detonancy rank for the region is 2.

A more rigorous definition for a term takeover is given below.

DEFINITION 5.6. We shall say that in an excitation region $A(j, i)$ of a variable z_j there is an excitation function term takeover for S_j (or R_j) if there exists a path $\alpha^1 \rightarrow \alpha^2 \rightarrow \dots \rightarrow \alpha^k$, where $\alpha^1, \alpha^2, \dots, \alpha^k \in A(j, i)$, such that all the terms of the reduced DNF of S_j (or R_j) which are equal to 1 in α^1 become equal to 0 in α^k .

The presence of a term takeover makes it more difficult to solve the problem of decomposing inherent functions of elements because *logical hazards* are possible between elements realizing the terms which are involved in the takeover.

DEFINITION 5.7. An asynchronous circuit is called *simple* (with respect to the initial state α) if all excitation regions in the set of operational states are free from term takeovers.

The freedom from term takeovers in a simple circuit implies that if the excitation of a variable z_j occurs for the first time in a single state, or in several unordered states, under the effect of one, or several (in the case of a detonant transition), terms of S_j or R_j , then none of the variables which enter into these terms can change its value until z_j changes.

A suitable way to represent the operation of circuits is to use a *signal change diagram* that is formally defined as a triple $\langle V, \leftarrow, \rightarrow \rangle$ where V is a set of pairs of positive integers (j, i) each of which is interpreted as a transition of variable, i.e. (j, i) is the i -th change (or transition) of z_j , \leftarrow and \rightarrow are precedence relations defined on the set V with different semantic interpretations. If $a, b \in V$ then $a \leftarrow b$ denotes that transition a strongly precedes transition b . The other relation, $a \rightarrow b$, denotes that a weakly precedes b . The term “strong precedence” is related to the fact that b can only occur after the occurrence of a , whereas the “weak precedence” implies that b occurs after either a , or another transition c , such that $a \rightarrow b$ and $c \rightarrow b$. Thus in the latter case b does not necessarily occur after a but may also occur after c .

It is essential that the operation of distributive circuits can be expressed by using only the \leftarrow -relation.

The study of signal change diagrams defining circuits and being a formal tool for their initial specification is by no means a self-goal, but, rather, is introduced as a useful illustration of their properties. We shall represent a signal change diagram as a directed graph with vertices of three types. Vertices of the first type are associated with transitions in a set V which are labelled with + or - depending on the direction of a signal change. These vertices are interconnected either directly by arcs corresponding to the \leftarrow -relation or indirectly, through auxiliary vertices, in the case of the \rightarrow -relation.

We should also note that the graph does not include those arcs which correspond to a transitive closure of the above relations, thus being different from what is accepted in using *Hasse diagrams*.

EXAMPLE 5.6. Fig. 5.21 illustrates three diagrams in which: $(+1, 1) \leftarrow (-3, 2)$ and $(+2, 1) \leftarrow (-3, 2)$ in Fig. 5.21(a); $(+1, 1) \rightarrow (-3, 2)$ and $(+2, 1) \rightarrow (-3, 2)$ in Fig. 5.21(b); $(+2, 1) \rightarrow (-3, 2)$ and $\{(+1, 1), (+4, 1)\} \rightarrow (-3, 2)$. The third diagram defines the case where the occurrence of $(-3, 2)$ is preceded by either the occurrence of $(+2, 1)$ or by the occurrence of both $(+1, 1)$ and $(+4, 1)$. Fig. 5.22 shows a signal change diagram for a circuit whose state-transition diagram was given in Fig. 2.11(a).

It is easily seen that the graphical representation of a signal change diagram for distributive circuits coincides with a signal graph with all tokens removed from arcs.

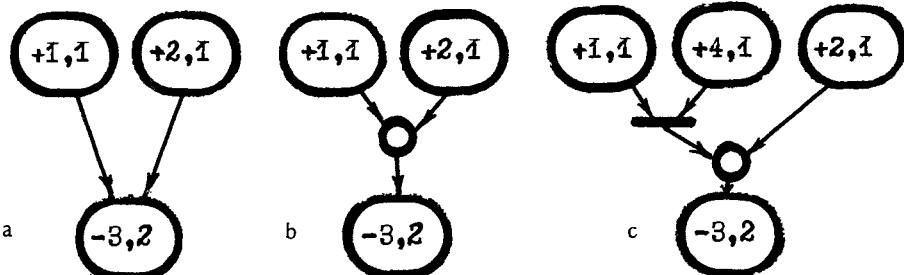


Figure 5.21 (a) - (c). Three signal change diagrams.

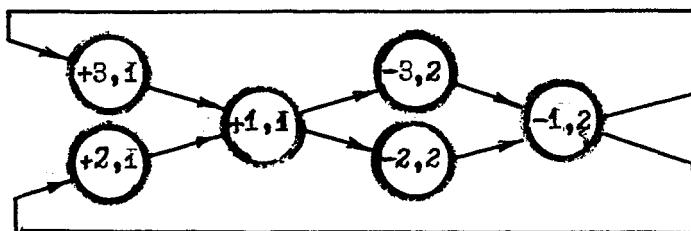


Figure 5.22. Signal change diagram with concurrent transitions which correspond to the state-transition diagram of Fig. 2.11(a).

We introduce another binary relation on a set of transitions V – the concurrency relation. We call *two transitions* (s, p) and (t, q) concurrent, as denoted by $(s, p) \parallel (t, q)$, if and only if neither $(s, p) \leftarrow (t, q)$ nor $(t, q) \leftarrow (s, p)$ holds. In other words, the occurrence of either of these transitions is not preceded by the occurrence of the other transition.

It is obvious that in a signal change diagram associated with a semi-modular circuit any two different transitions of the same variable must not be concurrent. In a state-transition diagram the concurrency relation manifests itself by the fact that $A(s, p) \cap A(t, q) \neq \emptyset$, i.e. there exists at least one such state in which both variables z_s and z_t are excited, and these excitations result in the p -th and q -th changes of these variables, respectively.

EXAMPLE 5.6 (continued). Fig. 5.22 illustrates that $(+3, 1) \parallel (+2, 1)$ and $(-3, 2) \parallel (-2, 2)$.

As has already been mentioned, each variable z_j in the circuit is associated with a finite number of transitions $(j, 0), \dots, (j, m_j - 1)$ before the operational cycle of the circuit is reiterated.

Let α be an arbitrary state in the excitation region $A(j, i)$, and Q_j^i is a part of the DNF of the excitation function R_j or S_j such that corresponds to the (j, i) -transition. A transition (k, l) is called the *input transition for the transition (j, i) and for the region $A(j, i)$* if there exists a state β in the set of operational states such that $\beta \xrightarrow[1]{z_k} \alpha$

and this transition is made as a result of the l -th change of the variable z_k . This implies that the l -th change of variable z_k is the immediate cause for the i -th excitation of variable z_j . Thus, if the excitation region $A(j, i)$ contains the change of at least one variable which is input for that region, then the region necessarily contains a takeover. Such a takeover will be called *complex takeover*. If a takeover occurs in the excitation region where none of the variables which are input for that region change, we shall call it a *simple takeover*. The presence of a takeover in the region $A(j, i)$ is a consequence of the concurrency between (j, i) and a current transition of the variable which enters into the excitation term Q_j^i which corresponds to the given state α . For the case of the above complex takeover, $(j, i) \parallel (k, l+1)$ holds.

The significance of the class of simple circuits is shown by the following theorem.

THEOREM 5.5. *For any semi-modular circuit, an equivalent semi-modular simple circuit can be constructed, and if the original circuit is distributive, then the equivalent simple circuit will also be distributive.*

Note that the theorem is formulated for initialized circuits, i.e. both semi-modularity and distributivity are considered with respect to initial state.

Proof. We should point out that any non-distributive but semi-modular circuit can be transformed such that the excitation functions which correspond to detonant transitions will assume the form $z_m^\alpha \vee z_r^\beta$, i.e. the excitation terms are elementary and the detonancy rank is equal to 2. Therefore, we should pay special attention to the problem of eliminating takeovers concerned only with non-detonant transition regions.

Elimination of complex takeovers. The general idea of eliminating a complex takeover is based on the insertion of an additional variable x playing the role of a junction between transitions (j, i) and $(k, l+1)$. Transition $(+x, 1)$ is performed immediately after (k, l) and precedes (j, i) and $(k, l+1)$. The reverse change of x , i.e. $(-x, 2)$, occurs directly before $(j, i+1)$. In such a transformation, we insert a delay before the transition $(k, l+1)$ in such a way that the latter is held up until $(+x, 1)$ occurs. The ordering of the original transitions is not violated by the insertion

because, originally, (k, l) precedes $(k, l+1)$. The excitation functions are then rewritten as follows : $Q_j^i = T_j^i \vee \phi_j^i$ is converted to $Q_j^i = x(T_j^i \vee \phi_j^i)$ where T_j^i is the term which corresponds to the minimum state α of the region $A(j, i)$, and T_j^{i+1} is obtained from T_j^i by deleting z_k ; Q_j^{i+1} is converted into $Q_j^{i+1} = \bar{x}$. For the added variable x : $S_x = z_k^{\alpha_l} \phi_x$ where $\alpha_l = 0$, if (k, l) is the 1-0-transition, and $\alpha_l = 1$, otherwise, and the function ϕ_x provides the locking of changes of x that may be caused by subsequent transitions $(k, l+2), (k, l+4), \dots$, if it is necessary; $R_x = Q_j^{i+1}$. Due to such transformations, a complex takeover in the region $A(j, i)$ will be eliminated. The region $A(x, 1)$ is free from a complex takeover. Any takeovers remaining in the region $A(j, i+1)$ are now "moved" into the region $A(x, 2)$. No other takeovers are left in the region $A(x, 1)$. In other regions, except $A(k, l+1)$, no change has been made. If, in the original circuit $(k, l+1)$ and $(j, i+1)$ are non-concurrent, then the region $A(k, l+1)$ is free from takeover with respect to x .

Let $Q_k^{l+1} = Q_k^{l+1} x$ which eliminates one complex takeover.

If $(k, l+1) \parallel (j, i+1)$, then in the region $A(k, l+1)$ there is a complex takeover with respect to x . For its elimination the above procedure can again be used, which adds another extra variable y . If in the original circuit $(j, i+1) \parallel (k, l+2)$, then in the region $A(x, 2)$ there will be a complex takeover with respect to y . To eliminate this takeover we have to insert a third additional variable z using a similar technique. This will result in the complete elimination of one complex takeover because the region $A(y, 2)$ is free from complex takeover with respect to z due to the fact that before the occurrence of $(-y, 2)$ there must be the occurrence $(-z, 2)$, i.e. $(-y, 2)$ and $(-z, 2)$ are non-concurrent.

Elimination of simple takeovers. The above technique for complex takeovers is not always applicable for simple ones because delaying the transition which causes a simple takeover may result in the violation of the original order of element switching.

Note that the excitation function terms Q_j^i , which are involved in a simple takeover, have a common sub-term P including at least all the variables which are inputs for the region $A(j, i)$. It is impossible to substitute the sub-term P for all the terms of the DNF of Q_j^i . This is because there is another part of the set of operational states of the circuit where the values of variables appearing in P , and that of z_j , coincide with their respective values in the region $A(j, i)$ but z_j is idle in that part. Introducing auxiliary variables into the sub-term P , we can modify the circuit in such a way that in the set of operational states of the modified circuit there will be no states outside the region $A(j, i)$ with z_j being idle and the given sub-term assuming the true values in them.

We introduce a variable x which, during the operational cycle, will be changing only twice. The $(+x, 1)$ -transition is an immediate predecessor for (k, l) , which is

the input for $A(j, l)$, and the $(-x, 2)$ -transition immediately follows (j, i) and precedes those changes (m, t) which in the original circuit may have followed (j, i) . The excitation functions are transformed as follows: $Q_j^l = xP$; $Q_k^l = x$; $Q_m^t = Q_m^t \bar{x}$; $R_x = z_j^{\alpha_i}$ where $\alpha_i = 0$, if (j, i) is the 1-0-transition, and $\alpha_i = 1$, otherwise; $S_x = Q_k^l z_k^{\alpha_l}$. Although $(+x, 1)$ immediately precedes (k, l) , the function S_x may not be simply set equal to Q_k^l . Indeed, since $(-x, 2)$ precedes $(k, l+1)$, the function Q_k^l may remain true (see Example 5.7), in the states that follow $(-x, 2)$ and precede $(k, l+1)$, and it may, thereby, cause a premature return of x to 0. To avoid the latter, Q_k^l can be strengthened by $\bar{z}_k^{\alpha_l}$. As $(+x, 1)$ is followed by (k, l) , $\bar{z}_k^{\alpha_l}$ changes to 0 and cannot go back to 1 until the change $(k, l+1)$ occurs. This guarantees that in all of the above mentioned states $S_x = 0$. *Q.e.d.*

EXAMPLE 5.7. (This illustrates the method for eliminating takeovers).
Let a circuit be given by the system of equations

$$z_1 = \bar{z}_4 (\bar{z}_4 \vee \bar{z}_4), \quad z_2 = \bar{z}_3 \vee z_2 \bar{z}_4,$$

$$z_3 = z_2 (z_1 \vee z_3), \quad z_4 = \bar{z}_1 z_2 z_3$$

whose state-transition diagram is shown in Fig. 5.23(a) and the signal change diagram is presented in Fig. 5.24(a). The circuit contains two takeovers. The first, complex one, is in the region $A(-4, 1)$ and is with respect to z_2 . It occurs in the sequence $001^*1^*-00^*01^*-0101^*$ in the first state of which the term \bar{z}_2 of the excitation function R_4^1 is true, and in the last state another term, \bar{z}_3 , becomes true. The second takeover, now in the region $A(+1, 1)$, is simple. It occurs in the sequence $0^*01^*0-0^*00-0^*100$ where the terms of S_1^1 shift from $\bar{z}_2 \bar{z}_4$ (in state 0^*01^*0) to $\bar{z}_3 \bar{z}_4$ (in state 0^*100).

First we eliminate the complex takeover by introducing an extra variable z_5 as shown in Fig. 5.23(b) and 5.24(b). Since $(+2, 2)$ and $(+4, 2)$ are non-concurrent it will be sufficient to insert just one extra variable. The inherent functions of elements will be modified according to the technique for eliminating a complex takeover and, thus, assume the form

$$z_1 = \bar{z}_4 (\bar{z}_2 \vee \bar{z}_3), \quad z_2 = \bar{z}_3 z_5 \vee z_2 \bar{z}_4, \quad z_3 = z_2 (z_1 \vee z_3),$$

$$z_4 = \bar{z}_5, \quad z_5 = \bar{z}_2 \vee z_5 (z_1 \vee \bar{z}_3).$$

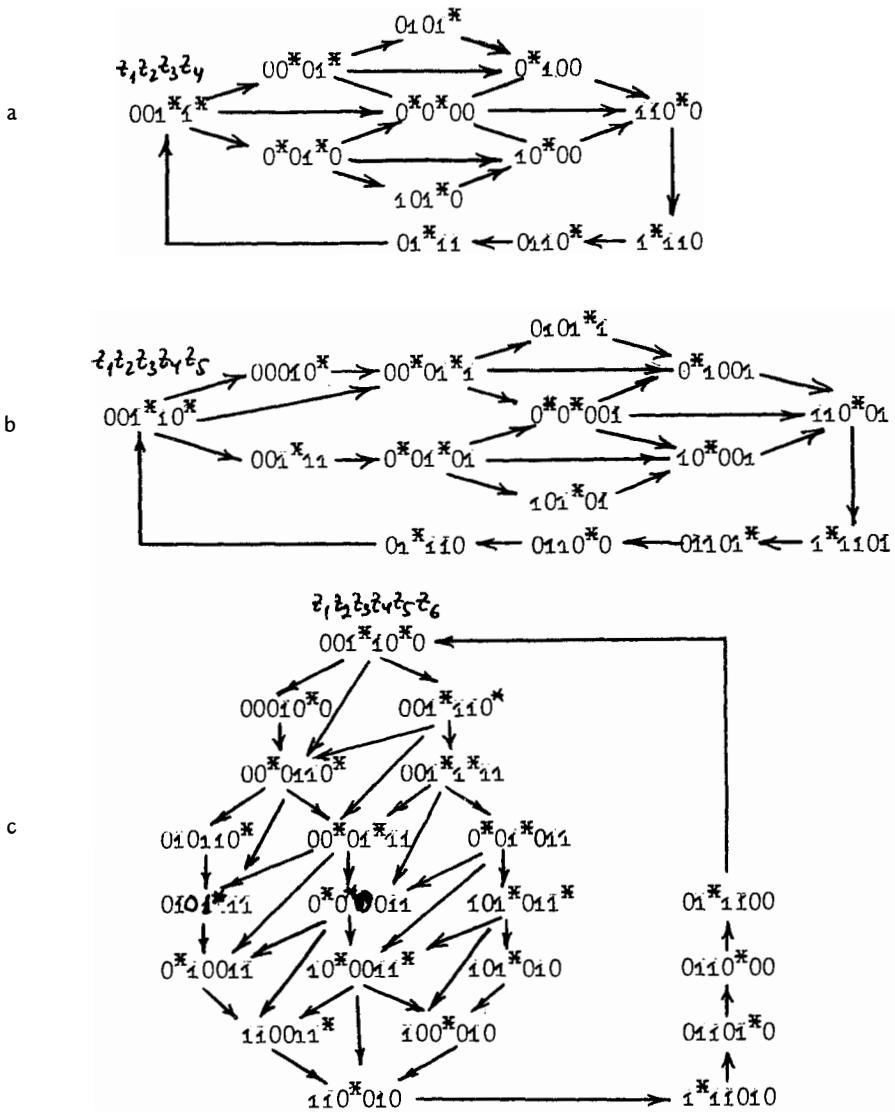


Figure 5.23 (a) - (c). Illustration of takeover elimination method: (a) original state-transition diagram, (b) the result of eliminating a complex takeover, (c) after eliminating a simple takeover.

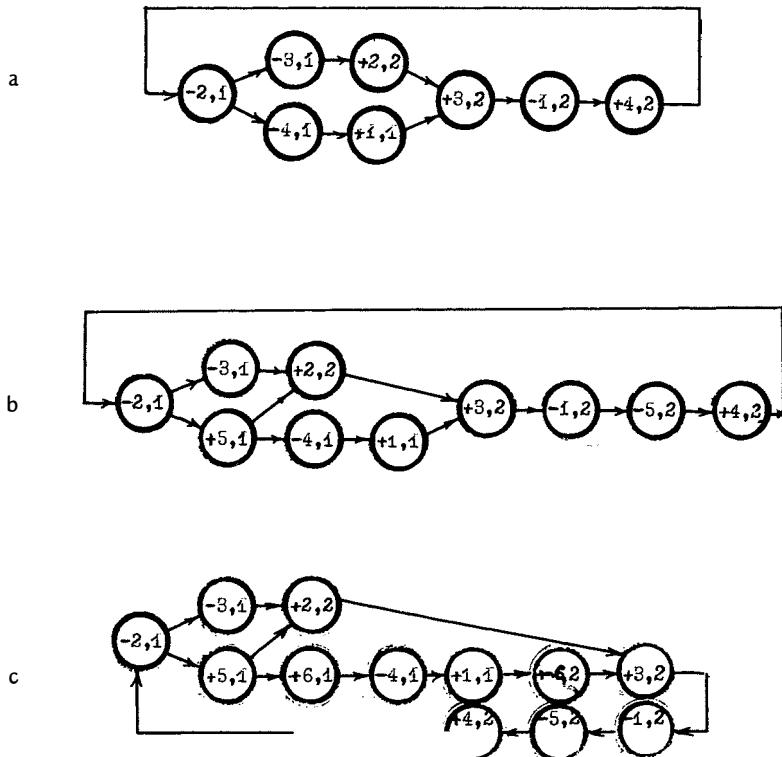


Figure 5.24 (a) - (c). Signal change diagrams corresponding to the takeover elimination steps of Fig. 5.23.

The circuit obtained still contains one simple takeover in the sequence $0^*01^*01-0^*0^*001-0^*1001$ of the region $A(+1, 1)$. It can be remedied by inserting one more auxiliary variable, z_6 , as shown in Fig. 5.23(c) and 5.24(c). The modified excitation functions have the form

$$\dot{S}_1 = \bar{z}_4 z_6, \quad R_1 = R_1 = z_4 \vee z_2 z_3, \quad \dot{S}_2 = S_2 = \bar{z}_3 z_5, \quad \dot{R}_2 = R_2 = z_4,$$

$$\dot{S}_3 = S_3 \bar{z}_6 = z_1 z_2 \bar{z}_6, \quad \dot{R}_3 = R_3 = \bar{z}_2, \quad \dot{S}_4 = S_4 = \bar{z}_5, \quad \dot{R}_4 = R_4 = z_6,$$

$$\dot{S}_5 = S_5 = \bar{z}_2, \quad \dot{R}_5 = R_5 = \bar{z}_1 z_2 z_3, \quad \dot{S}_6 = R_4 z_4 = z_4 z_5, \quad \dot{R}_6 = R_6 = z_1.$$

The system of equations defining the circuit which is takeover-free and, at the same time, equivalent to the original one has the form

$$z_1 = \bar{z}_1 \bar{z}_4 z_6 \vee z_1 \bar{z}_4 (\bar{z}_2 \vee \bar{z}_3), \quad z_2 = \bar{z}_3 z_5 \vee z_2 \bar{z}_4,$$

$$z_3 = z_2 (z_1 \bar{z}_6 \vee z_3), \quad z_4 = \bar{z}_4 \bar{z}_5 \vee z_4 \bar{z}_6,$$

$$z_5 = \bar{z}_2 \vee z_5 (z_1 \vee \bar{z}_3), \quad z_6 = \bar{z}_6 z_4 z_5 \vee z_6 \bar{z}_1.$$

By the partial minimization using states that do not belong to the set of operational states, these functions are represented as follows

$$z_1 = \bar{z}_4 (z_6 \vee \bar{z}_2 \vee \bar{z}_3), \quad z_2 = \bar{z}_3 z_5 \vee z_2 \bar{z}_4, \quad z_3 = z_2 (z_1 \bar{z}_6 \vee z_3),$$

$$z_4 = \bar{z}_4 \bar{z}_5 \vee z_4 \bar{z}_6 \text{ or } z_4 = \bar{z}_6 (z_4 \vee \bar{z}_5),$$

$$z_5 = z_1 \vee \bar{z}_2 \vee \bar{z}_3, \quad z_6 = z_4 z_5 \vee z_6 \bar{z}_1.$$

5.3.5. THE IMPLEMENTATION OF DISTRIBUTIVE AND TOTALLY SEQUENTIAL CIRCUITS

In sub-section 5.3.3 we described a method for constructing the so-called perfect implementation that enables us to synthesize semi-modular circuits using an AND-OR-NOT base. Here, we turn our discussion to the problem of constructing distributive and totally sequential circuits using minimal elementary functional bases of either NAND elements or, dually, NOR elements. This problem has its own specific theoretical importance. Furthermore, we should also take into account the particular assumptions relating the character of element delays and wire delays with respect to a chosen functional basis. The following example clarifies the latter issue.

EXAMPLE 5.8. The circuit of Fig. 5.25 is described by the system

$$z_1 = \bar{z}_3, \quad z_2 = \bar{z}_3, \quad z_3 = z_1 z_2 \vee z_3 (z_2 \vee z_2).$$

If the sub-circuit which is enclosed in the dashed rectangle is considered as an elementary unit, then the circuit operation will obviously be regarded as correct. However, if we build a state-transition diagram picking out variables K_1 , K_2 and K_3 in an explicit way, the picture will be totally different. Fig. 5.26 illustrates a fragment of such a diagram from which we firstly, derive potential sequences of incorrect operation (they are designated by double arrows), and, secondly, it is clear that if after the 1-0-transition of K_3 the elements K_1 and K_2 change faster than either of the elements z_2 and z_3 , then there will be no violation of the correct operation. On the other hand, the reader may be convinced that the circuit of Fig. 5.27 models the

given system of equations with no dependence of element delays, and is, therefore, correct.

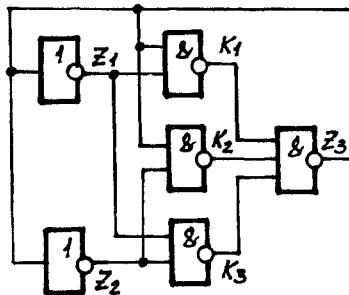


Figure 5.25. Incorrect implementation of the equation system for a two-input Γ -flip-flop and two inverters.

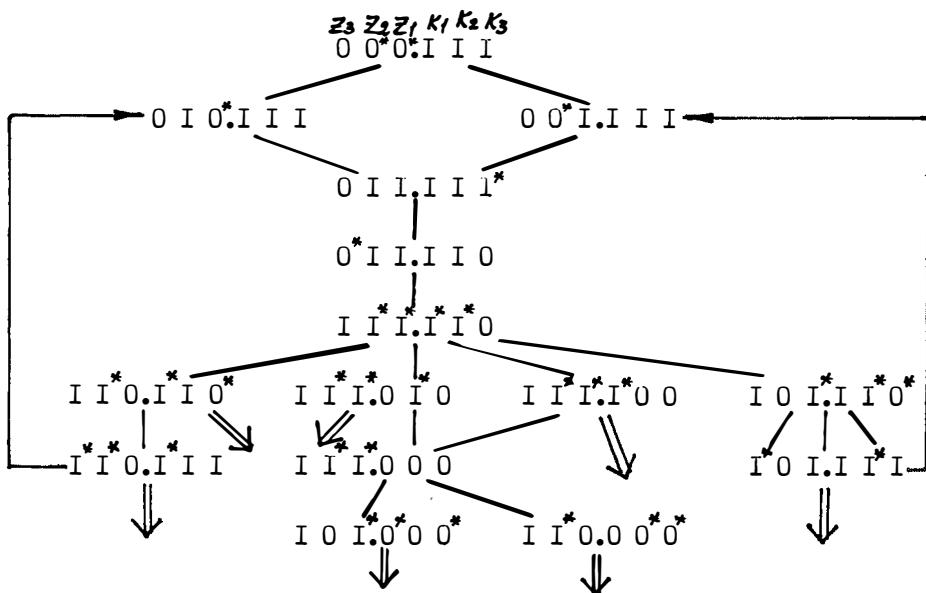


Figure 5.26. A fragment of the transition diagram for the circuit of Fig. 5.25.

These statements justify the search for a minimal circuitry basis for delay-independent circuits and for appropriate synthesis methods that will have, not only practical value, but also an obvious theoretical interest.

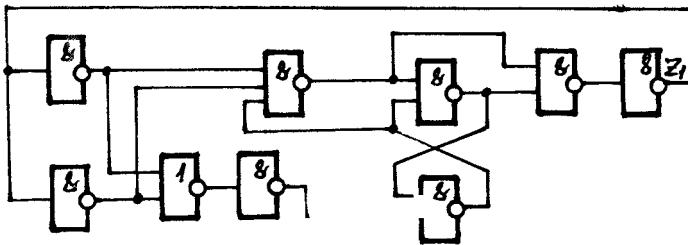


Figure 5.27. Correct implementation for a Γ -flip-flop and two inverters.

As before, assume that an inherent function of any element z_i in a circuit in U_n^* can be represented in the form (5.3). By analogy with (5.5) (see Theorem 5.2) we can obtain a decomposition of (5.3) associated with a two-layered circuit composed of NAND elements where the first layer realizes the RS-flip-flop outputs z_i and \bar{z}_i , and the second layer realizes the terms of functions w_i and v_i . The disjunction of these terms is realized directly by the element which constitutes the RS-flip-flops. However, the implementation obtained by this decomposition will not be correct. In fact, having a second layer, according to Theorem 5.2, we should provide input variables for functions w_i and v_i in complemented form. Such complementing implies, above all, the changing output values when $z_i = \bar{z}_i$. While in a single-layer implementation on NAND elements, the states in which $z_i = \bar{z}_i = 1$ are in the class $T_j, j \neq i$, in the double-layer implementation, we require that $z_i = \bar{z}_i = 0$. This requirement can be realized if flip-flops of the type shown in Fig. 5.28 are used.

We start our examination with totally sequential circuits, i.e. in the class K_n^* of circuits, where no more than one element may be in an excited state at a time.

THEOREM 5.6. *A system of inherent functions of NAND elements is functionally complete in the class K_n^* , i.e. it is possible to realize correctly any totally sequential circuit using only NAND elements.*

Proof. Each state $\alpha^k = \alpha_1^k \dots \alpha_n^k$ belonging to the set of operational states of a totally sequential circuit from $K_n^*, n > 1$, can be associated with an element t_k implementing

the function $t_k = t_{k-1} z_1^{\alpha_1^k} \dots z_n^{\alpha_n^k}$, and each element z_i of the circuit can then be associated with an RS-flip-flop built from NANDs. Element t_k is connected to either the input R_i of the flip-flop \dot{z}_i if $\alpha^k \in R_i$ or to the input S_i of the flip-flop \dot{z}_i if $\alpha^k \in S_i$, and has no connection at all to the inputs of \dot{z}_i if $\alpha^k \in T_i$.

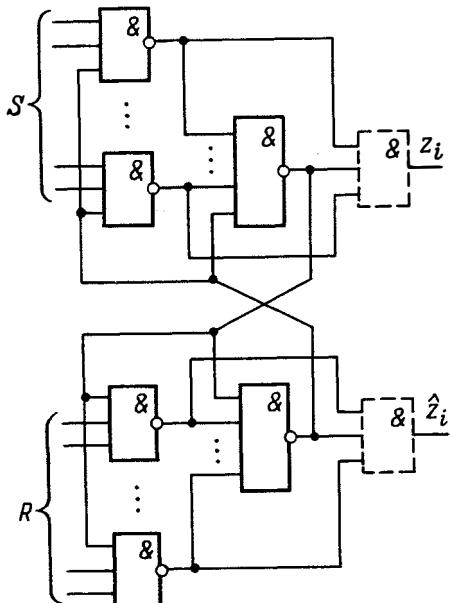


Figure 5.28. Basic construct for implementing distributive and totally sequential circuits.

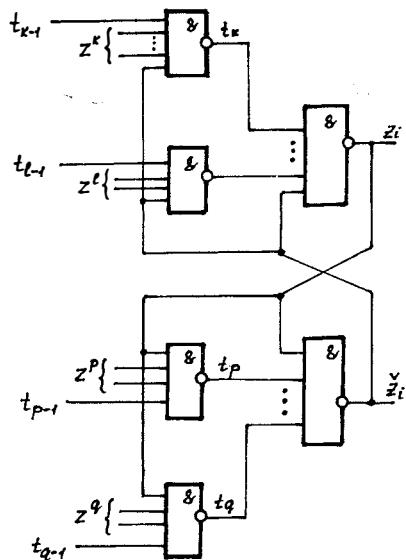


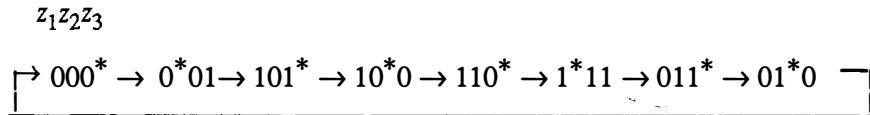
Figure 5.29. Illustration for the proof of correct implementability of totally sequential circuits within a NAND basis.

If the circuit is in state α^k then $t_k = 0$. Assume, without loss of generality that $\alpha^k \in S_i$ (Fig. 5.29) and $\alpha^l \in S_i$, $\alpha^p \in R_i$, $\alpha^q \in R_i$. Since $t_k = 0$ it causes flip-flop \dot{z}_i to be set. As soon as the flip-flop \dot{z}_i assumes the state where $z_i = 1$ and $\dot{z}_i = 0$, the element t_k changes to 1. After this change, the element t_{k+1} also changes, which is equivalent to the transition of the circuit from state α^k to state α^{k+1} . This transition is made by changing the value of z_i in the original circuit, whereas in the

implementation of the type in Fig. 5.29 it is done through a sequence of changes of z_i , \bar{z}_i , t_k , t_{k+1} .

The circuit obtained in this way is also totally sequential and belongs to the class K_m^* , $m \geq 4n$. The ordering of its flip-flop switchings is identical to the original circuit from K_n^* . Therefore, the implementation obtained is correct. *Q.e.d.*

EXAMPLE 5.9. Let a totally sequential state-transition diagram be given as follows



where the changes in z_3 are twice as frequent as those in z_1 and z_2 , i.e. this diagram describes the behaviour of a complementing flip-flop (or the so-called scale of two counter). From this diagram we derive (see Section 2.4) a system of equations of the form

$$z_1 = z_1 \bar{z}_3 \vee \bar{z}_2 z_3, \quad z_2 = z_1 \bar{z}_3 \vee z_2 z_3, \quad z_3 = z_1 z_2 \vee \bar{z}_1 \bar{z}_2.$$

According to the definition of t_k presented in the proof of Theorem 5.4, we obtain

$$\begin{aligned} t_0 &= \overline{t_7 \bar{z}_1 \bar{z}_2 \bar{z}_3}, & t_1 &= \overline{t_0 \bar{z}_1 \bar{z}_2 z_3}, & t_2 &= \overline{t_1 z_1 \bar{z}_2 z_3}, \\ t_3 &= \overline{t_2 z_1 \bar{z}_2 \bar{z}_3}, & t_4 &= \overline{t_3 z_1 z_2 \bar{z}_3}, & t_5 &= \overline{t_4 z_1 z_2 z_3}, \\ t_6 &= \overline{t_5 \bar{z}_1 z_2 z_3}, & t_7 &= \overline{t_6 z_1 \bar{z}_2 \bar{z}_3}. \end{aligned}$$

The outputs of NAND gates with inherent functions t_0-t_7 are connected to the RS-flip-flops \bar{z}_1 , \bar{z}_2 and \bar{z}_3 as shown in Fig. 5.30.

To implement a circuit from the class $U_n^* K_n^*$ (semi-modular but not totally sequential) by using flip-flops of the type shown in Fig. 5.29, the number of elements of the form t_k for each state α^k should be increased, respectively, with the number of elements of the form z_i excited in this state. Minimizing the inherent functions of elements z_i in accordance with Theorem 5.3 we obtain the representation in which each element t_k is associated with a single term of the reduced DNF of w_i or v_i in expressions of the form (5.5).

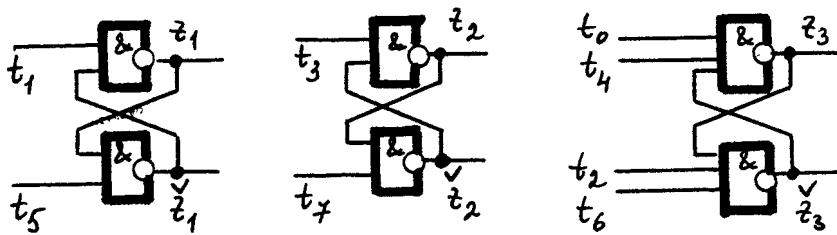


Figure 5.30. A variant of a complementing flip-flop circuit built on the base of Fig. 5.29 (eight four-input gates t_0-t_7 are not shown).

Furthermore, in these expressions for $z_j, j \neq i$, we substitute the terms $z_i t_k \dots t_l$ and $\bar{z}_i t_p \dots t_k$ for variables z_i and \bar{z}_i , respectively, because the 0–1 (1–0)-transitions of variable z_i in the circuit Fig. 5.28 result in a setting where

$$z_i = t_k = \dots = t_l = 1 \quad (\bar{z}_i = t_p = \dots = t_q = 1).$$

However, such a modification is by no means sufficient for obtaining a correct implementation for arbitrary circuits in $U_n^* \setminus K_n^*$ by NAND elements. In fact, since the excitation functions which cause transitions of an element may be multi-termal, the elements of the first layer (flip-flops of Fig. 5.28) may suffer from logical hazards leading to the violation of semi-modularity. As has been noted, previously, there are two main causes of non-unitermality: (i) detonant transitions, (ii) term takeovers. In compliance with Theorem 5.5, every semi-modular circuit can be transformed to a circuit free from takeovers, and furthermore, as distributive circuits are free from detonant transitions, the following theorem holds.

THEOREM 5.7. *A system of inherent functions of NAND elements is functionally complete in the class D_n^* , i.e. every distributive circuit can be correctly implemented using only NAND gates.*

Examples 5.7 to 5.9 illustrate the canonical approaches to the synthesis of semi-modular, totally sequential and distributive circuits. Circuits obtained by applying these approaches are referred to as those of the standard implementation class related to the implementations which guarantee the minimum cost of the design process, but at a penalty in the size of the circuit.

5.4 Synthesis of semi-modular circuits in limited bases

So far we have not touched on the question of how the implementability of a semi-modular circuit is affected by imposing limitations on the allowed number of inputs (fan-in) and fan-out capability of logical elements.

In this section, we prove, constructively, the fact that *any distributive circuit in D_n^* can be correctly implemented in the functional basis of 2-NAND elements, i.e. elements that have a minimum value (2) of fan-in and fan-out.*

During the operational cycle of a circuit, each variable z_j may change several, say m_j , times before the circuit returns to the initial state. Therefore, the number of changes of each variable z_i can be counted by modulo m_j . Every transition (j, i) where $1 \leq i \leq m_j$ is associated with a Boolean variable z_j^i that assumes the value of 1 after the transition (j, i) , but before the transition $(j, i+1)$, takes place. (Both addition and subtraction of indexes are meant to be modulo m_j , i.e. $m_j + 1 = 1$ while $1 - 1 = m_j$.) In other words, referring to a state-transition diagram of an original circuit, z_j^i is equal to 1 in the states which follow the state from the region $A(j, i)$ including the states of the region $A(j, i+1)$, and z_j^i is equal to 0 in all remaining states.

The sequential process of signal changes of variables $z_j^1 z_j^2 \dots z_j^{m_j}$ representing z_j can be expressed as a cyclic shift of 1 in the unitary encoding $00\dots01 \rightarrow 10\dots00 \rightarrow 01\dots00 \rightarrow \dots \rightarrow 00\dots10 \rightarrow 00\dots01 \rightarrow \dots$. Let the shift control function for the i -th variable, which is actually an excitation function for the 0–1-transition of z_j^i , be denoted by ψ_j^i . Q_j^i denotes the excitation function for (j, i) -transition in the original circuit, i.e. the part of S_j or R_j which causes the excitation of z_j in the region $A(j, i)$. For simple distributive circuits each Q_j^i contains exactly one term (unital). ψ_j^i can be obtained from Q_j^i by substituting variable z_k^l for each entry of z_k (to \bar{z}_k) where l is the number of the last transition of variable z_k which precedes the region $A(j, i)$ in the original circuit.

EXAMPLE 5.10. A particular case of a simple distributive circuit is defined by the totally sequential state-transition diagram shown in Fig. 5.31(a), (state 0^*00 is the initial state). This diagram corresponds to the system

$$z_1 = \bar{z}_2 \bar{z}_3 \vee z_2 z_3, \quad z_2 = z_1 \bar{z}_3 \vee z_2 \bar{z}_1, \quad z_3 = \bar{z}_1 z_2 \vee z_3 z_1.$$

Variables z_2 and z_3 change twice during the cycle, whereas z_1 changes four times a cycle.

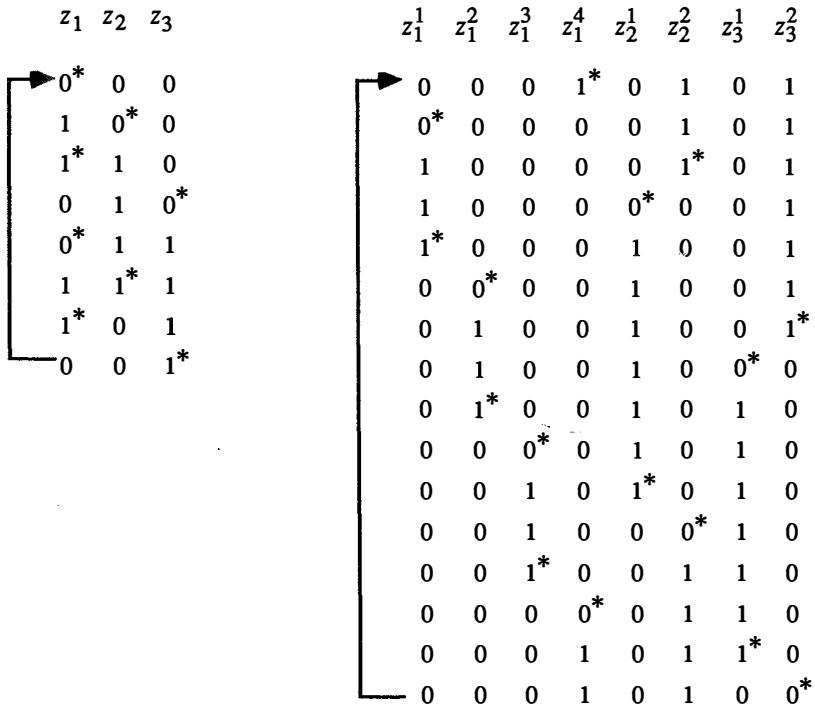


Figure 5.31 (a) and (b). (a) State-transition diagram for a totally sequential circuit, (b) diagram for variables z_k^l .

Associated excitation functions can be expressed in the form

$$Q_1^1 = \bar{z}_2 \bar{z}_3, \quad Q_1^2 = z_2 \bar{z}_3, \quad Q_1^3 = z_2 z_3, \quad Q_1^4 = \bar{z}_2 z_3,$$

$$Q_2^1 = z_1 \bar{z}_3, \quad Q_2^2 = z_1 z_3, \quad Q_3^1 = \bar{z}_1 z_2, \quad Q_3^2 = \bar{z}_1 \bar{z}_2.$$

The state-transition diagram for variables z_k^l is shown in Fig. 5.31(b). The shift control functions are

$$\psi_1^1 = z_2^2 z_3^2, \quad \psi_1^2 = z_2^1 z_3^2, \quad \psi_1^3 = z_2^1 z_3^1, \quad \psi_1^4 = z_2^2 z_3^1,$$

$$\psi_2^1 = z_1^1 z_3^2, \quad \psi_2^2 = z_1^3 z_3^1, \quad \psi_3^1 = z_1^2 z_2^1, \quad \psi_3^2 = z_1^4 z_2^2.$$

As a framework for the implementation of distributive circuits, we can use a signal distribution network, the token shifter, based on the so-called *David cell*,

whose circuit was shown earlier, in Fig. 5.3.⁽¹⁾ The behaviour of that cell in interaction with its neighbour cells can be described by the following diagram

y_{i-1}	\dot{x}_{i-1}	x_{i-1}	y_j	\dot{x}_i	
1*	1	0	1	1	
0	1	0*	1	1	– activation signal to the j -th cell from the $(j-1)$ -th cell,
0	1*	1	1	1	
0*	0	1	1	1	– reply from the j -th cell to the $(j-1)$ -th cell about the receipt of the activation signal,
1	0	1	1*	1	– removal of the activation signal by the $(j-1)$ -th cell,
1	0	1	0	1*	– activation signal to the $(j+1)$ -th cell from the j -th cell,
1	0*	1	0	0	– reply from the $(j-1)$ -th cell to the j -th cell about the receipt of the activation signal,
1	1	1*	0	0	
1	1	0	0*	0	
1	1	0	1	0*	– removal of the activation signal by the j -th cell.

Fig. 5.32(a) demonstrates a construction that can be used for the implementation of a simple distributive circuit by NAND elements.

Each variable z_j associated with a module V_j consisting of the circuit B_j , which is used for modelling the behaviour of variable z_j , and the token shifter S_j . The module V_j contains a series of switching sub-modules V_j^i which are structurally identical and interconnected to form a ring. The switching sub-module V_j^i consists of two basic David cells of the type of Fig. 5.3 designated as $D_{j,i}^1$ and $D_{j,i}^2$, an inverter whose output is applied to other modules as a value of variable z_j^{i-1} , and element $\dot{\psi}_j^i$ implementing the conjunction ψ_j^i . When the flip-flop in cell $D_{j,i}^1$ is set to ($x_{k-1} = 1$, $\dot{x}_{k-1} = 0$), the module V_j is ready for modelling the transition (j, i) and the termination of the preceding transition $(j, i-1)$ can be signalled to the

(1) The only difference is in the renamed variables: p and q are replaced by y and x , respectively.

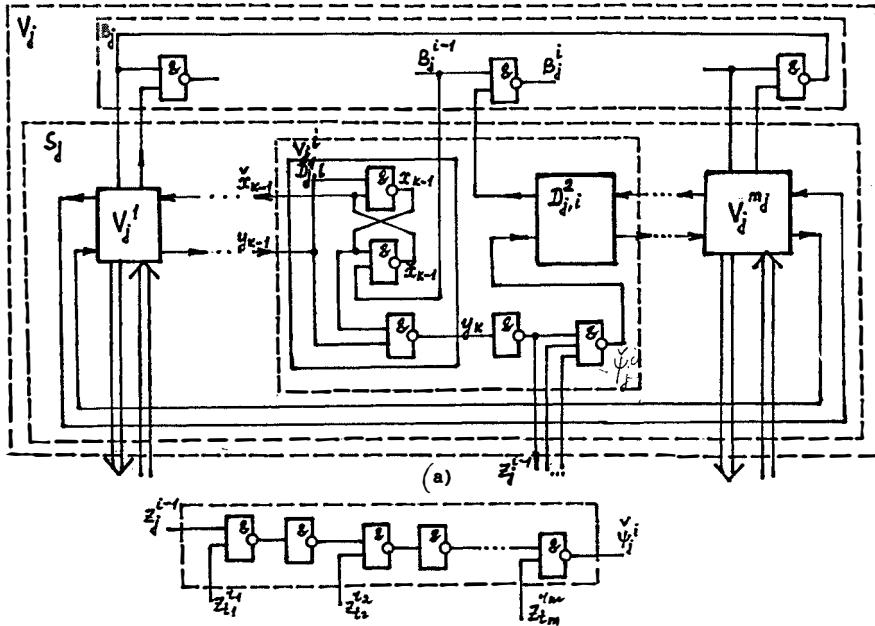


Figure 5.32 (a) and (b). Illustration of proof of the implementability of distributive circuits within 2-NAND-2 basis: (a) construction, (b) implementation of ψ_j^i term.

other modules from the output z_j^{i-1} . Once ψ_j^i becomes equal to 1, the flip-flop in the cell $D_{j,i}^2$ is set to 1 and the element B_j^i changes from 0 to 1, thus causing a sequence of changes in the flip-flop B_j . This sequence terminates with setting B_j^{i-1} to 0. This signal sets the flip-flop in $D_{j,i}^1$ back to 0, and the shift of 1 from $D_{j,i}^2$ to $D_{j,i+1}^1$ is made. The circuit V_j is ready for modelling the next transition of z_j , and the other modules are signalled about the termination of the (j, i) -th transition by the signal z_j^i .

It is easily seen that associating each variable z_j with any odd-numbered variable of B_j , say B_j^1 , if in the initial state of the original circuit, the value of z_j is equal to 0, and with any even-numbered variable of B_j , say B_j^2 , otherwise (if the value of z_j is equal to 1), we obtain a circuit that exactly models the ordering of the element switchings in the original circuit.

In the circuit of Fig. 5.32(a) all elements, except z_j^{i-1} have fan-outs equal to 2. The problem of increasing the loading capability of z_j^{i-1} can be easily solved by inserting a chain of inverters in series with z_j^{i-1} . The number of inverters should be twice the fan-out of elements. By this, the *minimum fan-out per element is reduced to 2*.

The other problem related to the *minimum number of inputs of NAND elements* can be tackled in the following way.

All the elements, except one that realizes the term ψ_j^i , have at most two inputs. Element ψ_j^i can be implemented by connecting a 2-NAND-2 element and inverters in series, as shown in Fig. 5.32(b). To obtain the resulting circuit which will be delay-independent with respect to element delays we should also take into account the allowed order of variables as they enter into the term. In fact, if $z_j^{i-1} = 1$, the variable $z_{t_1}^{r_1}$ must either stay equal to 1 or, if it was equal to 0, change by performing a single 0–1-transition. A similar requirement is imposed on variable $z_{t_2}^{r_2}$ if $z_j^{i-1} z_{t_1}^{r_1} = 1$ etc. It can be proved that, for simple circuits, such an ordering always exists. The proof is based on the fact that during the 0–1-change of variable z_j^{i-1} there is at least one variable entering into the term ψ_j^i and which is not equal to 1. Otherwise, the condition for the change would have been true already. Because each variable $z_{t_l}^{r_l}$ changes twice during the whole operational cycle, this variable will make the 0–1-change only once. Any of the variables which meets this requirement can be chosen as the first one. Once $z_j^{i-1} z_{t_1}^{r_1} = 1$, the transition condition is, again, either true, or there exists a variable entering into the term ψ_j^i which is not equal to 1, etc.

Thus, the following theorem can be stated.

THEOREM 5.8. *A system of inherent functions of 2-NAND-2 elements is functionally complete in the class D_n^* , i.e. every distributive circuit can be implemented correctly using only 2-NAND-2 elements.*

Indeed, according to Theorem 5.7, for any distributive circuit C we can build a distributive simple circuit C' equivalent to C . Using the above technique and basic construction, we can then transform C' into a circuit C^* satisfying the following three conditions:

- (1) C^* is built of 2-NAND-2 elements,
- (2) C^* is semi-modular and distributive,

(3) C^* is equivalent to C with respect to the set Z of variables of the original circuit.

Note that in the construction of Fig. 5.32(a) there is no direct connection between the elements which model the behaviour of basic variables, i.e. those variables that are given in the original circuit. However, the outputs of these elements may be used for the link with the environment by inserting matched aperiodic implementations into the breaks of wires connecting the elements V_j^i and V_j^{i+1} .

The next item to be discussed is *how to implement circuits from the class U_n^* in a limited basis of functional elements*. We have shown that it is sufficient to study only the implementations of simple semi-modular circuits whose detonancy rank is not higher than 2, and the terms of their excitation functions for detonant transitions are elementary, i.e. have the form $Q_j^i = z_k^\sigma \vee z_m^\sigma$.

The general idea of modelling corresponds to the structural representation shown in Fig. 5.33(a). Any deviations from that representation would be due to particular implementation techniques for detonant transitions, for which ψ_j^i has the form $\psi_j^i = z_k^l \vee z_m^t$ and, thus, can be implemented according to Fig. 5.33(b). A seemingly evident solution using the circuit realizing $z_j^{i-1}(z_k^l \vee z_m^t)$ would be incorrect because it does not guarantee semi-modularity of the circuit. Thus, the first task is to realize the function ψ_j^i which corresponds to the detonant transition (j, i) . The second task is to find a technique for appropriate idling of the circuit ψ_j^i . For non-detonant transitions, the idling precedes the removal of the indication signals from the inputs of modules associated with the other variables. Such an organization is not allowed for detonant transitions. Indeed, when a detonant transition occurs, there is no guarantee that both of its input transitions have already occurred: we can only guarantee the occurrence of at least one of them. The idling process, starting immediately after the detonant transition (j, i) caused by the first of its weak predecessors, may be too premature in the sense that it will be concurrent to the second input transition, thereby violating semi-modularity.

The idea of realizing a detonant transition (j, i) consists of organizing a parallel execution of processes in the module V_j^i . The latter is constructed as a two-layered structure (see Fig. 5.33(a)). The first layer which models the transition (j, i) consists of a bifurcator cell $D_{j,i}^1$ that enables these parallel processes, cells $D_{j,i}^2$ and $D_{j,i}^4$, and two inverters. From the output of the first inverter, a signal z_j^{i-1} , can be taken. As in the case of distributive circuits, the fan-out problem for z_j^{i-1} is solved by using a chain of inverters. The second layer in which the excitation

function is “assembled” consists of the circuit ψ_j^i , two cells $D_{j,i}^3$ and $D_{j,i}^5$, and, possibly, $D_{j,i}^6$.

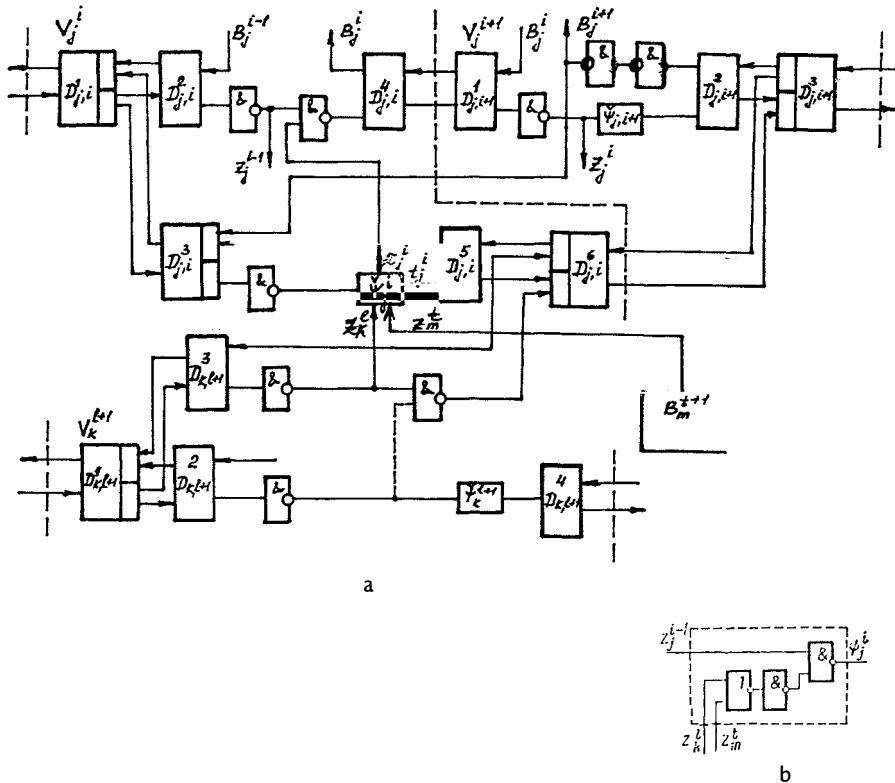


Figure 5.33 (a) and (b). Illustration of the implementation of semi-modular circuits within a limited basis: (a) variant of modelling a detonant transition, (b) incorrect implementation of excitation function for a detonant transition.

From the bifurcator cell shown in Fig. 5.34(a) the token (“1”) is shifted to the subsequent cells $D_{j,i}^2$ and $D_{j,i}^3$. After the setting of tokens in both these cells, the bifurcator cell is reset to 0, and then the processes start to flow in the layers: the output z_j^{i-1} assumes the value 1 which is accepted in associated modules as a signal indicating the transition $(j, i - 1)$, thus opening the circuit ψ_j^i to sense the changes to its inputs. The implementation of ψ_j^i shown in Fig. 5.34(b) contains a three-input Γ -flip-flop. Fig. 5.34(c) is an implementation of the latter by two-input Γ -flip-flops

which implement the function $z = x_1 x_2 x_3 \vee z(x_1 \vee x_2 \vee x_3)$ where x_1, x_2, x_3 are input variables and z is the output variable of the flip-flop. When any one of the transitions (k, l) or (m, t) ($z_k^l = 1$ or $z_m^t = 1$) occurs, one of the outputs τ_j^i of the circuit $\dot{\Psi}_j^i$ will go to 1. The flip-flop in $D_{j,i}^4$ will be set to 1, and a series of element switchings in the flip-flop B_j will finally cause the flip-flop in $D_{j,i}^2$ to be reset to 0, and a token will be shifted from $D_{j,i}^4$ to the first cell $D_{j,i+1}^1$ of the next module V_j^{i+1} . Then $z_j = 1$ and the subsequent process will be performed in the described way.

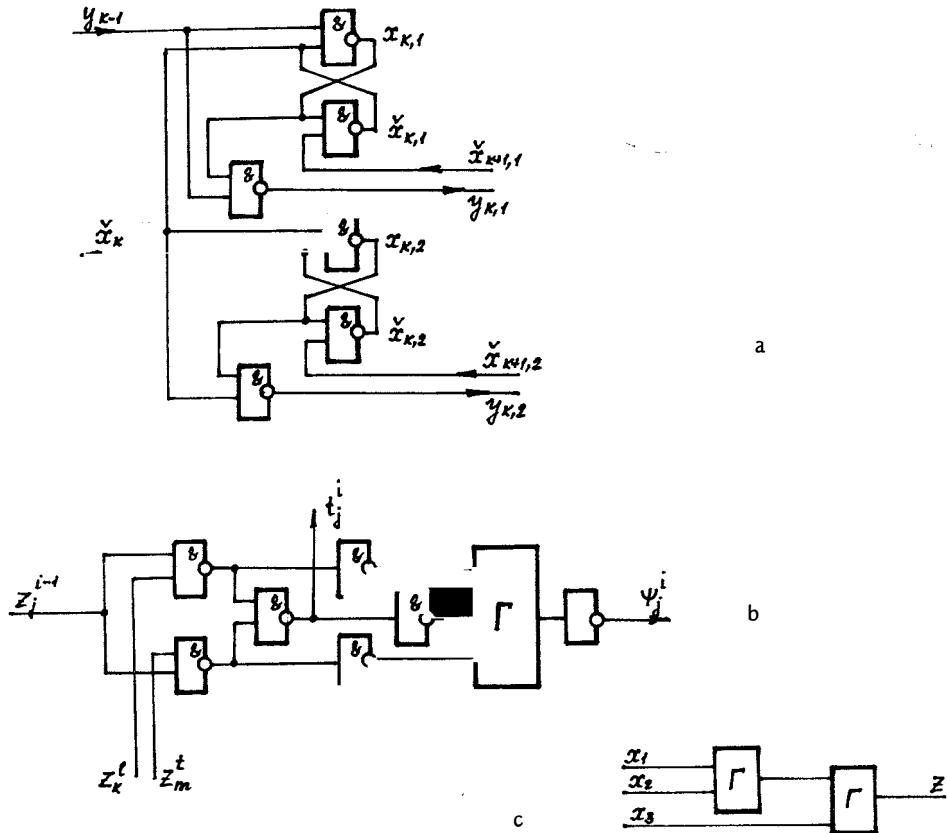


Figure 5.34 (a) - (c). Implementation of basic cells by David's elements: (a) bifurcator, (b) circuit $\dot{\Psi}_j^i$ with three-input Γ -flip-flop, (c) the implementation of the latter by two-input Γ -flip-flops.

To explain the idling process in circuit $\dot{\Psi}_j^i$ we introduce several new concepts.

DEFINITION 5.8. The *focus of a detonant transition* (j, i) is a transition (r, s) such that:

- (1) (r, s) cannot occur sooner than both of the transitions (k, l) and (m, t) which are input for (j, i) and the transition (j, i) itself, and
- (2) there is no transition (r', s') such that satisfies the condition (1) and precedes the transition (r, s) .

In general, a detonant transition may have several mutually un-ordered foci.

A transition $(j, i+p)$, $p > 0$, is called *synchronizing for a detonant transition* (j, i) if it is the first transition of z_j to follow (j, i) and not permitted to occur prior to each of the foci of (j, i) .

In other words, by the time a synchronizing transition occurs, all focus transitions and both input transitions for (j, i) should have occurred. This allows the synchronizing, or, better, say, joining, of the processes, which are forked in the module responsible for a detonant transition, in the module which is responsible for an appropriate synchronizing transition.

The definitions introduced are illustrated in Fig. 5.35(a). The synchronizing transition for (j, i) is the transition $(j, i+1)$. Fig. 5.35(b) shows that the module V_j^{i+1} has an additional cell, the synchronizer $D_{j,i+1}^3$, which is set to 1 only after both of the inputs connected to the preceding cells are changed to 0. After the setting to 1 of the synchronizer flip-flop, both of the preceding cells are reset to 0, and “1” is shifted to the subsequent cell in the module V_j^{i+2} . The cell $D_{j,i+1}^3$ of the second layer is built on the base of the bifurcator circuit.

If the next transitions $(k, l+1)$ and $(m, t+1)$ of the variables z_k and z_m , which are input for the transition (j, i) , follow the transition which is synchronizing for (j, i) , as shown in Fig. 5.35(a), then no more modifications of the circuit are needed. Rather, let $(k, l+1)$ occur prior to the synchronizing transition. Then the change z_k^l to 0 may occur earlier than the moment when the circuit $\dot{\Psi}_j^i$ is idled, which makes the latter non-semi-modular. Delaying the occurrence of transition $(k, l+1)$ until the completion of the idling process in $\dot{\Psi}_j^i$ is also disallowed, since it will necessitate insertion of an additional synchronizing connection from (m, t) to $(k, l+1)$ which will violate the principle of preservation of the original ordering semantics.

A correct solution can be obtained by making use of an extra cell $D_{k,l+1}^3$ in the module V_k^{l+1} that will play the role of a flag storing the fact of the completion of transition (k, l) specially for the circuit $\dot{\Psi}_j^i$. The resetting of such information is not a necessary condition for performing the subsequent transition $(k, l+1)$ because it will occur only after the idling of $\dot{\Psi}_j^i$.

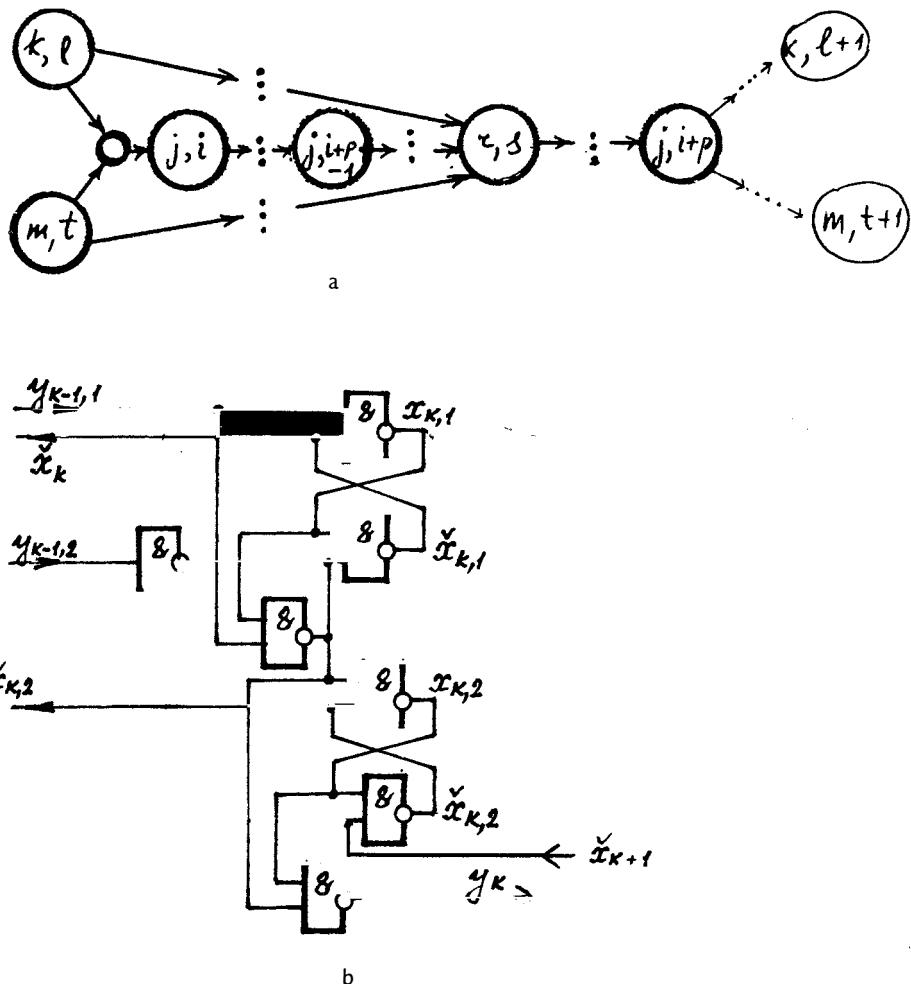


Figure 5.35 (a) and (b). (a) Signal change diagram with detonant transition; (b) the implementation of a synchronizer by David cells.

It can be easily seen, in the above construction, that a need to fork and join two more processes, may arise, as for example, in the case of nested detonant transitions. Appropriate implementations of the pyramidal form can be built of bifurcators (Fig. 5.34(a)) and/or synchronizers (Fig. 5.34(b)). Note that when the number of a synchronizing transition $i + p$ is taken modulo m_j and is equal to i , we can apply a very simple technique of doubling the number of transitions for variable z_j .

From the above method we can obtain the following.

THEOREM 5.9. *A system of inherent functions of 2-NAND-2 and 2-NOR-2 elements is functionally complete in the class U_n^* , i.e. any semi-modular circuit can be correctly implemented using only 2-NAND-2 and 2-NOR-2 elements.*

In this section, we achieved our major goal – to prove, in a constructive way, that the correct implementations of semi-modular circuits can be obtained by using elements with bounded fan-in and fan-out capabilities. Such a constraint often leads to very cumbersome realizations but the knowledge that they are feasible in principle provides the designer with assurance of the effectiveness of the synthesis process. Furthermore, the proposed implementations can be made substantially more compact as the allowed fan-in and fan-out factors are increased.

Another interesting problem concerns the minimality of the implementation basis of $\{2\text{-NAND-2} + 2\text{-NOR-2}\}$ for semi-modular circuits, i.e. *whether it is possible to implement any semi-modular circuit using only 2-NAND-2 (or, dually 2-NOR-2) elements*. This problem is still open. Moreover, it is also not known whether any semi-modular circuit can be built of NAND gates with arbitrary fan-in and fan-out capabilities. The solution for such a problem could be easily found if we abandoned the requirement of semi-modularity with respect to every element in the circuit. However, circuits obtained in such a “compromizing” way, i.e. semi-modular with respect to a sub-set of variables, could operate correctly only with elements having ideal inertial delays. The construction of such elements is obviously unrealistic.

We conclude this section with two significant observations. The first follows as a corollary of Theorem 5.9 and states that any semi-modular circuit can be correctly implemented using elements with inherent functions of the type $\overline{ab} \vee c$ (three-input AND-OR-NOT elements). The second fact is that all results obtained in Sections 5.3 and 5.4 for classes U^* , D^* and K^* can be generalized for classes U , D and K , respectively, without any restrictions on a set of operational states of circuits.

5.5 Modelling pipeline processes

Definition 2.10 gave us the idea of a *pipeline* asynchronous process. Note that in Example 2.7 the i -th event in the Petri net of Fig. 2.8(b) becomes enabled only when both $(i - 1)$ -th and $(i + 1)$ -th events have fired. According to Fig. 5.1(a), (b) and (c) we can build a circuit modelling this net. Taking into account that each task requires two phases (working phase and idle phase) in its operation, we define the ordering of the phases for three successive tasks P_{i-1} , P_i and P_{i+1} , interpreted as $(i - 1)$ -th, i -th and $(i + 1)$ -th events in a corresponding Petri net, by the state-transition diagram shown in Fig. 5.36.

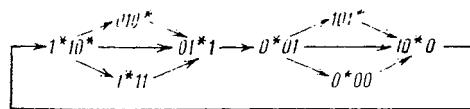


Figure 5.36. State-transition diagram for a pipeline process with variables z_{i-1} , z_i , z_{i+1} .

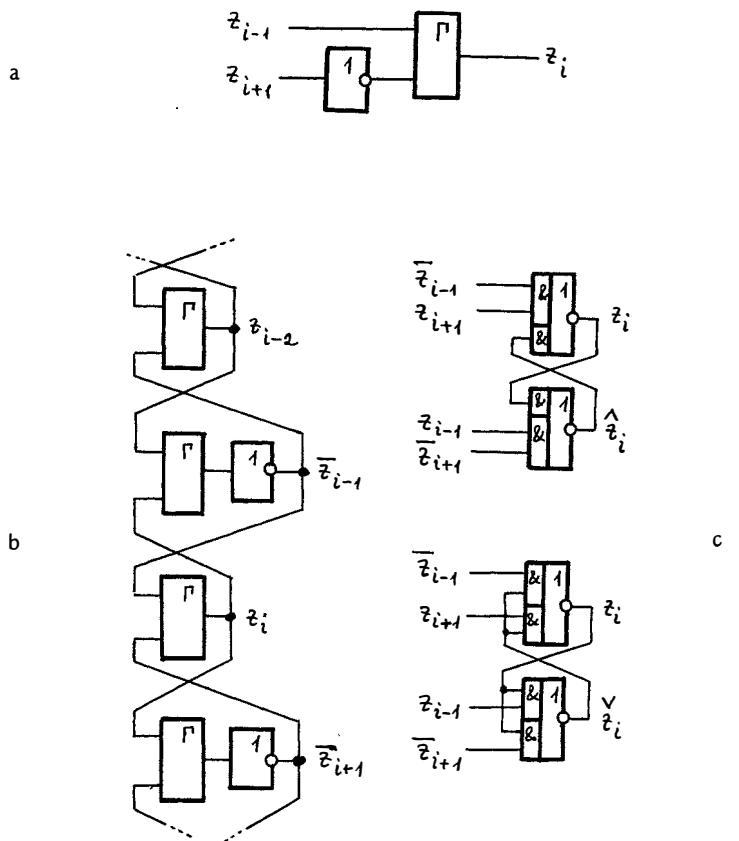


Figure 5.37 (a) - (d). Implementation of a pipeline process: (a) cell consisting of a Γ -flip-flop and inverter; (b) interconnection with a reduced number of inverters; (c) and (d) cells built of flip-flops with separate inputs.

From the truth table built for this diagram (Figure 5.36) we obtain the equation for the i -th element of the modelling circuit in the form

$$z_i = z_{i-1} \bar{z}_{i+1} \vee z_i (z_{i-1} \vee \bar{z}_{i+1}).$$

The direct implementation of this equation is presented in Fig. 5.37(a). The number of inverters can be reduced if we use an alternative circuit shown in Fig. 5.37(b). Fig. 5.37(c) and (d) illustrate the pipeline circuits built of flip-flops with all-zero and all-one transient states, respectively. The switching time for these flip-flops, as for Γ -flip-flops, is equal to a cumulative delay of two elements.

For convenience in the subsequent discussion, we accept that \mathcal{P}_i (working state) denotes the state with $z_i = 1$, and \mathcal{I}_i (idle state) denotes one with $z_i = 0$. The behaviour of a modelling circuit can be specified by a state graph such as, for example, that shown in Fig. 5.38 which corresponds to implementations of Fig. 5.37 (c) and (d).

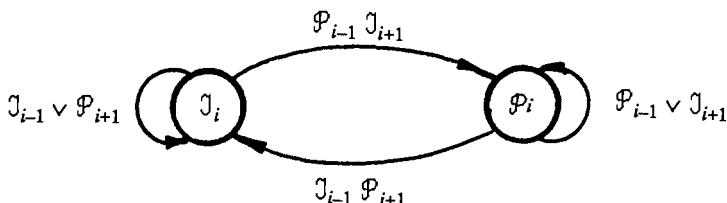


Figure 5.38. State graph for a circuit corresponding to implementations of Fig. 5.37.

5.5.1. SOME PROPERTIES OF MODELLING PIPELINE CIRCUITS

As the reader may have noticed in Fig. 2.8(b), event r_2 models the destination. If, due to some reason, the destination does not operate, i.e. event r_2 does not fire, then the source which is modelled by event r_1 may “inject” at most four tokens to the net. In general, if the destination is inactive the source may fire n times, i.e. initiate the change of phase in a pipeline process at most n times. Since each task has two phases, the overall number of initiated tasks is $n/2$. Similarly, the destination may “unload” the net, if the source for some reason halts the loading, and thus “consume” at most n tokens. Therefore, *pipeline modelling circuits exhibit certain buffering properties*.

It is important that *the rate with which tasks are enabled does not depend on the number of events in a modelling pipeline net*. For a totally sequential process (Fig. 2.8(a)) it is equal to nt where n is the number of events. Thus, in the pipeline organization of a process, its average speed grows up to $\lceil n/2 \rceil$ times that of a non-pipeline organization.

We now try to construct modelling circuits that will be basic for pipeline modules with the so-called “*dense*” information packing, the packing for which a net with n events allows the solution of n tasks at a time. Consider two identical circuits consisting of cells of the type shown in Fig. 5.37(c) (or, dually, Fig. 5.37(d)). Let the i -th cell of one of them be in the state \mathcal{P} , while the i -th cell of the other is in the state \mathcal{J} . The dense packing of information is achieved if the source loads two nets with tokens, alternately, and these tokens “move along” those two different nets, which correspond to the two circuits above. When composing these two nets into a single net, we should bear the following in mind. To avoid disturbance of the required order of tokens, we should inhibit the combination $\mathcal{P}\mathcal{P}$ of states of two adjacent cells which may be interpreted as the situation when one token outruns the other. The remaining combinations of states of adjacent cells in these two nets, viz. $\mathcal{P}\mathcal{J}, \mathcal{J}\mathcal{P}$ and $\mathcal{J}\mathcal{J}$ are allowed, and are denoted as \mathcal{P} , \mathcal{P}^* and \mathcal{J} , respectively. Recall that in the model transitions described $\mathcal{J}-\mathcal{P}$ and $\mathcal{J}-\mathcal{P}^*$ must alternate. Observing such precautions, the state graph for the i -th composite cell of the corresponding modelling circuit assumes the form shown in Fig. 5.39(a). The semi-modular implementation is shown in Fig. 5.39(b). The combinations 011, 101 and 110 of values z_1, z_2, z_3 encode the states \mathcal{J}, \mathcal{P} and \mathcal{P}^* , respectively.

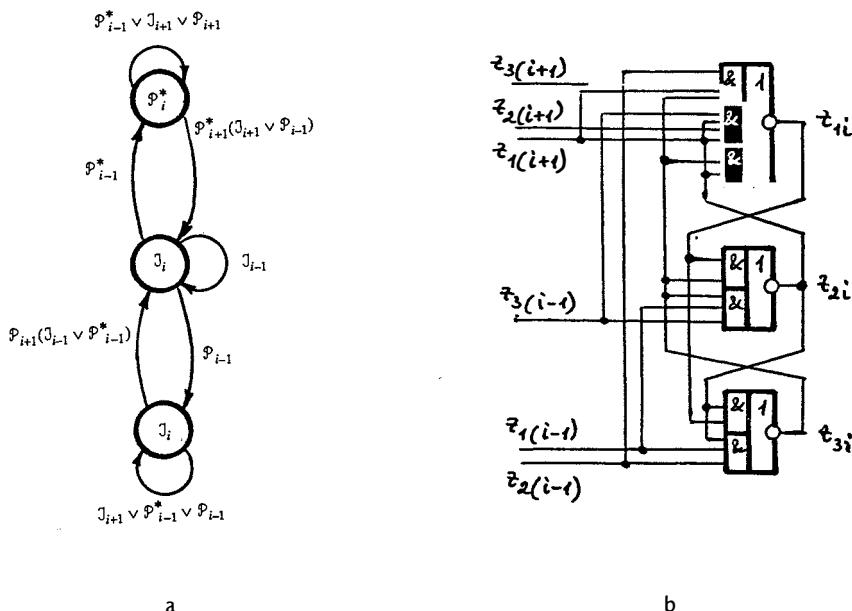


Figure 5.39 (a) and (b). (a) A cell for a “dense” modelling circuit: (a) state graph; (b) implementation.

Another modelling pipeline implementation with “dense” packing can be obtained by using an original Petri net with a doubled number of events. Assume that the net of Fig. 2.8(b) consists of $2n$ events, but only half of them (say, those having even numbers) are actively used, i.e. interpreted as executions of n tasks. Then the $2i$ -th event being in the phase \mathcal{P} should change to phase \mathcal{J} if the $(2i + 1)$ -th event has already come to phase \mathcal{P} . The state graph of Fig. 5.38 is modified in such a way as to make the $2i$ -th event change to \mathcal{J} after the $(2i + 2)$ -th event comes to \mathcal{P} , provided that the $(2i + 1)$ -th event has also done so as shown in Fig. 5.40(a). The state graph of Fig. 5.40(b) corresponding to the odd number event differs from the former graph of Fig. 5.38 only in the case where events with numbers $2i$, $2i + 1$ and $2i + 2$ are in phases \mathcal{J} , \mathcal{P} and \mathcal{J} , respectively. In the graph of Fig. 5.40(a) such a combination of states cannot arise.

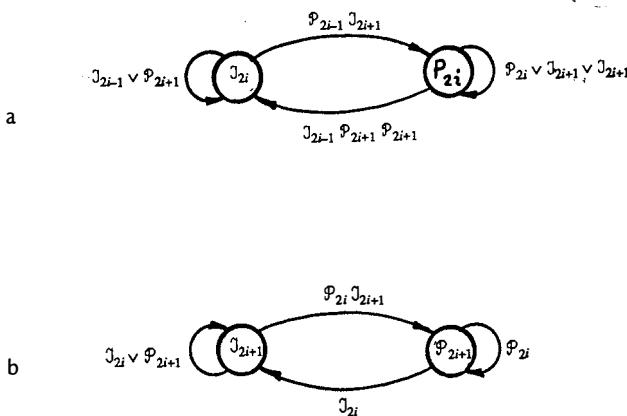


Figure 5.40 (a) and (b). Alternative variants of state graphs for cells with (a) even numbers and (b) odd numbers.

An implementation for the graphs of Fig. 5.40 can be built using flip-flops similar to those of Fig. 5.37(c) and (d).

In the implementations based on the graphs of Fig. 5.38, the combination of phases \mathcal{J} and \mathcal{P} of the $2i$ -th and $(2i + 1)$ -th cells, respectively, is unsteady, i.e. tends towards the combination $\mathcal{J}\mathcal{J}$, independently of the states of the neighbouring cells. Such an unsteady combination reduces the speed of the modelling circuit and this is undesirable. To remedy this problem, we combine the given pair of cells into one cell, by denoting the combinations of the states of the $2i$ -th and $(2i + 1)$ -th cells in the following way: $\mathcal{J}\mathcal{J}-\mathcal{J}$, $\mathcal{P}\mathcal{J}-\mathcal{P}^*$, $\mathcal{P}\mathcal{P}-\mathcal{P}$. As a result, we obtain the state graph shown in Fig. 5.41(a) with the semi-modular implementation presented in Fig. 5.41(b).

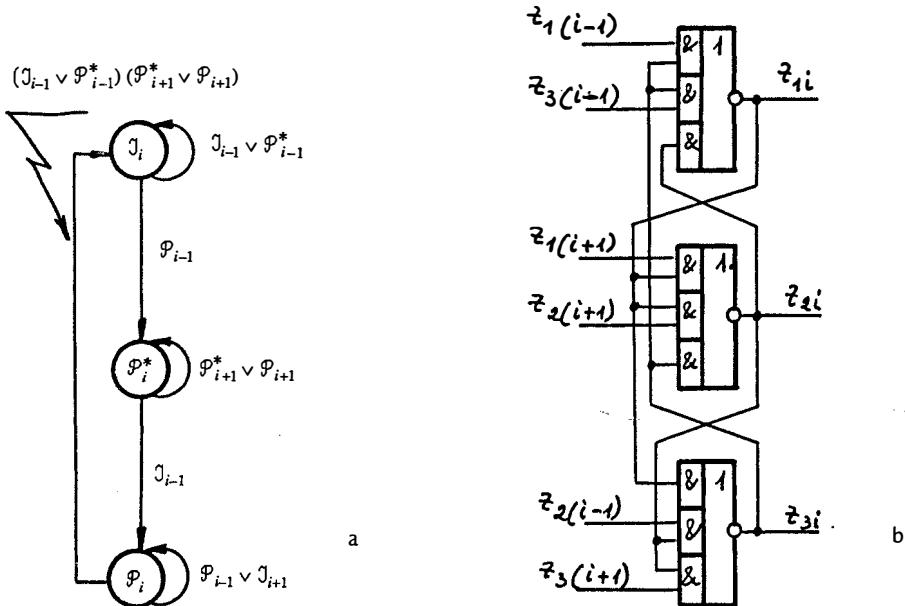


Figure 5.41 (a) and (b). (a) State graph obtained as a result of combining graphs of Fig. 5.40, and (b) the construction of the corresponding cell.

The states \mathcal{J} , \mathcal{P} and \mathcal{P}^* are encoded by the triples of values z_1 , z_2 and z_3 as 001, 010 and 100, respectively.

The net of Fig. 2.8(b) contains two chains of conditions: one is in the direction of the information flow, and the other is the reverse. Generally, if a process, which is not necessarily totally sequential, is defined by a Petri net, we may insert additional reversely directed chains of conditions into this net and thus obtain a pipeline process with the same ordering of events as the original one. In this case, the speed growth can only be achieved when the original net contains no loops with less than four events (conditions) in series.

Fig. 5.42 presents a table showing *pipeline analogues to fragments of simple Petri nets*. The implementation for this has been discussed in Section 5.2 (see Fig. 5.1). Modelling circuits for the fragments shown in Fig. 5.42 can be constructed in the manner described if the Petri nets containing these fragments, as well as the original nets, are safe and persistent. In the general case, the mechanical transformation to a pipeline process according to Fig. 5.42 does not guarantee persistence of the net obtained. Therefore, the latter should be augmented by some extra conditions and events, thus, making it persistent. By analysing the AP of the original net, these additions can be identified.

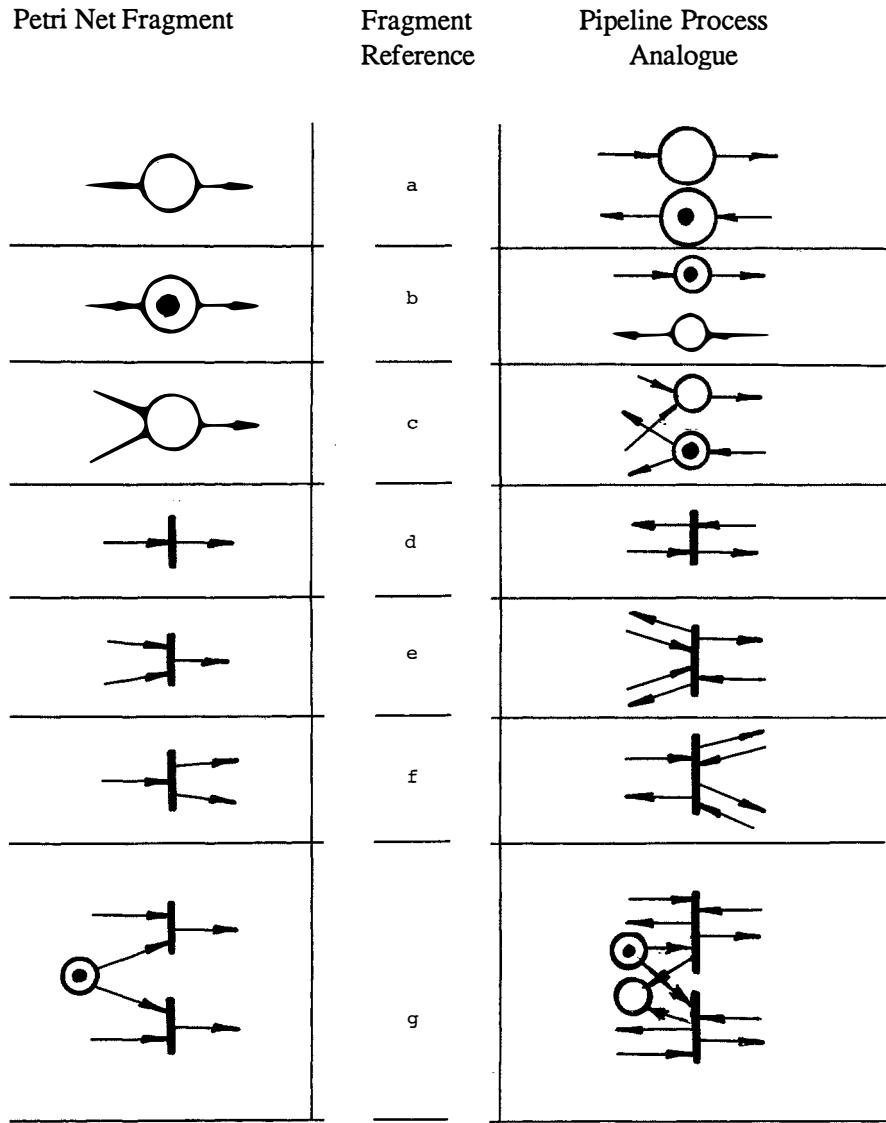


Figure 5.42 (a) - (g). Fragments of Petri nets and their pipeline analogues.

Another problem is to construct a pipeline analogue for a simple AP that will be adequate for the original process and guarantee the maximum efficiency of the solution.

Since checking whether an autonomous AP belongs to the class of pipeline APs is a rather difficult task, we prefer to use a Petri net interpretation of the AP.

The discussion is rather informal referring to the ideas of persistence, safeness and liveness.

We should point out that the transformation of totally sequential fragments has already been described (in Example 2.7).

5.5.1.1. Pipelinization of Parallel Fragments

An example of a simple Petri net, the so-called *reconvergence net*, is shown in Fig. 5.43(a).

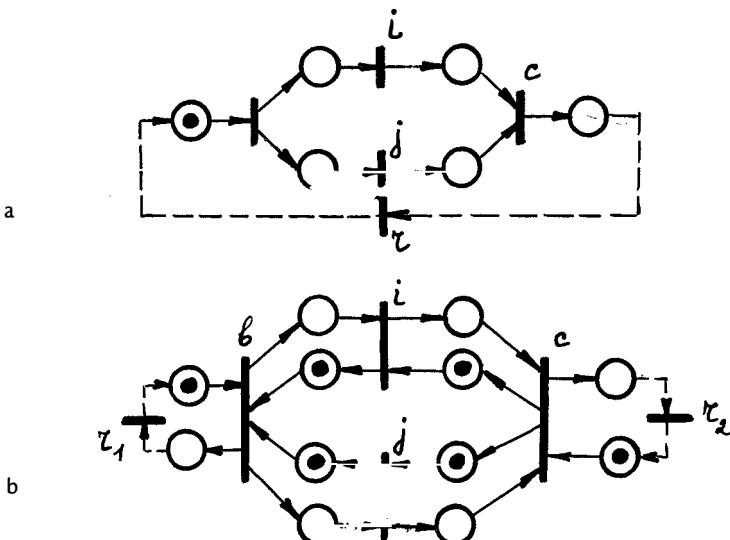


Figure 5.43 (a) and (b). (a) Persistent and safe Petri net with recovergent section, and (b) its pipeline analogue.

Events labelled with *i* and *j* are interpreted as two *concurrent operators*. The deadlock marking containing a single token in the output condition of event *c*, called *convergent event*, is a resultant, while the marking with a single token in the input condition of the event *b*, called a *bifurcant event*, is an initiator. A trivial reposition can be obtained by using the event *r*, as shown by the dashed line. This persistent net is safe. According to the assumption given at the beginning of Section 5.5 (an event is enabled only when both immediately preceding and succeeding events have fired), this net can be converted to its pipeline analogue, shown in Fig., 5.43(b). The reposition is now realized through two events *r*₁ and *r*₂. The persistence and safeness conditions are similar to those of a sequential process. It is obvious that the pipelinization of the original AP implies that the number of concurrently executed

tasks would be increased. The generalization for $m, m > 2$, parallel branches is straightforward.

5.5.1.2. Pipelinization of a Conditional Branch

An example of a process with a *conditional branch* is presented in Fig. 5.44(a).

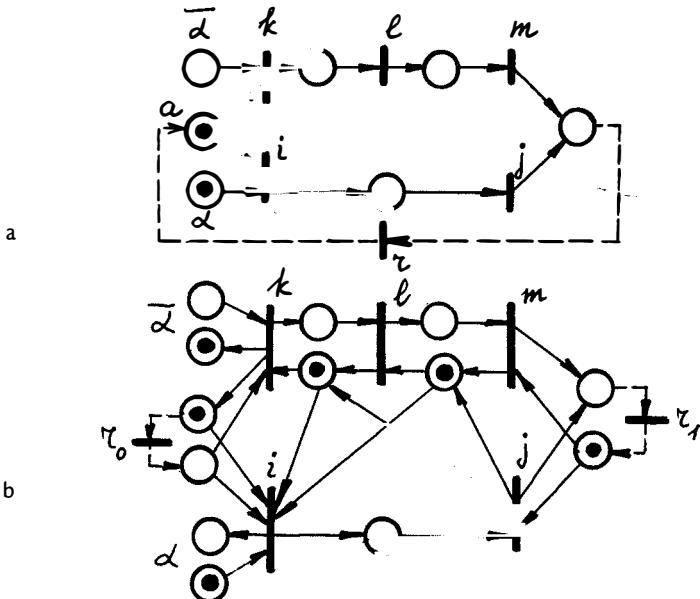


Figure 5.44 (a) and (b). (a) Persistent and safe Petri net with a conditional branch, and (b) its pipeline analogue.

This net is persistent with respect to those markings which do not contain a pair of tokens in conditions α and $\bar{\alpha}$ at the same time. The net is safe with respect to the initial marking given in Fig. 5.44(a) and the dual one, i.e. such that contains a token in α and $\bar{\alpha}$. The reposition of AP (a trivial reposition) is done through the event r . The Petri net depicts a process of executing only one task, and just one of the two branches is used in this case.

In finding a pipeline analogue using this approach, special problems related to net persistence arise that can be illustrated through the following analogy. Assume that balls roll along a railway in much the same way as tokens move along the net. A conditional branch can be a form of railway switch point. When a ball approaches the switch, it is then directed to one of two directions. What is supposed to happen next? Due to the fact that balls roll with different speed, and because different

branches may have different lengths, one ball may outrun another one if they are directed to different branches, after the switch. Therefore, at the merging point, there may either be a “ball collision”, thus violating persistence of the net, or the “ball train” may be re-ordered, which corresponds to task re-ordering in the net.

The idea, which forms the basis for constructing a persistent net for a pipeline process, is in synchronizing the task execution in parallel branches in the same way as it is done in sections of railways with limited throughput. A series of additional conditions can be inserted into the original net for its pipelinization, thereby, providing a required synchronization medium.

The result of transformation of the net with the conditional branch, shown in Fig. 5.44(a), is presented in Fig. 5.44(b). A pair of events taken in parallel branches, for example, j and m , have an additional input condition in common that provides a synchronization medium for task execution. When synchronizing an event with the alternative parallel branch consisting of more than one event (for example, in Fig. 5.44(a) event i is synchronized with a section consisting of events k and l in the other branch), the event may be enabled only if none of the events in that branch is enabled. This requirement is met if all complementary conditions of the alternative branch contain tokens. Synchronized sections of parallel branches can be chosen with regard to real parameters of the system, thus, optimizing the performance.

Another synchronization technique exploits the effect of inserting a buffer-operator which can be used for storing the values of the condition α for each executed task. These values allow the results of the tasks to be read in the same order as they have been written into the buffer-operator. The length of the buffer affects the system performance: if the number of buffer bits does not exceed the number of operators in the shortest branch, then the number of tasks in the system will be equal to the number of bits in the buffer.

5.5.1.3. Transformation of a Loop

An example of a Petri net containing a *loop* with a given number of iterations is shown in Fig. 5.45(a), from which we see that, depending on the value of α , there will either be the iteration of events, i, j, k, i, \dots , if $\alpha = 1$, or occurrence of the event l , if $\alpha = 0$. After this, a process reposition, designated by event r , may follow. This net is persistent and safe with respect to the given initial marking or the dual one.

Let us again resort to an analogy for examining the problems arising during the pipelinization. A cyclic section of a net can be associated with a railway loop having a switch. Depending on the position (or state) of the switch, a ball either leaves the loop, or continue on its way in it. In addition, more than one ball may be in the loop and their maximum number is determined by the length of the loop. If the loop

is not empty, i.e. it contains at least one ball, then at the loop exit a collision may occur between a new ball, that is approaching the switch, and the old one inside the loop. As has already been said, this situation is analogous to the violation of persistence of the net. The idea which is used to obtain a persistent net with a loop is as follows. The loop entry must be controlled by the same switch as the loop exit. As long as the loop exit is closed, the entry should also be closed. In this case any collision would be prevented. The loop entry (exit) is opened up to let a ball in (out) only after the switch allows another ball to leave (enter) the loop. Hence, the number of balls in the loop is kept constant.

The maximum throughput of a pipeline system is achieved when the system is half-loaded, i.e. then the number of tasks is half the number of operators. Therefore, being initially reset, the loop should be half-filled with tasks, some of which, of course, may be fake. The idea suggested is illustrated in Fig. 5.45(b).

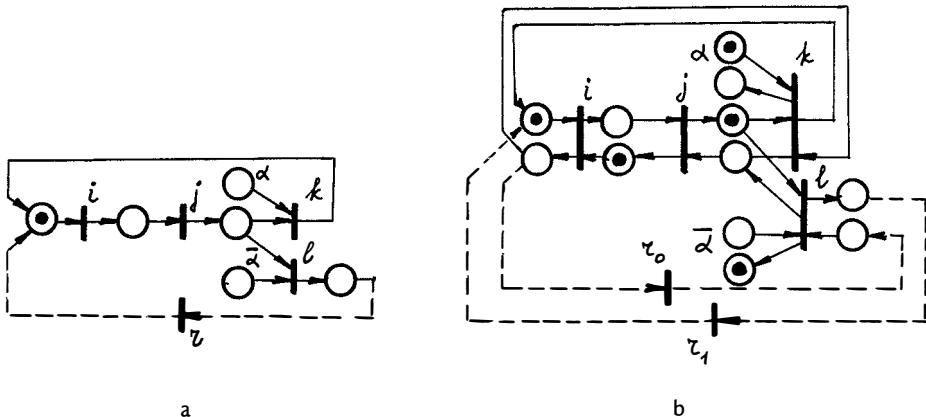


Figure 5.45 (a) and (b). Persistent and safe Petri net with (a) a loop, and (b) its pipeline analogue.

5.5.1.4. Pipelinization for Multiply Used Sections

The implementation of an AP can be simplified when some sections of the process are *multiply used*. Let us consider an example presented in Fig. 5.46(a). The fragment A of the net is used twice: both after event i and after event j . Conditions b and c are used for storing the information of a particular branch to which the process should return. They play the role of a return address when a sub-routine is called.

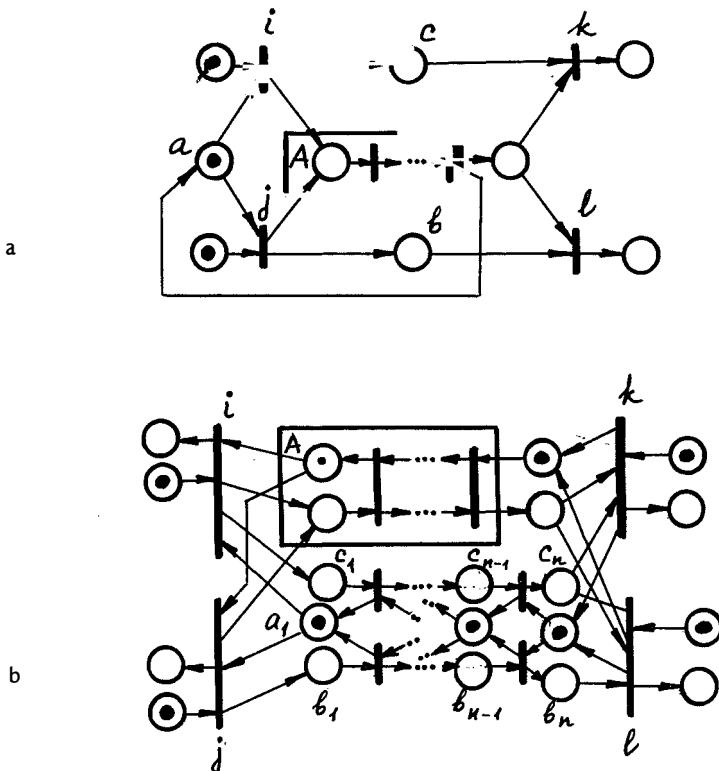


Figure 5.46 (a) and (b). (a) Petri net with a multiply used section, and (b) its pipeline analogue.

Note that such a net may generally appear to be non-persistent because events i and j may become enabled simultaneously if they are in two concurrent sections of the net. To resolve the conflict, we can resort to using synchronization primitives, for example, semaphores which are implemented with arbiters (see Chapter 9). In this case the marking of a condition a with at most one token (a safe marking) resolves the conflict. We may, thus, state that non-persistence of the given Petri net is *localized* in condition a . In pipelining we should also ensure that the violation of persistence is localized in exactly one specific place. Constructing a corresponding net, as shown in Fig. 5.46(b), we assume that the section A is also pipelined. To store the “return address”, we substitute a sequence of conditions $b_1c_1, b_2c_2, \dots, b_nc_n$ for the pair of conditions b and c , thus modelling a binary buffer. The length of the buffer is equal to the number of stages in section A . Here non-persistence is localized at points a_1 and a .

In conclusion to this chapter, we should note that the major part of this presentation has been inspired by M.A. Gavrilov's idea of a modular synthesis and his concept of a standard or canonical implementation. This implies that a circuit is composed of modules that are built of basic elements belonging to a functionally complete set. Furthermore, these modules are associated with particular fragments of an original structural specification. Such an approach substantially reduces design costs, but at a sacrifice in the size of circuitry.

On the other hand, the major methodological technique, originally suggested by M.V. Glushkov, is in partitioning a system into operational and control parts, even though such a partitioning is of a relative character: It is taken with respect to particular communication disciplines and sequencing of events in time. The qualities of aperiodic circuits give an additional impact to this idea. In these circumstances, a modelling circuit plays the role of a control mechanism, while operational units can be inserted into the breaks of wires connecting the elements of the modelling circuit. Such an approach makes it possible to synthesize a control mechanism in parallel with the construction of operational units. This simplifies the composing of the whole system.

5.6 Reference notations

The idea of implementing Petri net fragments by standard modules (with respect to events) has been discussed in [225] and [269]. A group of French researchers, working in the Aero-Space research centre in Toulouse, proposed a set of hardware models and a technique for the transformation of Petri net specifications into networks of such modules [197], [211] and [301]. It should, however, be pointed out that the authors did not attempt to assign, to the obtained implementations, membership within the class of semi-modular circuits. Thus, immediate use of their results is not possible. Nevertheless, the design of one of these modules, the so-called David cell [197], has been modified in such a way that an asynchronous signal distribution circuit (Fig. 5.3) has been proposed [27] and [40]. In [206], the problems of universality of another (hypothetical) set of modules have been considered. (A physical implementation of such modules can be obtained, if the same interpretation, which has been used for Petri net implementation with respect to conditions, is applied.)

Approaches to the implementation of parallel asynchronous flow charts (based on the modular syntheses idea of M.A. Gavrilov [69]) and pipeline processes have been proposed in [6] and developed in [53], [62] and [307].

The multiple use operator circuit which is an improvement on that of [6] is given in [56].

The concept of functional completeness presented in Section 5.4 differs from the commonly known concept, although there is no contradiction between them. The results of this section have been reported elsewhere [6], [64], [178], [180], and are based on the classification of speed-independent circuits presented in [120]. The proof of Theorem 5.9 can be found in [64].

A set of universal speed-independent modules (with memory) has been proposed in [249], where a question has been raised as to whether or not the number of modules is reducible, as well as the number of inputs and/or outputs of the modules. It seems that Theorems 5.8 and 5.9 give the final answer to this question (abandoning the requirement of the circuit to be insensitive to the wire delays). Another, rather complex, implementation method is proposed in [274] but this method does not guarantee semi-modularity of a synthesized circuit with respect to the variables which are not defined in the original state-transition diagram.

The material on synthesis and analysis of antitonus linear (totally sequential) circuits is not presented in this book but has been discussed in a series of papers [161], and, for a wider class of circuits, the discussion can be found in the book [163]. This approach is based on using a cyclogram language and its extension called taxogram notation. It exploits the general idea of representing a specification by signal changes which was originally proposed by M.A. Gavrilov (switch on/off tables) [71]. The linear writing form of cyclograms is close to that of A.A. Lyapunov [110]. This approach also uses ideas from the book [143].

CHAPTER 6

COMPOSITION OF ASYNCHRONOUS PROCESSES AND CIRCUITS

Everybody climbs his or her own wall, but, meanwhile, the truth is at the bottom at everyone's feet.

F. Krivin.

Modern computers and discrete systems are built using a modular approach where both hardware and software components are composed through the interconnection of functional and structural units (modules), thus, forming rather complex designs.

Since the main interest of this book is to study a special class of systems, namely self-timed systems, we need an adequate formalism to enable us to obtain the description of the behaviour of a system from its fragments. Such a formalism is based on the introduction of a certain set of operators defined on the processes being formed. It differs from the methods which are common for classical automata theory in their specific treatment of the dynamic qualities of processes.

This chapter a discussion of composition methods applicable for both asynchronous processes and aperiodic circuits, and transformation techniques for asynchronous circuits defined in terms of the Muller model, is presented.

6.1 Composition of asynchronous processes

6.1.1 REINSTATED PROCESS

Referring to Definitions 2.2 and 2.8, we introduce the following definition.

DEFINITION 6.1. Let $P = \langle S, F, I, R \rangle$ be an asynchronous process and $P' = \langle S', F', I', R' \rangle$ be a reposition of P satisfying Definitions 2.2 and 2.8, respectively. Then $P^r = \langle S^r, F^r, I^r, R^r \rangle$ in which $S^r = S \cup S^*$, $F^r \cup (F' \setminus (S' \times I))$, $I^r \subseteq I$, $R^r = R \cup S^*$ is called a *reinstated asynchronous process*.

In other words, a reinstated process is a composition of an AP and its reposition with those pairs of situations which define the transitions leading to the initiators of the process (forming pairs in $F' \cap (S' \times I)$) deleted from relation F' .

Depending on which of the reposition types is used for process P , the corresponding reinstated process P^r is either *completely reinstated* or *partially*

reinstated. If, for a given AP, a reposition does not exist, this AP is called *irreinstable*. The concepts of initiators and resultants for a reinstated process with situations structured by establishing input and output components can be particularized in the following way. A situation is referred to as a resultant if, in all situations belonging to the sequences outgoing from the given situation, the output component remains the same as in the given situation. A situation is referred to as an initiator if it can be obtained from the resultant situation by replacing the input component value in such a way that the situation obtained is no longer regarded as a resultant.

If a certain process is regarded as a model of the joint functioning of a group of sub-processes, then the situations of these sub-processes may be considered as structural units (components) of the situations of the given process.

EXAMPLE 6.1. (continued from Example 2.6). A possible reposition of a process P given by the Petri net of Fig. 2.7 can be defined as follows:

$$\begin{aligned} P : S' &= \{C, \emptyset, A, B, AB\}, \\ F' &= \{C, \emptyset, (C, A), (C, B), (C, AB), (\emptyset, A), (\emptyset, B), (\emptyset, AB), (B, AB)\}, \\ I' &= \{C\}, \quad S^* = \{\emptyset, A, B\}, \quad R' = \{AB\}. \end{aligned}$$

The reinstated process is defined by

$$\begin{aligned} P^r : S^r &= \{A, B, C, \emptyset, AB\}, \\ F^r &= \{(AB, C), (C, \emptyset), (C, A), (C, B), (\emptyset, A), (\emptyset, B)\}, \\ I^r &= \{AB\}, \quad R^r = \{C, \emptyset, A, B\}. \end{aligned}$$

Strictly speaking, this reinstated process is described by four, rather than one, Petri nets of Fig. 2.7 with initial markings AB, \emptyset, A, B .

6.1.2. PROCESS REDUCTION

In this sub-section, the reduction operation ,which consists of reducing a given AP to a more simple one, is discussed. It is obvious that such an operation is needed when specific interest is focussed on a part of the process description.

Let a non-reinstated AP be given by $P = \langle S, F, I, R \rangle$ with situations structured in much the same way as in the final paragraph of sub-section 2.1.4, i.e. the set of situations S can be represented as an ordered triple $S = (X, Y, Z)$ where X, Y, Z stand for the sets of values of input, output and internal components, respectively.

Let us consider a p -block π -partition of a set of situations S of the process P such that, for the situations of any one block, the input component assumes a fixed

value x_j , $1 \leq j \leq p$. We also select $r < p$ different values of the input component constituting the sub-set X^* , $X^* \subset X$. The situations that enter the blocks of the π -partition which correspond to the selected values of the input component constitute the sub-set S^* , $S^* \subset S$.

For each initiator $s_i \in I$ we construct a set $S(s_i)$ of the situations which belong to the sequences of the process P leading from the given initiator. We form a set $S(X^*)$ as a union of those sets $S(s_i)$ for which $S(s_i) \subseteq S^*$, i.e.

$$S(X^*) = \bigcup_{S(s_i) \subseteq S^*} S(s_i)$$

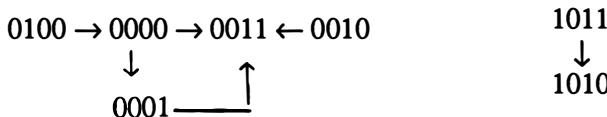
We also build

$$F(X^*) = F \cap (S(X^*) \times S(X^*)), \quad I(X^*) = I \cap S(X^*), \quad R(X^*) = R \cap S(X^*).$$

DEFINITION 6.2. The process $P(X^*) = \langle S(X^*), F(X^*), I(X^*), R(X^*) \rangle$ is called the *reduction* of a non-reinstated process $P = \langle S, F, I, R \rangle$ with respect to a given set X^* of input component values.

Similarly, the reduction $P(Y^*)$ of a non-reinstated process with respect to a given set Y^* of output component values can be defined.

EXAMPLE 6.2. Let process P be given by the following state-transition diagram



in which “*” are omitted. Here $S = \{0100, 0000, 0011, 0010, 0001, 1011, 1010\}$, $I = \{0100, 0010, 1011\}$, $R = \{0011, 1010\}$. The first two elements of the encoding combination of a full state are chosen as an input component. Then the reduction of P with respect to $X^* = \{00, 10\}$ will be the process $P(X^*)$ with the diagram of full states

$$0010 \rightarrow 0011 \quad 1011 \rightarrow 1010$$

and

$$S^* = \{0000, 0010, 0001, 0011, 1011, 1010\}, \quad S(X^*) = \{0010, 0011, 1011, 1010\}, \\ I(X^*) = \{0010, 1011\}, \quad R(X^*) = \{0011, 1010\}.$$

EXAMPLE 6.3 (continued from Example 2.7). If we assume for the net in Fig. 2.8(b) that A is the input condition and E is the output condition, then the pipeline process can be specified as a reinstated process. Its initiators are the situations with a token in place A , and its resultants are those with a token in place E , or without tokens in A and E .

In a similar way, taking the previously assumed discipline, the net in Fig. 2.8(b) can be specified as a reinstated process with input condition H and output condition D . If, by contrast, both A and H are referred to as input conditions while E and D are referred to as output conditions, then appropriate sets of initiators and resultants cannot be extracted.

From the construction rules for reduction, it follows that the reduction $P(X^*)$ of process P can be defined, not on the whole set X^* , but just on a subset X^{**} , $X^{**} \subset X$, because, in building the reduction, we normally delete entire groups of situation sequences, rather than any individual situation, from the entire process. Thus, generally, we have $X^{**} \subset X$. A similar thing holds for $P(Y^*)$. It is easily seen that for any X' , $X^* \supseteq X' \supseteq X^{**}$, $P(X') = P(X^{**}) = P(X^*)$.

We now consider a reposition of the reduction $P(X^*)$ of process P . We form a p -block π' -partition of the situations S' of the process P' , each block of which is associated with a fixed value x_j , $1 \leq j \leq p$, of the input component. The set S'^* is formed as a union of those blocks in π' whose situations contain $x_j \in X^*$. Then, for each initiator $i'_j \in R(X^*)$ of process P' and each resultant $r'_m \in I(X^*)$, we consider a set $S'_k(i'_j, r'_m)$ of situations belonging to the k -th sequence leading from (i'_j) to r'_m .

$$(1) \text{ the set } S'(X^*) = \bigcup_{s'_k(i'_j, r'_m) \subset S'^*} S'_k(i'_j, r'_m);$$

We build

- (2) the relation $F'(X^*)$ which defines the order between situations in the sequences $F'(X^*) = F \cap (S'(X^*) \times S'(X^*))$;
- (3) the sets $I'(X^*) = I' \cap S'(X^*)$ and $R'(X^*) = R' \cap S'(X^*)$.

The process $P'(X^*) = \langle S'(X^*), F'(X^*), I'(X^*), R'(X^*) \rangle$ will be called the *reposition of the reduction $P'(X^*)$* of the process P .

In a similar way to that for a reposition P' of a process P , a reposition of the reduction $P(Y^*)$ can be constructed. The reduction of a reinstated process P' is built

from the reduction of the non-reinstated process P and the reposition of the reduction $P'(X^*)$.

The following assertions can easily be deduced (see Definitions 2.3 and 2.4.).

1. The reduction of a non-effective process may be an effective process.
2. The reduction of an effective process, if it exists, is always an effective process.
3. The reduction of a controlled process, if it exists, is always a controlled process.
4. The reduction of a partially reinstated process may be a completely reinstated process, while the reduction of a completely reinstated process may appear to be a partially reinstated, or even a non-reinstatable, process.

6.1.3. PROCESS COMPOSITION

In this sub-section, we discuss operations that enable us to interconnect processes into a composition.

Let two APs be given, one of which $P_1 = \langle S_1, F_1, I_1, R_1 \rangle$ is not necessarily a reinstated process, whereas the other $P'_2 = \langle S_2, F_2, I_2, R_2 \rangle$ is a reinstated process. The situations of processes are structured in such a way that, in the situations of P_1 , the output component is picked out, while in those of P'_2 , the input component is picked out. Let all, or some, of the values of the output component y^1 of situations in S_1 be associated with all, or some, of the values of the input component x^2 of situations in S_2 . We denote the projections of a set of pairs of associated components on sets Y_1 and X_2 by Y_1^* and X_2^* , respectively. For sets Y_1^* and X_2^* , we build the reductions $P_1(Y_1^*)$ and $P'_2(X_2^*)$ of processes P_1 and P'_2 . Let $P'_2(X_2^*)$ be a completely reinstated process, and hence, $Y_1^{**} = Y_1^*, X_2^{**} = X_2^*$.

Now, we build, if possible, a process $P_3 = \langle S_3, F_3, I_3, R_3 \rangle$ whose situations can be represented as pairs $s^3 = (s^1, s^2)$ such that

- (1) $s^1 \in S_1(Y_1^*), s^2 \in S_2(X_2^*)$, i.e. $S_3 \subseteq S_1(Y_1^*) \times S_2(X_2^*)$;
- (2) the output component y^1 of situation s^1 , is identified with the input component x^2 of situation s^2 , i.e. $y^1 = x^2$;
- (3) if in s^3 the component $s^2 \in I_2(X_2^*)$, then $s^1 \in R_1(Y_1^*)$;
- (4) if $(s_i^1, s_j^2)F_3(s_k^1, s_l^2)$ then either $(s_i^1 F_1 s_k^1) \& (s_j^2 F_2 s_l^2)$, or $(s_i^1 F_1 s_k^1) \& (s_j^2 = s_l^2)$, or $(s_i^1 = s_k^1) \& (s_j^2 = s_l^2)$.

DEFINITION 6.3. The *sequential composition* of two asynchronous processes P_1 and P'_2 is an asynchronous process P_3 which is produced by the semantic identification of the values of the input and output components of situations of the reduced processes $P_1(Y_1^*)$ and $P_2(X_2^*)$, respectively, and satisfies the four constraints shown above.

Constraint (1) implies that the behaviour of P'_2 as a part of process P_3 is such that the reduction $P'_2(X_2^*)$ does not contain the situations which do not belong to the set $S_2(X_2^*)$. Constraint (4) implies that no new sequences may be created in the reduction $P'_2(X_2^*)$ of P_2 , being a part of P_3 .

It follows from Definition 6.3 that:

- 1) Since situations $s^3 \in S_3$ are composed of pairs of the form $(s^1 = (x^1, y^1, z^1), s^2 = (x^2, y^2, z^2))$ then the components of $s^3 = (x^3, y^3, z^3)$ are defined as $x^3 = x^1, y^3 = y^2, z^3 = (z^1, y^1 = x^2, z^2)$, $S_3 \subseteq S_1(Y_1^*) \times S_2(X_2^*)$ with projections of S_3 on $S_1(Y_1^*)$ and on $S_2(X_2^*)$ being equal to $S_1(Y_1^*)$ and $S_2(X_2^*)$, respectively;
- 2) $I_3 \subseteq (I_1(Y_1^*) \times R_2(X_2^*)) \cap S_3$ with projections of I_3 on $I_1(Y_1^*)$ and on $R_2(X_2^*)$ being equal to $I_1(Y_1^*)$ and $R_2(X_2^*)$, respectively;
- 3) $R_3 \subseteq (R_1(Y_1^*) \times R_2(X_2^*)) \cap S_3$ with projections of R_3 on $R_1(Y_1^*)$ and on $R_2(X_2^*)$ being equal to $R_1(Y_1^*)$ and $R_2(X_2^*)$, respectively.

Now consider two asynchronous (not necessarily reinstated) processes P_1 and P_2 with structured situations, i.e. with components x^1 and x^2 picked out. Let all, or some, of the values of the input component x^1 of situations in S_1 be semantically identified with all, or some, of the values of the input components x^2 of situations in S_2 . We denote the projections of identified pairs of component values on sets X_1 and X_2 by X^* . For X^* we build reductions $P_1(X^*)$ and $P_2(X^*)$ of processes P_1 and P_2 . Let $X_1^{**} = X_2^{**} = X^*$.

Let us construct, if possible, a process P_3 whose situations $s_3 \in S_3$ can be represented as pairs $s^3 = (s^1, s^2)$ such that:

- (1) $s^1 \in S_1(X^*), s^2 \in S_2(X^*)$, i.e. $S_3 \subseteq S_1(X^*) \times S_2(X^*)$;
- (2) the input components x^1 and x^2 of situations s^1 and s^2 are identical, i.e. $x^1 = x^2 = x^3$.

DEFINITION 6.4. The *parallel composition* of two asynchronous processes P_1 and P_2 is a process P_3 which is formed by the semantic identification of input component values in the situations of the reduced processes $P_1(X^*)$ and $P_2(X^*)$ and satisfies the above two constraints.

Now consider a reinstated process $P^r = \langle S^r, F^r, I^r, R^r \rangle$ that corresponds to some process P and its reposition P' . Let the situations of P^r be structured in such a way that both the input and the output components in them are picked out, and all, or some, of the values of the output components y of situations of this process are associated with some, or all, of the values of the input components x of situations of the same process. Let X^* and Y^* denote projections of the set of mutually identical pairs of component values on sets X and Y , respectively. For the sets X^* and Y^* we construct the reductions $P^r(X^*)$ and $P^r(Y^*)$ of the process P .

We then construct an autonomous process $P^c(X^*, Y^*) = \langle S^c, F^c \rangle$ satisfying the following constraints:

- (1) $S^c \subseteq S^r(X^*) \cap S^r(Y^*)$;
- (2) the output component y^c of situation s^c is identified with the input component x^c , i.e. $y^c = x^c$;
- (3) $F^c = (F(X^*) \cup F'(X^*)) \cap (F(Y^*) \cup F'(Y^*))$ where F and F' define the causal relationship between situations of the non-reinstated process and its reposition which corresponds to the process P^r .

DEFINITION 6.5. An autonomous process P^c that satisfies the constraints (1) to (3) is called the *closure* of a process P^r .

Since the closure is defined only for reinstated processes, the necessary condition for the existence of a closure, for the sequential or parallel composition, is membership of the composed processes in the class of reinstated ones.

The composition mechanism introduced allows us to consider any AP as a super-position of the sequential and parallel compositions as well as the closure of the composed processes. As a consequence, it is possible to compose processes that are specified in any of the languages which may be interpreted in terms of asynchronous processes.

EXAMPLE 6.4. Fig. 6.1(a) shows the process P_1 in the structured situations of which the output component y^1 is represented by the fifth and sixth positions. Fig. 6.1(b) depicts the process P_2 with the input component x^1 marked out in the second and third positions of the situation vector. The reposition of P_2 is presented in Fig.

6.1(c), and the autonomous process which is composed in P_2 and its reposition P'_2 is shown in Fig. 6.1(d).

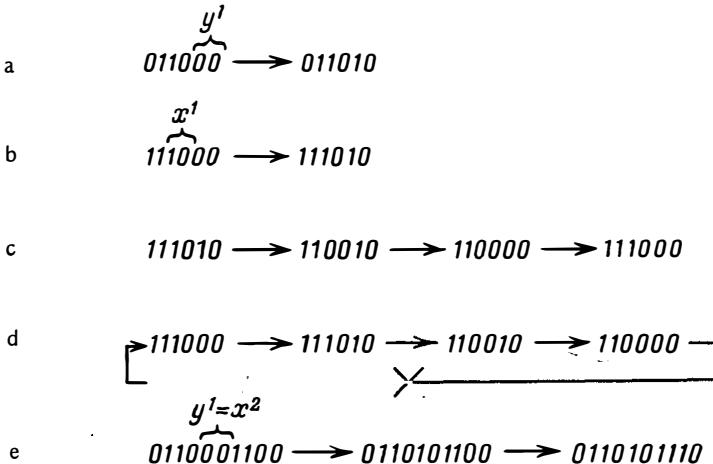


Figure 6.1 (a) - (e). Example of composition of two processes: (a) first process with output component y^1 ; (b) second process with input component x^2 ; (c) reposition of the second process; (d) autonomous process formed by the second process and its reposition (and reinstated process lacking the 110 000–111 000 transition); (e) the result of composition.

The reinstated process as has been defined above is obtained from the autonomous one by deleting the paths which lead directly to the initiators of P_2 , i.e. in this case we delete the arc which is marked with a cross in Fig. 6.1(d). If y^1 is semantically identified with x^2 then the sequential composition of P_1 and P'_2 will assume the shape in Fig. 6.1(e). In the situations of the composite process the first four positions are the corresponding positions of P_1 , the fifth and sixth positions represent the component $y^1 = x^2$, and the seventh to tenth positions stand, respectively, for the first, second, fifth and sixth positions in the situations of P_2 .

Special attention to the composition of aperiodic circuits is given in the following.

6.2 Composition of aperiodic circuits

Referring to Definition 2.25, we may interpret it in the following way. The combinations of input values for each element z_i in a circuit are divided into two classes: those causing the excitation of z_i when $z_i = 1$, and those making z_i excited

when $z_i = 0$. In an aperiodic circuit, the input combination of a certain class, applied to an element, cannot be changed until the element comes to the idle state. Thus, the operation of every element, as well as that of the entire circuit, is governed by the “request-acknowledge” principle.

6.2.1. THE MULLER THEOREM

The specific character of aperiodic circuits is concerned with using those composition methods which are not typical for classical synchronous and asynchronous circuits. One such method is presented by the following theorem which is called, in the literature, the *interconnection theorem*, or the *Muller theorem*. Several conventions about the nomenclature must be stated initially.

The expression $x = x(x_0, X)$ stands for the fact that element x realizes a certain Boolean function of x_0 and some other arguments in the set X (but which particular ones is unimportant). The expression $x^* = x(x_0 \equiv c, X)$ has the following meaning: the input x_0 is disconnected from element x and is replaced by the input c thereby realizing another function x^* .

THEOREM 6.1. *Let a pair of aperiodic circuits A and B be given such that, in the circuit A, an element y_0 and elements $y_j = y_j(y_0, Y)$, $1 \leq j \leq m$, while in the circuit B, an inverter z_0 and elements $z_i = z_i(z_0, Z)$, $1 < i < k$, are drawn out. Then the circuit composed of A and B without the inverter z_0 and with $y_j^* = y_j(y_0 \equiv \bar{z}_0, Y)$, $z_i^* = z_i(z_i \equiv y_0, Z)$, $1 \leq j \leq m$, $1 \leq i \leq k$, will also be aperiodic.*

The main idea of this theorem is illustrated in Fig. 6.2(a).

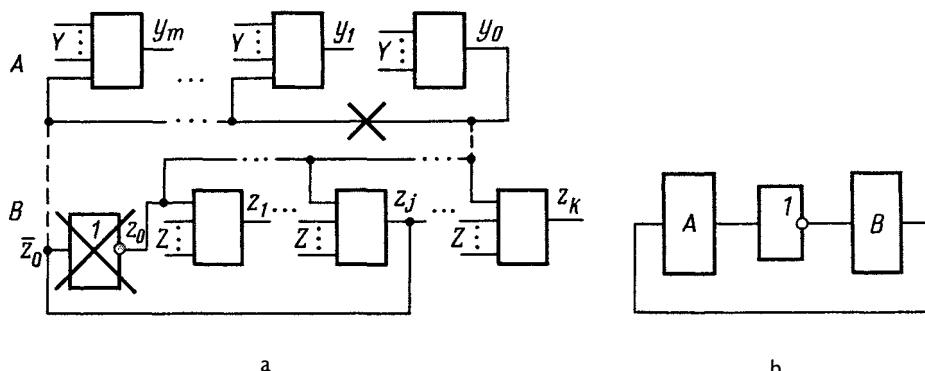


Figure 6.2 (a) and (b). Illustration of Muller theorem for the cases where (a) one of the interconnected circuits has an inverter, and (b) when neither have an inverter.

The line that connects the output of the element y_0 to the inputs of the elements y_1, \dots, y_m in the circuit A is broken, and the inverter is deleted from the circuit B , whereupon the inverter's input \bar{z}_0 is connected to the inputs of y_1, \dots, y_m and the output y_0 is connected to the inputs of z_1, \dots, z_k . The circuit obtained from such an interconnection is aperiodic.

The major disadvantage of the above interconnection method is in the necessity of having an inverter in one of the circuits. There is, however, a simple way to overcome this problem. We can insert a pair of inverters in series with one of the elements in circuit B . It is clear that the behaviour of all the other elements in B will not be changed by this, since such an insertion is equivalent to an increase in the delay in the chosen element without any effect on its logical operation. Now, in accordance with Theorem 6.1, one of the inverters may be deleted, thus yielding a composition consisting of A and B with an additional inverter, as shown in Fig. 6.2(b).

6.2.2. GENERALIZATION OF THE MULLER THEOREM

Recall the assumption of element delays presented in 4.1.1. In order to consider the proof of the theorem which is a generalized form of Theorem 6.1, we need an additional notation which is given here. The expression $x^* = x(x_0 \equiv f, X)$ will imply that the element x is replaced by the element x^* , realizing a function obtained by substituting the function f for the argument x_0 , in the function of x . For example, if $x = x_0x_1 \vee \bar{x}_1\bar{x}_2$, $f = \bar{x}_1y_1 \vee \bar{x}_1\bar{y}_2$, then $x^* = (\bar{x}_1y_1 \vee \bar{x}_1\bar{y}_2)x_1 \vee \bar{x}_1\bar{x}_2 = \bar{x}_1\bar{x}_2$.

THEOREM 6.2. *Let a pair of aperiodic circuits A and B be given such that, in the circuit A , an element y_0 and elements $y_j = y_j(y_0, Y)$, $1 \leq j \leq m$, while in the circuit B , an element z_0 and elements $z_i = z_i(z_0, Z)$, $1 \leq i \leq k$, are picked out. Then the circuit composed of A and B without the element z_0 and with $y_j^* = y_j(y_0 \equiv \bar{z}_0, Y)$, $z_i^* = z_i(z_i \equiv y_0, Z)$, $1 \leq j \leq m$, $1 \leq i \leq k$, will also be aperiodic.*

Proof. The element z_0 can be represented as a serial interconnection of a delayless decision element realizing the function \bar{z}_0 and an inverter with an arbitrary, but finite, delay. The accepted assumption about the character of element delays justifies such a representation. Now, according to Theorem 6.1, we can delete the indicated inverter and, thus, at the new output of the circuit B the function \bar{z}_0 will be realized. After disconnecting the output of element y_0 from the inputs of elements y_1, \dots, y_m in the

circuit A , we obtain these elements as realizing the functions $y_j(y_0 \equiv c, Y)$ where c stands for the remaining, so far unused, input common to these elements. Each of these elements can also be represented as a series of delayless decision elements realizing the function $y_j(c, Y)$ (or $\bar{y}_j(c, Y)$) and a built-in arbitrary, but finite, delay (or an inverter with an arbitrary, but finite, delay). Connecting the input c to the output of the delayless decision element \bar{z}_0 , we obtain the delayless decision elements which realize the functions $\bar{y}_j^* = y_j(c \equiv \bar{z}_0, Y)$ or their complements, if, of course, they are implementable by one element. Referring to real functional elements with finite delays, and closing the connection between y_0 and the common inputs of elements z_1, \dots, z_k , we obtain the desired result for the proof. *Q.e.d.*

Thus, according to Theorem 6.2, we should break the line which connects the output of element y_0 to the inputs of elements y_1, \dots, y_m in the circuit A and replace these elements by ones with inherent functions defined by \bar{y}_j^* . At the same time, we should delete the element z_0 from the circuit B and connect the newly formed outputs to the corresponding inputs of elements \bar{y}_j^* , while the new input line in A should be connected to the output of the circuit y_0 .

It is important to note that the resultant complexity of a composite circuit (say, the number of elements of the overall number of inputs/outputs in a circuit) may become less than the total complexity of the composing circuits. This is because of the deletion of the element z_0 and the replacement of elements y_j . Another issue is that the composite circuit may have a higher speed than the circuits composed by any other method. Such effects taken in combination cannot usually be achieved with traditional design methodologies.

EXAMPLE 6.5. Consider two aperiodic complementing flip-flops A and B shown in Fig. 6.3. Each is identical to that of Fig 4.12(b). They differ from each other only in that A has its “go” (a) and “done” (\bar{b}) signals disconnected, whereas in B these signals are connected, thus forming an autonomous circuit. The interconnection of A and B aims at obtaining an aperiodic two-bit counter, by using the results of Theorem 6.2. This is done in the following way.

In the circuit A we break the wire leading from the output of y_0 to the inputs of $y_1 = \overline{y_0 y_3}$ and $y_2 = \overline{y_0 y_4}$. This breaking is done at a point prior to the branching of this link. In B we delete the element $z_0 = \overline{z_3 z_4}$. Then the output of y_0 is connected to the inputs of z_1 and z_2 while the elements y_1 and y_2 are replaced, respectively, by \bar{y}_1^* and \bar{y}_2^* such that

$$y_1^* = y_1(y_0 \equiv \bar{z}_0) = \overline{z_3 z_4 y_0}, \quad y_2^* = y_2(y_0 \equiv \bar{z}_0) = \overline{z_3 z_4 y_4}.$$

Thus, the two-input elements y_1 and y_2 are replaced by three-input elements. The dashed lines in Fig. 6.3 designate the newly introduced connections.

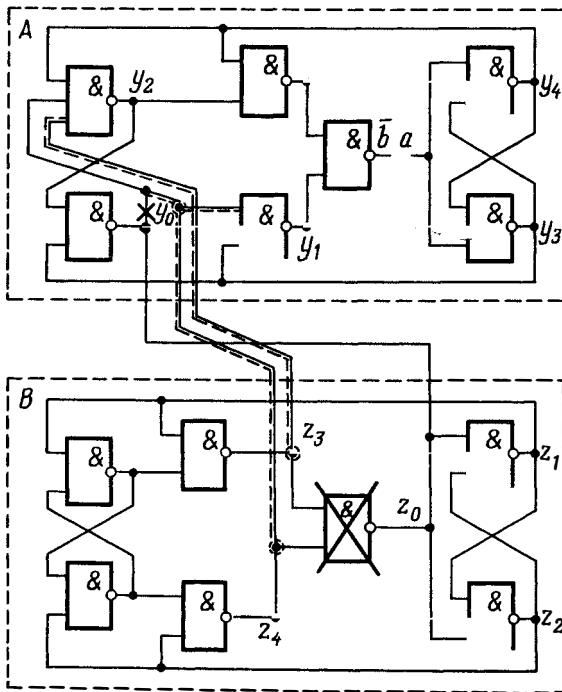


Figure 6.3. Illustration of building a counter consisting of two complementing flip-flops.

Compared to the ordinary counter obtained by the standard interconnection of flip-flops of the type in Fig. 4.12(b) (as will be shown in Fig. 11.22), this two-bit counter demonstrates that the above composition technique gives more economic and faster circuits. For numbers of bits greater than 2, its indicator will be simpler than that of the ordinary counter. Furthermore, the higher speed is guaranteed because the starting of the second, and subsequent, bit flip-flops is done by the signal on y_0 , and similar signals produced by subsequent bit flip-flops, rather than by the signal on y_1 , as in the circuit of Fig. 11.22.

One of the characteristic features of aperiodic circuits is the two-rail representation of (input, output and internal) signals. The following definition, and

two accompanying theorems, deal specifically with the interconnection of such two-rail circuits.

DEFINITION 6.6. An aperiodic circuit is called *mono-transient* if, for any change of its input variables, the output and internal variables have the same transient state (either $x_i = \hat{x}_i = 0$, all-zero transient state, or $x_i = \check{x}_i = 1$, all-one transient state), and *perfect* if the transient state of the inputs of its elements do not cause any excitation of these elements.

For perfect aperiodic circuits such, for example, as those that have been considered in Chapter 5 (perfect implementation), the following composition methods can be applied.

THEOREM 6.3. *Let two aperiodic circuits A and B be given such that within the circuit A we can pick out a perfect sub-circuit with a two-rail output \tilde{y}_0, y_0 and perfect sub-circuits with two-rail outputs $y_j = y_j(y_0, \tilde{y}_0, Y)$, $\tilde{y}_j = \tilde{y}_j(y_0, \tilde{y}_0, Y)$, $1 \leq j \leq m$, while in B $\tilde{z}_i = \tilde{z}_i(z_0, \tilde{y}_0, Z)$ is taken. Furthermore, all indicated sub-circuits have the same transient states. Let, also in the stable state, $z_0 = \tilde{x}_0$ and $\tilde{z}_0 = x_0$. Then the circuit C, obtained by the interconnection of A and B, with sub-circuit (z_0, \tilde{z}) deleted, and such that*

$$y_j^* = y_j(y_0 \equiv x_0, \tilde{y}_0 \equiv \tilde{x}_0, Y), \quad \tilde{y}_j^* = \tilde{y}_j(y_0 \equiv x_0, \tilde{y}_0 \equiv \tilde{x}_0, Y),$$

$$z_i^* = z_i(z_0 \equiv y_0, \tilde{z}_0 \equiv \tilde{y}_0, Z), \quad \tilde{z}_i^* = \tilde{z}_i(z_0 \equiv y_0, \tilde{z}_0 \equiv \tilde{y}_0, Z),$$

$$1 \leq j \leq m, 1 \leq i \leq k,$$

will also be aperiodic.

It is obvious that this theorem is an analogue of Theorem 6.1 for perfect implementations, and the role of the inverter in Theorem 6.1 is played, here, by a sub-circuit with two-rail outputs z_0 and \tilde{z}_0 . Hence, the proof of Theorem 6.3 follows from Theorem 6.1, taking into account, that no transient state in inputs can cause any excitation of the elements in the composition. We can take, as such a perfect sub-circuit, an RS-flip-flop built of NORs (or NANDs) having the all-zero (all-one) transient state. Another composition technique is outlined in the following theorem.

THEOREM 6.4. *Let two aperiodic circuits A and B be given such that within circuit A we can pick out a perfect sub-circuit with a two-rail output \tilde{y}_0, y_0 and flip-flops \tilde{y}_j , $1 \leq j \leq m$, with excitation functions S_j and R_j , $S_j = S_j(y_0, \tilde{y}_0, Y)$,*

$R_j = R_j(y_0, \tilde{y}_0, Y)$, while in circuit B we can pick out a flip-flop \tilde{z}_0 with excitation functions S_0 and R_0 and perfect sub-circuits (z_i, \tilde{z}_i) , $1 \leq i \leq k$, $z_i = z_i(z_0, \tilde{z}_0, Z)$, $\tilde{z}_i = \tilde{z}_i(z_0, \tilde{z}_0, Z)$. Furthermore, all indicated circuits and flip-flops have the same transient states. Then the circuit C obtained by the interconnection of A and B, with (z_0, \tilde{z}_0) deleted, and such that

$$S_j^* = S_j(y_0 \equiv R_0, \tilde{y}_0 \equiv S_0, Y), \quad R_j^* = R_j(y_0 \equiv R_0, \tilde{y}_0 \equiv S_0, Y),$$

$$z_i = z_i(z_0 \equiv y_0, \tilde{z}_0 \equiv y_0, Z), \quad \tilde{z}_i = \tilde{z}_i(z_0 \equiv y_0, \tilde{z}_0 \equiv \tilde{y}_0, Z),$$

$$1 \leq j \leq m, 1 \leq i \leq k,$$

will also be aperiodic.

This theorem is an analogue of Theorem 6.2 for the case of perfect implementation, and its proof follows from Theorems 6.2 and 6.3.

EXAMPLE 6.6. Fig. 6.4 shows a pair of complementing flip-flops A and B each of which contains three RS-flip-flops. The excitation functions of flip-flops \hat{y}_1 and \hat{y}_3 in A and that of \hat{y}_6 in B have their respective forms

$$S_1 = \overline{y_2 y_3}, \quad R_1 = \overline{\overline{y_2} y_3}, \quad S_3 = y_1 \oplus y_2, \quad R_3 = y_1 \oplus \bar{y}_2,$$

$$S_6 = y_4 \oplus y_5, \quad R_6 = y_4 \oplus \bar{y}_5.$$

To interconnect A and B, in accordance with Theorem 6.4, we should delete \hat{y}_6 from B and break the output lines of \hat{y}_2 on A. Then the excitation functions of flip-flops \hat{y}_1 and \hat{y}_3 should assume the form

$$S_1^* = \overline{y_3(y_4 \oplus y_5)}, \quad R_1^* = \overline{y(y_4 + \bar{y}_5)},$$

$$S_3^* = y_1 \oplus y_4 \oplus y_5, \quad R_3^* = y_1 \oplus y_4 \oplus \bar{y}_5,$$

while the inputs of the flip-flops \hat{y}_4 and \hat{y}_5 should be connected to the outputs of flip-flop \hat{y}_2 instead of those of \hat{y}_6 .

The resulting circuit will be a two-bit counter containing five, rather than six, RS-flip-flops.

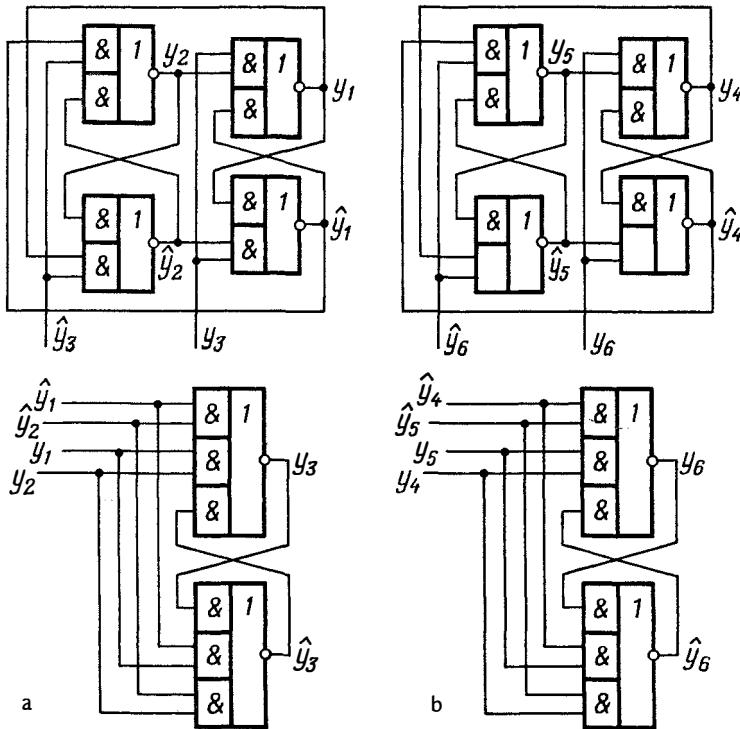


Figure 6.4 (a) and (b). Composing a counter of complementing flip-flops by deletion of a perfect sub-circuit.

Thus, we have considered the methods for interconnecting, or composing, aperiodic circuits which are not typical for conventional synchronous and asynchronous logic. In essence, all these methods are *generalizations of Muller's interconnection theorem*. Note that the techniques described do not cover all the potential interconnection capabilities of aperiodic circuits. For example, we have not mentioned the problem of interconnecting aperiodic implementations with different transient states.

6.3 Algebra of asynchronous circuits

The interconnection methods outlined in Section 6.2 are generally related to implementations that have higher dimensionality, i.e. a greater number of variables, in the composite circuit than in its composing circuits. In this section, we consider an approach to composing circuits in such a way that the dimensions of the composite

and composing circuits are equal. In other words, we are concerned here with circuit transformation techniques that can be used in the design of asynchronous circuits for eliminating certain undesired behavioural effects that may have been established during the process of analysis of the circuit. (see Chapter 8). Circuits are regarded, here, as algebraic objects, on the set of which, certain algebraic operations are defined, thereby providing the basis for appropriate transformations.

6.3.1. OPERATIONS ON CIRCUITS

Assume that asynchronous circuits (from now on, for the sake of brevity, called “circuits”) are defined on equal sets of variables $Z = \{z_1, z_2, \dots, z_n\}$. It is clear that there are 2^{n2^n} circuits defined on n variables. Let \mathbb{C} denote a set of circuits of n variables and two circuits S_1 and S_2 in \mathbb{C} are respectively associated with the following two systems of equations

$$z_i = f_{1i}(z_1, \dots, z_n), \quad z_i = f_{2i}(z_1, \dots, z_i, \dots, z_n), \quad i = 1, \dots, n$$

or, for brevity, $z_i = f_{1i}(Z)$ and $z_i = f_{2i}(Z)$.

DEFINITION 6.6. The *union (intersection)* of circuits S_1 and S_2 , denoted by S_{1+2} ($S_{1,2}$), is the circuit in which, for any pair of adjacent states a and b (taken in its state-transition diagram), the relation $a \xrightarrow{1} b$ holds, if and only if, this relation holds in either S_1 or in S_2 , or in both (in both S_1 and in S_2).

The following notations may also be used for the union and the interconnection of S_1 and S_2 : $S_{1+2} = S_1 + S_2$, $S_{1,2} = S_1 \cdot S_2$.

The following definition is equivalent to Definition 6.6.

DEFINITION 6.6*. The *union* S_{1+2} (*intersection* $S_{1,2}$) of circuits S_1 and S_2 is the circuit in which each variable z_i is excited in those, and only those, states, where it is excited in either S_1 or in S_2 , or in both (in both S_1 and in S_2).

Since for circuit S , the excitation condition z_i is $\bar{f}_i(Z) = 1$, when $z_i = 1$, and $f_i(Z) = 1$, when $z_i = 0$, then

$$z_i = z_i f_i(Z) \vee \bar{z}_i \bar{f}_i(Z). \quad (6.1)$$

It immediately follows from (6.1) and Definition 6.6* that for S_{1+2}

$$z_i = f_{(1+2)i}(Z) = z_i f_{1i}(Z) f_{2i}(Z) \vee \bar{z}_i (f'_{1i}(Z) \vee f'_{2i}(Z)) \quad (6.2)$$

and for $S_{1,2}$

$$z_i = f_{(1,2)i}(Z) = \bar{z}_i f_{1i}(Z) f_{2i}(Z) \vee z_i (f'_{1i}(Z) \vee f'_{2i}(Z)). \quad (6.3)$$

Note that in the union S_{1+2} , the “followed by” relation may also involve pairs of non-adjacent states that are neither in S_1 nor in S_2 . Assume, for example, that

$a \xrightarrow[z_i]{1} b$ ($a \xrightarrow[z_j]{1} c$) holds in S_1 (S_2), and variable z_j (z_i) is idle in state a . Furthermore,

there is no such d in S_1 or in S_2 that $a \rightarrow d$. Then in S_{1+2} both variables z_i and z_j are excited in state a and there should also exist a state d such that $a \xrightarrow[z_i, z_j]{1} d$.

EXAMPLE 6.7. Let two circuits S_1 and S_2 be defined on set $\{z_1, z_2, z_3\}$ by the following system of equations

$$z_1 = z_2 \vee z_3, \quad z_2 = z_2 \bar{z}_3, \quad z_3 = z_1 \vee \bar{z}_2 z_3 \quad (6.4)$$

and

$$z_1 = \bar{z}_3, \quad z_2 = z_1 \bar{z}_3, \quad z_3 = z_1 z_2. \quad (6.5)$$

The state-transition diagrams for these circuits are presented in Fig. 6.5(a) and 6.5(b), respectively. Because of (5.3) and (6.2), the intersection $S_{1,2}$ and union S_{1+2} are defined by the systems

$$z_1 = z_1 \vee z_2 \bar{z}_3, \quad z_2 = z_2 \bar{z}_3, \quad z_3 = z_1 z_2 \vee \bar{z}_2 z_3$$

and

$$z_1 = \bar{z}_1 \vee z_2 \bar{z}_3, \quad z_2 = z_1 \bar{z}_2, \quad z_3 = z_1 z_2 \vee z_1 \bar{z}_3.$$

The state-transition diagram for the intersection is shown in Fig. 6.5(c). It contains only those arcs which are in common for both diagrams Fig. 6.5(a) and (b). The diagram for the union, in addition to these common arcs, must also contain the arcs which connect non-adjacent states if they enter into the “followed by” relation in S_{1+2} . Because of its large size, it is not presented here.

Apart from the binary operators of union and intersection, we also introduce some unary operators. The major purpose of these operations is to replace the $a \rightarrow b$ relation in an original circuit by the $b \rightarrow a$ relation in the resulting circuit. We should, however, hint that such a replacement is not always possible because, for example, the number r ($0 \leq r \leq 2^n - 1$) of preceding states to a given state may be even,

whereas, the number of states following the given state is equal to $2^m - 1$, where m is the number of variables excited in the given state, and is thus, odd.

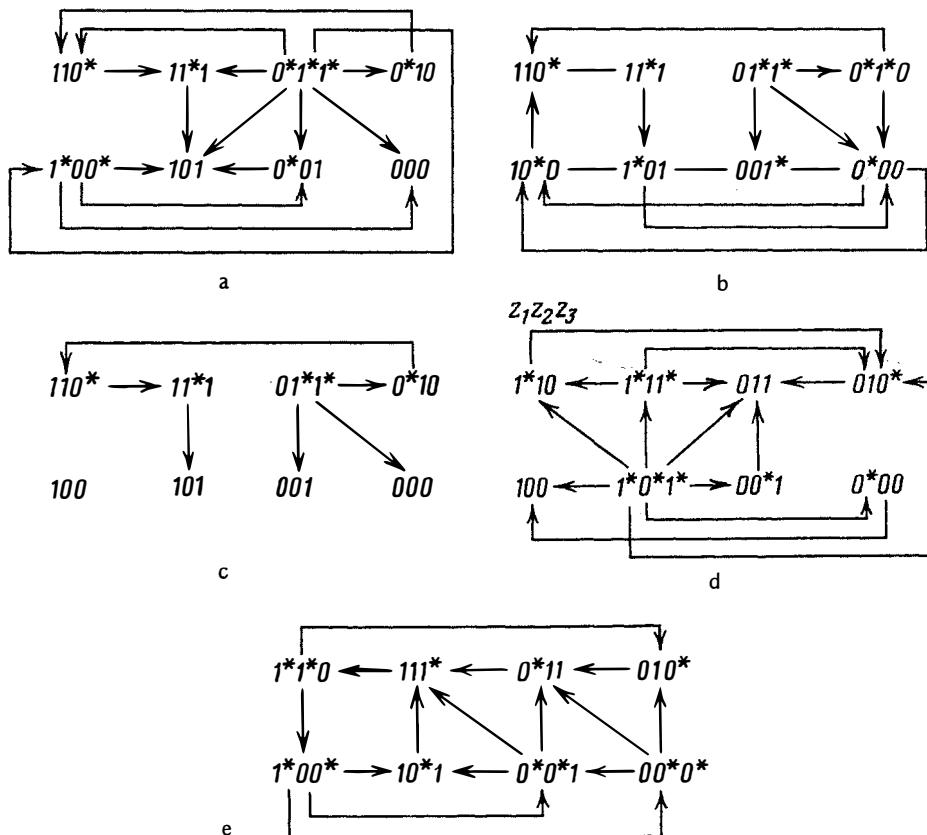


Figure 6.5 (a) - (e). (a) and (b) State-transition diagrams of original circuits; (c) their intersection; (d) and (e) their inverses.

DEFINITION 5.6. The *inverse (complement)* of a circuit S is a circuit, denoted by $\neg S$ (\bar{S}), such that for any pair of adjacent states a and b in $\neg S$ (\bar{S}) the $a \xrightarrow{z_i} b$ relation holds if, and only if, $b \xrightarrow{z_i} a$ holds (a is not immediately followed by b) in S .

Let $a \xrightarrow{z_i} b$ in S . If $a_i = 0$, then $b_i = 1$, i.e. $f_i(z_i = 0) = 1$, and, on the other hand, if $a_i = 1$, then $b_i = 0$, i.e. $f_i(z_i = 1) = 0$. Because of this, for the inverse of S ,

the following equation holds

$$z_i = \neg f_i(Z) = z_i \bar{f}_i(z_i = 0) \vee \bar{z}_i \bar{f}_i(z_i = 1). \quad (6.6)$$

The unary operation of inversion changes the $a \rightarrow b$ relation to the inverse relation only for pairs of adjacent states. Thus, in $\neg S$ the “followed by” relation may involve the pairs of non-adjacent states for which this has not been true in the original circuit S , while, on the other hand, some non-adjacent pairs of states that have been involved in the “followed by” relation in S , may come to be not in such a relation in $\neg S$.

It can also be seen from Definition 6.7 that, if in S , a variable z_i is excited (idle) in state a , then in the complement \bar{S} , this variable must be idle (excited) in a . Hence \bar{S} is defined by the system of the form

$$z_i = \bar{f}_i(Z) \quad (6.7)$$

which is obtained by complementing the right-hand sides of the equations for the original circuit.

EXAMPLE 6.7 (continued). The equations for $\neg S_1$ and $\neg S_2$ can be derived from (6.4) and (6.5), using (6.6), in the form

$$z_1 = \bar{z}_2 \bar{z}_3, \quad z_2 = z_2 \vee z_3, \quad z_3 = \bar{z}_1 z_2 \vee \bar{z}_1 z_3$$

and

$$z_1 = z_3, \quad z_2 = \bar{z}_1 \vee z_3, \quad z_3 = \bar{z}_1 \vee \bar{z}_2$$

respectively. The state-transition diagrams for these circuits are shown in Fig. 6.5(d) and (e). The equations for \bar{S}_1 and \bar{S}_2 are obtained from (6.4) and (6.5) using (6.7). For \bar{S}_1 the system is

$$z_1 = \bar{z}_2 \bar{z}_3, \quad z_2 = \bar{z}_2 \vee z_3, \quad z_3 = \bar{z}_1 z_2 \vee \bar{z}_1 \bar{z}_3$$

while the complement of S_2 is identical to its inverse.

6.3.2. LAWS AND PROPERTIES

Using expressions (6.2), (6.3), (6.6) and (6.7) for the union, intersection, inverse and complement, we can easily see that for the binary operations introduced, the laws of commutativity, associativity, idempotence, absorption and distributivity are satisfied. For the unary operations, the involution law is true. The complement satisfies deMorgan's laws $\overline{S_1 + S_2} = \overline{S_1} \cdot \overline{S_2}$, $\overline{S_1 \cdot S_2} = \overline{S_1} + \overline{S_2}$ whereas for the

inverse, the following laws hold: $\neg(S_1 + S_2) = \neg S_1 + \neg S_2$, $\neg(S_1 \cdot S_2) = \neg S_1 \cdot \neg S_2$.

In addition, for both unary operators, the following holds: $\neg \bar{S} = \overline{\neg S}$.

Thus the system $\langle \mathbb{C}, +, \cdot \rangle$ forms a *distributive lattice*. This lattice is complete because it contains the 0-element and the 1-element which are the circuits defined by the systems $z_i = z_i$ and $z_i = \bar{z}_i$, $i = 1, \dots, n$, respectively.

In the 0-circuit, for any states a , $a \rightarrow \emptyset$ and $\emptyset \rightarrow a$ hold, whereas in the 1-circuit, for any pair of states a and b , we have $a \rightarrow b$. Every circuit S in \mathbb{C} satisfies the following postulates $S \cdot 0 = 0$, $S \cdot 1 = S$, $S + 0 = S$, $S + 1 = 1$, $S \cdot \bar{S} = 0$, $S + \bar{S} = 1$. Hence, the system $\langle \mathbb{C}, +, \cdot, - \rangle$ is a Boolean algebra. This algebra contains $2^n \cdot 2^n$ elements where n is the number of variables in a circuit.

DEFINITION 6.8. The circuit S^0 which satisfies $\neg S^0 = S^0$ is called *self-inversible*.

Since 0-circuit and 1-circuit are self-inversible, i.e. $\neg 0 = 0$ and $\neg 1 = 1$, the sub-structure $\langle \mathbb{C}^0, +, \cdot \rangle$, where $\mathbb{C}^0 = \{S^0\}$ is a complete distributive lattice (sub-lattice), and the sub-structure $\langle \mathbb{C}^0, +, \cdot, - \rangle$ is a Boolean sub-algebra. This sub-algebra contains $2^n \cdot 2^{n-1}$ elements.

Let $[S]$ and $]S[$ denote the circuits which are defined by $S \cdot (\neg S)$ and $S + \neg S$, respectively. The Boolean equations for $[S]$ and $]S[$ can be expressed through the equations for S in the following form

$$z_i = \bar{z}_i f(z_i = 0) \bar{f}(z_i = 1) \vee z_i (\bar{f}(z_i = 0) \vee f(z_i = 1))$$

and

$$z_i = \bar{z}_i (f(z_i = 0) \vee \bar{f}(z_i = 1)) \vee z_i \bar{f}(z_i = 0) f(z_i = 1).$$

Circuits $[S]$ and $]S[$ are self-inversible. The expressions defining these circuits also define the mapping of the Boolean algebra of asynchronous circuits onto its sub-algebra of self-inversible circuits. Furthermore, $[S]$ is epimorphic with respect to the intersection, and $]S[$ is epimorphic with respect to the union.

THEOREM 6.5. In a circuit S such that $[S] = 0$ ($]S[= 1$) all the functions $f_i(Z)$ are isotonous (antitonous) in z_i . If both of these conditions are satisfied, then all variables z_i are non-self-dependent and $\neg S = \bar{S}$.

The algebra of asynchronous circuits, which is the Boolean algebra, is generated by the system containing at least $n + \lceil \log_2 n \rceil$ constituents where $\lceil \log_2 n \rceil$ is an integer such that $\log_2 n \leq \lceil \log_2 n \rceil \leq \log_2 2n$.

Consider a collection of the sets of circuits of two types $\langle \{S^z\}, \{S^m\} \rangle$ where the circuits of the first type $S^z_j, j = 1, \dots, n$, are defined by the systems of Boolean equations of the form $z_i = \bar{z}_i z_j \vee z_i \bar{z}_j, i = 1, \dots, n$ and the circuits of the second type $S^m, m = 1, \dots, \lceil \log_2 n \rceil$ by the system of Boolean equations of the form $z_i = \bar{z}_i \sigma_m(i) \vee z_i \bar{\sigma}_m(i)$. In these the coefficients $\sigma_m(i)$ are defined by the binary expansion of the variable number in the form

$$i = \sum_{m=1}^{\lceil \log_2 n \rceil} 2^{m-1} \sigma_m(i).$$

The two following lemmata hold for these types of circuits.

LEMMA 6.1. *In a circuit obtained from circuits of the first type by applying the union, intersection and complement operations to them, all variables in each state are either excited or idle.*

LEMMA 6.2. *In a circuit obtained from circuits of the second type by applying the union, intersection and complement operations to them, each variable is either excited or idle in all states of the circuit.*

From these lemmata, we can see that any circuit can be defined by the set of circuits $\langle \{S^z\}, \{S^m\} \rangle$ by applying the union, intersection and complement operations to them. To do this, we may, for example, pick out a set of states, and a set of variables excited in these states, from a given circuit. Then we can obtain, according to Lemmata 6.1 and 6.2, the circuits which correspond to these sets of states and variables, form the intersections of these circuits, and unify the intersections thus obtained, in all possible combinations.

Hence, the following theorem holds.

THEOREM 6.6. *The set of circuits in $\langle \{S^z\}, \{S^m\} \rangle$ is the minimum system of constituents of the algebra of asynchronous logical circuits using the union, intersection and complement operations.*

6.3.3. CIRCUIT TRANSFORMATIONS

Let $\phi(Z)$ be a logical function, and let Z^ϕ denote a set of states for a circuit S , where state a belongs to Z^ϕ if, and only if, $\phi(a) = 1$.

DEFINITION 6.9. Circuits S_1 and S_2 given by the Boolean equations $z_i = f_{1i}(Z)$ and $z_i = f_{2i}(Z)$, respectively, are called *equivalent* with respect to the set of states Z^ϕ if for every $a \in Z^\phi$, $f_{1i}(a) = f_{2i}(a)$, $i = 1, \dots, n$.

The first of the transformations considered consists in obtaining, for a given circuit S , the circuit $S^{[\phi]}$ which will be equivalent to the given circuit with respect to Z^ϕ . Moreover, this transformation demands that in $S^{[\phi]}$, any state a , $a \in Z^\phi$, is deadlock, i.e. $a \rightarrow \emptyset$. To obtain such a transformation, we use a circuit S^ϕ given by the system $z_i = \bar{z}_i \phi(Z) \vee z_i \bar{\phi}(Z)$, $i = 1, \dots, n$. In this circuit, for any state a , $a \in Z^\phi$, $a \rightarrow \emptyset$, while for any pair of states a and b such that $b \in Z^\phi$, $b \rightarrow a$. (The circuit S can be obtained from the constituents of the first type, according to Lemma 6.1). The required transformation is achieved via the intersection of the given circuit S and circuit S^ϕ , i.e. $S^{[\phi]} = S \cdot S^\phi$.

The second transformation is, in some sense, the inverse of the first one. It consists of obtaining, for a given circuit S , the circuit $S^{[\phi]!}$ in which no transition to state a in Z^ϕ from any state, adjacent to a , is possible, i.e. for any pair of adjacent states a and b , $a \in Z^\phi$, $b \xrightarrow{1} a$ does not hold. Hence, $S^{[\phi]} = S \cdot (\neg S^\phi)$.

The third transformation allows us to obtain, for a given circuit S , the circuit $S^{(\phi)}$ in which the states in set Z^ϕ are “isolated”, i.e. for any pair of adjacent states a and b , $a \in Z^\phi$, $a \rightarrow \emptyset$ holds and $b \rightarrow a$ does not. The circuit $S^{(\phi)}$ can be obtained as the intersection of $S^{[\phi]}$ and $S^{[\phi]!}$, i.e. $S^{(\phi)} = [S^\phi]$.

The fourth, and last, transformation allows us to obtain, for a given circuit S , the circuit $S^{[\phi]!}$ in which no transition to the states in Z^ϕ from those which are in Z^ϕ and adjacent to the former, is possible. Conversely, for any pair of adjacent states a and b such that $a \in Z^\phi$ and $b \in Z^\phi$, neither $a \rightarrow b$, nor $b \rightarrow a$ holds. The circuit $S^{[\phi]!}$ can be obtained as the union of circuits $S^{(\phi)}$ and $S^{(\phi)} : S^{[\phi]!} = S([S^\phi] + [S^\phi])$.

EXAMPLE 6.8. Let $\phi(Z) = z_j$, $j = 1, \dots, n$. Circuit S^{z_j} , being the constituent of the first type, is defined by the system $z_i = \bar{z}_i z_j \vee z_i \bar{z}_j$, $i = 1, \dots, n$ while circuit $\neg S^{z_j}$, is defined by the system $z_i = \bar{z}_i z_j \vee z_i \bar{z}_j$, $z_j = 1$, $i \neq j$. The system of equations for $[S^{z_j}]$ has the form

$$z_i = \bar{z}_i z_j \vee \bar{z}_j z_i, \quad z_j = z_j, \quad i \neq j.$$

For the last transformation, we use circuit $[S^j] + [\bar{S}^j]$ given by the system $z_i = \bar{z}_i$, $z_j = z_j$, $i \neq j$.

We can see, from this example, that using the inverse operator, together with the union, intersection and complement, we can obtain the constituents of the second type from those of the first type. This helps us in the formulation of the following theorem.

THEOREM 6.7. *The set of circuits S^j , $j = 1, \dots, n$, is the system of constituents of the algebra of asynchronous logical circuits using the union, intersection, inverse and complement operations.*

6.3.4. HOMOLOGICAL ALGEBRAS OF CIRCUITS

Algebraic structures with operations defined on equal sets of circuits are called *homological*. As far as asynchronous logical circuits are concerned, obtaining homological structures requires the extension of the list of algebraic operations on circuits. The most interesting are those operation sets, which, on the one hand, together with sets of circuits, form some known algebraic structures, and, on the other hand have a simple interpretation.

DEFINITION 6.10. The *disjunction (conjunction)* of circuits S_1 and S_2 , denoted by $S_{1 \vee 2}$ ($S_{1 \wedge 2}$), is the circuit defined by the system of Boolean equations of the form

$$z_i = f_{1i}(Z) \vee f_{2i}(Z), \quad i = 1, \dots, n \quad (z_i = f_{1i}(Z)f_{2i}(Z)).$$

A simple check may demonstrate that the newly introduced binary operations satisfy the commutative, associative, idempotent, absorption and distributive laws. Adding the previously defined complement operation, we can see that deMorgan's laws also hold: $\overline{S_1 \vee S_2} = \overline{S_1} \wedge \overline{S_2}$, $\overline{S_1 \wedge S_2} = \overline{S_1} \vee \overline{S_2}$ whereas for the inversion the similar laws are $\neg(S_1 \vee S_2) = \neg S_1 \wedge \neg S_2$, $\neg(S_1 \wedge S_2) = \neg S_1 \vee \neg S_2$. Hence, the triple $\langle \mathbb{C}, \vee, \wedge \rangle$ forms a *distributive lattice*. This lattice is complete, since it contains both the zero-element and the one-element, which are the circuits defined by $z_i = 0$ and $z_i = 1$, $i = 1, \dots, n$, respectively. We shall denote the zero-element as $0'$ and the one-element as $1'$.

Any circuit satisfies the following laws $S \wedge 0' = 0'$, $S \wedge 1' = S$, $S \vee 0' = S$, $S \vee 1' = 1'$ as well as $S \wedge \overline{S} = 0'$ and $S \vee \overline{S} = 1'$.

Thus, the system $\langle \mathbb{C}, \vee, \wedge, \neg \rangle$ is a *Boolean algebra*. Since this algebra, and the one considered above, $\langle \mathbb{C}, +, \cdot, \neg \rangle$, are homological and, hence, contain an equal number of elements - $2^n \cdot 2^n$ where n is the number of variables, the following theorem can be stated.

THEOREM 6.8. *The homological algebras of circuits $\langle \mathbb{C}, +, \cdot, \neg \rangle$ and $\langle \mathbb{C}, \vee, \wedge, \neg \rangle$ are isomorphic, i.e. there exists a one-to-one mapping ψ of one algebra onto the other, such as, for example*

$$\psi(S_1 + S_2) = \psi(S_1) \vee \psi(S_2), \quad \psi(S_1 \cdot S_2) = \psi(S_1) \wedge \psi(S_2).$$

Fig. 6.6 (a) and (b) show two Hasse diagrams for these algebras in the case when the circuit is defined on the set $\{z\}$ consisting of a single variable.

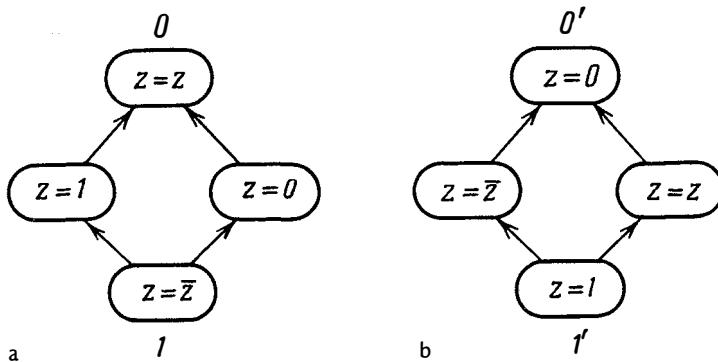


Figure 6.6(a) and (b). Hasse diagrams for two algebras.

For a Boolean algebra of circuits $\langle \mathbb{C}, +, \cdot, \neg \rangle$ ($\langle \mathbb{C}, \vee, \wedge, \neg \rangle$) we define a binary operation of identity (equivalence) as

$$S_{1 \equiv 2} = S_1 \equiv S_2 = S_1 S_2 + \bar{S}_1 \bar{S}_2 \quad (S_{1 \leftrightarrow 2} = S_1 \Leftrightarrow S_2 = (S_1 \wedge S_2) \wedge (\bar{S}_1 \wedge \bar{S}_2)).$$

The Boolean equations for identity $S_{1 \equiv 2}$ and equivalence $S_{1 \leftrightarrow 2}$ of circuits S_1 and S_2 are expressed in the following way:

$$z_i = z_i(f_{1i}(Z) \bar{f}_{2i}(Z) \vee \bar{f}_{1i}(Z)f_{2i}(Z)) \vee \bar{z}_i(f_{1i}(Z)f_{2i}(Z) \vee \bar{f}_{1i}(Z) \bar{f}_{2i}(Z))$$

$$z_i = f_{1i}(Z)f_{2i}(Z) \vee \bar{f}_{1i}(Z) \bar{f}_{2i}(Z)$$

respectively. In $S_{1 \equiv 2}$, each variable z_i is excited in state a if, and only if,

$f_{1i}(a) = f_{2i}(a)$, while in $S_{1\leftrightarrow 2}, f_{1\leftrightarrow 2i}(a) = 1$ holds in these, and only in these, states. Hence, for any circuit S , the following laws are satisfied

$$S \equiv S = 1 \text{ and } S \curvearrowleft S = 1',$$

$$S \equiv \bar{S} = 0 \text{ and } S \curvearrowleft \bar{S} = 0'.$$

Therefore, $S_{1\equiv 2}$ and $S_{1\leftrightarrow 2}$ may be considered as a certain kind of proximity between S_1 and S_2 in corresponding homological algebras.

From the definition of identity and equivalence, we can see that any circuit satisfies the following laws

$$S \equiv 1 = S \curvearrowleft 1' = S,$$

$$S \equiv 0 = S \curvearrowleft 0' = \bar{S};$$

it is also interesting that for any circuit

$$S \equiv 1' = S \curvearrowleft 1 \text{ and } S \equiv 0' = S \curvearrowleft 0,$$

i.e. the equal elements in these homological algebras are equally near, not only to the one- and zero- elements of their own algebras, but also to the one- and zero-elements of each other's algebras.

Let S' denote a circuit given by the expression $S \equiv 1'$ or $S \curvearrowleft 1$. The Boolean equations for S' expressed via those of S have the form $z_i = z_i \bar{f}_i(Z) \vee \bar{z}_i f_i(Z)$. The latter may be regarded as a definition of another unary operation in the circuit algebra. For this operation, the involution law is satisfied.

Because of satisfying the laws

$$(S_1 + S_2)' = S_1' \vee S_2' , \quad (S_1 S_2)' = S_1' \wedge S_2' ,$$

$$(S_1 \vee S_2)' = S_1' + S_2' , \quad (S_1 \wedge S_2)' = S_1' \cdot S_2' ,$$

$$(\bar{S})' = \bar{S}'$$

the expression defining S' (or corresponding unary operation) are isomorphic mappings of a Boolean algebra of the circuits $\langle \mathbb{C}, +, \cdot, - \rangle$ onto a Boolean algebra of circuits $\langle \mathbb{C}, \vee, \wedge, - \rangle$. The notations $0'$ and $1'$, introduced earlier independently, have now come to have the sense of the mapping of 0 and 1 , respectively. Due to the involution of this mapping $(0')' = 0$ and $(1')' = 1$. The inversion satisfies $\neg S' = (\neg S)'$.

The isomorphic mapping obtained for the homological algebras considered, allows the expression of the binary operations of one algebra through those of the

other, as

$$S_1 + S_2 = (S_1 \vee S_2) \wedge 1 \vee (S_1 \wedge S_2), \quad S_1 \cdot S_2 = (S_1 \vee S_2) \wedge 0 \vee (S_1 \wedge S_2)$$

$$S_1 \wedge S_2 = (S_1 + S_2) \cdot 1' + S_1 \cdot S_2, \quad S_1 \wedge S_2 = (S_1 + S_2) \cdot 0' + S_1 \cdot S_2.$$

The system of constituents of the circuit algebra $\langle \mathbb{C}, \vee, \wedge, - \rangle$ may be obtained with the aid of the given mapping from the constituents of the circuit algebra $\langle \mathbb{C}, +, \cdot, - \rangle$; the mapping $(S^z)^j$ of circuits of the first type S^z is defined by the system of the form $z_i = z_j, i = 1, \dots, n$, and the mapping of a circuit of the second type is defined by the system $z_i = \sigma_m(i), i = 1, \dots, n$.

It is obvious that various Boolean algebras of circuits can be obtained in the same way. Since, in this process, any circuit S (or its complement \bar{S}) may be chosen as the one (zero) element, the number of various homological algebras of circuits existing on the set of n variables is $2^{n \cdot 2^n}$. In particular, assuming $1' = 0, 0' = 1$, we can obtain the algebra which is dual to $\langle \mathbb{C}, +, \cdot, - \rangle$, and the isomorphic mapping on which will be given by the complement operation.

From the two homological algebras of circuits, considered above, the first is made up of operations which have a fairly clear behavioural interpretation (the union, intersection, complement and inverse of circuits), whereas, the other has the clear decomposition context (conjunction, disjunction). The fact that these algebras are Boolean and isomorphic to each other helps us to express the operation of one through the operations of the other. The transformations which exploit the proposed operations may be of use in the process of asynchronous logic design for eliminating mistakes in the analysis of circuits. The formal analysis techniques will be discussed later, in Chapter 8.

6.4 Reference notations

The discussion of the methods of composition of asynchronous processes is presented in [9]. The proof of the Muller theorem [272], [273] can be found in [120]. The generalizations of this theorem are presented in [63]. The counter shown in Fig. 6.4 was proposed in [23], and is also referred to in [145]. The fact that further generalizations of the Muller theorem exist is justified by the possibility of a deeper “incorporation” into each other of the bit stages in the counter, as shown in [26]. Other composition techniques, which may also be regarded as generalizations, are in [247].

The algebra of asynchronous circuits was also studied in [176] and [177]. The ideas presented in these works are close to those on equivalent transformations of asynchronous circuits discussed in [126], [182] and [185].

CHAPTER 7

THE MATCHING OF ASYNCHRONOUS PROCESSES AND INTERFACE ORGANIZATION

When signalling with a horn, be always so wise
To be equally fair, strict and precise.

Kozma Prutkov.

The design of modern digital systems and computers is based on a modular principle. Hardware units, computers themselves, as well as software components, are built of separate functional and structural modules. The latter can be combined to form rather complex systems. The integration of modules into a system implies the need for the organization of interaction between modules, and that, in its turn, requires substantial time, hardware and software resources. The key idea reflecting the interaction aspect in a system is the concept of interfacing.

In a broad sense, *the interface is a set of rules defining the co-operative functioning of hardware and software components*. Sometimes, an implementation constructed in accordance with these rules can also be regarded as an interface.

The interface, in contrast to an interfacing adapter, is not a part of the hardware, but is the border between modules of the system. A set of signals and communication procedures is defined at such a border. The interface should also provide a unified technique for data transfer, independent of the internal characteristics of the interacting modules, and referring only to the signal exchange between devices. The properly specified meaning of interface is the necessary condition of correct interface design.

The set of rules and facilities comprising an interface structure is usually divided into three major layers: mechanical, electrical and logical.

The *mechanical layer* defines the requirements which must be satisfied by the design of connectors – plugs and jacks, cables, adaptor boards, etc.

The *electrical layer* defines allowances for input and output signal parameters: voltage and/or current levels, loadings, impedances. They are usually technology dependent.

The *logical layer* provides co-ordination of information flow between modules. It defines representation modes for signals, symbols, and messages and provides denotation and classification of signals and messages. This layer is the most complicated part of the interface specification. The logical matching of modules integrated into a system implies that the functioning of these modules is performed in exact compliance with established algorithms. The major notion reflecting the

co-ordinated operation of the logical layer entities is the concept of a communication protocol.

A *communication protocol*, called from now on, simply a *protocol*, is a set of rules that must be satisfied in providing the ordered information interchange between two or more entities.

The term “protocol” has found widespread usage in the literature on computing and data communication. Its appearance is somewhat akin to the concept of a diplomatic protocol (a set of rules ordering various diplomatic acts), or a protocol of a meeting (a written form of the meeting proceedings). A whole theory, the *theory of protocols*, has even arisen in communication technology, whose main subject is formal modelling and methods for analysis and synthesis of interface interactions. Despite the obvious success in this direction, the practical ways of specifying interface protocols are still more concerned with informal, or semi-formal tools, such as natural language, timing diagrams, cyclograms and flow charts. Due to this, they are subject to ambiguity and error. The requirements that protocol specifications should comply with are the compactness of representation of communication procedures, their unambiguous interpretation, the possibility of checking whether they are complete and consistent, and the possibility of the direct conversion of a protocol to an interfacing hardware logic, to name but a few. The only way to achieve such goals, is through formalizing the description notations, for example, in terms of asynchronous processes (see Sections 2.1 and 6.1).

In this chapter, our discussion will be oriented towards the systems in which communication protocols are implemented in hardware. Nevertheless, the proposed approach may also be used for computer networks where protocols are normally realized in software.

Among the various problems that arise in interface organization, the current discussion concentrates only on some of them. First of all, we present a formal notion of a protocol itself which is the key issue in the area. Then we provide an instructive example for describing an interface protocol with the subsequent translation of it into hardware. It will also be shown that in solving problems intrinsic to lower level interfaces, some “logical” remedies can be effectively used. For example, the well-known phenomenon of a skew in parallel bundles of wires can be avoided by means of aperiodic circuit design in combination with self-synchronizing codes.

7.1 Matched asynchronous processes

We consider a pair of reinstated processes (see Definition 6.1) $P_1^r = \langle S_1, F_1, I_1, R_1 \rangle$ and $P_2^r = \langle S_2, F_2, I_2, R_2 \rangle$ where the situations are structured to have input and output components. The elements of a sub-set Y_1^* of the output component values in the

situations of the first process ($Y_1^* \subseteq Y_1$) are associated, by means of *semantic identification*, with the elements of a sub-set X_2^* of the input component values in the situations of the second process ($X_2^* \subseteq X_2$). Similarly, the elements of a sub-set Y_2^* of the output component value in the situations of the second process ($Y_2^* \subseteq Y_2$) are associated with the elements of a sub-set X_1^* of input component values in the situations of the first process ($X_1^* \subseteq X_1$).

We first build, if possible, a sequential composition of P_1^r and P_2^r assuming that $Y_1^* = X_2^*$ (or $Y_2^* = X_1^*$). The result of the composition is an asynchronous process $P_{1,2}^r$ ($P_{2,1}^r$). In the situations of $P_{1,2}^r$ ($P_{2,1}^r$) we pick out the input component which is identical to the input component in the situations of P_1^r (P_2^r). Then, if again possible, we construct the closure $P_{1,2}^c$ ($P_{2,1}^c$) of process $P_{1,2}^r$ ($P_{2,1}^r$), bearing in mind that $Y_2^* = X_1^*$ ($Y_1^* = X_2^*$) in their elements. Using the reduction and closure notations, that were defined earlier, we can easily show that $P_{1,2}^c = P_{2,1}^c$.

The situations of a process thus built have the following structure:

$$s = (u, v, z)$$

where $u = y^1 = x^2$, $v = y^2 = x^1$, $z = (z^1, z^2)$ with components u and v assuming the values from sets $U \subseteq Y_1^* = X_2^*$ and $V \subseteq Y_2^* = X_1$, respectively.

DEFINITION 7.1. Two asynchronous processes are called *matched* with respect to sets U and V if the result of applying sequential composition and closure operations to these processes yields an autonomous process defined on a non-empty set of situations whose components u and v are the semantically identified input and output components in the situations of these processes.

7.2 Protocol

Let an asynchronous process $P = \langle S, F, I, R \rangle$ be given. The situations of the process have input (x) and output (y) components defined in sets X and Y , respectively. Consider the projections

- (1) $S(X \times Y)$ of set S on $X \times Y$,
- (2) $F(X \times Y)$ of set F on $X \times Y \times X \times Y$,
- (3) $I(X \times Y)$ of set I on $X \times Y$,
- (4) $R(X \times Y)$ of set R on $X \times Y$.

DEFINITION 7.2. The asynchronous process given by $P(X \times Y) = \langle S(X \times Y), F(X \times Y), I(X \times Y), R(X \times Y) \rangle$ is called the *projection* of process P on $X \times Y$.

DEFINITION 7.3. The projection of the autonomous process $P_{1,2}^c$, obtained as a result of matching processes P'_1 and P'_2 , on $U \times V$ will be called the *protocol* $\Pi(U, V)$ of matching P'_1 and P'_2 , with respect to sets U and V .

Protocol $\Pi(U, V)$ is a formal specification of process interaction, and it is realized by the $P_{1,2}^c$ process.

Consider the projections of the matched processes P'_1 and P'_2 on $U \times V$. It is clear that these projections have the following properties

$$\begin{aligned} I_1(U \times V) &\subseteq R_2(U \times V), \\ I_2(U \times V) &\subseteq R_1(U \times V), \\ F_1(U \times V) \cup F_2(U \times V) &= F_{1,2}^c(U \times V). \end{aligned}$$

The practical statement of the problem of operational co-ordination of a pair of mechanisms can be done as follows. Two hardware entities are given, as well as an interconnection protocol that they must satisfy in their joint behaviour. The protocol specifies a set of mutually identified values of signals, and thereby, the inputs and outputs which must be interconnected, as well as the order of changes of these signals.

In order to solve these problems we should:

- represent the interconnected modules as APs with the input and output components picked out,
- define the sets of mutually equivalent (due to semantic identification) components in accordance with the protocol,
- match, if possible, the processes to each other, with respect to the semantically identified sets,
- construct the projection of the AP that defines the joint behaviour of the matched processes, and
- make sure that the protocol of matching obtained describes the given protocol (a semantic interpretation is usually required for the latter).

There are at least three cases when this problem cannot be solved.

- 1) Sometimes the values of the output component of one AP are not equivalent to those of the input component of the other AP, because, for example, of difference in the physical parameters of signals representing these values (directly unmatchable levels of signals, their different physical origin, different encodings of signals that have the same meaning, etc.).

2) It may happen that sets U and V with respect to which the APs can be matched do not include all the values required by the protocol, and because of this, the protocol is either not implementable, or cannot be implemented completely.

3) Even though all members of the identified sets are present in the autonomous process describing the joint behaviour of the matched processes, there are certain sequences of changes of component values prescribed by the protocol that are not realizable by any means.

In all such cases, the modules should be interconnected through a special interfacing module, called an adapter, rather than directly to each other.

7.3 The matching asynchronous process

The interconnection of two mechanisms through an adapter can be formally defined by the APs that are realized by the mechanisms and adapter, viz. P'_1 , P'_2 and P'_m whose situations have the following structure

$$s^1 = (x^1, y^1, z^1), \quad s^2 = (x^2, y^2, z^2), \quad s^m = (y^1, y^2, x^1, x^2, z^m),$$

(in the situations of process P'_m associated with the adapter, components y^1 and y^2 are the inputs, and x^1 and x^2 are the outputs).

The pair of APs, P'_1 and P'_m , is matched with respect to sets U_1 and V_1 of allowed values of components y^1 and x^1 . Similarly, APs P'_2 and P'_m are matched with respect to sets U_2 and V_2 of allowed values of components y^2 and x^2 . Such a composition of APs P'_1 , P'_2 and P'_m may be represented as an autonomous P_{1m2} defined on situations of the form

$$s^{1m2} = (s^1, s^m, s^2).$$

DEFINITION 7.4. An asynchronous process P'_m is called the *matching process* (or *match-maker*) for APs P'_1 and P'_2 if in the P_{1m2} process:

- (1) there exists a situation s_i^{1m2} with component $s_i^1 \in R_1$ such that any sequence starting in s_i^{1m2} in process P_{1m2} will inevitably lead to situation s_j^{1m2} with component $s_j^2 \in I_2$, and
- (2) there exists a situation s_k^{1m2} with component $s_k^2 \in R_2$ such that any sequence starting in s_k^{1m2} in process P_{1m2} will inevitably lead to situation s_l^{1m2} with component $s_l^1 \in I_1$.

In other words, P_m^r will be the matching process if, and only if, amongst its situations there can be found one such that the appearance of a certain value of the input component y^1 (y^2) when process $P_1(P_2)$ reaches its resultant, will necessarily cause the change of the output component $x^2(x^1)$ which will result in the transition of $P_2(P_1)$ from a resultant to an initiator.

EXAMPLE 7.1. Consider a system consisting of a channel processor (CP), an adapter (Ad) and a pair of server modules – Serv1 and Serv2, as shown in Fig. 7.1(a). CP, Serv1 and Serv2 are linked via a common data bus. Each of these modules may acquire the bus to perform a data transfer operation on it. The system operates using the “master-slave” principle, and the data may be transferred between CP and servers. The master-ship is assigned to CP, whereas servers have the slave status. The CP sets-up a server address on A lines, and then activates the request signal $a = 1$. The adapter initiated by this signal decodes the address to produce either $a_1 = 1$ or $a_2 = 1$, which are the signals initiating the reception or transmission of data. After the data transfer operation is completed, the corresponding server sets $b_i = 1$, $i = 1, 2$. The adapter receives this signal and then produces $b = 1$. After this, the process of server disconnection begins: the CP releases a to 0 (in doing so it may also change the address) upon which the adapter releases to 0, the a_i signal, which has been set to 1. The i -th server, in response to $a_i = 0$, releases b_i to 0, whereupon the adapter also resets b to 0, and the system returns to its original state.

The Petri net shown in Fig. 7.1(b) specifies the interaction between the above modules on the level of establishing and breaking the connection. This net defines a matched AP and consists of four fragments related to the given modules. Because the CP may address only one of the servers at a time, the token may appear either in place 5 or in place 6, and hence, either in place 7 or place 8, but not in both.

For the fragments enclosed by dashed lines, we can outline the situations which are initiators and resultants.

The resultants of the CP fragments are manifested by:

- 1) tokens in places 4 and 7 thus implying the setting-up of the Serv1 address and the change of a to 1;
- 2) tokens in places 4 and 8 thus implying the setting-up of the Serv2 address and the change of a to 1;
- 3) tokens in places 2 and 4 thus implying that $a = 0$.

The initiator of the CP fragment is the 1) resultant of the Ad fragment.

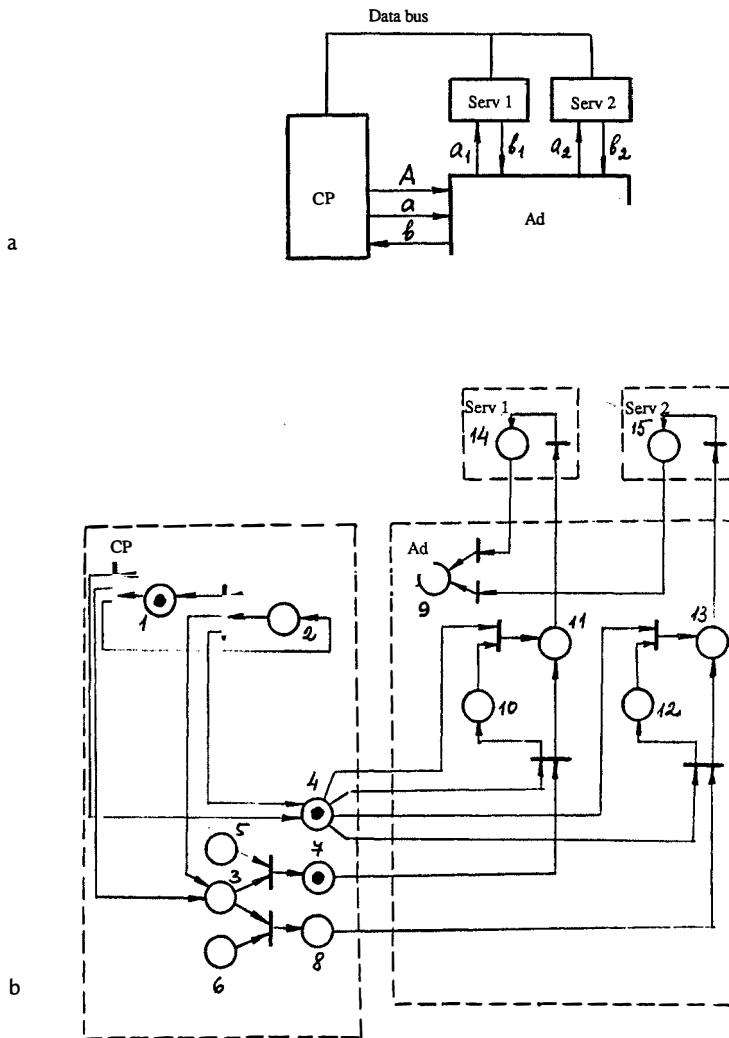


Figure 7.1(a) and (b). A system consisting of a channel processor (CP) and two servers (Serv1 and Serv2) interconnected via adapter (Ad): (a) block diagram; (b) Petri net specifying a protocol of logical connection and disconnection.

The resultants of the Ad fragment are manifested by:

- 1) a token in place 9 which implies that $b = 1$ or $b = 0$;
- 2) a token in place 11 which implies that $a_1 = 1$ or $a_1 = 0$;
- 3) a token in place 13 which implies that $a_2 = 1$ or $a_2 = 0$.

The initiators of the Ad fragment are the resultants of CP, Serv1 and Serv2 fragments.

For the Serv1 fragment, the resultant is manifested by the presence of a token in place 14 meaning that $b_1 = 1$ or $b_1 = 0$, and the initiator is the 2) resultant of the Ad fragment.

For the Serv2 fragment, the resultant is marked by a token in place 15 meaning that $b_2 = 1$ or $b_2 = 0$, and the initiator is the resultant 3) of the Ad fragment.

Note the following. In this example, the level of protocol description presented in Fig. 7.1(b) is such that the protocol can be used directly for the implementation of the adapter. This is due to the persistence and safeness of the net of Fig. 7.1(b), and for such a class of Petri nets, appropriate implementation disciplines were presented in Section 5.1. The result of the implementation will be a controller state machine that, being integrated with the operational part (using techniques outlined in Chapter 6), will produce the entire implementation of the adapter.

Another example of protocol specification, together with a method for its translation into hardware logic, will be discussed in the following section.

7.4 The T2 interface

In this section we discuss a real interface design for a data communication system. The interface entitled T2 is, on the one hand, rather specific, which is reflected in a peculiar terminology used for the data transfer entities and signals, but, on the other hand, is common enough to be close to existing standard interfaces. The authors aimed at two major goals when including this material in the book. The first is to help the reader to become familiar with a typical graphical notation for interface and protocol specification at the level of control signals. This is presented in 7.4.1 and 7.4.2. The other goal is to show the technique for the translation of a given protocol description into its hardware implementation; and this can be found in 7.4.2 and 7.4.3.

7.4.1. GENERAL NOTATIONS

The T2 interface is used for the communication between a central control unit (CCU) and individual servers, each of which is called a group control unit (GCU). Each GCU has an interface board (IB)⁽¹⁾, that is able to realize a matching process, and a number of peripheral devices (PDs). Note that teletypes, CRT video terminals,

⁽¹⁾ An *interface board*, or *adapter*, is usually a device that interconnects modules with different data representations, or modules using different types of unified connector options.

printers, data communication devices, special-purpose processors, etc. may be used as PDs.

IBs and the CCU have input and output *buses* which are sub-divided into *data* and *control* lines. The data lines from the CCU to the IBs are used to transmit commands and data. The data lines from IBs to the CCU are used for transmitting data and status information. The latter can be used to inform the CCU of the status of the IBs and the status of the related PDs.

The signals on control lines have a two-fold role. On the one hand, they are used to identify the type of information on the output data lines, and on the other, they can be used for the notification, or acknowledgement, of the reception of information from the input data lines. The following discipline must be satisfied in such transactions. The setting-up of signals on data lines must always strictly precede the setting-up of signals on control lines. This ordering discipline is shown in Fig. 7.2.

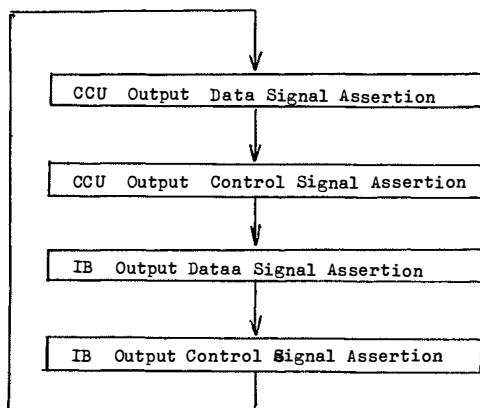


Figure 7.2. Signal ordering in the T2 interface.

In every transaction cycle both in the CCU and the IB a single output control line signal is allowed to change. Hence, the model considered in Section 4.6 can adequately describe the behaviour of the CCU and the IBs.

The CCU has 9 output data lines CCD1-CCD9 and 2 output control lines CCC1-CCC2. Each GCU has 9 output data lines GCD1-GCD9 and 2 output control lines GCC1-GCC2. The signals produced by all the GCUs are merged in a wired-OR way.

Table 7.1 contains the meaning of the signal value combinations on the CCC lines, their respective symbols and the description of the type of information with which they are associated.

Table 7.2 presents similar records related to the signals on the GCC.

CCC1-2	Meaning	Symbol	Information on CCD
0 0	The command from CCU to GCU; The acknowledgement of the GCU status code reception	C	A command byte and its odd parity bit
1 0	The write signal from CCU to GCU in the write or exchange operations; The acknowledgement of the data reception in the CCU in the read operation	W	A data byte with its odd parity bit in the write or exchange operations or the spacer byte
0 1	The GCU address	A	A unitary code value of the GCU address
1 1	The permission signal allowing the GCUs to send calls to the CCU	P	A unitary code value of the GCU address or the spacer byte.

Table 7.1

GCC1-2	Meaning	Symbol	Information on GCD
0 0	The GCU status; The acknowledgement of the address reception	S	A GCU status byte with its odd parity bit, if the GCU is logically connected to the CCU, and the spacer byte, otherwise
1 0	The read signal from GCU to CCU in the read or exchange operations; The acknowledgement of the write operation command reception	R	A data byte with its odd parity bit in the read and exchange operations or the spacer byte in the write operation
0 1	The call from GCU for CCU	Z	A unitary code value of the GCU address
1 1	The modified status of GCU; The acknowledgement of the data reception in the write and exchange operations; The acknowledgement of the transmission of the final byte in the read operation	M	A GCU status byte with its odd parity bit

Table 7.2

Some of these specifications require additional comments.

The command byte has the following structure. CCD1 and CCD2 carry the operation code (00 - control operation, 10 - read operation, 01 - write operation,

11 - exchange operation). The CCD3 = 1 signal is produced to notify the final byte transfer. CCD4 - CCD6 are used for the address code of the PD to which the command is sent. The values CCD7 and CCD8 are left application dependent, and CCD9 is used to send the odd parity bit for CCD1-8 lines.

The *control* operation is not related to data transfer and results in changing the state of an IB or a particular PD.

The *read* operation involves the data byte transfer from a chosen PD to the CCU.

The *write* operation involves the data byte transfer from the CCU to a chosen PD.

The *exchange* operation involves both read and write operations.

In addressing a GCU, the unitary encoding is used in which the j -th GCU is addressed while the remaining eight positions stay equal to 0. Thus the T2 interface allows up to nine GCUs to be interconnected to the CCU.

The code combination with all nine bits equal to 0 is called the *spacer byte*.

If neither the GCU address byte, nor the spacer byte is transmitted on CCD₈ (GCD) lines, the CCD₉ (GCD₉) line carries the odd parity bit ($\text{CCD}_9 = \bigoplus_{i=1}^8 \text{CCD}_i$).

The status byte has the following structure. GCD1 and GCD2 indicate the potential operational mode (00 - ready, 10 - ready to write, 01 - ready to read, 11 - ready to exchange). GCD3 is set to 1 when the final byte has just been transferred. The signals on GCD4-6 represent the address code of the PD which is currently connected to the IB. The signals on GCD7 and GCD8 are left for further applications. GCD9 is used for the transmission of the odd parity bit for the status code on GCD1-8.

7.4.2. COMMUNICATION PROTOCOL

At the control signal level the protocol can be defined by the directed graph shown in Fig. 7.3.

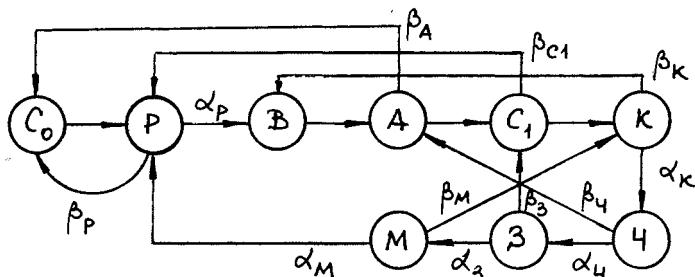


Figure 7.3. Specification for the T2 interface on the control signals level.

Some of the arcs in this graph are labelled with the conditions which, when true, enable the corresponding transitions. The vertices associated with the states of the output control lines of the GCU (S_0, S_1, Z, R, M) are connected by arcs to the vertices associated with the states of the output control lines of the CCU (P, A, C, W), and vice versa.

The state S , shown in Table 7.2, splits into two vertices S_0 and S_1 , where S_0 is the original state of the CCU. In this state, the GCD output lines carry the spacer byte. S_1 corresponds to the logical connection between the GCU and the CCU. In this state, the GCD output lines carry a status byte value.

The need for splitting S into a pair is caused by the specific features of operation on the bus. The GCUs which do not participate in communication at a given transaction set their outputs to zeros, and it is against the “background” of these zeros that the CCU can identify the values of outputs of the GCU which participates in the current transaction with the CCU.

Transitions of GCUs are determined, not only by the states of the output control lines of the CCU, but also by the conditions $\alpha_P, \beta_P, \alpha_A, \beta_A, \alpha_C, \beta_C, \alpha_W, \beta_W$. The conditions of the form α and β labelled with the same sub-script are alternative (complementary) to each other. They are dependent on the internal state of the GCU and its data inputs.

According to the T2 interface specifications, the above conditions can be expressed in the following form

$$\begin{aligned}\alpha_P &= CCD_i \vee z_1 \vee z_2; \quad \alpha_A = CCD_i \cdot \left(\bigwedge_{j \neq i} CCD_j \right); \\ \alpha_C &= \left(\bigoplus_{i=1}^9 CCD_i \right) \cdot \left(\bigwedge_{i=1,2,4,5,6} (CCD_i \oplus \bar{z}_i) \right) \cdot (z_1 \vee z_2); \\ \alpha_W &= z_2 \left(\left(\bigoplus_{i=1}^9 CCD_i \right) \vee (\text{Count GW} = 3) \right) \vee z_1 \bar{z}_2 z_3\end{aligned}$$

where z_i is the i -th bit of the GCU status register, and the ($\text{Count GW} = 3$) condition is the predicate that becomes true if a data byte from the CCU has been assessed, three times in succession, as incorrect in parity checks (in this case, the GCU accepts this byte in spite of the presence of the error).

The graph of Fig. 7.3 and the above conditions allow the derivation of the excitation functions for memory elements of the device realizing the transition functions (see the structure of Fig. 4.18).

We now demonstrate a synthesis procedure for that device. To make the synthesis easier, we redefine the behaviour of the device in an appropriate form. Assume that the set of signals, both input and output, constitute a full state of the

machine. If, for example, the machine is in the state S_0 and its input becomes equal to P, then the full state is presumed to be expressed as PS_0 .

Fig. 7.4 shows the state-transition diagram in which a full state is encoded by five bits, where the first two bits stand for the input signal value (see Table 7.1), the third and fourth bits are assigned to the output signal value (see Table 7.2), and the fifth bit indicates the state of the flip-flop T1, which is the logical connection flag. It is set to state 0 in state S_0 and to 1 in the other states.

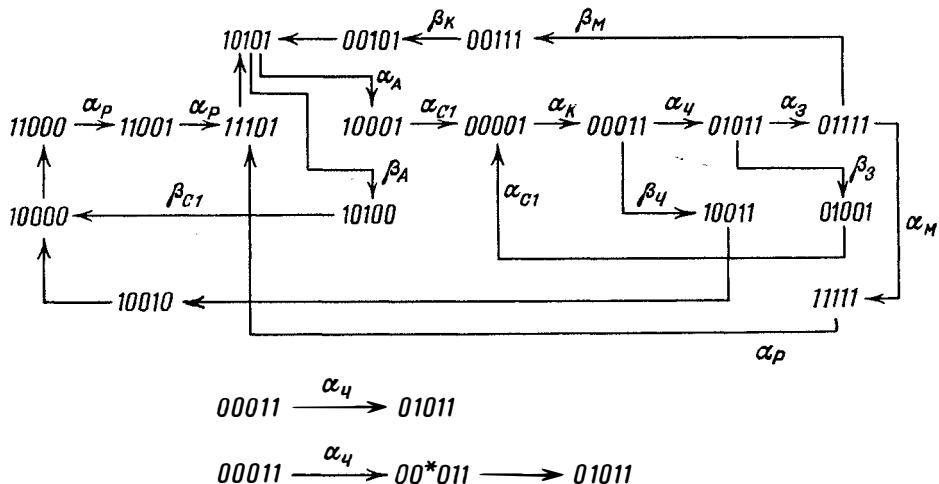


Figure 7.4. State-transition diagram for the T2 interface.

This diagram differs from the ordinary state-transition diagrams in that it contains transitions associated with arcs labelled by logical conditions, and these transitions are enabled if, and only if, such conditions assume the value of 1. This diagram exactly corresponds to that of Fig. 7.3, except for two items:

- according to the interface specification, the transition from full state PM to state PS is not allowed, and hence, not shown in the diagram, and
- in the transitions PS-PZ, AR-AS and AZ-AS, the state of flip-flop T1 is changed and, to keep the discipline of maximum adjacency, these transitions are performed through transient states PS, ART1 and AZT1, respectively.

The diagram contains only stable full states. Let us examine one of the transitions in greater detail. For example

$$00011 \xrightarrow{\alpha_R} 01011.$$

This transition is done in the following way. If the $\alpha_R = 1$ predicate is true, the variable associated with the second position in the state combination is excited and then changes. This change can thus be redefined in the form

$$\begin{array}{c} \alpha_R \\ 00011 \xrightarrow{\alpha_R} 00^*011 \rightarrow 01011. \end{array}$$

If we also take into account that from the conditions $\alpha_i = 1$ and $\beta_i = 1$, because of their alternation, only one may be true at a time, then we may deduce that in the system only one variable may be excited at a time, and the removal of this excitation is done only through the change of the value of this variable. This statement allows the implication that the diagram can be implemented in an aperiodic circuit provided, of course, that conditions α_i and β_i are correctly realized.

7.4.3. IMPLEMENTATION

In the following, we describe a sequence of steps leading towards a logical implementation. The first step of this procedure is the building of a truth table for functions x_1, x_2, y_1, y_2, T_1 (the first two columns in Table 7.3). This can be done in the following way.

1. Column 1 contains all 32 binary combinations of variable values in their respective positional notation order.
2. Against those combinations which are defined in the state-transition diagram, we write the full states to which the system goes after completing an associated transition. If some variable changes its value independently of additional conditions, then its value after this change is written into a corresponding position in column 2. If the change of the variable value from 0 to 1 depends on condition α_i (β_i), we should enter symbol α_i (β_i) in a corresponding position in column 2. If the change of variable from 1 to 0 is dependent on α_i (β_i), then we enter $\bar{\alpha}_i$ ($\bar{\beta}_i$) in a corresponding position.

For example, state 10001 may change either to 00001, if $\alpha_{S1} = 1$, or to 11001, if $\beta_{S1} = 1$. Then in the second column of the table we should enter $\bar{\alpha}_{S1}\beta_{S1}001$ against 10001.

3. The values of functions for the combinations which are not presented in the state-transition diagram may be arbitrary (don't cares) and thus are marked by "*" in corresponding entries.

Combinations 00000 and 01000 are not used in the diagram. However, the diagram reflects the operation of the CCU and a GCU connected to the CCU. When the GCU is disconnected from the CCU, the states corresponding to these combinations may occur, and in such cases, the GCU should stay in its state as indicated by rows 1 and 3 of Table 7.3.

x_1	x_2	y_1	y_2	T1	x_1	x_2	y_1	y_2	T1	S_1	R_1	S_2	R_2	S_3	R_3
0	0	0	0	0	0	0	0	0	0	*	0	*	0	*	*
1	0	0	0	0	1	1	0	0	0	*	0	*	0	*	*
0	1	0	0	0	0	1	0	0	0	*	0	*	0	*	*
1	1	0	0	0	1	1	0	0	α_P	0	*	0	*	α_P	*
0	0	1	0	0	*	*	*	*	*	*	*	*	*	*	*
1	0	1	0	0	1	0	0	0	0	1	0	*	0	*	*
0	1	1	0	0	*	*	*	*	*	*	*	*	*	*	*
1	1	1	0	0	*	*	*	*	*	*	*	*	*	*	*
0	0	0	1	0	*	*	*	*	*	*	*	*	*	*	*
1	0	0	1	0	1	0	0	0	0	*	0	1	0	*	*
0	1	0	1	0	*	*	*	*	*	*	*	*	*	*	*
1	1	0	1	0	*	*	*	*	*	*	*	*	*	*	*
0	0	1	1	0	*	*	*	*	*	*	*	*	*	*	*
1	0	1	1	0	*	*	*	*	*	*	*	*	*	*	*
0	1	1	1	0	*	*	*	*	*	*	*	*	*	*	*
1	1	1	1	0	*	*	*	*	*	*	*	*	*	*	*
0	0	0	0	1	0	0	β_C	α_C	1	β_C	0	α_C	0	*	0
1	0	0	0	1	$\bar{\alpha}_{S1}$	β_{S1}	0	0	1	0	*	0	*	*	0
0	1	0	0	1	β_{S1}	$\bar{\alpha}_{S1}$	0	0	1	0	*	0	*	*	0
1	1	0	0	1	1	1	α_P	0	1	α_P	0	0	*	*	0
0	0	1	0	1	1	0	1	0	$\frac{1}{2}$	*	0	*	0	*	0
1	0	1	0	1	1	0	$\bar{\alpha}_A$	0	β_A	0	α_A	0	*	0	β_A
0	1	1	0	1	*	*	*	*	*	*	*	*	*	*	*
1	1	1	0	1	1	0	1	0	1	*	0	0	*	*	0
0	0	0	1	1	β_R	α_R	0	1	1	0	*	*	0	*	0
1	0	0	1	1	1	0	0	$\bar{\alpha}_A$	$\bar{\beta}_A$	0	*	0	α_A	0	β_A
0	1	0	1	1	0	1	α_W	$\bar{\beta}_W$	1	α_W	0	0	β_W	*	0
1	1	0	1	1	*	*	*	*	*	*	*	*	*	*	*
0	0	1	1	1	0	0	$\bar{\alpha}_C$	$\bar{\beta}_C$	1	0	α_C	0	β_C	*	0
1	0	1	1	1	*	*	*	*	*	*	*	*	*	*	*
0	1	1	1	1	α_M	$\bar{\beta}_M$	1	1	1	*	0	*	0	*	0
1	1	1	1	1	1	1	1	$\bar{\alpha}_P$	1	*	0	0	α_P	*	0

Table 7.3

The implementation is built of flip-flops y_1, y_2 and T1. The excitation functions for these flip-flops can be derived from the truth table, using the fact that the canonical equation of an RS-flip-flop has the form $y' = S \vee \bar{R}y, SR = 0$.

Expressions of S and R as functions of y and y' (the current and next states of the flip-flops, respectively) can be obtained using Table 7.4.

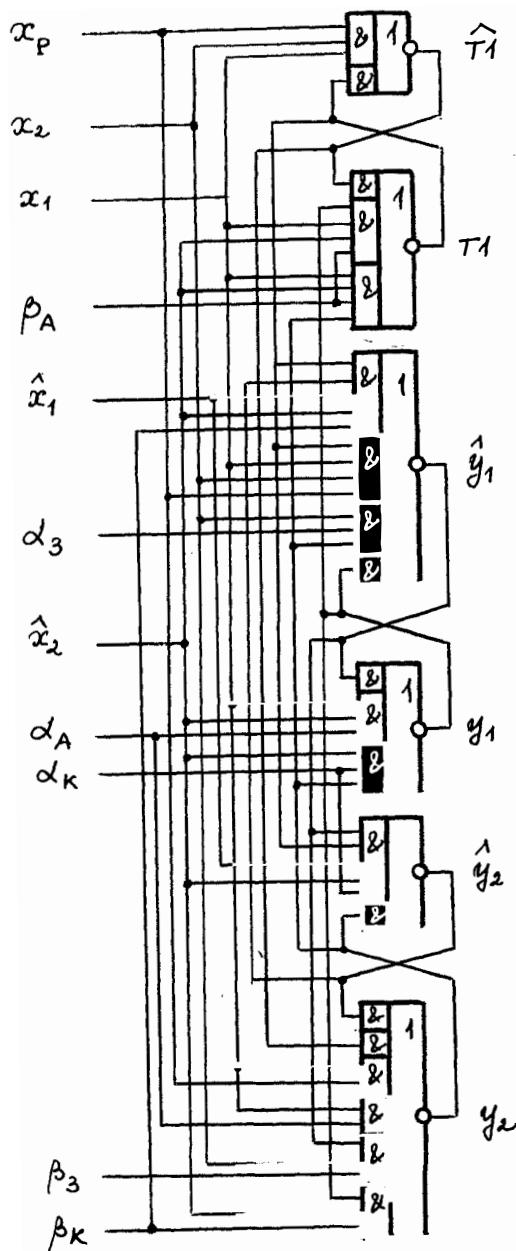


Figure 7.5. Implementation of a control machine for the T2 interface.

y	y'	S	R	y	y'	S	R
0	0	0	*	0	*	*	*
1	0	0	1	1	*	*	*
0	1	1	0	0	α	α	0
1	1	*	0	1	$\bar{\alpha}$	0	α

Table 7.4

Columns 3, 4, and 5 of Table 7.3, filled in using Table 7.4, represent the excitation functions for flip-flops y_1, y_2 and T_1 , respectively. Using “don't cares”, the functions are minimized to the following disjunctive forms:

$$S_1 = \bar{x}_1 \bar{x}_2 \bar{y}_2 T_1 \beta_C \vee x_1 x_2 T_1 \alpha_P \vee x_2 y_2 \alpha_W;$$

$$R_1 = x_1 \bar{x}_2 \alpha_A \vee \bar{x}_2 y_2 \alpha_C;$$

$$S_2 = \bar{x}_1 \bar{x}_2 \bar{y}_2 T_1 \alpha_C;$$

$$R_2 = T_1 \vee x_1 \alpha_P \vee x_1 \alpha_A \vee x_2 \bar{y}_1 \beta_W \vee \bar{x}_2 y_1 \beta_C;$$

$$S_{T1} = x_1 x_2 \alpha_P; \quad R_{T1} = x_1 \bar{x}_2 y_1 \beta_A \vee x_1 \bar{x}_2 y_2 \beta_A.$$

Finally, it should be remembered that all the flip-flops in the implementation shown in Fig. 4.18 should have an all-zero transient-state. Hence, the control machine will finally consist of seven flip-flops $T_1, T_2, T_1, T_1, T_1, y_1, y_2$ and T_1 . The circuit implements the excitation functions as shown in Fig. 7.5. Conditions α_i and β_i are produced in the operational machine.

7.5 Asynchronous interface organization

We now want to discuss the problem at the lower level of asynchronous interface logic. Special emphasis is put on techniques for delay-insensitive data transfer.

Traditionally, when an asynchronous data transfer is mentioned, it implies the use of at least two groups of lines for sending data in each direction. The first group is formed by *information lines*, a parallel bunch of wires for transmitting data, addresses and/or commands. The other group is formed by *control lines*, which are sometimes called *identification lines*. The operational rules for such a transfer demand that the following discipline is an absolute necessity: the signals in the

information lines must be set-up before those on the control lines. As this takes place, the delay of setting an identifier on the control lines must be such that the same order is guaranteed at the opposite end of the bus, i.e. at the inputs of the receiver. If this is not the case, there is a risk of the receiver strobing incorrect values from the information lines. Therefore, in implementing traditional interfaces, we must take account of the scatter in line delays. This is usually referred to as a *skew phenomenon*, which is dependent on the distance between the sender and the receiver. The delay value for the identifier, to compensate a skew, is usually chosen from the transmission parameters of the worst case and this significantly degrades the data transmission rates.

In this section, we present a practical technique for the organization of an interface which is *invariant to the skew*.

To eliminate the influence of a skew on the functioning of interface modules, we should use self-synchronizing codes for information representation, and implement the interface logic in the aperiodic way.

The following two assumptions are accepted in implementing an asynchronous interface:

- (1) delays in transmission lines and the scatter in their values may be arbitrary but finite;
- (2) element delays and wire delays inside the interface machine satisfy the assumption of Chapter 4 related to aperiodic state machines.

It is now clear that these assumptions do not allow the use of input variables in their single-rail representation with a control (phase) signal because such representation will be subject to a skew effect. Therefore, in this case, we should transmit data in self-synchronizing codes, particularly, using the *differential encoding* introduced in Section 3.8.

Each of the two interacting modules may function in one of two operation modes: the sender mode and the receiver mode. The interaction is based on the handshake principle, i.e. the module playing the part of a sender, after transmitting an item of data, must receive an *acknowledgement* from the receiver about the reception and acceptance of the data, before it will proceed to further transactions.

We agree that each of the interacting modules is structurally represented as an interface machine consisting of *transmitter*, *receiver* and *controller* units. The latter is intended to co-ordinate the alternating operation of the former units. The transmitter consists of the encoder, which converts given information into self-synchronizing code combinations, and the output converter, which converts the self-synchronizing code into the differential encoding. The receiver consists of the input converter which converts input signals into self-synchronizing code combinations, the input code checker and, possibly, the decoder which would convert the self-synchronizing code into a representation suitable for further use in the receiver.

All these components of the interface machine, with the exception of the input converter and checker, can be constructed using the approaches to aperiodic circuit design that have already been discussed in the previous chapters.

A discussion of a method for implementing an interface machine of the above kind or one-byte-width data transfer is given in the subsequent sub-sections.

7.5.1. USING THE CODE WITH IDENTIFIER

One of the potential structures of the interface machine for one-byte-width data transfer using the code with identifier, introduced in Section 3.3, is shown in Fig. 7.6.

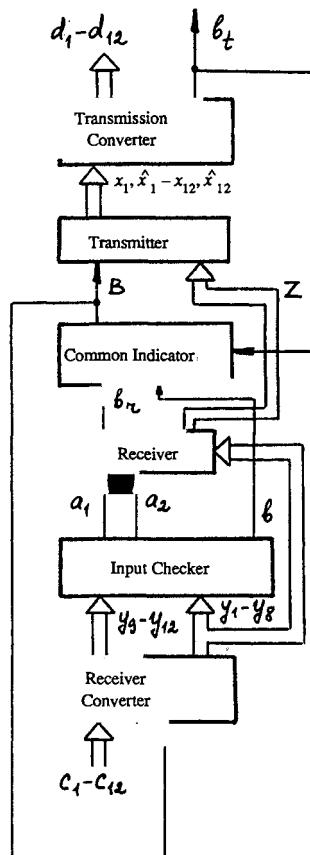


Figure 7.6. Block diagram for an interface machine implementing the byte-width data transfer, using code with identifier (Berger code).

We assume the following notations.

$Z = \{(z_1, \dots, z_8)\}$ is a set of combinations of the binary positional representation of bytes transferred from the receiver part to the transmitter part.

$X = \{(x_1 \hat{x}_1, \dots, x_{12} \hat{x}_{12})\}$ is a set of combinations for the two-rail representation of transmitted bytes in the code with identifier.

i_X is the all-zero spacer for combinations in X .

$D = \{(d_1, \dots, d_{12})\}$ is a set of combinations in the differential representation at the transmitter output.

$Y = \{(y_1, \dots, y_{12})\}$ is a set of combinations in the code with identifier at the output of the receiver converter.

e_Y is the all-one spacer for combinations in Y .

$C = \{(c_1, \dots, c_{12})\}$ is a set of combinations in the differential representation at the receiver input.

$Y^* = \{(y_1, \dots, y_8)\}$ is a set of combinations of the values of basic bits in the representation with identifier at the output of the receiver converter.

A combination in some set U will be denoted by a small letter with a superscript, that is, for example, u^i .

The transmitting part of the machine consists of the transmitter and the transmission converter. The transmitter accepts z^i and produces x^i . The control of the transmitter operation is done by the phase signal B , the output of the common indicator. When $B = 1$, the output of the transmitter is set to i_X . When $B = 0$, the output is set to x^i .

The transmitter converter realizes the mapping of X into D . It contains the indicator b_t which indicates completion in the entire transmitting part of the circuit. If the inputs of the converter are set to i_X , its output stores the previous value d^p , and $b_t = 0$, but if they are set to some value x^q , then the outputs change to d^q , and $b_t = 1$.

The receiving part of the circuit incorporates the receiver converter, the checker of y^i and the receiver itself.

The receiver converter realizes the mapping of C into Y . It is controlled by the phase signal B , the output of the common indicator. When $B = 0$, the converter stores the previously written combination c^i . If c^i coincides with the combination loaded into the converter, the latter produces the spacer (the use of the all-one spacer is because of the desire to simplify the circuit implementation). After the setting of a new combination c^j at the inputs, the outputs of the converter change to a corresponding combination y^j . When $B = 1$, the combination c^j is loaded into the converter's register, after which the converter output becomes equal to e_Y .

The input combination checker produces $a_1 = 1, a_2 = 0$ ($a_1 = 0, a_2 = 1$), if the combination y^j is compulsory (auxiliary), and then $b = 1$. If the converter outputs are equal to e_y , the checker produces $a_1 = a_2 = 0$, and then $b = 0$.

The receiver uses combination y^{j*} with its subsequent conversion, if necessary, into the positional representation. Signals a_1 and a_2 play the role of phase signals for the receiver. When $a_1 = 1, a_2 = 0$ or $a_1 = 0, a_2 = 1$, the receiver accepts combination y^{j*} and produces z^l which may be used by the transmitter for producing a new byte of the transmitted data. After the completion of the transient processes in the receiver, b_r goes to 1. When $a_1 = a_2 = 0$, the state of the receiver does not depend on y^{j*} , and its outputs remain equal to z^l with $b_r = 0$.

In addition to the indicated units, the structure also contains the common indicator with output B which co-ordinates the operation of the receiving and transmitting parts.

The composite description of the operation of both parts can be given by the signal graph shown in Fig. 7.7.

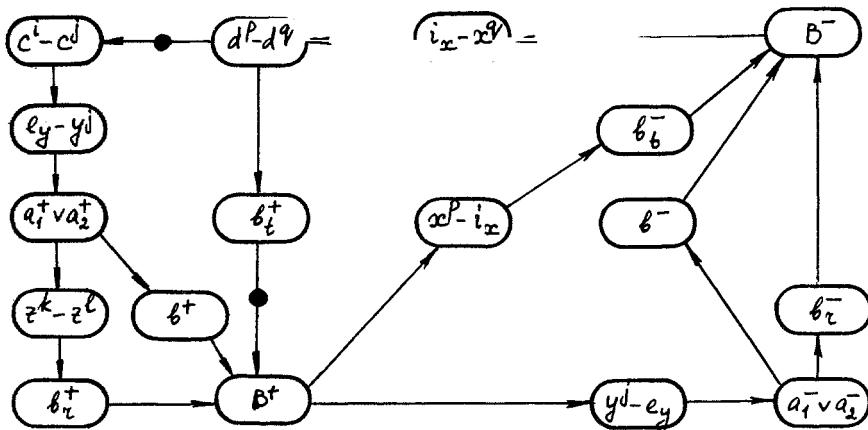


Figure 7.7. Signal graph for the machine of Fig. 7.6.

This graph may be interpreted as a protocol of interaction between the receiver and the transmitter within one interchange machine, but it is, of course, by no means a representation of the communication between the two interface machines.

In the initial state, which corresponds to the initial marking, the receiver is ready to receive the current data byte, and the transmitter is in the phase of data transmission.

The functioning of the signal graph in Fig. 7.7 can be commented on as follows.

In the original state a new code combination is applied to the inputs of the receiver converter (transition c^i-c^j), and then it is converted into a corresponding combination y^j (transition e_Y-y^j), after which $a_1 \vee a_2 = 1$ and $b = 1$. When either $a_1 = 1$ or $a_2 = 1$ ($a_1 a_2 = 0$), the receiver begins to accept the data y^j , and then produces z^l associated with its state (transition z^k-z^l), whereupon b_r is set to 1. The output B of the common indicator changes to 1 only after the condition $b_t b_r b = 1$ (the synchronization of $b_t = 1$, $b_r = 1$ and $b = 1$) becomes true. The change of B to 1 indicates the completion of the reception and acceptance phase in the receiver. The $B = 1$ signal enables the receiver to enter the preparation phase for the reception of the next data byte transmission. The same signal also indicates the storing of data in the converter, after which its outputs become equal to e_Y (transition y^j-e_Y). This fact is manifested by the input combination checker ($a_1 = a_2 = b = 0$). After $a_1 = a_2 = 0$, the second phase of the receiver operation (the preparation for the next data reception) begins, after which $b_r = 0$. When $b_r = b = 0$, the second phase (the storing of the received data) is assumed to be finished.

In the transmitting part, after $B = 1$, the transient process results in the setting up of i_X (transition x^p-i_X), and then $b_t = 0$. During this, the output converter stores the previous data combination d^p . Signal $b_t = 0$ indicates that the corresponding phase of the transmitter operation is over. When $b_t = b_r = 0$, the output of the common indicator B goes to 0. After this, the transmitter enters the transmission phase, and its outputs assume the next code value (transition i_X-x^q). This value is further converted at the converter outputs (transmission d^p-d^q), whereupon b_t goes to 1, and the circuit returns to its original state.

The following issues are of particular concern when speaking about the implementation.

1. The transmitter can be designed using the canonical methods discussed in Chapter 4. We should also bear in mind that the transmitter output is represented in the two-rail code.
2. The transmitter converter corresponds to that of Example 4.3 in 4.5.3.
3. The receiver converter is built as a register based on the D-flip-flops of Fig. 4.11(c) (Section 4.4).
4. The input data checker of the receiver is presented in Section 4.2.
5. The receiver is built as an ordinary matched machine with phase signal $a_1(a_2)$.

6. The common indicator of the interface machine is implemented as a Γ -flip-flop according to the equation

$$B = b_l b_r b \vee B(b_l \vee b_r \vee b).$$

7.5.2. USING THE OPTIMALLY BALANCED CODE

The following two examples are intended to illustrate the implementability of the interface machine using the optimally balanced code.

7.5.2.1. Half-Byte Data Transfer

The circuit using $C_6^3 = 20$ code combinations is built in such a way that 16 combinations, the compulsory ones, are decoded into the four-bit positional code. The structure and the operation of the circuit are similar to those of Fig. 7.6 with the exception of the different number of information lines. The only units to be modified are the input converter and checker.

The input, or receiver, converter with inputs c_1-c_6 and outputs $y_1-y_4, \hat{y}_3, \hat{y}_4$, together with decoding the differential representation into the optimally balanced code on p_1-p_6 , also converts the latter into the 4-bit positional representation. The converter circuit is implemented according to Table 7.5. Redefining the functions on the "don't care" combinations, the realizations for $y_1-y_4, \hat{y}_3, \hat{y}_4$ have the form:

$$\begin{aligned} y_1 &= p_1, & y_2 &= p_2, & \hat{y}_3 &= \overline{\overline{p_1 p_6} \vee p_2 p_4 \vee p_4 p_6}, \\ y_3 &= \overline{\overline{p_1 p_3} \vee p_2 p_5 \vee p_3 p_5}, & \hat{y}_4 &= \overline{\overline{p_1 p_5} \vee p_2 p_6 \vee p_5 p_6}, \\ y_4 &= \overline{\overline{p_1 p_4} \vee p_2 p_3 \vee p_3 p_4}. \end{aligned}$$

The input data checker with inputs $y_3, \hat{y}_3, y_4, \hat{y}_4$ generates three signals a_1, a_2 and b . The combination of $a_1 = 1$ and $a_2 = 0$ means that the outputs y_1-y_4 of the receiver converter carry one of the 16 compulsory positional code combinations that is identified by $\hat{y}_3 \neq y_4, \hat{y}_4 \neq y_4$, i.e. the third and fourth bits are represented by the two-rail code. When $a_1 = 0$ and $a_2 = 1$, it means that the outputs $y_1-y_4, \hat{y}_3, \hat{y}_4$ carry one of the additional combinations, separated by the line at the bottom rows of Table 7.5. When $a_1 = 0$ and $a_2 = 0$, the outputs are in the spacer state i_Y . The signalling order for signals a_1, a_2 and b is the same as given by the graph of Fig. 7.7.

p_1	p_2	p_3	p_4	p_5	p_6	y_1	y_2	y_3	\hat{y}_3	y_4	\hat{y}_4
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	0	0	0	1	0	1
0	0	1	0	1	1	0	0	1	0	0	1
0	0	1	1	0	1	0	0	0	1	1	0
0	0	1	1	1	0	0	0	1	0	1	0
1	0	0	0	1	1	1	0	0	1	0	1
1	0	1	0	1	0	1	0	1	0	0	1
1	0	0	1	0	1	1	0	0	1	1	0
1	0	1	1	0	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	1	1	0	0	1
0	1	1	1	0	0	0	1	0	1	1	0
0	1	1	0	1	0	0	1	1	0	1	0
1	1	0	0	0	1	1	1	0	1	0	1
1	1	0	0	1	0	1	1	1	0	0	1
1	1	0	1	0	0	1	1	0	1	1	0
1	1	1	0	0	0	1	1	1	0	1	0

1	0	0	1	1	0	1	0	0	0	1	1
1	0	1	0	0	1	1	0	1	1	0	0
0	1	0	1	1	0	0	1	1	1	0	0
0	1	1	0	0	1	0	1	0	0	1	1

Table 7.5

The signals a_1 , a_2 and b are implemented in the form:

$$\bar{a}_1 = \overline{y_3 y_4 \vee y_3 \hat{y}_4 \vee \hat{y}_3 y_4 \vee \hat{y}_3 \hat{y}_4}, \quad \bar{a}_2 = \overline{y_3 \hat{y}_3 \vee y_4 \hat{y}_4},$$

$$\bar{b} = \bar{a}_1 \bar{a}_2 \bar{a}_3 \vee \bar{b}(\bar{a}_1 \bar{a}_2 \vee \bar{a}_3),$$

where

$$\bar{a}_3 = \overline{y_3 \vee \hat{y}_3 \vee y_4 \vee \hat{y}_4}.$$

The membership of the checker circuit within the class of aperiodic circuits can be established by checking its indicatability through Statement 4.4 of sub-section 4.3.1.

7.5.2.2. Byte Data Transfer

The above implementation can be used as a basis for the byte data transfer. In this case, we need 12 information lines, i.e. the inputs of the receiver converter will be

c_1c_{12} . Its outputs are y_1-y_4 , $\hat{y}_3, \hat{y}_4, y_5, y_6, y_7, \hat{y}_7, y_8, \hat{y}_8$. Signals $y_3, \hat{y}_3, y_4, \hat{y}_4$ and $y_7, \hat{y}_7, y_8, \hat{y}_8$ are connected in their respective order to the inputs of two indicators of the input data checker. The outputs of these indicators are denoted by $\overline{a}_1^1, \overline{a}_2^1, \overline{a}_3^1$ and $\overline{a}_1^2, \overline{a}_2^2, \overline{a}_3^2$ (the circuits for signals \overline{b}^1 and \overline{b}^2 are not used and may be deleted). Then a_1, a_2 and b can be realized in the form

$$a_1 = \overline{\overline{a}_1^1 \vee a_1^2}, \quad a_2 = \overline{\overline{a}_1^1 \overline{a}_2^1 \vee \overline{a}_1^1 \overline{a}_2^2 \vee \overline{a}_1^2 \overline{a}_2^1 \vee \overline{a}_1^2 \overline{a}_2^2}, \\ b = a_1 a_3 a_4 \vee a_2 a_3 a_4 \vee b(a_1 \vee a_2 \vee a_3 \vee a_4),$$

where

$$a_3 = \overline{\overline{a}_3^1 \overline{a}_3^2}, \quad a_4 = \overline{\overline{a}_3^1 \vee a_3^2}.$$

In this construction the total number of code combinations is $20 \times 20 = 400$, from which 256, the compulsory ones, are intended for the transmission of the 8-bit positional code.

7.5.2.3. Using Non-Balanced Representation

The following example demonstrates the implementation of the interface which uses the minimum number of bits and a non-balanced encoding discipline.

The minimum number of bits for representing $N^0 = 2^8 = 256$ combinations of the 8-bit positional code in the optimally balanced representation is eleven. This is due to the constraint $C_{10}^5 = 252 < 256 < C_{11}^5 = 462$. Since only 256 combinations (out of 462) are to be encoded, then abandoning the balanced representation, we can simplify the conversion function and, at the same time, stay within the same code space, which is minimum.

We shall use the code which is further referred to as the modified code with identifier. Six of its less significant bits are used without any conversion as they are simply those of the positional representation. The remaining five bits of the modified code with identifier are assigned to the identifier, which should be used for the following purposes:

- determining the weight w of the combination in the main (less significant) six bits,
- encoding the remaining two most significant bits of the positional code.

Since each combination of the main bits is associated with four combinations of values in the two most significant bits, each w -class must be associated with at least four identifiers.

The following variant of encoding in which each w -class is associated with 4 identifiers satisfies three conditions imposed on the code with identifier in Section 3.3. The correspondence between identifiers and the classes of combinations of the main bits, as well as the encoding of the two most significant bits of positional representation, are presented in Table 7.6.

w	p_1	p_2	p_3	p_4	p_5	y_7	\hat{y}_7	y_8	\hat{y}_8
0	0	1	1	1	1	0	1	0	1
0	1	0	1	1	1	1	0	0	1
0	1	1	0	1	1	0	1	1	0
0	1	1	1	0	1	1	0	1	0
<hr/>									
1	0	1	1	1	0	0	1	0	1
1	1	0	1	1	0	1	0	0	1
1	1	1	0	1	0	0	1	1	0
1	1	1	1	0	0	1	0	1	0
<hr/>									
2	0	0	1	1	1	0	1	0	1
2	1	0	1	0	1	1	0	0	1
2	0	1	0	1	1	0	1	1	0
2	1	1	0	0	1	1	0	1	0
<hr/>									
3	0	0	1	1	0	0	1	0	1
3	1	0	0	1	1	1	0	0	1
3	0	1	1	0	1	0	1	1	0
3	1	1	0	0	0	1	0	1	0
<hr/>									
4	0	1	0	1	0	0	1	0	1
4	1	0	0	1	0	1	0	0	1
4	0	1	1	0	0	0	1	1	0
4	1	0	1	0	0	1	0	1	0
<hr/>									
5	0	1	0	0	1	0	1	0	1
5	0	0	0	1	1	1	0	0	1
5	0	0	1	0	1	0	1	1	0
5	1	0	0	0	1	1	0	1	0
<hr/>									
6	0	1	0	0	0	0	1	0	1
6	0	0	0	1	0	1	0	0	1
6	0	0	1	0	0	0	1	1	0
6	1	0	0	0	0	1	0	1	0

Table 7.6

The structure of the given interface machine is similar to that of Fig. 7.6. The receiver converter and input code checker are built in a different way.

The receiver converter:

- realizes the conversion of the differential representation into the modified code with identifier,
- decodes identifiers of the modified code into combinations of the two-bit positional representation,
- generates intermediate signals g_1 and g_2 which are necessary for the indication of completion of the transient processes in the converter circuit.

The conversion of the differential representation into the modified code with identifier is performed by the 11-bit circuit, the implementation for each bit of which has already been described (see also Section 3.8).

Redefining the functions $y_7, \hat{y}_7, y_8, \hat{y}_8$ with the use of "don't care" combinations, we obtain

$$y_7 = \overline{p_7 p_9} \vee u_1 u_2 p_7 p_8 p_{11} \vee u_1 u_2 u_3 p_7 \vee u_5 p_{10} p_{11} \vee u_5 u_6 p_{10},$$

$$\begin{aligned} \hat{y}_7 = & \overline{p_8 p_{10}} \vee u_1 u_2 p_9 p_{10} p_{11} \vee u_1 u_2 u_3 p_9 p_{10} \vee u_1 u_2 u_3 p_8 p_9 \vee \\ & \vee u_5 p_9 p_{11} \vee u_5 u_6 p_9 \vee u_1 u_2 u_3 u_4 p_8, \end{aligned}$$

$$\begin{aligned} y_8 = & \overline{p_7 p_8 p_{11}} \vee u_1 p_7 p_8 \vee u_1 u_2 p_8 p_{10} p_{11} \vee u_3 p_8 p_9 p_{11} \vee \\ & \vee u_1 u_2 p_9 u_3 u_4 \vee u_4 u_5 p_7 p_{11} \vee u_4 u_5 u_6 p_7, \end{aligned}$$

$$\begin{aligned} \hat{y}_8 = & \overline{p_9 p_{10} p_{11}} \vee u_1 p_9 p_{10} \vee u_1 u_2 p_7 p_9 p_{11} \vee u_3 p_7 p_{10} p_{11} \vee \\ & \vee u_5 p_8 p_{11} \vee u_5 u_6 p_8 \vee u_1 u_2 u_3 u_4 p_{10}, \end{aligned}$$

where

$$u_1 = \overline{e h_1}, \quad u_2 = \overline{e_1 e_2 h_2 \vee e_2 h_1 h_2},$$

$$u_3 = \overline{e_1 e_3 h_3 \vee e_2 e_3 h_2 h_3 \vee e_3 h_1 h_3},$$

$$u_4 = \overline{e_1 \vee h_1 \vee e_2 h_2 \vee e_2 h_3 \vee e_3 h_2}, \quad u_5 = \overline{e_2 \vee h_2 \vee e_3 h_3},$$

$$\begin{aligned} u_6 &= \overline{\overline{e_3 \vee h_3}}, \quad e_1 = \overline{\overline{p_1 \vee p_2 \vee p_3}}, \quad e_2 = \overline{\overline{p_1 p_2 \vee p_1 p_3 \vee p_2 p_3}}, \\ e_3 &= \overline{\overline{p_1 \vee p_2 \vee p_3}}, \quad h_1 = \overline{\overline{p_4 \vee p_5 \vee p_6}}, \\ h_2 &= \overline{\overline{p_4 p_5 \vee p_4 p_6 \vee p_5 p_6}}, \quad h_3 = \overline{\overline{p_4 p_5 p_6}}. \end{aligned}$$

The input data checker is described by the equations

$$g_1 = \overline{\overline{(\bigvee_{i=1}^6 u_i) \vee p_{11}}} \cdot \overline{\overline{\bigvee_{i=7}^{10} p_i}}, \quad g_2 = \overline{\overline{(\bigvee_{i=1}^6 u_i) \vee p_{11}}} \vee \overline{\overline{\bigvee_{i=7}^{10} p_i}},$$

$$b = (y_7 \vee \hat{y}_7) \cdot (y_8 \vee \hat{y}_8) \cdot g_1 g_2 \vee b(y_7 \vee \hat{y}_7 \vee y_8 \vee \hat{y}_8 \vee g_1 \vee g_2).$$

7.6 Reference notations

A comprehensive bibliography on communication protocols, the methods for their specification, classification, verification, syntheses, etc. are presented, for example, in [223]. It is therefore possible to speak about the theory of protocols as an independent trend. Several special issues of journals and conference proceedings can be recommended, amongst which [244] contains a series of articles on various aspects of this theory [201], [202], [220] and [303]. A number of books [51], [93], [154], [174] and [190] with special reference to protocols have been published. Nevertheless there is room for further development of the formal theory of protocols beyond what may be found in existing papers [203], [204], [233], [241], [265], [267], [303], [309], [313], [314], and surveys [266] and [298]. The main statements of [148] are actually “borrowed” from the results of [9]. The notion of a protocol given in Chapter 7 is a reformulated version of those of [9], [61] and [147]. It can be concluded from the formal constructions that protocols can be specified in the languages used to represent asynchronous processes. The representation of a protocol as a set of “initiator-resultant” pairs, allows us to reduce the length of the specification in contrast to the majority of other approaches which often produce redundant specifications.

Example 7.1 was inspired by the outlines of [315].

The organization of an asynchronous interface using optimally balanced codes with identifiers was discussed in [6], [129] and [130]. The skew phenomenon was mentioned in [114].

Approaches to the construction of models for the controlled protocol were considered in [146] and [255].

CHAPTER 8

ANALYSIS OF ASYNCHRONOUS CIRCUITS AND PROCESSES

There's a story about a physicist who earned tremendous prestige by deriving a formula for the radius of the Universe from rather general mathematical notions. It was an impressive formula richly stuffed with constants e, c, h . There were also several π 's and many square roots in it. Since he was a confirmed theoretician, he did not care very much about actual numerical values. Several years had passed before any one found himself calculating the radius of the Universe.

It turned out to be ten centimetres.

Ian Stewart

The complexity of the control of asynchronous processes is the motivation for the development of methods of analysis for them. The most common analysis problems for asynchronous computations involve the detection of deadlock, and the checking that the functioning of the system is independent of the duration of actions performed by the components of the system. Such problems emerge at all stages of the design of systems controlling asynchronous processes. For example, in the development of parallel program notations and protocols of inter-modular communications, in the synthesis of the micro-program control of concurrent processes, in the implementation of the hardware control of processes, say, by means of aperiodic circuits, to mention but a few.

Here some issues in the *functional analysis of asynchronous circuits* will be discussed. However, the methods presented in this chapter may equally well be used for analysing parallel asynchronous processes on other levels of computer systems.

The dynamic behaviour of a digital circuit consists in the ordering of events concerned with the switching of the elements of the circuit. The element switching times may differ substantially from each other, as they depend on the physical characteristics of the elements, and change with time. In the general case, an arbitrary number of circuit elements may be simultaneously active. Thus, the behaviour of a digital circuit can be described by asynchronous switching processes.

The advantages of aperiodic circuits are mainly due to their speed-independent behaviour; such a circuit operates correctly independently of the values of the switching delays of its elements. Proving the characteristics providing such

operation is a topic of the analysis of aperiodic circuits. This analysis is essentially different from the traditional analysis of synchronous and asynchronous circuits.

When analysing synchronous circuits, we normally simulate the functions realized by the circuit by finding the response of the circuit to a given input and/or, conversely, determining the input value producing the given response, etc.

As far as asynchronous circuits are concerned, it is usual to think that if a circuit is free from critical races, then it is correct, and hence, the traditional problem for the analysis of asynchronous circuits is proving the absence of critical race conditions. The models of asynchronous logical circuits usually imply that transition processes in a circuit will terminate within some fixed time interval, which is determined by assuming the delay of any circuit, even in the worst case, to be less than some given value. If this constraint is abandoned, and the circuit is required to operate correctly for any finite element delays, then the *absence of critical races will not be sufficient for the desired operation of the circuit*.

It is a well-established fact that speed-independent circuits, including semi-modular, distributive and totally sequential ones, operate correctly for arbitrary finite element delays. Naturally, all these classes of circuits are absolutely free from any undesired glitches and hazards at the outputs of their elements, and hence, are free from critical race conditions.

The description of the operation of a circuit, where the latter is considered as some kind of asynchronous process, can be effectively used for the analysis of aperiodic circuits at the earlier stages of their design. An asynchronous process allows the representation of the circuit operation at its highest level of abstraction, without any regard to concrete implementation details, which should be refined at the later stages. In the beginning of the circuit design process, when only major “milestones” in the circuit operation are defined, the number of situations in the corresponding process is rather small. By gradual refinement of the description (for, example, by encoding the process situations or by interpreting the process in target models like the Muller model or Petri nets) some local properties, whose presence will guarantee the correct behaviour of the system (for example, the controllability of the corresponding asynchronous process), can be elicited.

8.1 The reachability analysis

When studying the functioning of discrete systems it is often necessary to find the causal relationships between situations or between sets of situations (we restrict ourselves to using the terms introduced for asynchronous processes for the sake of consistency). These relationships assume the form of the “followed by” relation, and the corresponding problem for its analysis will be called the *reachability problem*.

There are two major statements for this problem:

- 1) to construct a set of situations that are reachable from a given situation or set of situations – the *forward reachability problem*, and
- 2) to construct a set of situations from which a given situation or set of situations can be reached – the *backward reachability problem*.

Looking back to the material of Section 2.4, within the context of solving the reachability problem in the language of the Muller model, we shall call the object of study a *circuit*, and any variable z_i of the circuit, an *element*.

The reachability relation on a set A of states of a circuit is defined as a transitive closure of the “immediately followed by” relation introduced in Section 2.4. In other words, *state β is reachable from state α* if there exists a path in the transition diagram that leads from α to β . We denote this fact as $\alpha \Rightarrow \beta$. Hence $\alpha \Rightarrow \beta$ implies that in the transition diagram we can find a sequence of states $\delta, \gamma, \dots, \psi$ for which $\alpha \rightarrow \delta \rightarrow \gamma \rightarrow \dots \rightarrow \psi \rightarrow \beta$ where \rightarrow is the “immediately followed by” relation.

We also introduce the notion of *stepwise reachability*, or the so-called *neighbour reachability*, denoted by $\alpha \xrightarrow{1} \beta$, which implies that every transition in the sequence of the form $\alpha \rightarrow \delta \rightarrow \gamma \rightarrow \dots \rightarrow \psi \rightarrow \beta$ is neighbouring (related with changing the value of only one variable). In other words, $\alpha \xrightarrow{1} \beta$ holds, if, in accordance with the notations of Section 2.4, $\alpha \xrightarrow{1} \delta \xrightarrow{1} \gamma \xrightarrow{1} \dots \xrightarrow{1} \psi \xrightarrow{1} \beta$.

The concept of reachability can be generalized for sets of states of a circuit.

A set B of a circuit *immediately precedes* set A , if for each $\beta \in B$ there exists $\alpha \in A$ such that $\beta \rightarrow \alpha$ and for each $\alpha \in A$ there exists $\beta \in B$ such that $\beta \rightarrow \alpha$. In a similar way we can define a set that is *immediate successor* of A .

Therefore, the forward reachability problem can be stated in the following way. For a given set A of states of a circuit we should find all such β that for some $\alpha \in A$ there will be $\alpha \Rightarrow \beta$. The backward reachability problem involves the search of all β such that for $\alpha \in A$ we shall have $\beta \Rightarrow \alpha$.

Both reachability problems with respect to stepwise reachability are concerned with searching all β such that $\alpha \xrightarrow{1} \beta$ and $\beta \xrightarrow{1} \alpha$.

This section presents a discussion of algorithms for solving all reachability problems.

It is possible to suggest an iterative algorithm for reachability analysis. At every step of this algorithm we determine the immediate successors (forward reachability) or immediate predecessors (backward reachability) of a set of states which has been built in the previous iteration step (at the first iteration step, for the set of initial states). The process of building new sets of states stops when no new state can be found at the next iteration step. Thus, reachability analysis is the search for immediate predecessors and successors of a given set of circuit states.

The reachability problems can, in principle, be solved by the direct use of transition diagrams, but being a tool for describing the operation of real circuits, such

diagrams require too much space to be effective for that purpose. For this reason there is an obvious need for methods using more compact circuit representation – the Muller model, i.e. the system of logical equations.

A circuit state α is assigned a constituent $\omega(\alpha)$ in the same way as it was in Section 3.1. Then the set of circuit states can be given by a *characteristic Boolean function* $\phi_A(Z)$ such that

$$\phi_A(Z) = \bigvee_{\alpha \in A} \omega(\alpha).$$

We introduce two operators, P and Q , which, for a set A of states of the circuit, define the *immediate predecessor* set B and the *immediate successor* set D , i.e. $P(\phi_A(Z)) = \phi_B(Z)$ and $Q(\phi_A(Z)) = \phi_D(Z)$. For a particular case, the stepwise reachability, the corresponding operators will be denoted by P^1 and Q^1 , respectively.

STATEMENT 8.1. $P(\phi_A(Z))$ is obtained from $\phi_A(Z)$ by substituting the normal form of $z_i \vee f_i(Z)$ for each appearance of z_i , and $\bar{z}_i \vee \bar{f}_i(Z)$ for each \bar{z}_i , $1 \leq i \leq n$.

Proof. The proof is limited to justifying the general idea of the statement. Let, in the set A' , $A' \subseteq A$ of states, the value of z_i is equal to 1. What conditions must satisfy z_i in states belonging to the immediate predecessor of A' ? It is obvious that either $z_i = 1$, i.e. the value of z_i does not change in the transition to a state in A' , or $f_i(Z) = 1$, i.e. if $z_i = 0$ then z_i is excited and changes its value in the above transition. The case $z_i = 0$ can be handled in a similar way. *Q.e.d.*

COROLLARY 8.1.

$$P^1(\phi_A(Z)) = \phi_A(Z) \vee \bigvee_{i=1}^n \phi_A(z_i = f_i(Z)) \quad (8.1)$$

$$P^1(\phi_A(Z)) = \phi_A(Z) \vee \bigvee_{i=1}^n \phi_A(z_i = \bar{z}_i) (z_i \oplus f_i(Z)). \quad (8.2)$$

In these formulae, $\phi_A(z_i = c)$ has the same meaning as in Definition 3.1, i.e. $\phi_A(z_i = c) = \phi_A(z_1, z_2, \dots, z_{i-1}, c, z_{i+1}, \dots, z_n)$. $f_i(Z)$ is the inherent function of the i -th element of the circuit. It can be easily seen that both (8.1) and (8.2) hold for any form of representation of $\phi_A(Z)$.

In (8.2), $\phi_A(z_i = \bar{z}_i)$ defines a set B of states that may differ from the states in A only by the value of z_i ; $(z_i \oplus f_i(Z))$ defines a set of states where z_i is excited, and hence, their conjunctions defines the states in B in which z_i is excited. Therefore, from these states we can reach the states in A by the immediate stepwise reachability relation.

An algebraic representation of the forward reachability operator would be rather complicated, and since it is unlikely to be used for analysis purposes, we restrict ourselves to obtaining formulae for the stepwise forward reachability operator, Q^1 . Note that the expression $\phi_A(Z) \cdot (z_i \oplus f_i(Z))$ defines a set G of states in A in which z_i is excited, while $\phi_A(z_i = \bar{z}_i) (\bar{z}_i \oplus f_i(z_i = \bar{z}_i))$ defines a set of states that are different from those in G by the value of z_i . It is easily seen that each state α in D is immediately stepwise reachable from states $\alpha' \in G \subseteq A$, and α' differs from α only in the i -th position. Hence

$$Q^1(\phi_A(Z)) = \phi_A(Z) \vee \bigvee_{i=1}^n \phi_A(z_i = \bar{z}_i) (\bar{z}_i \oplus f_i(z_i = \bar{z}_i)). \quad (8.3)$$

It can also be shown that

$$Q^1(\phi_A(Z)) = \phi_A(Z) \vee \bigvee_{i=1}^n \phi_A(z_i = \bar{f}_i(z_i = \bar{z}_i)). \quad (8.4)$$

EXAMPLE 8.1. Let us return to the system of equations of Example 2.8 which corresponds to the circuit of Fig. 4.2(d). The sets of states in which the variables of the circuit are excited are given by the following characteristic functions

$$z_1 \oplus f_1 = \bar{z}_1 z_2 z_3 \vee z_1 \bar{z}_2 \bar{z}_3,$$

$$z_2 \oplus f_2 = \bar{z}_1 \bar{z}_2 \vee z_1 z_2,$$

$$z_3 \oplus f_3 = \bar{z}_1 \bar{z}_3 \vee z_1 z_3.$$

We take, as the initial set of states, $A = \{101, 110\}$ for which $\phi_A = z_1 \bar{z}_2 z_3 \vee z_1 z_2 \bar{z}_3$. The values of forward and backward reachabilities are derived as follows

$$\begin{aligned} P^1(\phi_A(Z)) &= z_1 \bar{z}_2 z_3 \vee z_1 z_2 \bar{z}_3 \vee (\bar{z}_1 z_2 z_3 \vee z_1 \bar{z}_2 \bar{z}_3) (\bar{z}_1 \bar{z}_2 z_3 \vee \bar{z}_1 z_2 z_3) \vee \\ &\quad \vee (\bar{z}_1 \bar{z}_2 \vee z_1 z_2) (z_1 z_2 z_3 \vee z_1 \bar{z}_2 \bar{z}_3) \vee (\bar{z}_1 \bar{z}_3 \vee z_1 z_3) (z_1 \bar{z}_2 \bar{z}_3 \vee z_1 z_2 z_3) \\ &= z_1 \bar{z}_2 z_3 \vee z_1 z_2 \bar{z}_3 \vee z_1 z_2 z_3 = z_1 z_2 \vee z_1 z_3. \end{aligned}$$

$$\begin{aligned}
Q^1(\phi_A(Z)) &= z_1 \bar{z}_2 z_3 \vee z_1 z_2 \bar{z}_3 \vee (z_1 z_2 z_3 \vee \bar{z}_1 z_2 \bar{z}_3) (\bar{z}_1 \bar{z}_2 z_3 \vee \bar{z}_1 z_2 \bar{z}_3) \vee \\
&\vee (\bar{z}_1 z_2 \vee z_1 \bar{z}_2) (z_1 z_2 z_3 \vee z_1 \bar{z}_2 \bar{z}_3) \vee (\bar{z}_1 z_3 \vee z_1 \bar{z}_3) (z_1 \bar{z}_2 \bar{z}_3 \vee z_1 z_2 z_3) \\
&= z_1 \bar{z}_2 \vee z_1 \bar{z}_3.
\end{aligned}$$

It is worthwhile replacing the general reachability problem with its special case, the stepwise reachability, because of the computational simplicity of the operators P^1 and Q^1 . The evaluation of the complexity of reachability algorithms, in both the general and the particular cases, is outside the scope of this book. We should, however, point out that in many real circuit analysis examples, the stepwise reachability algorithms are more efficient than the algorithms for the general reachability problem.

Firstly, we should consider the conditions under which solutions for the general and special cases coincide.

We reformulate Definition 2.21.

DEFINITION 8.1. A state α is called *conflict in z_i* , if there exists a state β immediately reachable from α ($\alpha \rightarrow \beta$) such that $\alpha_i = \beta_i = f_i(\beta) \neq f_i(\alpha)$, i.e. z_i is excited in α and is idle in β . We call α a *1-conflict state* if β is such that $\alpha \xrightarrow{1} \beta$.

THEOREM 8.1. *If all successors of a state α are non-conflict, then the set of states reachable from α is identical to the set of states that are stepwise reachable from α .*

Proof. Let, for some $\beta, \alpha \rightarrow \beta$ but not $\alpha \xrightarrow{1} \beta$. There can be found a shortest sequence $\alpha^0 \alpha^1 \dots \alpha^k \alpha^{k+1}$ where $\alpha^0 = \alpha$ and $\alpha^{k+1} = \beta$. We substitute, wherever this is possible, a sequence $\alpha^i \xrightarrow{1} \beta^i \xrightarrow{1} \dots \xrightarrow{1} \beta^r \xrightarrow{1} \alpha^{i+1}$ for each pair $\alpha^i \rightarrow \alpha^{i+1}$, and finally will obtain the maximally neighbouring sequence $\beta^0 \beta^1 \dots \beta^m \beta^{m+1}$ where $\beta^0 = \alpha$ and $\beta^{m+1} = \beta$. Assume that β^i is found such that $\beta^i \xrightarrow{1} \beta^{i+1}$ is not true. Let there be a $\gamma \in (\beta^i, \beta^{i+1})$ which is neighbouring to β^{i+1} . Then $\beta^i \rightarrow \gamma$, but $\gamma \rightarrow \beta^{i+1}$ is not true, because if it is, then due to $\gamma \xrightarrow{1} \beta^{i+1}$, the constructed sequence will no longer be maximally neighbouring. Hence β^i is conflict which contradicts the condition of the theorem. *Q.e.d.*

It follows from the above theorem that if all predecessors of a state α are non-conflict, then the set of states, predecessors of α , is identical to the set of stepwise predecessors of α .

EXAMPLE 8.2. In the transition diagram shown in Fig. 8.1 the 000 state is conflict, and hence, the set of states reachable from 111 does not coincide with those which are stepwise reachable from 111. The latter set does not contain state 110.

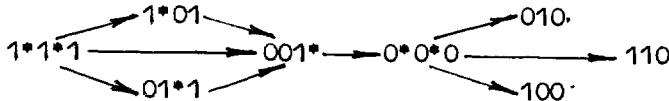


Figure 8.1. A state-transition diagram with conflict state.

The problem of forward reachability can be reduced to the problem of forward stepwise reachability with the aid of the following circuit transformation.

DEFINITION 8.2. The *shift of a circuit C* with respect to the state set A will be a circuit C^A defined by the system of Boolean equations:

$$\begin{aligned} z_i = f_i^A(Z) = \bar{z}_i(f_i(z_i = 0) \vee Q^1(\phi_A(Z) f_i(z_i = 0))) \vee \\ \vee z_i(\overline{\bar{f}_i(z_i = 1) \vee Q^1(\phi_A(Z) \bar{f}_i(z_i = 1))}). \end{aligned}$$

The idea of such a transformation consists in that all 1-conflict states in A in the original circuit become non-1-conflict states in the new circuit. This happens because some variable excitations shift along the transition diagram, and this “opens access” through the stepwise manner to those states which were not stepwise reachable in the original circuit.

The main properties of the circuit obtained by the shift transformation are formulated as follows.

- 1) The state-transition diagram of C^A contains all variable excitations that the diagram of the original circuit contains.
- 2) All the states in the set A, with respect to which the shift of C is performed, will not be 1-conflict in C^A .
- 3) In the state-transition diagram of C^A there may appear some new excited variables as compared to that of C, and this will alter the inherent functions of only those variables for which there has been a violation of the semi-modularity in the original circuit.

In a more exact form, this is expressed by the following theorem.

THEOREM 8.2. For the shift C^A of a circuit C the following statements hold:

- 1) if $\alpha_i \neq f_i(\alpha)$, then $\alpha_i \neq f_i^A(\alpha)$;

2) all states in A are non-1-conflict in C^A ;

3) if $f_i(\beta) \neq f_i^A(\beta)$, then there exists an $\alpha \in A$ such that $\alpha \xrightarrow{1} \beta$ and α is 1-conflict in z_i in C .

Proof. Point 1) is proved by direct substitution.

Let $\alpha \in A$ be 1-conflict in z_i , i.e. there exists β such that $\alpha \xrightarrow{1} \beta$ and $\alpha_i = \beta_i = f_i(\beta) \neq f_i(\alpha)$. Assume that $\alpha_i = 0$. Then α belongs to the set defined by the function $\phi_A(Z) f_i(z_i = 0)$, and β to the set defined by $\phi^1(\phi_A(Z) f_i(z_i = 0))$. Due to this, $\beta_i \neq f_i^A(\beta) = 1$. The case $\alpha_i = 1$ can be handled in a similar way. Hence, point 2) is also true.

Point 3) is fulfilled because the functions $f_i(Z)$ and $f_i^A(Z)$ are different only in those states α for which there can be found a 1-conflict state β such that $\beta \xrightarrow{1} \alpha$. *Q.e.d.*

From this theorem, it immediately follows that, if in the circuit C , B is the set of all 1-conflict states in A ($B \subseteq A$), then $C^A = C^B$.

The possibility of reducing the forward reachability problem to the forward stepwise reachability problem is explored in the following theorem.

THEOREM 8.3. *Let B be a set of states reachable from the states of a set A in a circuit C . Then the set of states reachable from the states of A , through the stepwise reachability in the shift C^B of C will be identical to B .*

Proof. Let $\alpha^1 \in A$ and $\alpha^N \in B$. We demonstrate that $\alpha^1 \xrightarrow{1} \alpha^N$ is true in C^B . If $\alpha^1 \alpha^2 \dots \alpha^N$ is a maximally neighbouring sequence in C and, for some α^j , $\alpha^j \xrightarrow{1} \alpha^{j+1}$ is not true in C , then due to Theorems 8.1 and 8.4 (the latter will be presented in Section 8.2) α^j is a 1-conflict state. On the other hand, according to Theorem 8.2, α^j is non-1-conflict in C^B and hence $\alpha^j \xrightarrow{1} \alpha^{j+1}$ will be true in circuit C^B .

Let $\alpha^1 \in A$, and $\alpha^1 \xrightarrow{1} \alpha^N$ hold in C^B . We show that $\alpha^N \in B$. If the sequence of states $\alpha^1 \alpha^2 \dots \alpha^N$ is such that $\alpha^k \xrightarrow{1} \alpha^{k+1}$ holds in C^B ($1 \leq k \leq N-1$) and a state α^j ($1 \leq j \leq N$) is the last of such states for which $\alpha^j \xrightarrow{1} \alpha^{j+1}$ is not true in C , then due to Theorem 8.2, there can be found a 1-conflict in z_l state γ , $\gamma \in B$, for which $\gamma \xrightarrow[1]{z_m} \alpha^j$ will be true in C . Since α^{j+1} differs from γ in the values of variables z_l and z_m , and both of these variables are excited in γ in C , then $\gamma \rightarrow \alpha^{j+1}$. As a consequence, $\alpha^{j+1} \in B$, then because $\alpha^{j+1} \xrightarrow{1} \alpha^N$ in C , we have $\alpha^N \in B$. *Q.e.d.*

The direct use of this theorem is rather difficult: the set of states, to which it is necessary to make the shift of an original circuit, is *a priori* unknown. We, therefore, solve the reachability problem in the following step-by-step, manner.

In the first step, the set of states D is defined, those states which are stepwise reachable from the states of a given set A in the original circuit C . Then the shift C^D of C is built and the set E of states that are stepwise reachable from A in the C^D circuit is defined. If $E = D$ (D does not contain conflict states), then D is the desired set of states, otherwise the procedure is repeated with C^D as an original circuit. By virtue of Theorem 8.2, the procedure should converge.

EXAMPLE 8.3. The circuit whose diagram is shown in Fig. 8.2(a) is defined on the set of variables $\{z_1, z_2, z_3\}$ by the system

$$z_1 = (\bar{z}_2 \vee \bar{z}_3)z_1 \vee \bar{z}_2 \bar{z}_3,$$

$$z_2 = (\bar{z}_1 \vee \bar{z}_3)z_2 \vee \bar{z}_1 \bar{z}_3,$$

$$z_3 = (z_1 \vee z_2)z_3 \vee z_1 z_2.$$

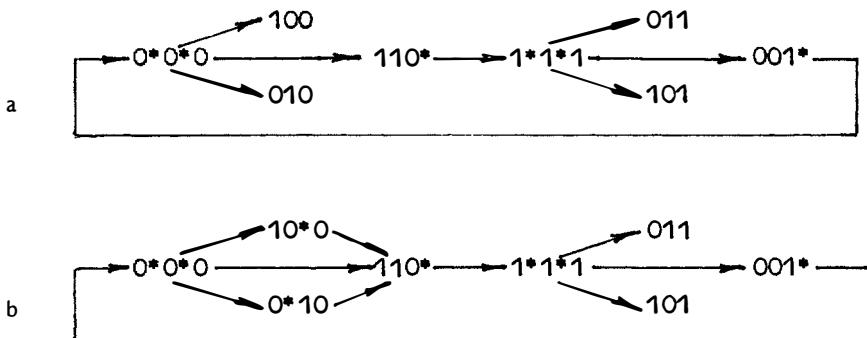


Figure 8.2(a) and (b). (a) Original state-transition diagram and, (b) its shift.

Let the initial state set be given by the function $\bar{z}_1 \bar{z}_2 \bar{z}_3$. During the iterative process of solving the forward reachability problem we can obtain: $Q(\bar{z}_1 \bar{z}_2 \bar{z}_3) = z_3$, $Q(z_1) = z_3 \vee z_1 z_2$, $Q(z_3 \vee z_1 z_2) = 1$. The solution process for the stepwise forward reachability problem halts at the second step: $Q^1 = (\bar{z}_1 \bar{z}_2 \bar{z}_3) = \bar{z}_1 \bar{z}_3 \vee \bar{z}_2 \bar{z}_3$, $Q^1(\bar{z}_1 \bar{z}_3 \vee \bar{z}_2 \bar{z}_3) = \bar{z}_1 \bar{z}_3 \vee \bar{z}_2 \bar{z}_3$. The difference between the above solutions is due to the presence of a 1-conflict state, the 000 state, in the state set computed.

The shift of the original circuit with respect to the set defined by the expression $\bar{z}_1\bar{z}_3 \vee \bar{z}_2\bar{z}_3$ is illustrated in Fig. 8.2(b) and represented by the system

$$\begin{aligned} z_1 &= \bar{z}_3 \vee z_1 \bar{z}_2, \\ z_2 &= \bar{z}_3 \vee z_2 \bar{z}_1, \\ z_3 &= (z_2 \vee z_1)z_3 \vee z_1 z_2. \end{aligned}$$

Continuing the solution process for the stepwise forward reachability problem, we obtain, for that circuit, the following: $Q^1(\bar{z}_1\bar{z}_3 \vee \bar{z}_2\bar{z}_3) = \bar{z}_3$, $Q^1(\bar{z}_3) = \bar{z}_3 \vee z_1 z_2$, $Q^1(\bar{z}_3 \vee z_1 z_2) = \bar{z}_3 \vee z_1 \vee z_2$, $Q^1(\bar{z}_3 \vee z_1 \vee z_2) = \bar{z}_3 \vee z_1 \vee z_2$. This result also differs from the solution of the general reachability problem for the original circuit, because the set obtained contains a 1-conflict state, the 111 state. The shift of the circuit with respect to the set defined by $\bar{z}_3 \vee z_1 \vee z_2$ gives the circuit shown in Fig. 2.11(a) and defined by the system of equations from Example 2.8. For the latter circuit, $Q^1(\bar{z}_3 \vee z_1 \vee z_2) = 1$, which coincides with the solution of the general reachability problem for the original circuit.

8.2 The classification analysis

A vital role in the general area of circuit analysis is given to the problems of the circuit membership within the class of speed-independent circuits in which we distinguish the classes of semi-modular, distributive, parallel-sequential and totally sequential circuits. This section presents a discussion of methods for such *classification analysis*. First, we should clarify some of the notations introduced in the previous sections.

DEFINITION 8.3. A set of all such states that for any pair of states α, β belonging to this set we have both $\alpha \Rightarrow \beta$ and $\beta \Rightarrow \alpha$ will be called an *equivalence class*.

An equivalence class is called:

- *fake*, if in all its states a variable z_i has the same value and is excited;
- *closed*, if an immediate predecessor of the class coincides with the class itself, i.e. if a state α belongs to the class and $\alpha \Rightarrow \beta$, then β also belongs to this class.

A trivial example of a closed class is the *deadlock* state, i.e. the state in which all variables are idle.

DEFINITION 8.4. A circuit is called *speed-independent* with respect to a state α if, and only if, the set of states reachable from α contains one and only one non-fake equivalence class.

It follows from this definition that the set of operational states of the speed-independent circuit (i.e. states reachable from a certain initial state) contains exactly one closed class. In particular, this set may contain at most one deadlock state.

DEFINITION 8.5. A circuit is *semi-modular* with respect to a state α if every state β reachable from α in this circuit is non-conflict in all variables.

Proving circuit membership in the semi-modularity class requires the search for conflict states. The following theorem is aimed at the simplification of such a search.

THEOREM 8.4. *For any state α conflict in z_i , there can be found a state β , 1-conflict in z_i , such that $\alpha \rightarrow \beta$ and $\alpha \xrightarrow{1} \beta$.*

Proof. Let α be conflict in z_i and $\{\alpha^1, \dots, \alpha^l\}$ be a set of all states such that $\alpha \rightarrow \alpha^j$ and $\alpha_i = \alpha_i^j = f_i(\alpha^j) \neq f_i(\alpha)$. Assume that α^j differs from α in the values of k_j variables; $k = \min_{1 \leq j \leq l} k_j$ is the rank of conflict of state α in z_i . Apply mathematical induction to the rank of conflict. For $k = 1$ the proof is obvious. Assume the theorem is true for $k = m$. Let us now have $k = m + 1$, i.e. for some j state α differs from α^j in the values of $m + 1$ variables. Let a state γ be adjacent (neighbouring) to α^j and $\gamma \in (\alpha, \alpha^j)$. Then $\alpha \rightarrow \gamma$ and $\gamma \rightarrow \alpha^j$, otherwise the conflict rank of α would be less than $m + 1$. Hence, state γ is 1-conflict in z_i and satisfies the first condition of the theorem. The other condition also holds — $\alpha \xrightarrow{1} \gamma$, because otherwise there can be found a state $\sigma \in (\alpha, \gamma)$ such that $\alpha \rightarrow \sigma$, $\alpha_i = \sigma_i = f_i(\sigma) \neq f_i(\alpha)$, and this will make the rank of conflict of α in z_i be less than $m + 1$. *Q.e.d.*

Thus, a circuit is semi-modular with respect to state α if, and only if, every state reachable from α is non-1-conflict.

Furthermore, from Theorems 8.1. and 8.4 it follows that *if a circuit is semi-modular with respect to state α then the set of states reachable from α in this circuit will be the same as the set of states that can be stepwise reached from α .*

We make a concise clarification of why the behaviour of semi-modular circuits is invariant to the delays of circuit elements.

In the general case, a certain excited variable can become idle by two possible routes: 1) by changing its values, or 2) as a result of the change of values of other variables, on which it depends. In a semi-modular circuit, we allow only the first of these ways. As a consequence, in a semi-modular circuit the switching of one, or several, excited variables cannot “disturb” the switching of the other excited variables.

In order to find the state set with respect to which a circuit is not semi-modular, the *set of states which are conflict* (1-conflict) in some variables is first defined, and then the backward reachability problem is solved for this set. For this purpose, because of Theorem 8.1, it is sufficient to solve the stepwise version of this problem.

STATEMENT 8.2. *The sets of conflict and 1-conflict states in variable z_i are given, respectively, by the functions*

$$z_i \bar{f}_i(Z) P(z_i f_i(Z)) \vee \bar{z}_i f_i(Z) P(\bar{z}_i \bar{f}_i(Z))$$

and

$$z_i \bar{f}_i(Z) P^1(z_i f_i(Z)) \vee \bar{z}_i f_i(Z) P^1(\bar{z}_i \bar{f}_i(Z)).$$

The validity of these formulae follows from the fact that they define the intersections between the sets of states, $z_i \bar{f}_i(Z)$ and $\bar{z}_i f_i(Z)$, in which z_i is excited, and the sets of states, $P(z_i f_i(Z))$ and $P(\bar{z}_i \bar{f}_i(Z))$ ($P^1(z_i f_i(Z))$ and $P^1(\bar{z}_i \bar{f}_i(Z))$), from which, without any change in z_i , there can be immediately reached (stepwise reached) the states where z_i is idle.

The speed-independence property is “global” in the sense that it is impossible to indicate those concrete states in which this property is violated (it is perhaps the major reason why it is so difficult to develop any regular methods for the synthesis of circuits within this class, rather than those in its narrower sub-classes). By contrast, the semi-modularity can be called a “local” property. It is exactly the “localization” of the semi-modularity violation that makes the circuit semi-modularity analysis more effective. Similarly, distributive and totally sequential circuits, as well as some other sub-classes of speed-independent circuits, have characteristic “local” properties.

To analyse circuit memberships within the semi-modularity class, we need several more definitions.

DEFINITION 8.6. A state α is called:

- *detonant* in z_i , if there exists a pair of states β and γ that are immediately reachable from α ($\alpha \xrightarrow{*} \beta$, $\alpha \xrightarrow{*} \gamma$) and such that z_i is idle in α and excited in both β and γ ($\alpha_i = \beta_i = \gamma_i = f_i(\alpha)$ and $f(\beta) = f_i(\gamma) \neq \alpha_i$);
- *bifurcant*, if it has more than one excited variable;
- *hammock*, if in every β , which is the immediate successor of α , except perhaps for just one that differs from α in the values of all variables excited in α , only those variables are allowed to be excited which are excited in α and have not changed their values ($\alpha_i = \beta_i \neq f_i(\alpha)$).

EXAMPLE 8.4. In the state-transition diagram shown in Fig. 8.3(a), the 000 state is detonant, non-hammock and non-conflict. The other states are neither detonant nor hammock. The bifurcant states are 100 and 010. Non-bifurcant states are 101, 110, 011 and 111. In Fig. 8.3(b), the 000 state is neither hammock nor detonant.

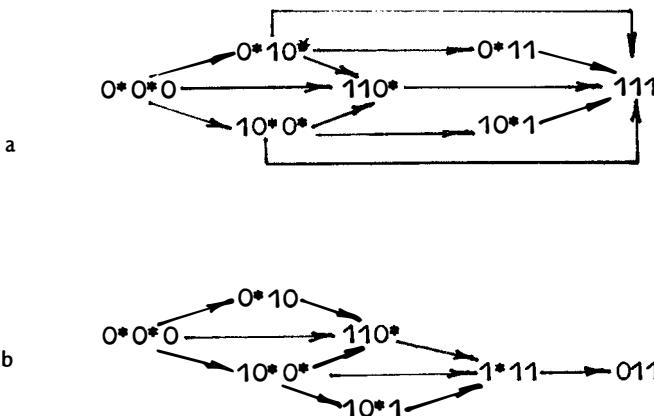


Figure 8.3(a) and (b). Illustration of bifurcant, detonant and hammock states: examples of transition diagrams.

STATEMENT 8.3. The sets of detonant, bifurcant and non-hammock states are defined by the following formulae

$$(\bar{z}_i \oplus f_i) \bigvee_{j=1, i \neq j}^n P^1((z_i \oplus f_i) z_j) P^1((z_i \oplus f_i) \bar{z}_j), \quad (8.5)$$

$$\bigvee_{1 \leq i, j \leq n, i \neq j} (z_i \oplus f_i) (z_j \oplus f_j), \quad (8.6)$$

and

$$\bigvee_{1 \leq i, j \leq n} (\bar{z}_i \oplus f_i) (z_j \oplus f_j) P((z_i \oplus f_i)(z_{ij} \oplus f_j)). \quad (8.7)$$

respectively.

The validity of these formulae comes from the following. (8.5) defines the intersection between the sets of states in which z_i is idle, $(\bar{z}_i \oplus f_i)$, and the sets of states given by $P^1((z_i \oplus f_i)z_j)$ and $P^1((z_i \oplus f_i) \bar{z}_j)$, from which two different states with z_i excited are immediately stepwise reachable. (8.6) defines the states with at least two excited variables. (8.7) defines the intersection between the set of states in which z_i is idle while z_j is excited, and the set of states that are immediate predecessors of the states in which z_j has not changed while z_i has already become excited.

Now we can define the following special sub-classes of semi-modular circuits.

DEFINITION 8.9. 1) A circuit is called *distributive* with respect to state α , if every state β reachable from α in this circuit is neither conflict nor detonant in each of its variables.⁽¹⁾

2) A circuit is called *parallel-sequential* with respect to state α , if every state β reachable from α in this circuit is neither conflict nor hammock in each of its variables.

3) A circuit is called *totally sequential* with respect to state α , if every state β reachable from α is not bifurcant.⁽¹⁾

It is characteristic for distributive circuits that their local prehistory is unique in the sense that every excitation of a variable can be caused in exactly one way – by the changes of values of all variables from a single, for this variable, minimum set of variables.

In parallel-sequential circuits, every variable can be excited only after all earlier variables have become idle. This class is a sub-class of distributive circuits and forms an extension of the class of totally sequential circuits.

The search for the states for which a given circuit is not distributive (not parallel-sequential, not totally sequential) is organized in the same way as the search for states violating the semi-modularity of the circuit.

EXAMPLE 8.5. The circuit presented in Fig. 8.4 is defined by the system of equations

⁽¹⁾ It can be shown that a circuit is distributive with respect to state α if, and only if, its cumulative diagram with respect to α forms a distributive lattice.

$$\begin{aligned} z_1 &= \bar{z}_3, \\ z_2 &= \bar{z}_1, \\ z_3 &= \bar{z}_2 \end{aligned}$$

and is totally sequential with respect to all its states, except for 000 and 111. With respect to the latter states, the circuit is not even semi-modular.

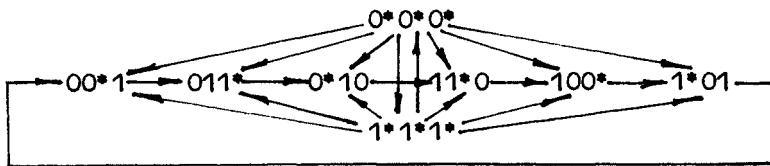


Figure 8.4. State-transition diagram for three inverters interconnected into a loop.

The circuit of Fig. 2.11(a) is parallel sequential with respect to all states, while the circuit of Fig. 8.3(b) is distributive, but not parallel-sequential with respect to 000. The circuit of Fig. 8.3(a) defined by the equation system

$$\begin{aligned} z_1 &= 1, \\ z_2 &= 1, \\ z_3 &= z_1 \vee z_2 \end{aligned}$$

is not distributive, but semi-modular, with respect to state 000.

Here, we introduce a concept of *complete analysis* by which we mean the procedure of splitting the set of states of a circuit into two sub-sets, called *forbidden* and *allowed* states, or regions. The forbidden region may – depending on which classification property is under study – contain the states with respect to which a circuit is non-semi-modular, non-distributive etc. Thus the complete analysis makes it possible to investigate the circuit's behaviour for all sets of its states.

EXAMPLE 8.6. Consider the procedure of the complete analysis of the circuit defined by the following system of equations

$$\begin{aligned} z_1 &= \bar{z}_5, \quad z_2 = \bar{z}_5, \quad z_3 = z_1 z_2 \vee z_3(z_1 \vee z_2), \\ z_4 &= z_2, \quad z_5 = z_3 z_4 \vee z_5(z_3 \vee z_4). \end{aligned} \tag{8.8}$$

The full transition diagram containing all 32 states is not presented here since it would not be useful in illustrating the principles.

As a result of analysis, the following facts have been obtained.

The set of 1-conflict states of the circuit is expressed by the characteristic function

$$z_2\bar{z}_4z_5 \vee \bar{z}_2z_4z_5 \vee z_1\bar{z}_3z_5(z_2 \vee \bar{z}_4) \vee \bar{z}_1z_3\bar{z}_5(z_2 \vee z_4)$$

and contains 12 states.

The set with respect to which the circuit is not semi-modular (the forbidden region) is described by the function

$$\bar{z}_5(\bar{z}_1z_3 \vee \bar{z}_2z_4) \vee z_5(z_2\bar{z}_4 \vee z_1\bar{z}_3)$$

and consists of 14 states.

With respect to the other states, the circuit is semi-modular. The region of allowedness is characterized by the function

$$z_3z_5(\bar{z}_2 \vee \bar{z}_4) \vee \bar{z}_3\bar{z}_5(z_2 \vee \bar{z}_4) \vee z_1\bar{z}_5(\bar{z}_4 \vee z_2z_3) \vee \bar{z}_1(z_4z_5 \vee \bar{z}_2\bar{z}_3\bar{z}_4)$$

and consists of 18 states.

It should be noted that, although the result of the complete analysis procedure gives full information about the circuit's behaviour, it is not always necessary, nor convenient to perform it.

8.3 The set of operational states

The total number of states that a circuit may enter during the process of its functioning is finite, 2^n (where n is the number of elements in the circuit). Hence, there are two alternatives: either 1) the circuit moves from state to state until it enters a deadlock, the state having no successors, or 2) starting at a certain state, a certain sequence of states is infinitely repeated by the circuit.

If some initial state is picked out in a circuit, then all states that are reachable from the *initial* one will be called *operational states*. The cardinality of the set of operational states is often considerably smaller than 2^n . The circuit analysis can be simplified by taking this point into consideration. Such a simplification is based on eliciting characteristic properties of the set of operational states.

The construction and analysis of the set of operational states for a circuit with initial state (initial state set) can be based on iterative methods for solving the problem of forward reachability. Furthermore, it will be worthwhile to alternate the reachability problem solving with proving the circuit membership with in a particular class of speed-independent circuits.

We now turn our attention towards those aspects of building the set of operational states which are related to the more efficient analysis procedures. Let W denote the set of operational states. To compute it, we may use the formulae (8.3) and (8.4) defining the forward stepwise reachability operator. As an initial set $A \subseteq W$ of states, we take the initial state (or the set of initial states) of the circuit. At the first step we build a set of immediate stepwise successors for A . This set is covered by the second terms in formulae (8.3) and (8.4) that have the form

$$R^1(\phi_A(Z)) = \bigvee_{i=1}^n \phi_A(z_i = \bar{z}_i) (\bar{z}_i \oplus f_i(z_i = \bar{z}_i)). \quad (8.9)$$

or

$$R^1(\phi_A(Z)) = \bigvee_{i=1}^n \phi_A(z_i = \bar{f}_i(z_i = \bar{z}_i)) \quad (8.10)$$

which may generally contain a certain sub-set of states in A . At the second step, using formulae (8.9) and (8.10), we compute all immediate stepwise successors for the states computed at the first step. It is obvious that continuing this computation we build a whole set W . The process terminates when there are no more new states produced. At a first glance, it seems that, for this, all earlier computed states should be stored at every computation step. In the following, we shall suggest an alternative organization for computing set W that will not require manipulation with each current value of W .

DEFINITION 8.8. The set of states stepwise reachable from α for k steps will be called the k -th layer (L_k) of the set W of operational states of a circuit with initial state α .

The zero layer $L_0 = \{\alpha\}$ is formed by the initial state.

From the definition it follows that the set of layers is ordered. Since the number of circuit states is finite, then starting at some step, the layers will be repeated.

The layer in W which totally coincides with one of the previously built layers will be called *multiple*.

It is clear that if L_m is a multiple layer, i.e. $L_m = L_k$ for $m > k$, then this is an indication for the termination of the process of building the set of operational states:

$$W = \bigcup_{i=0}^{m-1} L_i.$$

Generally speaking, not every layer obtained in the analysis process is multiple. A certain sub-set of operational states may be met only once in the circuit operation . For the layers which do not contain repeated states, there are no layers which cover them.

If there were any constructive considerations that could manifest that a certain k -th layer would be repeated in the sequel (we shall call such a layer a *reference layer*), then we would orient ourselves towards checking that a current layer coincides with the reference layer, and hence, would no longer need to store the current value of W , but store only the reference layer. One of such considerations can be the *width of a layer* (the number of states in it). As a reference layer, we may select the widest layer.

The following observation can be stated: *if the set of operational states presents a single equivalence class, then the widest layer is multiple*. If the state-transition diagram corresponding to W contains a “starting” section (the set of states that will no longer be met in the operation of the circuit, but from which the states belonging to the closed class are reachable), then, in general, the widest layer may belong to the “starting” section and, hence, not be multiple. In such a case, the above criterion does not guarantee the termination of the analysis process. However, in most circuits met in practice, these “starting” sections, if any, are not too wide because they are not normally concerned with any considerable parallel actions, as opposed to cyclic sections, of the transition diagrams.

EXAMPLE 8.7. The layers of the set of operational states for the circuit in Fig. 2.11(a) with the initial state 000 are expressed by the functions

$$L_0 = \bar{z}_1 \bar{z}_2 \bar{z}_3, \quad L_1 = \bar{z}_1 (z_2 \bar{z}_3 \vee z_2 z_3), \quad L_2 = \bar{z}_1 z_2 z_3,$$

$$L_3 = z_1 z_2 z_3, \quad L_4 = z_1 (z_2 \bar{z}_3 \vee \bar{z}_2 z_3), \quad L_5 = z_1 \bar{z}_2 \bar{z}_3,$$

$$L_6 = \bar{z}_1 \bar{z}_2 \bar{z}_3, \quad L_7 = \bar{z}_1 (z_2 \bar{z}_3 \vee \bar{z}_2 z_3).$$

As a reference layer, we may take the L_1 layer with the width of 2. It is multiple because $L_1 = L_7$. As we may be convinced, none of the layers contain conflict states, and $\phi_W(Z) = \bigvee_{i=0}^6 L_i = 1$. Thus, the circuit is semi-modular with respect to all its states.

The algorithm for building W , checking the termination condition on the widest layer, took seven steps in this case.

The minimum width layers are those consisting of a single state. Such layers are also attractive as to their possible use as reference ones. Let us consider the conditions when it is possible.

DEFINITION 8.9. 1) A sequence of states $\alpha^1 \alpha^2 \dots \alpha^k$ such that $\alpha^i \rightarrow \alpha^{i+1}$ is called *complete* if it is either finite and ends with a deadlock state, or it is infinite, and contains no infinite section where all states have some variable excited and having the same value.

2) Let W be a set of states reachable from the states of A , $A \subseteq W$. State β will be called *nodal* for A if all complete sequences beginning in the states of W contain β .

Let us comment on the term “complete sequence”. It is obvious that sequences in a state-transition diagram can either be finite (if they do not form a *cycle*) or infinite (if they do form a cycle). From those that are finite, we may consider to be complete, only those which cannot be extended, i.e. those which end with deadlocks. Among those sequences which are infinite (and, hence, contain cycles) the complete ones are those along which the circuit may “travel”, however long. But, due to the assumption of the finiteness of delays in elements. This property is not seen in those cycles where some variable is both excited and does not change its value in all states of the cycle, because, sooner or later, the excited variable should change its value.

From the above, we notice that the complete sequences are those which fully characterize a possible ordering of states in the circuit.

The concept of a nodal state can be interpreted in the following manner. In some aperiodic circuits the value of a signal at the output of the indicator of transient process completion (if such an indicator is present) normally changes last compared to other elements of the circuit, and hence, the output element of the indicator becomes excited only after all other elements have changed their values. Then the state in which the output element of the indicator is excited will be the nodal state.

The following theorem, whose proof will be omitted because of its length, demonstrates a property of nodal states.

THEOREM 8.5. *Let a circuit be semi-modular with respect to states in A . If α is the nodal state for A , then in every state β , reachable from the states in A , such that β does not belong to a fake equivalence class, $\beta \neq \alpha$ and $\beta \xrightarrow{*} \alpha$, exactly one variable is excited. Conversely, if a single non-fake equivalence class B is reachable from the states of set A , and an $\alpha \in B$ is such that in all of its immediate stepwise predecessors β , $\beta \neq \alpha$, reachable from the states of A , there is exactly one excited variable, then α is the nodal state for set A .*

The following relationship can be established between nodal states and layers.

THEOREM 8.6. *Let a circuit be semi-modular with respect to state α and no fake classes are reachable from α . If some layer contains a state nodal for set $\{\alpha\}$, then it*

does not contain any other states. Conversely, if a multiple layer contains a single state, then this state is nodal for the initial set of states.

Proof. 1. Assume to the contrary that layer L_j contains at least two states, one of which is nodal. Then, in view of the fact that each immediate stepwise predecessor of the nodal state that is reachable from the initial state is non-bifurcant (see Theorem 8.5), we can easily show, by induction, that each layer L_i , $0 \leq i \leq j-1$, defines more than one state. Hence, we arrive at a contradiction.

2. The converse statement can be easily seen if we bear in mind that between L_k and L_m there is contained the entire closed class of states reachable from the initial state. *Q.e.d.*

Two more interesting observations can be made about nodal states that characterize speed-independent circuits.

- 1) If state α is nodal for set A , then α belongs to some closed class B , and if A forms itself a non-fake class, then $A = B$.
- 2) If in a circuit there exists state α nodal for set A , then this circuit is speed-independent with respect to all states in A .

According to Theorem 8.6 the process of constructing the set of operational states of a circuit can be considered as terminated when the same nodal state is obtained twice.

Unfortunately, not every circuit, semi-modular with respect to the initial state, has a nodal state; see, for example, Fig. 8.5.

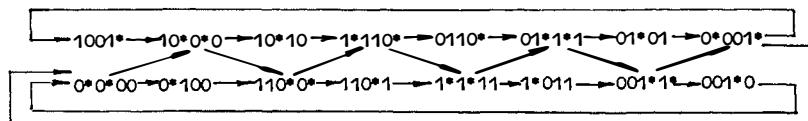


Figure 8.5. State-transition diagram for a circuit without nodal states.

The pipeline circuits also have no nodal states in their diagrams.

However, some sub-classes of semi-modular circuits have their operational state sets with nodal states. These are, for example, parallel-sequential circuits.

EXAMPLE 8.8 (continued from Example 8.7). The circuit of Fig. 2.11(a) is parallel-sequential with respect to all its states. The nodal states are 000, 011, 111, 100 that are given, respectively, by the functions L_0, L_2, L_3, L_5 . If we check the

termination of the process of building the set of operational states on nodal states, then we shall require only 6 steps rather than 7 as in Example 8.7.

There are also the circuits which are by no means parallel-sequential, but have, nevertheless, operational sets with nodal states.

For example, a circuit corresponding to the equation system (8.8) is presented in Fig. 8.6 and is not parallel-sequential with respect to, say, the 00000 state, but if we take that state as initial, then we shall obtain the W set with four nodal states 00000, 11110, 11111 and 00001.

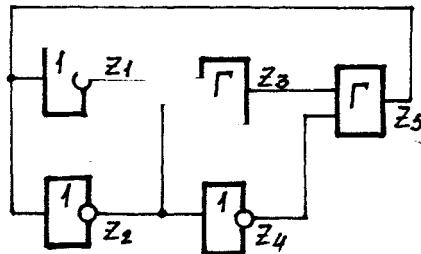


Figure 8.6. A circuit with nodal states which is not parallel-sequential.

EXAMPLE 8.9. (This illustrates the “interaction” between analysis and synthesis in the process of designing aperiodic circuits.)

Earlier, in Fig. 5.3, we saw a circuit for a shifter built as the interconnection of three so-called David cells, (each consisting of an RS-flip-flop and a NAND gate). This circuit, as any other circuit of the type, with a larger number of cells, is correct with respect to the initial states in which only one element is excited. The question arises as to whether such circuits operate correctly with respect to the initial states in which two elements are excited at a time, i.e. the question of whether such circuits can, or cannot, be used in a pipeline mode. Analyzing the set of operation states has shown that, in such conditions, even the register consisting of four David cells does not operate well. It goes rapidly to the deadlock state, which is unacceptable for autonomous circuits. A similar circuit with six David cells does not go to deadlock for initial states with a pair of excited variables, but its operational state set contains conflict states, and hence, the circuit is not semi-modular. To obtain the correct circuit shown in Fig. 8.7, we introduced some additional inputs to the elements and their corresponding interconnections.

In the above analysis methods, we did not investigate each complete sequence starting at α , separately, but instead organized sequential movement along all paths of the transition diagram together. This technique presents some kind of “parallel search” in branches in the transition diagram which speeds up the circuit analysis process as a whole.

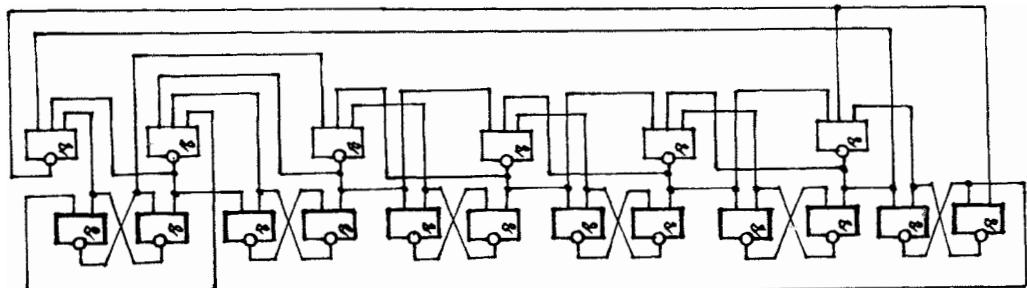


Figure 8.7. Pipeline analogue for a signal distribution circuit built of David's elements.

8.4 The effect of non-zero wire delays

Earlier, problems of asynchronous logic analysis have been tackled within the assumptions that delays in interconnecting wires are negligibly small as compared to the element delays. This assumption is no longer valid for circuits in which the speed of the logical elements is so great that the magnitude of element delays approaches that of wire delays. This is the case, for example, for circuits based on the ECL-technology or on the Josephson junction technology.

Furthermore, there exists a wide range of interface devices incorporating rather long interconnections, say, communication lines, whose delays may considerably exceed the delays in the decision elements. The problem of compensating skew values is characteristic for such devices. With a constant growth in integration scale, wire delays have an increasing effect on the behaviour of circuits, and in VLSI circuits, can be comparable with the delays in logical elements.

All these points are in favour of analyzing the operation of asynchronous logical circuits in the presence of wire delays.

Whenever we need to analyze circuits with interconnections having long delays, we may introduce special elements, repeaters, that will substitute for such wires, and thus all accepted assumptions will remain valid. As a consequence, we may use the above analysis methods for cases having significant delays in certain wires.

EXAMPLE 8.10. Consider the circuit shown in Fig. 8.8. The analysis of the set of operational states has established that the circuit is semi-modular, for example, with respect to the 0111100 state, if the delays in the wires are insignificant. If we need to take such delays into account, we shall obtain the following results. Those delays which do not affect the behaviour of the circuit are marked with asterisks in Fig. 8.8.

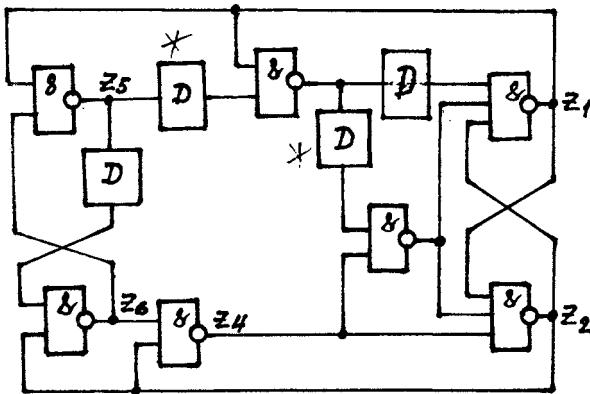


Figure 8.8. Example of circuit semi-modularity violation due to wire delays.

Those delays which are not marked are crucial for the operation. Being significant, they violate the circuit's semi-modularity. The remaining wires are similar to those marked with asterisks, and hence, we do not need to analyze the circuit bringing in their delays.

It should be noted that introducing delays into all (or many) interconnections in a circuit sharply increases the dimensions of the analysis problem, and can make it difficult to solve. Therefore, it would be interesting to investigate how the behaviour of the circuit is affected by the wire delays, and to develop on this basis, corresponding analysis techniques, without explicitly introducing additional variables into the circuit. Such techniques may, in some cases, be effective in the analysis of circuits with significant wire delays.

First, we should formalize the notion of a wire (interconnection) between circuit elements defined by

$$z_i = f_i(Z), \quad 1 \leq i \leq n.$$

DEFINITION 8.10. A *circuit wire* is a pair of positive integers (i, j) such that $f_j(Z)$ essentially depends on z_i , and z_i and z_j are the input and output variables of the (i, j) wire, respectively.

Let $L = \{(i_1, j_1), \dots, (i_m, j_m)\}$ be a set of wires in circuit C . $I(L)$ and $O(L)$ denote the sets of input and output variables of wires in L . If $z_j \in O(L)$, then $I(j, L)$ will denote the set of input variables for those wires in L which have z_j as an output variable.

To investigate the effect of non-zero wire delays on the circuit's behaviour we shall compare the behaviour of the initial circuit $C = \langle Z, F \rangle$ with that of the circuit C^L obtained from the initial circuit by inserting the elements (repeaters) into all significant wires. This new circuit $C^L = \langle Z^L, F^L \rangle$ will be called the *expansion of circuit C with respect to the set of wires L*.

The construction of circuit C^L is concerned, firstly, with introducing a set of additional variables associated with wires in L and denoted as $l_{i1}^{j1}, l_{i2}^{j2}, \dots, l_{im}^{jm}$, and, secondly, with the appropriate transformation of the system of Boolean equations. For the additional variable l_p^r we introduce the equation $l_p^r = z_p$ while the original equations are transformed in the following way. If $z_r \notin O(L)$, then the equation for z_r is not changed, but if $z_r \in O(L)$, then the function $f_r(Z)$ for each variable $z_p \in I(r, L)$ is replaced by a new variable l_p^r .

EXAMPLE 8.11. Let the circuit of Fig. 8.9 be given by the following system of equations

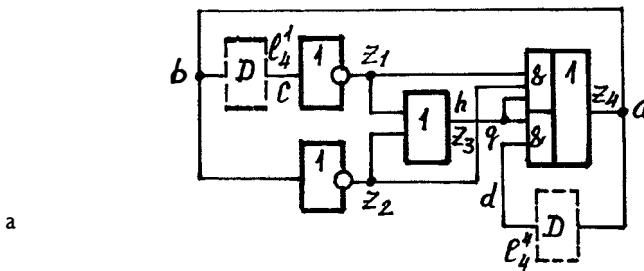
$$z_1 = \bar{z}_4, \quad z_2 = \bar{z}_4, \quad z_3 = z_1 \vee z_2, \quad z_4 = z_3(z_1 z_2 \vee z_4). \quad (8.11)$$

The set $\{(4, 1), (4, 4)\}$ is the set of significant wires. We introduce two additional variables l_4^1 and l_4^4 and transform the equation system to

$$z_1 = \bar{l}_4^1, \quad z_2 = \bar{z}_4, \quad z_3 = z_1 \vee z_2, \quad z_4 = z_4 z_1 z_2 \vee z_3 l_4^4,$$

$$l_4^1 = z_4, \quad l_4^4 = z_4.$$

The output element of z_4 is connected to other elements of the circuit by three wires $(4, 1), (4, 2), (4, 4)$, and according to the topology of Fig. 8.9(a) we have two branching nodes, a and b . It follows from the model introduced that the repeater l_4^4 is inserted into the break of section ad of wire $(4, 4)$, and l_4^1 into the break of bc of wire $(4, 1)$. There is no need to insert a repeater into the break of ab . If we were interested in wire $(3, 4)$, then we would have inserted element l_3^4 into the break hg because the delays of the sections of wire after the branching point that are connected with the inputs of the same element (z_4) are supposed to be negligibly small.



b

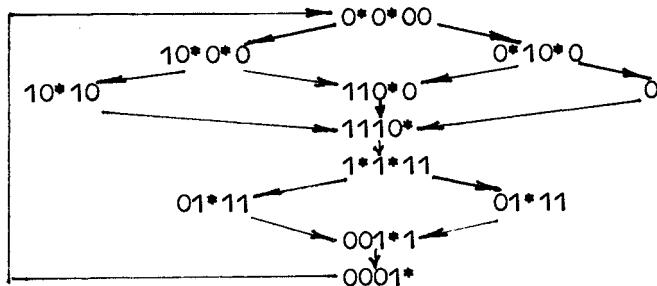


Figure 8.9(a) and (b). Illustration of a method of inserting delay elements into a circuit for the analysis of wire delay effects.

We introduce several auxiliary definitions.

DEFINITION 8.11. 1) The *projection of a state* $\alpha = \langle \alpha_1, \dots, \alpha_n \rangle$ of a circuit on a sub-set $Z' = \{z_{i1}, \dots, z_{ik}\} \subseteq Z$ is a vector (combination) of values of variables belonging to Z' in state α .

2) Let $\alpha^1 \alpha^2 \dots \alpha$ be a sequence of states of a circuit where $\alpha^i \rightarrow \alpha^{i+1}$, $\alpha^i \neq \alpha^{i+1}$. The *projection of this sequence on the sub-set* $Z' \subseteq Z$ of variables is obtained from the sequence of projections of its states on this sub-set by deleting each member identical to the preceding member of the sequence.

3) Two initialized circuits $C_1 = \langle Z_1, F_1, \alpha \rangle$ and $C_2 = \langle Z_2, F_2, \beta \rangle$ are called *equivalent* with respect to a set $Y \subseteq Z_1 \cap Z_2$ if the set of projections of the sequences

starting from α on Y in C_1 and the set of projections of the sequences starting from β on Y in C_2 are identical to each other. We denote that C_1 is equivalent to C_2 with respect to Y as $C_1 \underset{Y}{\sim} C_2$.

Assume that C^L is the expansion of an original circuit C with respect to a wire set L . A state α of C generally corresponds to a set of states of C^L whose projection on the set of variables of C is equal to α . Among these states we can find the state $\hat{\alpha}$ in which all variables corresponding to wires in L are excited, and the state $\hat{\alpha}$ in which they are idle. In the following, we shall be interested in the behaviour of the circuit expansion with respect only to initial states of the form $\hat{\alpha}$. Thus, if we are given an initialized circuit $C = \langle Z, F, \alpha \rangle$, then we should take, as its expansion with respect to the wire set L , the initialized circuit of the form $C^L = \langle Z^L, F^L, \hat{\alpha} \rangle$. For the sake of brevity, we shall say that the initialized circuit is semi-modular, delay-insensitive etc. if it is such, with respect to the initial state.

DEFINITION 8.12. A circuit C is called *insensitive to a set of wires* L if $C \underset{Z}{\sim} C^L$.

A semi-modular initialized circuit C is called *strictly insensitive to a wire set* L if its expansion C^L is semi-modular (with respect to the initial state $\hat{\alpha}$) for the set of variables Z .

Every non-semimodular circuit is sensitive to wires because non-semimodularity means dependence of the circuit operation on the delays of the elements.

The concepts introduced demonstrate different degrees of how the behaviour of the circuit is affected by wire delays. There is some inter-relation.

If a circuit C is strictly insensitive to wire set L , then it is insensitive to L . The converse is not always true because, generally, for circuits which are not strictly sensitive, even though they may be insensitive to L , the expansion C^L may be non-semimodular. However, if circuit C is insensitive to the set L of wires, the input for z_j ($O(L) = \{z_j\}$), and the z_j variable of C is non-self-dependent, then C is strictly insensitive to L .

Clearly, if a circuit is sensitive to L , and (i, j) is an arbitrary wire, then the circuit is sensitive to $L \cup \{(i, j)\}$. On the other hand, it can be shown that for proving a circuit to be insensitive to wire delays, we do not always need to check through all possible sets of wires in the circuit. Rather, it is enough to consider only those wire sets that have the same output variables.

To proceed with the analysis, we should use the concept of Boolean derivative introduced in Definition 4.6. Let $\alpha(Z')$ denote the state which is complement to α

with respect to values of variables in Z , $Z' \subseteq Z$. Similarly, $f(Z = \bar{Z})$ will denote the function obtained from $f(Z)$ by complementing all variables in Z' . In particular, if $Z = \{z_i\}$ then the latter denotation degenerates into $f(z_i = \bar{z}_i)$.

DEFINITION 8.13. 1) A variable z_j is called *active for z_i* in a state α if $\frac{\partial f_i}{\partial z_j}(\alpha) = 1$;

otherwise z_j is called *passive for z_i* in α .

2) The function of the form $\frac{Sf_i(Z)}{SZ'} = f_i(Z) \oplus f_i(Z = \bar{Z})$ is called the *sensitivity function* of $f_i(Z)$ in variable set Z' .

3) A state β of circuit C is called *critical* for a variable z_j to set $Z' \subseteq Z$, if z_j is idle in β and $\frac{Sf_j(\beta)}{SZ'} = 1$.

4) A state β is called *strictly critical* for z_i to set Z' if (a) it is critical for z_j to set Z' or if (b) z_j is excited in β , $\frac{Sf_j(\beta)}{SZ'} = 0$ and there exists a z_k active for z_j in state $\beta(Z')$ such that if $z_k \notin Z'$, then z_k is excited and passive for z_j in β .

EXAMPLE 8.11 (continued). Return to the system (8.11) which corresponds to the state-transition diagram in Fig. 8.9(b).

1) Variable z_1 is active for z_3 in the 0000 state:

$$\frac{\partial f_3}{\partial z_1}(Z) = f_3(Z) \oplus f_3(z_1 = \bar{z}_1) = (z_1 \vee z_2) \oplus (\bar{z}_1 \vee z_2) = \bar{z}_2,$$

$$\frac{\partial f_3}{\partial z_1}(0000) = 1.$$

The fact that z_1 is active means that in the 0000-1000 transition the value of function f_3 changes. Notice, however, that z_4 is active for z_1 in 1010, even though states 1010 and 1011 are not in the immediately followed by relation – there is no such link in the diagram.

2) Let $Z' = \{z_1, z_2\}$. Then

$$\frac{Sf_3}{SZ'} = f_3 \oplus f_3(z_1 = \bar{z}_1, z_2 = \bar{z}_2) = (z_1 \vee z_2) \oplus (\bar{z}_1 \vee \bar{z}_2) = z_1 z_2 \vee \bar{z}_1 \bar{z}_2.$$

3) The 0000 state is critical for z_3 to $Z' = \{z_1, z_2\}$ because z_3 is idle in 0000

and $\frac{Sf_3}{SZ'}(0000) = 1$.

4) The 0000 state is not critical for z_1 to $Z' = \{z_3\}$ but is strictly critical for z_1 to $\{z_3\}$. Indeed, z_1 is excited in 0000, $Sf_1/S\{z_3\} \equiv 0$, and z_4 is active for z_1 in 0011 ($\partial f_1/\partial z_4 \equiv 1$).

The sensitivity function provides a generalization of the Boolean derivative concept in the sense that for $Z' = \{z_i\}$ we have $Sf_j/SZ' = \partial f_j/\partial z_i$.

The following theorem present necessary and sufficient conditions for a circuit to be sensitive and strictly sensitive to wires.

THEOREM 8.7. *Let L be a set of wires with the output variable z_j ($O(L) = \{z_j\}$). For a circuit C with the initial state α to be sensitive (strictly sensitive) to the set of wires L it is necessary and sufficient that in the circuit C there exists a state β critical (strictly critical) for z_j to $I(L')$, $L' \subseteq L$, and such that : (1) there exists a sequence of states leading from α to β in which all variables in $I(L')$ change their values, and (2) if in this sequence z_j changes, i.e. $\alpha \xrightarrow{\gamma} \delta \xrightarrow{z_j} \beta$ holds, then $Sf_j/SZ(\gamma) = 0$*

where $Z' \subseteq I(L')$ and Z' contains all such (and only such) variables in $I(L')$ which do not change in the sequence leading from state δ to state β .

Because of its length, we omit the proof of this theorem.

This theorem can be used as a background for a sensitivity (strict sensitivity) analysis method which will be similar to methods for checking the membership within some sub-classes of speed-independent circuits. At the first stage, the states which are critical (strictly critical) to L are obtained, and the second stage involves solving the backward reachability problem and checking the conditions of Theorem 8.7.

A method for picking out critical (strictly critical) states of a circuit is provided by the following statement.

STATEMENT 8.4. *Let L be a set of wires with the output variable z_j ($O(L) = \{z_j\}$). The sets of states critical and strictly critical for z_j to $I(L)$ are respectively given by the functions*

$$\frac{Sf_j(Z)}{SI(L)} (\bar{z}_j \oplus f_j(Z))$$

and

$$\frac{Sf_j(Z)}{SI(L)} (\bar{z}_j \oplus f_j(Z)) \vee \frac{\overline{Sf_j(Z)}}{SI(L)} \left(\bigvee_{z_k \in I(L)} \frac{\partial f_j(I(L) = \overline{I(L)})}{\partial z_k} \vee \right. \\ \left. \vee \bigvee_{z_k \in I(z_j) \setminus I(L)} \frac{\partial f_j(I(L) = \overline{I(L)})}{\partial z_k} \cdot \frac{\overline{\partial f_j(Z)}}{\partial z_k} (z_k \oplus f_k(Z)) \right)$$

where $I(z_j)$ is the set of variables on which the $f_i(Z)$ function is essentially dependent.

Notice that the strictly critical for z_j to $I(L)$ states can also be elicited by, first, finding, in the expansion C^L , the states which are 1-conflict in z_j , and, second, by computing the projection of these states on the set of original variables Z .

Let us find a relationship between the circuit sensitivity to wire delays and the properties of Boolean functions defining a circuit. To do this, we need to introduce several new terms.

A state sequence $\alpha^1 \alpha^2 \dots \alpha^k (\alpha^i \rightarrow \alpha^{i+1})$ is called *stationary* in variable z_i if $\alpha_i^1 = \alpha_i^2 = \dots = \alpha_i^k$ and $f_i(\alpha^1) = \dots = f_i(\alpha^k)$. There are four possible variants for a sequence to be stationary in z_i :

(1) $\alpha_i^e = f_i(\alpha^e) = 0$ and (2) $\alpha_i^e = f_i(\alpha^e) = 1$ are associated with stationarity in an idle variable, and

(3) $\alpha_i^e = 0, f_i(\alpha^e) = 1$ and (4) $\alpha_i^e = 1, f_i(\alpha^e) = 0$ are associated with stationarity in an excited variable.

Furthermore, in case (1), $\bar{f}_i^0(\alpha^e) = 1$; in case (2), $f_i^1(\alpha^e) = 1$; in case (3), $f_i^0(\alpha^e) = 1$; and in case (4), $\bar{f}_i^1(\alpha^e) = 1$. In the above expressions $1 \leq e \leq k$, and $f_i^0 = f_i(z_i = 0)$ and $f_i^1 = f_i(z_i = 1)$ are the terms of Shannon's expansion for $f_i(Z)$. The functions $\bar{f}_i^0, f_i^1, f_i^0, \bar{f}_i^1$ are called the *decisive functions* of z_i for a stationary sequence of the respective type. In particular, f_i^0 and \bar{f}_i^1 are called the *excitation functions*, while \bar{f}_i^0 and f_i^1 are called the *equilibrium functions* of z_i .

In any sequence stationary in z_i , the corresponding decisive function remains equal to 1, i.e. in each state of the sequence there is at least one term of the DNF of this function that is equal to 1. As this takes place, different terms of the new DNF of the decisive function may be equal to 1 in different states of the stationary sequence.

We say that in a sequence of states $\alpha^1 \alpha^2 \dots \alpha^k$ stationary in z_i there is a *term takeover* if all the terms in the reduced DNF of the decisive, for this sequence, function that have been equal to 1 in α^1 become equal to 0 in α^k . Such a takeover

will be called the *takeover for variable z_i* with respect to the set Z' of variables changing in this sequence if, for each $z_j \in Z'$, the terms of the reduced DNF of the decisive function which contain z_j , include, also, a term T_1 such that $T_1(\alpha^k) = 1$ and no terms T_2 for which $T_2(\alpha^1) = 1$.

THEOREM 8.8. *If in a circuit, from some initial state we can reach the first state of the sequence in which a term takeover for z_i with respect to Z' takes place, then the circuit is strictly sensitive to the set of wires L where $I(L) = Z'$ and $O(L) = \{z_i\}$.*

Proof. Let $\alpha^1 \alpha^2 \dots \alpha^m$ be stationary in z_i . Let the given takeover take place in this sequence, and, furthermore, none of the prefixes of the form $\alpha^1 \dots \alpha^l$, $l < m$ have such a takeover, i.e. there is a term T in the reduced DNF of the decisive function of z_i such that $T(\alpha^1) = \dots = T(\alpha^{m-1}) = 1$ and $T(\alpha^m) = 0$. In the expansion C^L where the L set satisfies the conditions of the theorem, we build a sequence of states $\hat{\alpha}^1 \hat{\alpha}^2 \dots \hat{\alpha}^m$ that can be obtained from $\alpha^1 \alpha^2 \dots \alpha^m$ in the following way. If

$$\alpha^i \xrightarrow[z_j]{1} \alpha^{i+1}, \text{ then to the sequence being built we add } \alpha' \xrightarrow[z_j]{1} \alpha'', \text{ and if } z_j \in Z', \text{ and } z_i$$

changes in the $\alpha^{i+1} \dots \alpha^m$ sequence, then $\alpha'' \xrightarrow[z_j]{1} \alpha'''$. It is easily seen that the

projection of a sequence built in this way on the set Z' coincides with the $\alpha^1 \dots \alpha^m$ sequence and $f_i^L(\hat{\alpha}^m) \neq f_i(\alpha^m)$ holds because all the terms of the decisive function that are equal to 1 in α^m become equal to 0 in $\hat{\alpha}^m$. Hence, if z_i is idle in α^m of the circuit C , then it is excited in $\hat{\alpha}^m$ of the expansion C^L , and, conversely, if z_i is excited in α^m then z_i is idle in $\hat{\alpha}^m$. The first case contradicts to $C \sim C^L$, and in the other case C^L is not semi-modular with respect to z_i . As a result, C is strictly sensitive to L . *Q.e.d.*

COROLLARY 8.2. *If in an initialized circuit, from some initial state, we can reach the first state of a sequence in which a term takeover for z_i with respect to z_j takes place, and z_j enters into all terms equal to 1 in the final state α^k of the sequence, then the circuit is strictly sensitive to the (j, i) wire.*

The functioning of some asynchronous logical circuits does not depend on wire delays. Among such circuits there are, for example, a trivial class of semi-modular

circuits in which wires have no branches. A wider class of circuits insensitive to wire delays can be given in the following theorem.

THEOREM 8.9. *If an initialized circuit C is semi-modular and a function $f_i(Z)$ essentially depends on a single variable z_i , then the circuit C is strictly insensitive to the (i, j) wire.*

Generally, Theorem 8.9. will also be true for circuits in which the excitation function of the variable z_j essentially depends on a single variable, as, for example, in the case $z_j = z_j z_i \vee \bar{z}_j z_k$. Here, the circuit will be strictly insensitive to the wires (i, j) and (k, j) . However, even this class of asynchronous logical circuits is rather small – it does not even cover the class of totally sequential circuits.

EXAMPLE 8.12. The circuit presented in Fig. 5.9(a) consists of a pair of RS-flip-flops and is totally sequential with respect to any states, except those in which both arms of the same flip-flop are reset to 0. This circuit is strictly insensitive to the set of wires that connect flip-flops to each other. On the other hand, it is sensitive to the wires of flip-flop feedbacks. As this structure is typical for all “Master-Slave”-based flip-flops, for them, the results indicated can be directly implied.

The parallel-sequential circuit of Fig. 4.2(d) has five wires – (2, 1), (3, 1), (1, 2), (1, 3), (1, 1). It is strictly insensitive to all wires except the last. The structure of this circuit is typical for parallel-series interconnections of semi-modular circuits. This also makes it possible to generalize the results of the analysis.

8.5 Circuit Petri nets

In the previous sections of this chapter, we have discussed methods for analyzing circuits with the direct use of the Muller model. As a result of such a “local” representation of circuit operation, we have achieved rather effective algorithms of analysis.

It is, however, the case that the Muller model, without an accompanying transition diagram, is unable to express the operational dynamics of a circuit in a vivid way. To have the dynamics in an explicit form, we may turn to Petri nets that were discussed in Section 3.3. It was established that Petri nets, being a convenient formal language for parallel systems, also provide a local view of a system, in the sense that they represent some local relationship between the system's components. The “global” behaviour of the system is studied as a result of local interactions between these components.

This section presents a rather short discussion of another sub-class of Petri nets that is *in one-to-one correspondence with the Muller model*. This class is intended, primarily for analyzing asynchronous circuits. It is transparent enough to reflect the dynamics of their functioning and is free from certain of the drawbacks of the classes considered in 2.2.2.

As the basis for such an approach, we use the idea of representing an element of a circuit by a sub-net of a Petri net.

Let $P^* \subseteq P$ and $T^* \subseteq T$ be, respectively, sub-sets of conditions (places) and events (transitions) of a net N .

We also make the following denotations

$$I(P^*) = \bigcup_{p_i \in P^*} I(p_i), \quad O(P^*) = \bigcup_{p_i \in P^*} O(p_i),$$

$$C(P^*) = I(P^*) \cap O(P^*), \quad V(P^*) = O(P^*) \setminus C(P^*),$$

$$J(P^*) = I(P^*) \setminus C(P^*), \quad I(T^*) = \bigcup_{t_i \in T^*} I(t_i),$$

$$O(T^*) = \bigcup_{t_i \in T^*} O(t_i).$$

DEFINITION 8.15. 1) Let a Petri net $N = \langle P, T, M_0, H, F \rangle$ and some sub-set $P^* \subseteq P$ be given. A net $N^* = \langle P^*, T^*, M_0^*, H^*, F^* \rangle$ such that

$$T^* = I(P^*) \cup O(P^*), \quad F^* : P^* \times T^* \rightarrow \{0, 1\},$$

$$H^* : P^* \times T^* \rightarrow \{0, 1\}, \quad M_0^* : P^* \rightarrow \{0, 1, \dots\}$$

will be called the *sub-net* N^* of N with respect to P^* .

2) A *simple loop* is a sub-net $N^l = \langle P^l, T^l, M_0^l, H^l, F^l \rangle$ of a net N where $P^l = \{p_i\}$, $T^l = \{t_j\}$, $M_0^l = M_0(p_i)$, $H^l = H(p_i, t_j)$, $F^l = F(t_j, p_i)$ and $H^l = F^l = 1$.

We prefer to use a more economical designation of a simple loop as shown in Fig. 8.10(b) rather than its usual form of Fig. 8.10(a). It is clear that an event t_i cannot fire if $M(p_i) = 0$, and that any firing of t_j cannot change $M(p_i)$.

With a z_i in a circuit, we associate a pair of conditions z_i and \bar{z}_i in a Petri net. Two events s_i and r_i will be associated with corresponding element switchings, i.e.

$s_i(r_i)$ stands for the 0-1 (1-0) transition of z_i and appear to be the output event for the $z_i(\bar{z}_i)$ condition. Thus, the behaviour of an element is modelled by the sub-net of the form shown in Fig. 8.10(c). We shall call it an *element cycle*.

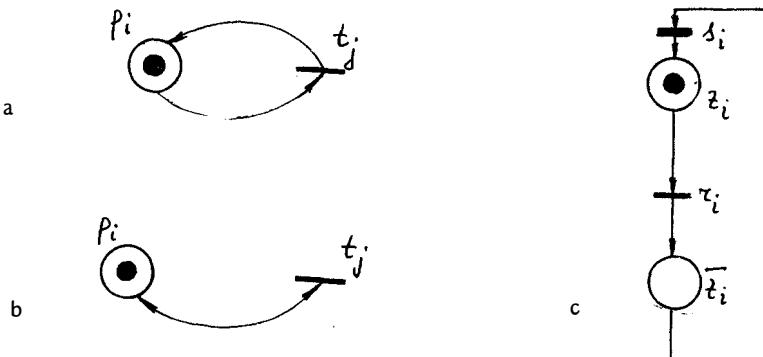


Figure 8.10(a)-(c). Representing a loop (a) and (b) and an element cycle (c) in a circuit Petri net.

We write the inherent function of an element in the form $z_i = z_i \bar{R} \vee \bar{z}_i S$. The 0-1 transition is associated with $S_i = 1$ while the 1-0 transition is associated with $R_i = 1$. Since an element's switching does not affect its inputs, the arcs of a Petri net that reflect the connections between the inputs of one element with the outputs of another should belong to simple loops.

Let us use a conjunctive expansion of S_i and R_i functions for z_i . Then, in modelling a Petri net, we should have some auxiliary conditions that correspond to the disjunctive terms of these functions. Such conditions will be input for corresponding events of the type $s_j(r_j)$ (that enter into the cycles of elements other than z_i), and output for events of the type $r_j(s_j)$. Thus, it is possible to build a net that will model the functioning of any given circuit.

EXAMPLE 8.13. Fig. 8.11(a) shows a circuit, and the corresponding modelling Petri net is shown in Fig. 8.11(b). The initial marking of the net corresponds to the 111 state of the circuit. The function $S_1 = \bar{z}_1 \vee \bar{z}_3$ contains one disjunctive term. It is associated with a condition for which $J(P) = \{r_1, r_3\}$, $V(P) = \{s_2, s_3\}$ and $C(P) = \{s_1\}$.

The Petri net described can be related to a special class of Petri nets. Such nets have certain structural peculiarities – every event enters into some element cycle, and all conditions that are linked to this event, and which are not in the given element

cycle, form a simple loop with this event. We shall call such nets *circuit Petri nets*. To be more exact, circuit Petri nets that correspond to the conjunctive representation form for inherent functions of elements will be called *conjunctive circuit Petri nets*. Similarly, with the use of a disjunctive expansion for S_i and R_i functions, we can also build *disjunctive circuit Petri nets*.

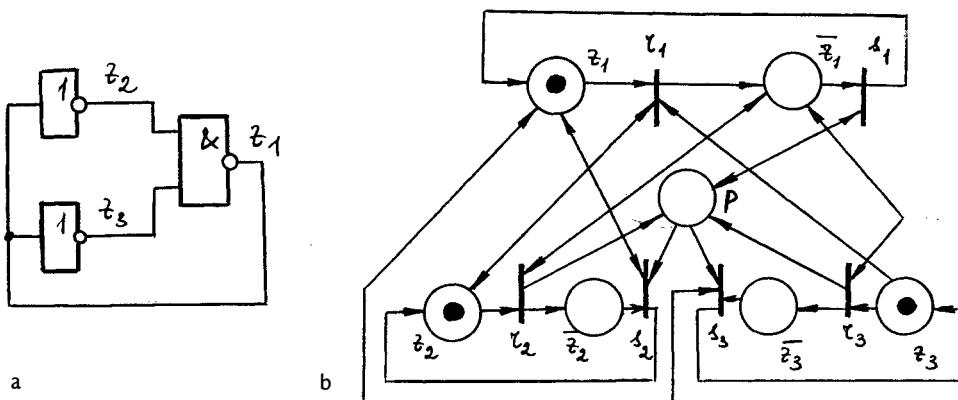


Figure 8.11(a) and (b). Example of a circuit Petri net.

For a circuit consisting of n elements, a conjunctive net will contain $2n$ events. In the P set we can pick out two sub-sets: P_1 , consisting of inherent conditions entering into some element cycles, $|P_1| = 2n$, and P_2 , consisting of conditions that have been added with disjunctive terms of inherent functions, $|P_2| \leq \sum_{i=1}^n m_i$, where m_i is the number of terms in the conjunctive normal forms (CNF) of S_i and R_i functions of z_i .

A disjunctive net would contain $2n$ conditions, and the number of transitions (events) would be $|T| = \sum_{i=1}^n m_i$, where m_i is the number of terms in the DNFs of S_i and R_i functions.

Each marking of a circuit Petri net is associated with some state of a modelled circuit. As a result, the set R_N of all markings of the net that can be reached from the initial marking M_0 is limited by $|R_N| \leq 2^n$ where n is the number of elements in the circuit.

For our purposes, the main analysis problem that should be solved for circuit Petri nets is the proof of their persistence. Let us recall that the local form of the violation of the persistence of a Petri net is the conflict of a marking of the net.

A marking M is considered to be conflict if, after the firing of a certain number of events (T^*) that have been enabled for M , another marking M^* , $M \xrightarrow{T^*} M^*$, “does not provide” enough tokens in some place (p_i) to fire the events enabled for M . If there are “enough” such tokens, M is not conflict.

The following theorem gives a conflict condition for a marking in a conjunctive net.

THEOREM 8.10. *Let, in a conjunctive circuit Petri net, a condition $p_r \in P$ and an event $t_j \in O(p_r)$ be given. Let also $V'(p_r)$ be a sub-set of the set of input events for p_r , ($V'(p_r) \subseteq V(p_r)$), $t_j \notin V'(p_r)$, and $M(p_r)$ be some marking for which $l = M(p_r)$. The marking M is conflict if, and only if, all events in $V'(p_r)$ and t_j are enabled and, in addition, $|V'(p_r)| \geq l$.*

The proof follows directly from the definition of a conflict marking and the rule forming the set $V(P)$.

Now we should formulate the main result for using Petri nets in the analysis of asynchronous circuits.

THEOREM 8.11. *Let a marking M of a conjunctive circuit Petri net N be associated with a state α of a circuit C . The circuit is semi-modular with respect to α if every marking M' reachable from M in N is non-conflict.*

The result follows directly from the comparison between the definition of a persistent Petri net and that of a semi-modular circuit.

A method for searching for a cover of the set of all conflict markings in a circuit Petri net can be based on Theorem 8.10 which implies the existence of an algorithm for searching for a cover of the set of circuit states with respect to which the circuit is not semi-modular. The first part of such an algorithm would consist of finding conflict markings of the Petri net, and the second would search the markings from which the conflict ones are reachable. In a similar way, we could demonstrate the possibility of reformulating all the analysis algorithms outlined in the earlier sections to fit the framework of a circuit Petri net model. Notice that the circuit analysis using circuit Petri nets is thus reduced to solving a reachability problem in Petri nets (keeping the fact of their boundedness in mind).

Circuit Petri nets have some additional properties, not mentioned here, that can be of special interest for analysing a circuit's freedom from deadlocks and liveness, i.e. the possibility for all elements to switch an unlimited number of times, during the cyclic operation of a circuit.

The use of such a well-known model, as provided by Petri nets, for parallel processes, vividly illustrates both the asynchrony and the concurrency of switching processes in a circuit. Furthermore, it opens up the possibility of using the algorithms discussed for proving the correctness of asynchronous parallel programs.

It is interesting to note that, in some cases, there is an advantage in combining the methods of this chapter with those for checking the circuit indicatability. Such combinations will be discussed further in Chapter 9.

8.6 On the complexity of analysis algorithms

In accordance with common notation, the assertion that a function $f(n)$ has an order not exceeding that of $g(n)$, will be denoted as $f(n) = O(g(n))$, if $f(n) \leq cg(n)$, for almost all n , where c is some constant.

The proving of reachability, liveness, speed-independence, semi-modularity, distributivity etc. can be carried out in terms of a non-interpreted asynchronous process by using, as initial data, the specification of relation F in the form, for example, of an adjacency matrix, on the set of situations of a process. The reachability problem is thus reduced to the manipulation of such a matrix, by means of, say, Warshall's algorithm with complexity $O(|S|^3)$ where $|S|$ is the number of situations of the process. The search for points where the violation of semi-modularity, liveness and total sequentiality takes place is carried out by checking some local properties of the process graph (adjacency matrix) and requires $O(|S|^2)$ steps. Thus, checking whether an asynchronous process is semi-modular, distributive or totally sequential with respect to a given situation can be carried out for $O(|S|^3)$ steps.

It is not always convenient to perform analysis in such a way since the number of situations of a process $|S|$ can be large, and, hence, the specification, whose length is of $O(|S|^2)$ would be rather unwieldy. Another weakness, is that such an approach does not provide any efficient ways for analyzing sets of situations.

The techniques operating on asynchronous process interpretations, say, in terms of the Muller model (or circuit Petri nets), are more economical, since due to the encoding of situations and the representation of variables by formulae, the process definition length becomes of the order of $c(m) \log_2(|S|^2)$ where $c(m) = O(2^m)$ is a parametric constant. The m parameter is the maximum number of encoding variables on which a given variable essentially depends ($m < \log_2|S|$, but in practice, $m \ll \log_2|S|$ holds). Furthermore, within the indicated interpretations, the efficiency of analysis may be increased by using various heuristic methods and, of course, with the aid of succinct forms of representation of situation sets, as discussed in previous sections of this chapter.

Table 8.1 contains a brief summary of complexity bounds for solving analysis problems within the framework of the Muller model and circuit Petri nets, where the parametric constants are as follows

$$c_1(m) = O(2^{\min(m^2, \log_2 |S|)}),$$

$$c_2(m) = O(2^{\min(m^3, \log_2 |S|)}),$$

$$c_3(m) = O(2^{\min(m^2, \log_2 |S|)}).$$

Problem (Property)	Complexity Order
Reachability	$ S ^3$
Search for Situations with Semi-modularity Violation	$c_1(m) \log_2 S $
Search for Situations with Distributivity Violation	$c_2(m) \log_2^3 S $
Search for Situations with Total-Sequentiality Violation	$c_3(m) \log_2^2 S $

Table 8.1.

We conclude this chapter with the note that the methods for analysis of asynchronous processes and circuits that have been developed using the characteristic function approach in combination with the Muller model framework, facilitate the design of reasonably efficient CAD products. Some software packages have been constructed on their basis and used for verification of real circuits of limited dimensions. The results of the work indicate that the complexity bounds, above, are rather pessimistic.

8.7 Reference notations

Analysis of asynchronous parallel systems based on the models examined earlier (oriented, however, somewhat towards programming problems) has been a topic of great interest for the last two decades. Among the work, we can pick out some publications on Petri net analysis [119], [197], [238], [239], [245], [254], [305] and [306], marked graph analysis [260] and the study of transition systems [248] (a

model closely related to an asynchronous process [9]). More complete information can be obtained from surveys [197], [198] and [284] and special report collections [275] and [292].

Various statements of the circuit analysis problem, different from those discussed in this chapter, can be found in [169], [181], [187] and [229].

The work on speed-independent circuit analysis was done by D.E. Muller.

Solution methods for forward and backward reachability problems can also be found in [4] and [181].

An approach for the design of circuits whose operation is independent of wire delays was discussed in [249] where a universal set of modules was suggested. Circuits built of such modules are both independent of the speed of the modules and insensitive to the delays in intermodular wires. The weak point of this approach is the size of the circuits obtained and, more important, the modules themselves are dependent on the values of composing elements and intra-modular wires.

The relative growth of wire delays with the increase in the integration scale is discussed in [207] and [291].

The use of Petri nets as a special class for switching circuit analysis is the topic of [99] and [237].

Warshall's algorithm for constructing a connectivity matrix from a graph is described in [308].

Methods for checking circuits for liveness and freedom from deadlock can be found in [99], [199], [229] and [232].

The material of Chapter 8 is, in general, based on [4] and [99]. The proofs of Theorems 8.5 and 8.11 are presented in [4].

The methods of analysis discussed in this chapter can also be applied to the verification of data transfer protocols, [61] and [98], and the control structures of parallel programs.

CHAPTER 9

ANOMALOUS BEHAVIOUR OF LOGICAL CIRCUITS AND THE ARBITRATION PROBLEM

I began to understand that I had to choose
between them once and for all.
My two natures shared a common memory ...

Robert Louis Stevenson

The evolution of computer architecture styles and the constant growth of systems performance require an adequate updating of all design solution and approaches related to the problems of sharing common resources in a system, and timing of interactions between system components. The first references to the difficulties that arise in solving these problems at the hardware level were noticed in the mid-sixties. Such difficulties can be illustrated by processes in a synchronous device whose input receives an asynchronous signal, as shown in Fig. 9.1.

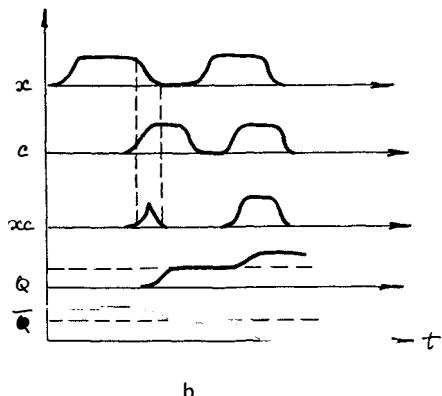
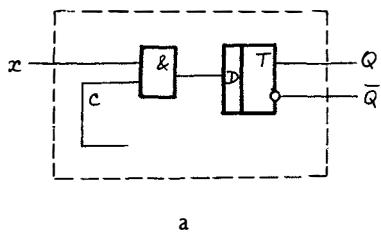


Figure 9.1(a) and (b). Behaviour of a synchronous circuit with an asynchronous input applied to it: (a) circuit, (b) waveforms.

The device consists of a D -flip-flop with direct and complement Q , \bar{Q} outputs and an AND element whose output is connected to the D input of the flip-flop. The input element of the device receives an external asynchronous sequence x and a sequence of internal synchronization impulses c . As can be seen from Fig. 9.1(b), short

impulses may appear at the output of this gate. The energy of such an impulse may suffice to start a transition process in the flip-flop, but at the same time, it will be insufficient to change the state of the latter. As a result, the flip-flop may enter a special state, the so-called *meta-stability condition*, in which both output signals will coincide on a level representing neither a logical 0, nor a logical 1. Devices of this type, that match asynchronously arriving sequences, are usually called *synchronizers*, and the behaviour indicated by these devices is called *anomalous*.

Later, it was discovered that similar anomalous behaviour is also typical of *arbiters*. The arbiter is a device that grants a common resource to exactly one of the processes competing for this resource. An *arbitration phenomenon* is abstract enough to be independent of the nature of the processes and their common resource. Even if these processes are in software, the problem, though being tackled at the software level, will, by some conflict resolution primitives, finally focus on the construction of a hardware arbiter which is required to operate correctly. A general structure of the k -input arbiter in its inter-connection with the requesting processes can be illustrated by Fig. 9.2.

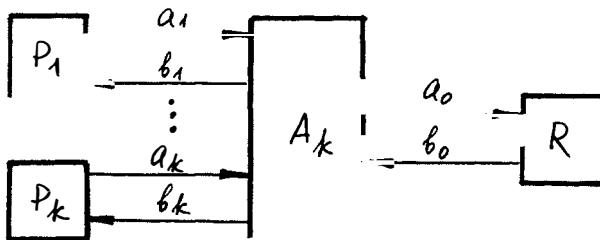


Figure 9.2. Block diagram for a k -input arbiter (A_k -circuit).

The main signals are

- a_1, a_2, \dots, a_k , the request signals from the processes P_1, P_2, \dots, P_k competing for the shared resource R ,
- b_1, b_2, \dots, b_k , the acknowledge signals from the arbiter to the processes P_1, P_2, \dots, P_k ,
- a_0 , the request signal from the arbiter to the common resource,
- b_0 , the acknowledge signal from the resource to the arbiter.

There are four known types of anomalies in the behaviour of circuits implementing synchronizers and arbiters. These are

- (i) the *meta-stability anomaly* shown in Fig. 9.3(a),
- (ii) the *oscillatory anomaly*, shown in Fig. 9.3(b), which is a synphase (coherent) oscillation of signals of both outputs,
- (iii) *instable transition process* shown in Fig. 9.3(c),

(iv) dragging in one of the edges during a transition process, as shown in Fig. 9.3(d).

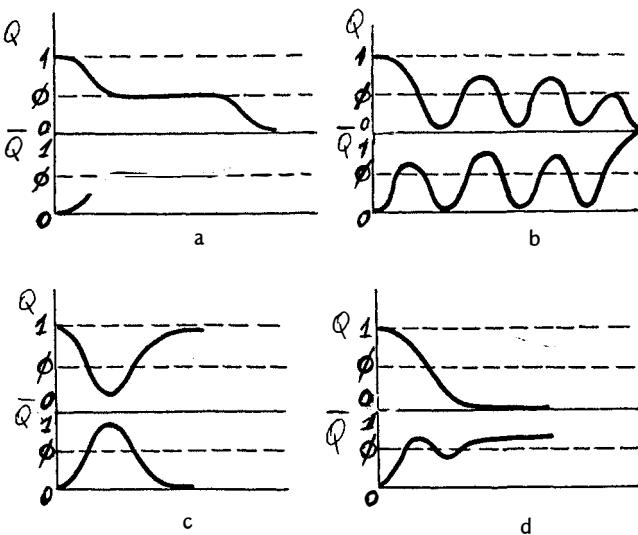


Figure 9.3(a) - (d). Various kinds of anomalies in circuits implementing arbiters: (a) meta-stability, (b) oscillation, (c) unstable transition process, (d) edge dragging on.

The anomalous behaviour may arise as a result of applying a short impulse (strobe) to one of the inputs of an arbiter or a synchronizer, especially when the magnitude of such a strobe only slightly exceeds the switching threshold of an element. An anomalous condition may also occur in cases where requests for a shared resource arrive from different processes with a time interval shorter than the delay of a transition process completion in the arbiter. The possibility of occurrence of the last two types depends on the structure of the device, and it is not too difficult to ensure their avoidance. On the other hand, anomalies of the first two kinds are fundamental in nature, and their elimination is very difficult to achieve.

Experiments indicate that anomalies in arbiters and synchronizers implemented with two-valued logical elements present an inherent source of faults in computer and digital control systems. A theoretical study of anomalous phenomena and methods for designing correctly functioning arbiters will be discussed in the following sections. Such a discussion is of practical value – until now, examples of incorrect implementations of arbiters and synchronizers have kept appearing.

9.1 Arbiters

Before giving a formal definition of arbitration circuits, we introduce some auxiliary definitions.

An *asynchronous logical circuit* C is a triple $\langle Z, Z_0, F \rangle$ where $Z = \{z_1, z_2, \dots, z_n\}$ is a set of binary variables, $Z_0 \subset Z$ is a set of input variables, and a system F of Boolean functions has the form $z_i = f_i(z_1, \dots, z_n)$, $z_i \in Z \setminus Z_0$.

A vector γ consisting of the values of variables in $Z' \subset Z$ will be called a *circuit sub-state*. A sub-state with dimension l defines a sub-cube consisting of 2^{n-l} states. The sub-state will be called *stable* (with respect to a given fixed value of input variables) if variables $z_i \in Z'$ ($z_i \in Z \cap (Z \setminus Z_0)$) are idle in all states defined by it. As the behaviour of the arbiters will be investigated for fixed values of input request signals, it is sufficient to consider the cases where the input variables are idle in all the states.

The set of all states reachable from an α will be denoted as $R(\alpha)$, and the set of all states immediately reachable from α as $r(\alpha)$. Let us recall that a set of states $D = \{\alpha_1, \alpha_2, \dots, \alpha_m\}$ is called a closed class if for every $\alpha_j \in D$ we have $R(\alpha_j) = D$. Let $E(\alpha)$ denote the set of closed classes reachable from α .

The projection of a set of states $B = \{\alpha_1, \alpha_2, \dots, \alpha_m\}$ on Z' , denoted by $B|Z'$, is defined and is equal to the projection of α_1 on Z' , denoted by $\alpha_1|Z'$, if $\alpha_1|Z' = \dots = \alpha_m|Z'$, and is not defined otherwise. For brevity, we shall only show in the projection those variables in Z' whose values are equal to 1 in a state (set of states) being projected. The states themselves will be represented, not by their binary combinations, but by product terms corresponding to them. For example, if $Z' = \{z_1, z_2\}$, then $\bar{z}_1 z_2 \bar{z}_3 \bar{z}_4|Z' = z_2$ and $\bar{z}_1 \bar{z}_2 z_3 z_4|Z' = 0$.

A circuit C_1 covers a circuit C_2 of the same dimension if each arc belonging to the transition diagram of C_2 can also be found in the diagram of C_1 .

Note that the model of an asynchronous logical circuit, used here, is based on the same assumptions about its physical implementation as have been outlined in Chapter 4.

DEFINITION 9.1. A circuit $A_k = \langle Z, Z_0, Z, A, B, F \rangle$ where $A = \{a_1, a_2, \dots, a_k\} \subseteq Z_0$ is a set of input request variables and $B = \{b_1, b_2, \dots, b_k\} \subset Z$ is a set of output acknowledge signals is called an *arbiter of the k-th rank* (an A_k -circuit) if it satisfies the following conditions:

- 1) the *acknowledgement set condition*: if the state α is such that $\alpha|A = a_{i1} \dots a_{il}$, $l \leq k$, then (a) for every closed class $D \in E(\alpha)$ we have $D|B = b_{ir}$, where $1 \leq r \leq l$, and (b) every complete sequence from α leads to a closed class;

2) the *acknowledgement reset condition*: if the α state is such that $\alpha|a_i = 0$, then $E(\alpha)|b_i = 0$;

3) the *acknowledgement stability condition*: if the α state is such that $\alpha|a_i = a_i$ and $\alpha|B = b_i$, then $r(\alpha)|B = b_i$, then $r(\alpha)|B = b_i$.

It should be noted that here A_k -circuits are not required to be race-free with respect to the acknowledge variables. This means that no restrictions are imposed on the acknowledge variables during the transition process. It is also assumed, without loss of generality, that the direct encoding of signals is used. Furthermore, since the major cause of anomalies in arbiters is the interaction between the arbiter and the processes, we are interested here only in the part of the arbiter which is responsible for such interaction.

OBSERVATION 9.1. *Every A_k -circuit for $k > 1$ is also an A_{k-1} -circuit.*

OBSERVATION 9.2. *Every A_k -circuit for $k > 1$ can be built of A_{k-1} - and A_2 -circuits.*

The proof for the case of $k = 2$ is trivial. For $k > 2$ we can use the construction of Fig. 9.4 for which it can be easily demonstrated that if A_{k-1} and A_2 satisfy the conditions 1) to 3) of Definition 9.1, then so does A_k . From these observations we can see that for the design of any A_k -circuit, it is necessary and sufficient to be able to design an A_2 -circuit.

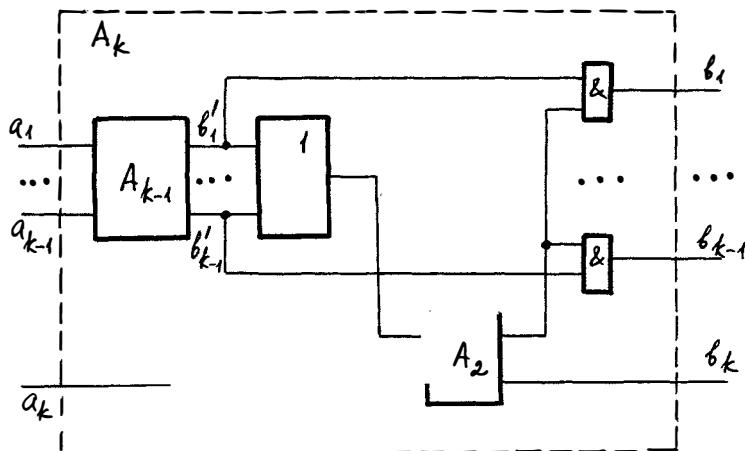


Figure 9.4. Illustration of constructability of arbitrary arbiters using two-input form.

9.2 Oscillatory anomaly

To investigate the effect of internal feedbacks of elements on the behaviour of arbitration circuits we should introduce several more concepts.

A variable z_i is called *self-dependent* if $f_i(z_1, z_2, \dots, z_n)$ essentially depends on z_i , i.e. $f_i(z_i = 0) \neq f_i(z_i = 1)$. Otherwise, z_i is *non-self-dependent*. A circuit C is *non-self-dependent* if each of its variables is non-self-dependent, otherwise C is *self-dependent*.

A variable z_i is called (*non*)*antitone-self-dependent* if it is self-dependent and f_i is (non)antitonus in z_i . A circuit C is *antitone-self-dependent* if all its self-dependent variables are antitone-self-dependent. Otherwise C is *non-antitone-self-dependent*.

OBSERVATION 9.3. A circuit C is *non-self-dependent* in z_i if, and only if, for any pair of states α and α' adjacent with respect to z_i , either $\alpha \in r(\alpha')$ or $\alpha' \in r(\alpha)$ (not both) holds.

OBSERVATION 9.4. A circuit C is *antitone-self-dependent* in z_i if, and only if, for any pair of states α and α' adjacent with respect to z_i , either $\alpha \in r(\alpha')$ or $\alpha' \in r(\alpha)$ holds, and for at least one pair, both of them hold.

DEFINITION 9.2. We say that a circuit C has an *oscillatory anomaly* with respect to the set Z' of variables if there exists a state cycle $\alpha_1 \alpha_2 \dots \alpha_p \alpha_1 \alpha_2 \dots \dots (\alpha_i \in r(\alpha_{i-1}), 1 \leq i \leq p - 1, \alpha_1 \in r(\alpha_p))$ such that for some $\alpha_i, \alpha_j, 1 \leq i, j \leq p$, $\alpha_i|Z' \neq \alpha_j|Z'$ holds.

THEOREM 9.1. No A_k -circuit belongs to the class of *non-self-dependent* or *antitone-self-dependent* circuits.

Proof. Due to Observations 9.1 and 9.2 and because of the fact that a construction based on Observation 9.2 preserves non-self-dependency and antitone-self-dependency, it is sufficient to prove the theorem for A_2 -circuits. Hence, $A = \{a_1, a_2\}$ and $B = \{b_1, b_2\}$.

Consider two cases.

- For each state α such that $\alpha|A = a_1a_2$ and for arbitrary $D_i, D_j \in E(\alpha)$, $D_i|B = D_j|B$ holds. Due to symmetry, we assume $D_i|B = b_1$. Consider two states $\alpha = a_1a_2 \ \bar{b}_1 \ \bar{b}_2 \beta$ and $\alpha' = a_1a_2 \ \bar{b}_1b_2 \beta$ adjacent with respect to b_2 where β is the vector of values of the remaining variables. According to Observations 9.3 and 9.4 either $\alpha \in r(\alpha')$ or $\alpha' \in r(\alpha)$ must hold. The first condition is, however, in

contradiction with the acknowledgement stability condition in an A_k -circuit, and the second one contradicts either the initial assumption that $D_1|B = b_1$ or the acknowledgement stability conditions. Case 1 thus leads to contradiction.

2. There exists a state α such that $\alpha|A = a_1a_2$ and there are two closed classes $D_1, D_2 \in E(\alpha)$ for which $D_1|B \neq D_2|B$ holds. Due to symmetry, we assume $D_1|B = b_1, D_2|B = b_2$. Consider two sets of states R_1 and R_2 such that $R_1|A = R_2|A = a_1a_2, R_1|B = b_1, R_2|B = b_2$. It is clear that $D_1 \subset R_1, D_2 \subset R_2$ and $R_1 \cap R_2 \neq \emptyset$, and due to the acknowledgement stability conditions, R_1 and R_2 are closed sets of states ($a_1 = a_2 = 1$). Now consider four states $\alpha_1 = a_1a_2 \bar{b}_1 \bar{b}_2 \beta, \alpha_2 = a_1a_2b_1 \bar{b}_2 \beta, \alpha_3 = a_1a_2 \bar{b}_1b_2 \beta$ and $\alpha_4 = a_1a_2b_1b_2 \beta$. It is obvious that $\alpha_1, \alpha_4 \notin R_1, R_2$ nor do α_1 and α_4 belong to any closed class in $E(\alpha_1)$. State α_1 is adjacent to α_2 and α_3 , state α_4 is adjacent to α_2 and α_3 , and furthermore, $\alpha_1, \alpha_4 \notin r(\alpha_2), r(\alpha_3)$. Hence, according to Observations 9.3 and 9.4 $\alpha_2, \alpha_3 \notin r(\alpha_1), r(\alpha_4)$. However, this implies $\alpha_1 \in r(\alpha_4), \alpha_4 \in r(\alpha_1)$ which contradicts the acknowledgement set condition, case (b). *Q.e.d.*

COROLLARY 9.1. *Any non-self-dependent or antitone-self-dependent circuit, covering an A_k -circuit, has an oscillatory anomaly with respect to the acknowledgement variables.*

This corollary is directly based on the proof of Theorem 9.1. Indeed, α_1, α_4 is an oscillatory anomaly with respect to the acknowledgement variables, regardless of β .

If we admit a semi-conductor gate to be an element, then no element can be self-dependent. Nevertheless, the interconnection of gates situated within a chip, having an inverter-amplifier, can be considered as an element whose delay is attached to the output inverter. Such a model adequately describes the behaviour of existing discrete elements and integrated circuits. Connecting the output of such an element to the input gives a circuit that, quite precisely, can be considered as an antitone-self-dependent element. To implement a non-antitone-self-dependent circuit we should implement a positive gain loop by one of the following ways: (1) there are no inverters in the loop, or (2) there is an even number of inverting cascades in the loop. In the first case, there is no way of getting any signal amplification in the loop. In the second case, the assumption of attaching the element delay to the output inverter is no longer correct. When analysing the behaviour of such an element, each inverting cascade should be assessed as a separate element, and, hence, the element will not be self-dependent.

Thus, an arbiter cannot be implemented using binary logical circuitry as any such design will be subject to an oscillatory anomaly with respect to the acknowledgement variables.

9.3 Meta-stability anomaly

A *ternary function* (T-function) is a mapping of the form $\{0, \emptyset, 1\}^n \rightarrow \{0, \emptyset, 1\}$. Note that $0 < \emptyset < 1$. The class of *Boolean ternary functions* (BT-functions) is the set of ternary functions preserving the $\{0, 1\}$ set, i.e. for any combination of 0's and 1's, a BT-function assumes the value in $\{0, 1\}$ but not \emptyset .

Examples of elementary T-functions are as follows

1. $0, \emptyset, 1$ are constants.
2. $x \vee y = \max(x, y)$ is the generalized disjunction ($0 < \emptyset < 1$).
3. $x \cdot y = \min(x, y)$ is the generalized conjunction.
4. $\bar{x} = 1 - x$ is the generalized complement ($1 - \emptyset = \emptyset$).
5. $x^a = \begin{cases} 1 & \text{if } x = a \\ 0 & \text{if } x \neq a \end{cases}$ are characteristic functions for three values of a
($a = 0, \emptyset, 1$).

All these functions are BT-functions except for the constant \emptyset .

DEFINITION 9.3. A *ternary circuit* C^T is a triple $\langle Z^T, Z_0^T, F^T \rangle$ where $Z^T = \{z_1, \dots, z_n\}$ is a set of ternary variables, F^T is a system of ternary equations of the form $z_i = F_i(z_1, \dots, z_n)$, $z_i \in Z^T \setminus Z_0^T$ where F_i is a T-function.

All the concepts for asynchronous circuits, introduced earlier, can be generalized for ternary circuits. A variable $z_i \in Z^T$ is *meta-stable* in a state α if $\alpha|z_i = F_i(\alpha) = \emptyset$. A sub-state γ that corresponds to the values of variables in $Z' \subset Z$ will be called meta-stable if, in all states defined by γ , at least one of the variables $z_i \in Z'$ is meta-stable, and the remaining ones in Z' are not excited. An obvious particular case of a meta-stable sub-state is a meta-stable state for $Z' = Z$. The concept of a ternary transition diagram can also be introduced for a ternary circuit, in the same way as has been done for binary circuits.

Let the behaviour of a circuit element (variable) z_i be given by the Boolean equations $z_i = f_i(z_1, \dots, z_n)$ where the inherent function f_i is presented in the AND-OR-NOT functional basis. A BT-function derived from the original Boolean

function $f_i(z_1, \dots, z_n)$ through a generalization of the conjunction, disjunction and complement operators, and through the substitution of ternary variables for the binary variables, will be called a generalization of the function f_i and will also be denoted by $f_i(z_1, \dots, z_n)$. In order to derive generalizations of Boolean functions given in some other bases it is necessary to express each operator in the basis by a formula in the AND-OR-NOT basis, and then to generalize both operators and variables.

The behaviour of an element z_i in a ternary circuit describing the operation of a real circuit during a transition process can be expressed by the ternary equation $z_i = F_i(z_1, \dots, z_n)$. The values of the T-functions F_i depend on the values of the generalized inherent functions $f_i(z_1, \dots, z_n)$ of elements as well as upon the signal levels representing \emptyset , 0 and 1. Let V_L , V_H , V_\emptyset indicate the threshold voltage levels of an element in its switching to logical 0, 1 and \emptyset , respectively.

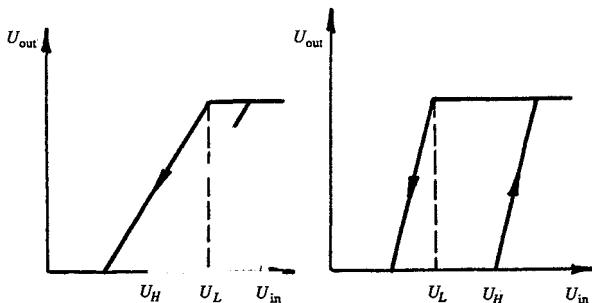


Figure 9.5. Idealized input-output characteristics of logical elements with different slopes and hysteresis width.

Analysis of input-output characteristics (see Fig. 9.5) of logical elements show that the following relationship variants are possible depending on the characteristic slope and hysteresis width:

- 1) $V_\emptyset \leq V_L \leq V_H$,
- 2) $V_L \leq V_\emptyset \leq V_H$,
- 3) $V_L \leq V_H \leq V_\emptyset$,
- 4) $V_\emptyset \leq V_H \leq V_L$,
- 5) $V_H \leq V_\emptyset \leq V_L$,
- 6) $V_H \leq V_L \leq V_\emptyset$,
- 7) the V_\emptyset level is not fixed ("floats") and, in some instances, may correspond to different relationships.

These cases correspond to four different types of T-functions that can be defined by the following ternary formulae:

$$\begin{aligned}
 F_{1i}(Z) &= \bar{z}_i^0 f_i(Z) \vee \emptyset f_i^1(Z) \vee \emptyset z_i^1, \\
 F_{2i}(Z) &= \bar{z}_i^0 f_i(Z) \vee \emptyset \bar{f}_i^0(Z) \vee \emptyset z_i^1, \\
 F_{3i}(Z) &= \bar{z}_i^0 f_i(Z) \vee z_i^1 f_i^0(Z) \vee \emptyset \bar{f}_i^0(Z) \vee \emptyset z_i^1, \\
 F_{4i}(Z) &= z_i^0 f_i(Z) \vee z_i^1 f_i^0(Z) \vee \emptyset f_i^1(Z) \vee \emptyset z_i^1.
 \end{aligned} \tag{9.1}$$

Note that in all of the variants, if $z_i = f_i(Z) = \emptyset$ then $F_i(Z) = \emptyset$, i.e. z_i is not excited in the \emptyset state. This is due to the fact that the \emptyset level is associated with the gain of a logical element equal to 1. A ternary circuit defined by the equations $z_i = F_{ji}(Z)$, $j \in \{1, 2, 3, 4\}$, where for different variables z_i the type (j) of the function may generally be different, has the following peculiarities. If a z_i is idle in a state α in the original circuit, then it is idle in that state in the ternary circuit. If z_i is excited in α in the original circuit then it will be excited in that state in the ternary function, but in this case, $F_i(\alpha) = \emptyset$ — the switching of an element from 0 to 1 and from 1 to 0 must always proceed through \emptyset . If z_i is non-self-dependent in the original circuit, then after changing to \emptyset it will remain in the excited state, thus proceeding to 0 or to 1. Finally, if $f_i(\alpha) = \emptyset$, then, depending on the value of z_i in that state and on the relationship between V_L , V_H and V_\emptyset , z_i may be either excited or idle as determined by the F_i function type.

The ternary functions F_{1i} , F_{2i} , F_{3i} , F_{4i} thus given will be called *normal extensions of the Boolean function $f_i(Z)$ of the types 1 to 4*, respectively.

OBSERVATION 9.5. Let $f_i(Z)$ be a Boolean function and $F_i(Z)$ its normal extension of any type. Then

- 1) if $f_i(\alpha) = 0$ and $\alpha|z_i \neq 1$, then $F_i(\alpha') \leq \emptyset$ where the ternary vector α' is obtained from α by replacing some binary values with \emptyset ;
- 2) if $f_i(\alpha) = 1$ and $\alpha|z_i \neq 0$, then $F_i(\alpha') \geq \emptyset$;
- 3) if $f_i(\alpha) = 0$, $f_i(\beta) = 1$, $\alpha|z_i \neq \beta|z_i$ then $F_i(\alpha|\beta) = \emptyset$ where $\alpha|\beta$ is a ternary combination whose i -th component $(\alpha|\beta)_i$ is determined by

$$(\alpha|\beta)_i = \begin{cases} \alpha_i & \text{if } \alpha_i = \beta_i, \\ \emptyset & \text{if } \alpha_i \neq \beta_i. \end{cases}$$

DEFINITION 9.4. A ternary circuit C^T is called a *ternary representation* of a circuit C if $|Z^T| = |Z|$, $|Z_0^T| = |Z_0|$ and each T-function F_i is a normal extension (of arbitrary

type) of the corresponding Boolean function f_i .

DEFINITION 9.5. A ternary circuit has a *meta-stability anomaly* with respect to the set of variables Z' if, for some binary combination of input variables, there exists a meta-stable (sub)state γ reachable from some α and, in γ , there is at least one meta-stable variable $z_i \in Z'$.

THEOREM 9.2. Let C be a non-self-dependent or an antitone-self-dependent circuit covering some A_k -circuit and satisfying the acknowledgement stability condition (according to Definition 9.1). Then its ternary representation C^T has a meta-stability anomaly with respect to the set of acknowledge variables.

Proof. It is sufficient to prove the theorem just for an A_2 -circuit.

Consider the states $\alpha_2 = a_1 a_2 b_1 \bar{b}_2 \beta$ and $\alpha_3 = a_1 a_2 \bar{b}_1 b_2 \beta$ of the circuit C where β is an arbitrary combination of values of the remaining variables of the circuit.

As the acknowledgement stability condition holds we have $f_{b_1}(\alpha_2) \equiv f_{b_2}(\alpha_3) \equiv 1$ and $f_{b_1}(\alpha_3) \equiv f_{b_2}(\alpha_2) \equiv 0$ regardless of the value of β . In the ternary representation of the circuit, according to Observation 9.5 (3), we have $F_{b_1}(\alpha_2|\alpha_3) \equiv F_{b_1}(a_1 a_2 \emptyset \emptyset \beta) \equiv F_{b_2}(\alpha_2|\alpha_3) \equiv F_{b_2}(a_1 a_2 \emptyset \emptyset \beta) \equiv 0$ regardless of the type of the extension and independently of the values of the variables in β . As a consequence, the sub-state defined by $a_1 = a_2 = 1$ and $b_1 = b_2 = \emptyset$ is meta-stable. We should now demonstrate that such a sub-state is reachable from the binary states, using the proof of Theorem 9.1. Consider the states $\alpha_1 = a_1 a_2 \bar{b}_1 \bar{b}_2 \beta$ and $\alpha_4 = a_1 a_2 b_1 b_2 \beta$. Variables b_1 and b_2 are excited in α_1 and α_4 . Hence, in the ternary representation, the sub-state $a_1 a_2 b_1^\emptyset b_2^\emptyset \beta$ is reachable from α_1 and α_4 . *Q.e.d.*

From this theorem, we may deduce that any implementaion of an arbiter by means of binary logical circuitry will be susceptible to a meta-stability anomaly with respect to the acknowledgement variables.

Now we present a property that will give us a method for the search of meta-stable (sub)states.

The *excitation function* for a variable z_i is defined by

$$\phi_i(Z) = z_i^0 \overline{F_i^0(Z)} \vee z_i^\emptyset \overline{F_i^\emptyset(Z)} \vee z_i^1 \overline{F_i^1(Z)}$$

where $F_i(Z)$ is the inherent T-function of z_i in a ternary circuit. It is obvious that

- z_i is meta-stable in state α , if $\phi_i(\alpha) = 0$ and $\alpha|z_i = \emptyset$,
- z_i is idle in α , if $\phi_i(\alpha) = 0$ and $\alpha|z_i \neq \emptyset$,
- z_i is excited in α , if $\phi_i(\alpha) = 1$.

OBSERVATION 9.6. Let C^T be a ternary circuit and γ a (sub)state corresponding to the values of variables in $Z' \subset Z$. Then γ is meta-stable if, and only if, $\bigvee_{z_i \in Z'} \phi_i(\gamma) \equiv 0$ where $\phi_i(\gamma)$ is the value of the excitation function of z_i in the γ sub-state.

EXAMPLE 9.1. A simple non-self-dependent implementation for a two-input arbiter (with complemented request variables) can be given by the following asynchronous logical circuit

$$b_1 = \overline{\overline{a}_1 \vee b_2}, \quad b_2 = \overline{\overline{a}_2 \vee b_1}.$$

If we build a fragment of a state-transition diagram for $a_1 = a_2 = 1$, then we shall see that an oscillatory anomaly of the form $a_1 a_2 b_1 b_2 - a_1 a_2 \overline{b}_1 \overline{b}_2$ is possible for the acknowledgement variables.

Analysing the binary representation of this circuit shows that the state which is defined by $a_1 = a_2 = 1$ and $b_1 = b_2 = \emptyset$ will be meta-stable with respect to the acknowledgement variables and is reachable from the binary states $a_1 a_2 b_1 b_2$ and $a_1 a_2 \overline{b}_1 \overline{b}_2$ regardless of the type of the extension.

9.4 Designing correctly-operating arbiters

For the design of correctly-operating arbiters we can use elements whose behaviour, not being adequately expressed by means of binary logic, can be modelled by BT-functions.

There are two possible types of correctly-operating arbiters: “*bounded*” and “*unbounded*”. Arbiters of the first type have *bounded acknowledgement time* for the process requests, regardless of whether they come to the arbiter’s inputs separately, or together, with other requests. In other words, if at some instant in time, one, or several, requests arrive at the inputs of a “vacant” arbiter, then, after a finite time interval, bounded by some (linear) function of inherent delay values of the elements, an acknowledge signal to one of the requests will appear at the arbiter output. It seems that a possible approach to the design of a bounded arbiter should be based on the prevention of anomalies by the use of elements that do not behave like binary logical elements in transition processes. Oscillatory anomalies cannot, however, be

excluded with this method – they are independent of the character of the transition process. Meta-stability anomalies can be eliminated, for example, by using a *rectangular hysteresis element* (RHE) whose inherent function is described by the following BT-function

$$f(Z) = x^1 \vee x^2 z^1$$

where x is the input variable and z is the output variable of the element.

Fig. 9.6(a) illustrates a circuit for the input cascade of an arbiter with RHE. By means of methods described in the previous section, we can prove that this circuit is free from meta-stability anomalies.

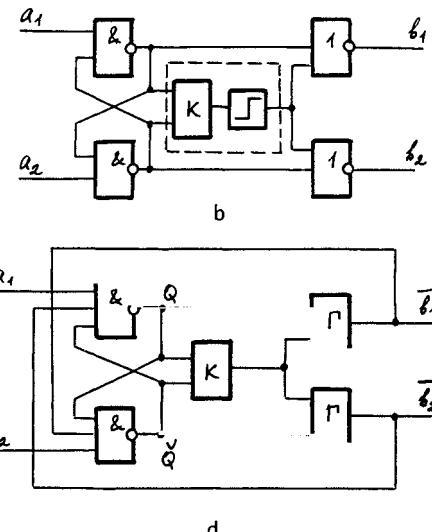
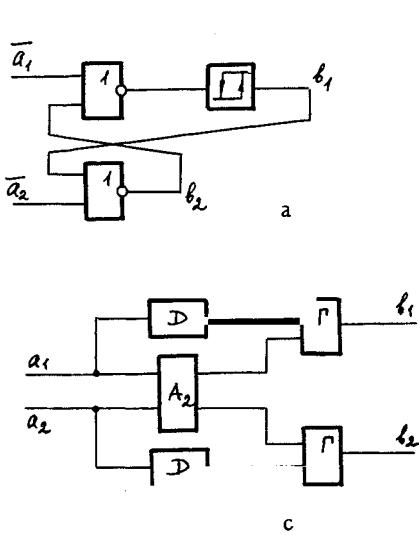


Figure 9.6(a) - (d). Implementation of arbiters: (a) using an element with rectangular hysteresis; (b) with Γ -flip-flops and delays; (c) with comparators; (d) with comparators and Γ -flip-flops.

The rise of oscillatory anomalies is, however, possible in the above circuit. Furthermore, since an ideal RHE is unrealizable, there is a non-zero probability that a meta-stable state may occur.

Using a *Schmidt trigger* instead of a RHE we can analyze the transition characteristics, for the circuit of Fig. 9.6(a), which reveal the existence of a meta-stable state.

Arbiters of the second (unbounded) type have undefined response times for simultaneous or “nearly” simultaneous requests. This means that the acknowledgement time is not defined by the delay values of the elements in such an arbiter.

The main idea for the construction of an unbounded arbiter consists in (a) locking the arbiter output signals while the circuit is in an anomalous state, and (b) allowing these signals to change only after the circuit has left the anomalous mode. The correctness of such an arbiter is attributable to objective physics – statistical laws – circuits leave an anomalous state after a, possibly long, but finite, time period having an upper bound.

The locking mentioned can be organized by built-in delays as shown in Fig. 9.6(b). To ensure a rather small probability that an anomaly will “skip” to the acknowledge outputs b_1 and b_2 we should make the value D of built-in delays tens, or even hundreds, of times greater than the delay of a normal transition process in the circuit. As a consequence, the overall speed of the arbiter falls by a corresponding factor. Note that, for the sake of correct operation of the arbiter, we should use Γ -flip-flops rather than AND gates, at the circuit's outputs.

Another technique ensuring the locking of outputs is based on the use of a threshold *comparator* which is used as a completion indicator for anomalous behaviour of the arbiter. Such an indication is possible because the arbiter output signals change in a synphase (coherent) way during an oscillatory anomaly (see Fig. 9.3(b)) and, as also happens in a meta-stability anomaly, they coincide with a certain accuracy at every moment during the anomalous state. The comparative threshold value must be sufficiently large in order to guarantee that a divergence of the arbiter output values exceeding the threshold value indicates the completion of the withdrawal from the anomalous state.

An input cascade circuit for such arbitration is shown in Fig. 9.6(c). Note that a slow transition process for such a solution – unlike the preceding solution – is invoked only under conditions of anomalous behaviour in the input flip-flop of the arbiter. The comparator and the threshold element are shown separately in Fig. 9.6(c) for clarity. They can, however, be implemented as a single element. The solution of Fig. 9.6(c) has a drawback in that the variable modelling the behaviour of the comparator is non-semi-modular. This allows short impulses to appear at the comparator's output in case of real (non-ideal) inertial delay in the comparator. Such pulses may reach the outputs b_1 and b_2 of the circuit, and to obtain a correct solution, a means of filtering them out should be provided. As a consequence of that drawback we have a circuit that is no self-checking with respect to a stuck-at-0 fault (see Section 10) at the comparator output. In the case of such a fault, the comparator ceases to provide the required locking function, and an anomalous condition may occur at the outputs b_1 and b_2 .

The drawbacks indicated can be avoided if, in the circuit of Fig. 9.6(c), we use Γ -flip-flops instead of the output gates shown. We should also connect the outputs of the Γ -flip-flops to the inputs of the elements of the input flip-flops, as shown in Fig. 9.6(d). A more attractive solution can be achieved through the idea of splitting the comparator into two symmetric sub-circuits, each of which can be implemented by one transistor.

Using the arbitration circuits indicated as a basis – these circuits are aperiodic in the sense that they are independent of the delays of their elements – we can build arbiters with arbitrary input disciplines, and multi-channel arbiters, that serve more than one pair of processes.

Multi-channel arbiters can be divided into two major groups: *centralized* arbiters and *decentralized (distributed)* arbiters. Arbiters of the first type can be implemented in the form of stand-alone components because they have a large number of internal interconnections that are not shared. Distributed arbiters are built using the idea that signals between composing modules are transmitted on a small number of common buses. Centralized arbiters can be built on the basis of tree structures, recursively, as shown, for example in Fig. 9.4, or using multi-state flip-flops. In contrast, the structure of distributed arbiters is likely to be cyclic as shown, for example, in Fig. 9.7.

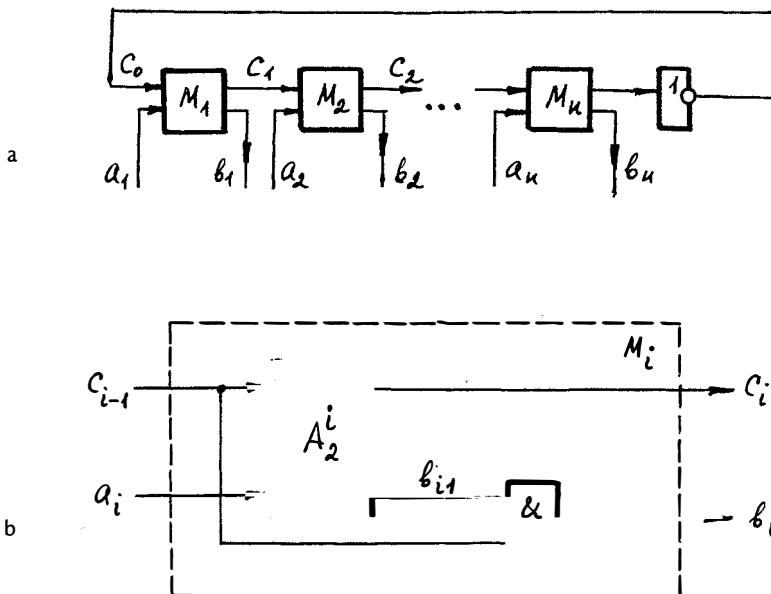


Figure 9.7(a) and (b). Distributed arbiter: (a) loop structure; (b) implementation of a module.

The circuit of Fig. 9.7(a) operates in the following way. Initially, $c_0 = 1$, $c_1 = c_2 = \dots = c_n = 0$. The arbitration function is thus carried out by the first module M_1 , which chooses between the “system's request”, c_0 and the external request a_1 , if any. If the system request has been chosen, then it proceeds to the second module in the form of $c_1 = 1$. Otherwise, the external acknowledge signal is set as $b_1 = 1$, and the value $c_1 = 1$ will be set only after the release of the external request $a_1 = 0$. As the system request continues to move, the second module is activated etc. After the last module, M_n has completed its arbitration function, we have $c_n = 1$ and then $c_0 = 0$. As a result, an “idling” wave will spread along the loop that will bring back all the internal requests, $c_0 = c_1 = c_2 = \dots = c_n = 0$, and the original system request c_0 will return to 1. In such an arbiter, the process of arbitration of external requests and the process of moving the system request, flow in parallel in the sense that for $c_{i-1} = 0$ and $a_i = 1$, the arbiter of the i -th module (see Fig. 9.7(b)) picks up the external request (by setting $b_{i_1} = 1$) and produces the external acknowledgement (by setting $b_i = 1$) immediately after the appearance of the respective system request (the setting of $c_{i-1} = 1$) without losing time on the arbitration process. As an A_2^i circuit, we may use, for example, arbiters of the type presented in Fig. 9.6.

To organize self-diagnosis in the arbiter, we may use a circuit with indication facilities. The setting of a certain value at the output of the indicator should block the serving of a request which is waiting but should not interfere with the process of serving the current request.

A circuit for an arbiter with self-diagnosis is presented in Fig. 9.8. The bus lines A and B are used for a wired-OR merging of all external requests and acknowledgements, respectively. The common indication signal \bar{I} is produced by complementing a signal on line B . The circuit of a module M_i is shown in Fig. 9.8(b). A stuck-at fault at the output of any element in the module will result in the absence of changes at the output of the indicator. Furthermore, if the fault has arisen in elements in the arm of the module which serves the system request, the above stoppage is due to the absence of signal changes on the system bus line.

Let us now consider techniques for designing devices that *distribute a common resource* between processes. An arbiter is their major part. We shall call such devices *allocators*.

The interaction between a process and a shared resource can be considered as consisting of two stages:

- 1) *acquiring an access* to the resource via the arbitration system, and
- 2) *resource handling*, as such.

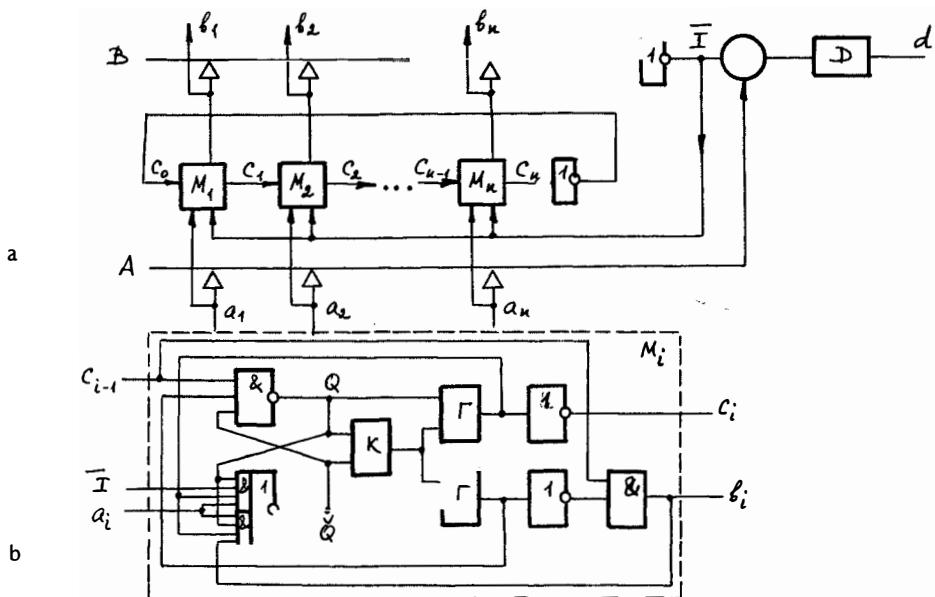


Figure 9.8(a) and (b). Self-diagnosing arbiter: (a) block diagram; (b) implementation of a module.

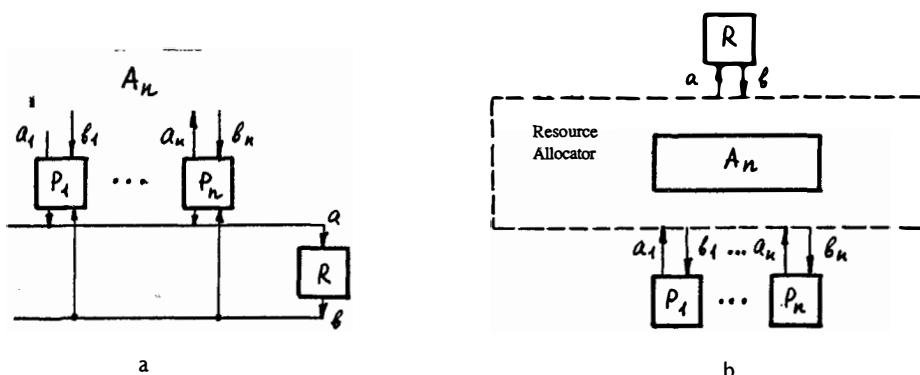


Figure 9.9(a) and (b). Resource allocation: (a) access through arbitration system; (b) using an allocator circuit with an arbiter.

There are two possible strategies for organizing such interaction. In the first, the system is assigned to only give permission for process work with a common resource, while both phases of the aperiodic resource (the idling and working phases) are organized by the process itself (see Fig. 9.9(a)). In this case the allocator is just a multi-channel arbiter. The advantage of such an approach is the relative simplicity of an allocator, whereas, the disadvantage is in a very small overlap between the arbitration process and the process of resource functioning.

In the second strategy, the allocator not only provides the arbitration between requests but also manages the interaction between a process and a resource, as shown in Fig. 9.9(b). In this case, each of the processes P_1, P_2, \dots, P_n , from the viewpoint of control, has access to the resource only via the allocator whose key component is the arbiter.

In Chapter 5 we saw a device for resource distribution that was called a multiple use circuit. That device could only operate for two consecutive requests which were not allowed to arrive at the same time (see Fig. 5.9). Fig. 9.10(a) shows a signal graph that describes the order of change of external signals in a multiple use circuit serving the i -th request. After the arrival of a request, a_i^+ , the circuit activates the working phase in the resource R^+ , then the idling phase R^- and only after that produces the acknowledgement, b_i^+ . The release of the request, a_i^- , causes the release of the acknowledgement, b_i^- , and does not “touch” the resource. At the moment of release the resource, we must not reset the information which was produced by the resource. The problem of preserving the information can be solved by using special switching networks which are inserted into breaks in the wires α_i and which control the latches where the information is stored. The serving of the next request is only possible after the release of the resource.

The multiple use circuit of Fig. 9.10(b) realizing the signal graph shown in Fig. 9.10(c) allows the resource idling phase and the processing of the acknowledgement b_i inside a request process to be launched in parallel. This is possible due to the introduction of an internal memory r_i into the circuit serving the i -th request. This additional memory flag indicates the fact that the resource has been released after the serving of the i -th request. The serving of the next request by the resource can be started only after the fact that the resource has been released from the previous request has been stored. In the circuit, the carrier of such information is the signal b_i .

To build a resource allocator capable of serving requests arriving concurrently, we should interconnect a circuit for an n -channel arbiter A_n and an n -channel multiple use circuit S_n . An example of such interconnection is shown in Fig. 9.11(a). Note that the conjunction of request a_i and signal d_i can be implemented directly by

elements of the arbiter A_n . When the circuit of Fig. 5.9 is used as an S_n circuit, the order of signal changes can be described by the signal graph shown in Fig. 9.11(b). Furthermore, if we decide to use a distributed arbiter as an A_n , the entire design will be distributed – there are no interconnections between separate cells of the multiple use circuit of Fig. 5.9.

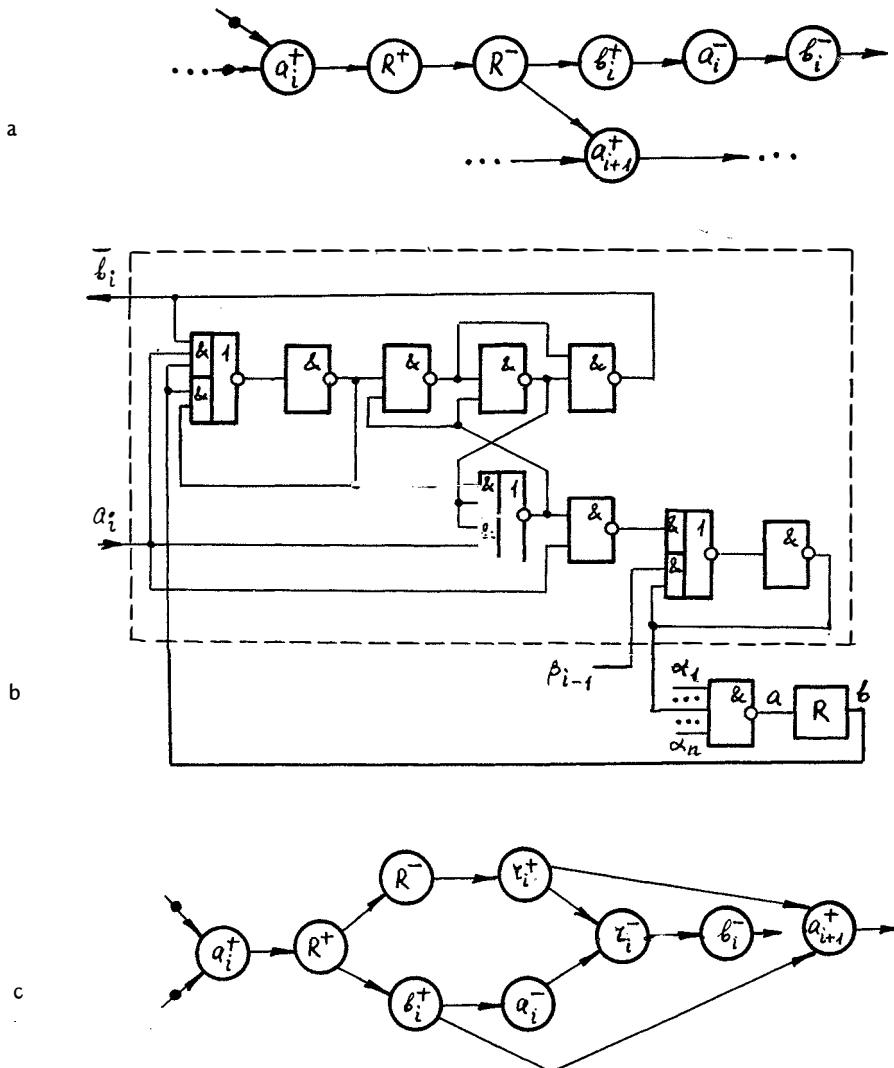


Figure 9.10(a) - (c). Resource allocation: (a) signal graph specifying the signalling order for the case using a multiple use circuit; (b) a variant of a multiple use circuit; (c) signal graph for the latter.

If we take the circuit of Fig. 9.10(b) as a multiple use circuit, then a signal graph shown in Fig. 9.11(c) will be realized. However, to have this device implemented in a distributed way, we must solve the problem of interconnection between the cells of the multiple use circuit. One of the potential variants will be such that through a wired-OR merging, the common signal β can be formed from signals β_i and this β is then applied to the β_{i-1} inputs of each i -th cell.

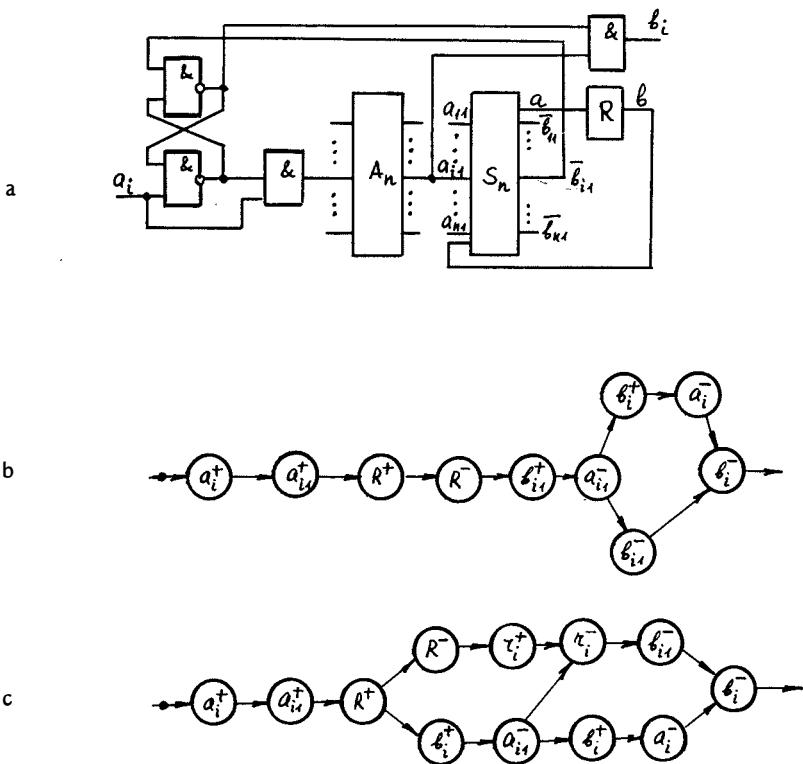


Figure 9.11(a) - (c). Resource allocation for serving parallel requests: (a) circuit; (b) signal graph for the case using the circuit of Fig. 5.9(b); (c) signal graph for the case using the circuit of Fig. 9.10(b).

9.5 “Bounded” arbiters and safe inertial delays

The problem of constructing a correct “bounded” arbiter correlates with the problem of implementing a safe inertial delay. This section presents a discussion of the difficulties which accompany the solution of these problems in practice.

A delay that occurs between the time that the input signal is applied to an element and the time that the output assumes a corresponding value can be one of the following two main types: a *perfect* delay and an *inertial* delay. The output of a perfect delay, at any time t , is an exact replica of its input at time $t - D$, where D is the magnitude of the delay. A perfect delay cannot, however, be physically reproduced, since a physical implementation cannot respond to input values which are lower than its sensitivity threshold. The behaviour of such a delay is modelled, approximately, by a transmission line.

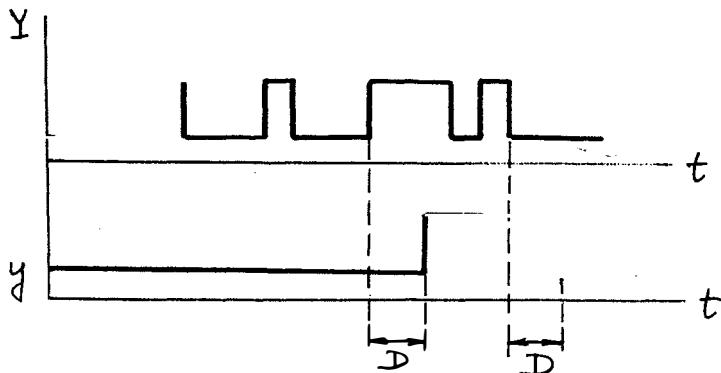


Figure 9.12. Behaviour of an ideal inertial delay.

The behaviour of an inertial delay is illustrated in Fig. 9.12. Signals with a duration less than D at the Y input are not “allowed” to the y output (they are filtered out), whereas a signal whose duration exceeds D will appear at the output with a time shift of the value of D . An ideal inertial delay can be implemented by a circuit using RC-filters and a majority function element M as shown in Fig. 9.13.

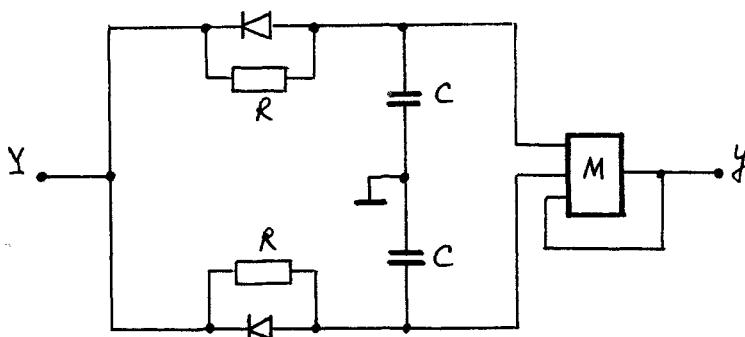


Figure 9.13. Implementation of an ideal inertial delay.

For any $\epsilon > 0$, however small, the ideal inertial delay should filter out signals with a duration $D - \epsilon$ and allow signals with duration $D + \epsilon$, regardless of the duration of the next signal, however small that may be. However, due to the reasons, already indicated, related to the sensitivity threshold, there is no adequate physical implementation of such a model. In other words, every inertial delay has some "uncertainty" range $[D - \epsilon, D + \epsilon]$ and it is impossible to predict how such a delay will respond to a signal with a duration within the bounds of this range.

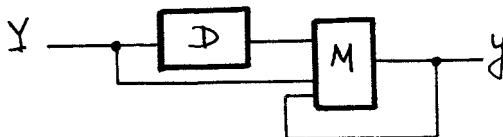


Figure 9.14. Model for an ideal inertial delay.

Fig. 9.14 presents the well-known Friedman's circuit which models the behaviour of an ideal inertial delay through a perfect delay D and a majority element M with zero delay. Input pulses ("zero" pulse — 1-0-1 — if $y = 1$, and "one" pulse — 0-1-0 — if $y = 0$) with duration less than D are filtered out under the condition that the time interval between successive pulses is not less than D . Hence, the behaviour of an inertial delay is dependent on the input signal change disciplines. This is due to the inertia intrinsic to the physical elements — a circuit always requires some time "to prepare itself" for filtering another short pulse (this is the so-called relaxation time), after filtering the current pulse.

The above considerations give rise to the introduction of another model of inertial delay that, being adequate for the ideal characteristics, will also incorporate the idea of an uncertainty range and input disciplines.

We introduce the following parameters of inertial delays.

1. *Input discipline.* A sequence of alternating binary signals having low and high logical levels is applied to the input of an inertial delay. It can be interpreted as a sequence of alternating positive and negative pulses (see Fig. 9.12) of the form $\tau_0^+ \tau_1^- \tau_2^+ \tau_3^- \dots$ or $\tau_0^- \tau_1^+ \tau_2^- \tau_3^+ \dots$ where τ_i^+ and τ_i^- are durations of positive and negative pulses, respectively. An input discipline for the inertial delay imposes certain constraints on the allowed inter-relations between the durations of successive pulses. For example, the input discipline for the circuit of Fig. 9.14 imposes the following constraint on input sequences: if $\tau_i < D$ then $\tau_{i+1} > D$.

2. *Filtration thresholds.* The positive real numbers D_m^+ and D_m^- are called filtration thresholds if positive pulses of duration $\tau^+ < D_m^+$ and negative pulses of duration $\tau^- < D_m^-$ are passed to the output.

If we assume that at the input of the delay, after the τ_i^+ (τ_i^-) pulse, there may arrive, not only the binary signal τ_{i+1}^- (τ_{i+1}^+) but also a signal τ_{i+1}^\emptyset having some intermediate amplitude, then it is natural to deduce that if $\tau_{i+1}^\emptyset < D_m^-$ ($\tau_{i+1}^\emptyset < D_m^+$) then this signal will be filtered out. No restrictions are, however, needed in the handling of longer pulses with intermediate amplitude.

3. Pass thresholds. The positive real numbers D_M^+ and D_M^- are called pass thresholds if positive pulses of duration $\tau^+ > D_M^+$ and negative pulses of duration $\tau^- > D_M^-$ are passed to the output without any distortion, or with an allowed timing distortion. For the evaluation of the timing distortion we may use the largest possible, relative (δ) or absolute (Δ), distortion allowed for the signal passed. Thus, if an input pulse τ_i is transformed to a pulse τ'_i at the delay output, then $|\tau'_i - \tau_i| < \Delta$ or $|\tau'_i - \tau_i| < \delta \cdot \tau_i$.

The intervals $[D_m^+, D_M^+]$ and $[D_m^-, D_M^-]$ will be called the uncertainty ranges for positive and negative pulses, respectively.

4. Delay values. These are the values D^+ and D^- which define the length of a shift for a passed positive and negative signal, respectively. It is obvious that $D^+ \geq D_m^+$ and $D^- \geq D_m^-$ – the delay cannot decide whether to pass a pulse to the output or not until it determines, in real time mode, whether $\tau^+ \geq D_m^+$ or $\tau^- \geq D_m^-$. Hence, a signal change on the delay output cannot start earlier than D_m^+ (or D_m^-) after the corresponding input change has occurred.

Note that, in the general case, the parameters D_m^+ , D_m^- , D_M^+ , D_M^- , D^+ and D^- may vary in time and may also be the functions of input sequences.

Therefore, inertial delays have three types of timing ranges. They are filtration ranges, uncertainty ranges and pass ranges. If an input sequence corresponding to the input discipline is applied to an inertial delay, the pulses from the filtration ranges are not passed to the output, while the pulses from the pass ranges pass to the output with a delay, but with durations preserved (within a required accuracy).

DEFINITION 9.6. An inertial delay will be called *safe* if every pulse having a duration within a given uncertainty range ($D_m^+ \leq \tau^+ \leq D_M^+$ or $D_m^- \leq \tau^- \leq D_M^-$) and applied to the input of the delay in accordance with some input discipline is either filtered out or passed to the output with delay but without distortion (within a given accuracy level).

If we apply a pulse, of a duration within an uncertainty range, to the input of a non-safe delay, a strongly distorted signal may occur at the output of the delay. In particular, such a signal may be a short parasitic glitch whose appearance, as has

already been mentioned, may result in a fault and an occurrence of an anomaly in the circuit behaviour.

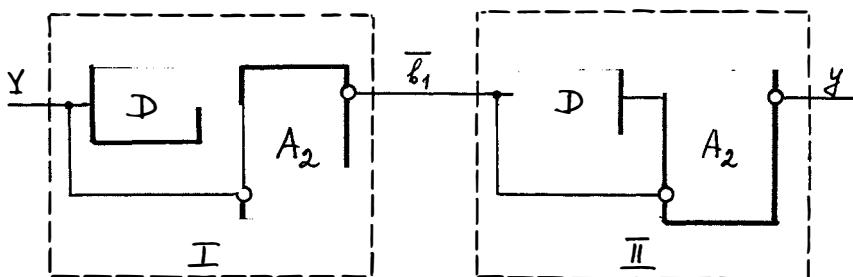


Figure 9.15. Ideal inertial delay based on an arbiter.

Consider the circuit in Fig. 9.15 consisting of two identical cascades, each of which is composed of a delay and an arbiter. The second request input and the first acknowledgement output of each arbiter are complemented. Furthermore, only the first acknowledgement output is used in the circuit. For the sake of simplicity, we shall assume the delays and the arbiters of the cascades to be identical. Let d be a perfect delay and A_2 an ideal non-inertial arbiter. Then it is clearly seen that the circuit of Fig. 9.15 implements an ideal inertial delay which filters out short pulses of duration less than d , provided that the time interval between successive short pulses is longer than d . The pulses whose length is longer than d are passed without distortion but with a $2d$ -delay. Note that positive short ($\tau^+ < d$) pulses are filtered out by the first cascade, while the negative ones ($\tau^- < d$) are filtered out by the second cascade. The pulses whose duration is equal to d may, in a non-deterministic way, be either filtered out or allowed to pass with the $2d$ -delay.

If we use as a d delay a non-safe inertial delay with the filtration thresholds such that $d_m^+ = d_m^- = 0$, and take a bounded arbiter as an A_2 , the construction introduced will be an implementation of a safe inertial delay, but with different parameters.

It is known that a bounded arbiter can be implemented using ideal inertial delays. Safe inertial delays can also be used. Thus, the problems of synthesizing a bounded arbiter and a safe delay are *equivalent* in the sense that having one solved, the other may be solved immediately⁽¹⁾.

(1) The problem of designing a *bounded synchronizer*, a synchronizer with an upper bound on the time of transition processes, is also equivalent to that of designing a bounded arbiter and a safe delay.

We conclude this section by analyzing certain difficulties that may be met in trying to implement a bounded arbiter.

An idea for constructing an arbiter with a bounded acknowledgement time delay could be as follows. In a similar way to that of Fig. 9.6(c) we should indicate the anomalous behaviour of the input cascade, but in doing so, we should not only lock the outputs of the arbiter for the whole duration of the anomaly, but also affect the inputs of the arbiter by blocking one of the requests. The aim of this is to obtain a faster recovery from the anomalous state.

Consider the circuit shown in Fig. 9.16(a) which differs from that of Fig. 9.6(c) only in having an additional AND-OR-NOT element, locking the a_2 request, and a pair of non-safe inertial delays at the outputs of the arbiter.

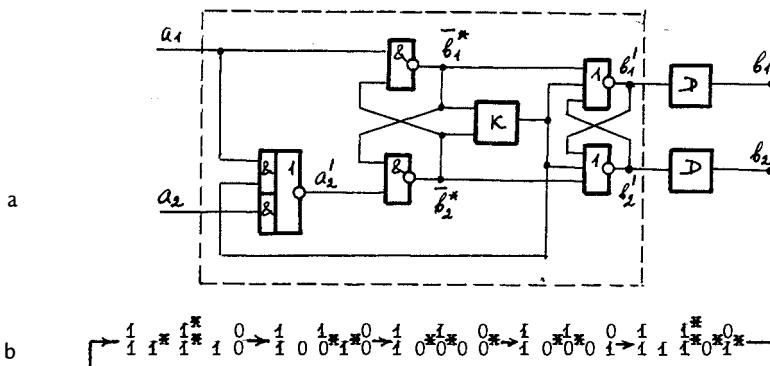


Figure 9.16(a) and (b). An attempt to realize an arbiter with bounded response time: (a) circuit; (b) state-transition diagram.

Furthermore, the a_2 request is applied in the complemented form (although this complementing could also be arranged inside the arbiter). The state-transition diagram of the circuit contains two regions: the region of normal operation and the hazard region with respect to the acknowledge variables. The latter in turn consists of a section of static and dynamic hazard⁽¹⁾ with respect to variables b'_1 and b'_2 , the duration of which is bounded by the switching time of several elements of the circuit, and, of a section of dynamic hazard of unlimited duration with respect to b'_2 . The latter point is due to the fact that the length of the anomalous behaviour in flip-flop (b'_1, b'_2) is unpredictable. It is possible that the (b'_1, b'_2) flip-flop leaves an

⁽¹⁾ A static hazard is considered here to be a possible appearance of a single pulse, while a dynamic hazard stands for several successive pulses, at the outputs of elements b'_1 and b'_2 . In the region of normal operation, the variables b'_1 and b'_2 are hazard-free.

anomalous state at the same time instant as an indication of anomalous behaviour occurs and the second request begins to be released under the effect of the feedback signal. The comparator is capable of indicating this situation, and the second request will be revived. The (b_1^*, b_2^*) flip-flop may enter another anomalous behaviour loop and so on. The fragment of a transition diagram corresponding to the described operation is presented in Fig. 9.16(b) (the variable values are spatially arranged in states in the same way as the elements are positioned in Fig. 9.16(a)).

As can be seen from this diagram, the feedback signal facilitates an inner “sub-motion” of the anomalous behaviour in the (b_1^*, b_2^*) flip-flop. The circuit can, therefore, enter an oscillation mode that will have an undefined duration even for fixed input values. A hazard of a limited duration can be filtered out because the arbiter operates in a “request-acknowledge” fashion.

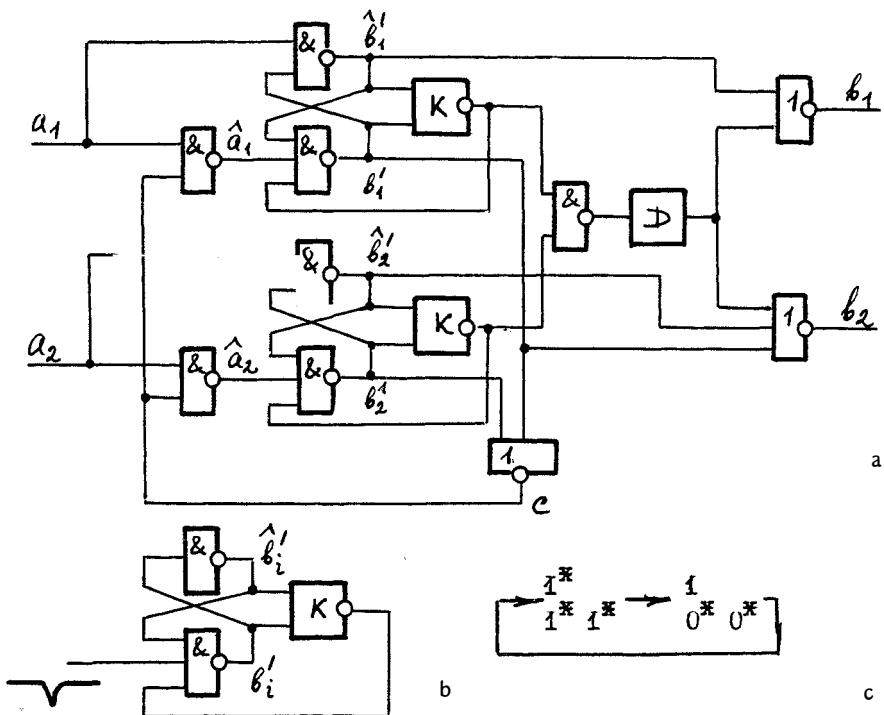


Figure 9.17(a) - (c). Another attempt to implement an arbiter with bounded response time: (a) circuit; (b) a circuit fragment; (c) transition diagram demonstrating a possibility for the fragment to enter the generation mode.

However, the existence of the unlimited hazard section with respect to b_2 makes the circuit of Fig. 9.16(a) be regarded as incorrect.

Similar difficulties arise when using another method for composing an arbiter, say, by introducing an input register (latch) for storing requests and self-synchronizing, as shown in Fig. 9.17(a). If two requests arrive at the inputs a_1, a_2 within a short time interval, then it is possible that the second one, say, a_1 , arrives at the input at the moment when the arbiter inputs are being locked (cut-off) by signal $c = 0$ resulting from the effect of the first request (in this case, a_2). The (\dot{b}_1', b_1') flip-flop may then enter an anomalous state under the effect of a possible short negative pulse at the input. If the indication of an anomalous state, by the comparator, occurs at the time when the flip-flop changes its state from the anomalous one to $(1, 0)$, then the section of the circuit, picked out separately in Fig. 9.17(b), may enter the oscillation mode shown in Fig. 9.17(c), due to the effect of the feedback from the output of the comparator. As a consequence, a dynamic hazard of unlimited duration may arise at the arbiter output b_2 .

Thus, several arguments proving the impossibility of pushing a circuit out of an anomalous state have been presented here. These practical arguments are confirmed by theoretical results.

It can, therefore, be stated that the approach for constructing arbiters which was presented in Section 9.4 is the only possible way to design arbiters with correct functioning independent of element delay values.

9.6 Reference notations

The difficulties arising in solving arbitration and synchronization problems were probably first reported in 1966 [210] and [256]. Many later works have been published showing experimental results of anomalous behaviour in various devices (mostly flip-flops) [214], [215], [216], [219], [250], [251] and [282]. Statistical treatment of the data obtained, carried out by some authors, shows that anomalous behaviour of arbiters and synchronizers is one of the major sources of faults in computer systems [219], [251] and [262]. For example, the probability of a synchronization failure in an interrupt handling circuit was evaluated in [251]. For a frequency of 1 MHz and a gate delay of 10 ns, an average of up to four system failures per second may be expected. It is clear that the higher the operating frequency of the system, and the greater its dimension (the number of synchronized signals), the higher the probability of a failure.

From valuable experimental data it was possible to make recommendations concerning the architecture of computer systems that aimed at a lower probability of failures caused by the anomalous behaviour of system elements [251] and [297].

A series of works were dedicated to proposals for various arbiter and synchronizer designs [213], [216], [228], [251], [268], [280], [281], [290] and [310]. The circuit with a Schmidt trigger shown in Fig. 9.6(a) was proposed in [310]. Later, in [212], this solution was shown not to be free from anomalous operation modes and recognized by the original author of the circuit [311]. An attempt to build a bounded arbiter was made in [290] using the circuit solution shown in Fig. 9.12(a). However, as has been demonstrated in Section 9.5, this solution is not correct. Methods for designing multi-channel arbiters were described in [217], [218], [242] and [286]. In particular, those with different priority disciplines can be found in [287].

The proof that the occurrence of oscillatory and meta-stability anomalies cannot be completely avoided in binary logic implementations of arbiters, and a discussion of methods for designing correctly operating arbiters, was also presented in [55]. It was based on using non-binary logical elements whose behaviour could be given in terms of Boolean-ternary logic [6] and [312].

The proof of the impossibility of implementing non-deterministic state machines within the class of non-self-dependent circuits was presented in [127]. However, it is arguable whether the concept of a circuit implementing a non-deterministic state machine [127] is an adequate model for arbiters and synchronizers.

The concept of a safe inertial delay introduced in Section 9.5 is quite close to that of an ideal inertial delay from [296]. The circuits for inertial delays shown in Fig. 9.7 were proposed in [230] (also see [5]). Analysis of various constructions of inertial delays can be found in [259]. The circuit of Fig. 9.10 was proposed in [296]. The inter-relation between the problem of building ideal arbiters and synchronizers having a bounded acknowledgement time with the problem of building an ideal inertial delay was reported in [207] and [296].

The fact that, for a wide class of dynamic systems, a short enough input sample can be chosen, that when applied to the system, will bring it to an anomalous state, has been reported in [258]. This result is a strong argument in favour of the impossibility of implementing a bounded synchronizer, an ideal arbiter and a safe inertial delay.

CHAPTER 10

FAULT DIAGNOSIS AND SELF-REPAIR IN APERIODIC CIRCUITS

James Cook, of Newark, pulled from the water a certain Miss Miller, who, with her puppy, nearly drowned. Shortly afterwards, Mr Cook received a postal order for one pound sterling as an "encouraging award" from the Life Saving Society. Recently, Mr Cook received another cheque, this time for ten pounds. This was an award from the Animal Life Society for saving Miss Miller's dog.

News Agency Report

In the early stages of the development of the theory of aperiodic circuits it looked as if their advantages over traditional circuits were mainly concerned with increase in speed due to their ability to produce a completion signal after each operation and, thereby, to function independently of variations of operation time lengths. Such variations are possible because of tolerances, aging of components or other effects. Today, it increasingly becomes the case that the *decisive advantage of aperiodic circuits is the relative simplicity of failure diagnosis in them*. From the interpretation of completion signals in aperiodic circuits, it can be seen that if some element fails to operate when it is excited, the completion signal will never indicate termination of the operation containing this element. The circuit is thus made to stop. This stop condition and the failure of the element are correlated in the sense that the latter is interpreted as an increase, to infinity, of the element delay, which can be modelled by the so-called stuck-at fault model. A *stuck-at fault* in a circuit manifests itself as a fixed, "constant-0" or "constant-1", signal at the output of the element.

The type of fault is governed by a number of factors. Firstly, it is governed by the site of the occurrence of the fault. The theory of aperiodic circuits deals with self-diagnosis and self-repair problems only as far as faults at the output are concerned; in general, it provides no treatment of faults inside the element or at the inputs. Secondly, by the behaviour of the failed element. We consider stuck-at faults of both directions for which the equation $z_i = f_i(Z)$ describing the behaviour of the element before the occurrence of the fault, is then replaced by either $z_i = 0$ or $z_i = 1$. Thirdly, by the multiplicity of faults in a circuit. In the fourth place, it is by the time of the occurrence of the fault, or, more exactly, by the state of the element at the time that a fault occurs at its output. According to this, last, attribute, all stuck-at faults are classified into one of two types: conservative faults and mutable faults.

Conservative faults are faults of the form $z_i = \sigma$, $\sigma \in \{0, 1\}$. At the time of their occurrence, the z_i element has, at its output, the signal value corresponding either to its idle state σ or to its excited state $\bar{\sigma}(z_i \neq f_i = \sigma)$. The moment of the occurrence of such a fault is unnoticed by the circuit because, at this moment, the fault does not have any effect upon the circuit operation – the faulty value σ is either equal to the output value of a non-faulty element z_i , or to the value of the inherent function f_i of the element; the detection of such a fault in an aperiodic circuit will be done later, at the time when the z_i element is switched.

Mutable faults show themselves in the following way: at the occurrence time of the fault $z_i = \sigma$, the output of the z_i element has the idle state $\bar{\sigma}(z_i = f_i \neq \sigma)$. This fault causes spontaneous switching of the element and, hence, affects the circuit behaviour at the time of its occurrence. Mutable faults may result in the appearance of glitches at the outputs of faulty elements that will, in turn, be the causes for anomalous conditions in the circuit.

Circuits that are capable of indicating faults of certain classes during the on-line operation are called *self-checking* circuits. The concept of self-checking is based on the partitioning of input and output signal values into classes. For such a partition, the responses of the circuit to the same class of input values, for faulty and non-faulty circuits, will belong to different output value classes. Self-checking circuits are usually equipped with special devices, also self-checking, that are capable of detecting what class an output value belongs to. Any fault in a circuit having such a built-in checker is detected at the circuit's outputs.

It is quite common to use the following encoding techniques for the outputs of a *built-in checker* (we consider the case of the two-output checker). The pairs 01 or 10 are generated in the case of normal operation, while in the presence of faults of a given class in a circuit, the checker's outputs will either be 00 or 11. A drawback of this encoding style is the rise of critical races in the 01-10 (or 10-01) transitions. During these transitions, the pairs 00 or 11 appear for a short time at the outputs of a correct circuit, thereby, giving false evidence about the faults. This problem cannot be fully overcome in a traditional approach.

The theory of totally self-checking circuits is, in essence, only developed for the class of synchronous circuits. The use of the term “totally self-checking” is, however, not particularly accurate since the faults of the synchronization (clock) mechanism cannot be detected.

In synthesizing asynchronous designs of totally self-checking circuits we should solve some serious problems related to achieving, in addition to self-checking, an avoidance of critical races in a circuit. Attempts to tackle these two problems separately result in rather clumsy designs.

Aperiodic circuits, which are free from a clock mechanism and critical races, are, as will be shown later, totally self-checking. Studying structural properties of aperiodic circuits is thus important, not only for circuit analysis as such, but also from the point of view of solving some problems in functional diagnostics. As aperiodic circuits exhibit a totally self-checking quality with respect to conservative faults, we have methods of self-detection and self-repair, both oriented towards these faults. The development of more general methods of self-diagnosis, or, even more, of self-recovery, for the cases of mutable faults appears to be rather difficult.

10.1 Totally self-checking combinational circuits

Unsuccessful attempts to design totally self-checking asynchronous circuits can probably be attributed to underestimation of certain specific features of such circuits. These features require

- a revision of the well-known definitions of fault-tolerance, self-testing, and total self-checking, and
- a more careful examination of the issues concerning variable encoding and allowed disciplines of changes of code values; these questions are closely linked with the problem of the elimination of logical hazards in asynchronous circuits.

Thus, a fundamental relationship should exist between the concept of indicability (see Section 4.3) and that of total self-checking. Clearly, to be totally self-checking an asynchronous circuit must be provided with input code combinations that are changed in accordance with a particular transition discipline, say, a two-phase discipline.

In order to discuss totally self-checking asynchronous circuits, we shall stay within the framework of the basic definition of an allowed transition (Definition 3.6) and the notation introduced in Section 4.3.

Assume that a correctly operating circuit implements the system of functions $F(X)$. A fault p manifests itself in such a way that the combinational circuit implements another system of functions $F_p(X)$ for which $F_p(X) \neq F(X)$.

DEFINITION 10.1. An asynchronous combinational circuit is called *totally self-checking* with respect to faults in the class P , if for all $p \in P$ the following conditions are satisfied:

- (a) for every allowed transition $a-b$, either

$$F_p(b) = F(b) \quad (10.1)$$

or

$$F_p(b) \notin L \quad (10.2)$$

holds;

- (b) for every allowed transition $b-a$, either

$$F_p(a) = F(a) \quad (10.3)$$

or

$$F_p(a) \notin K \quad (10.4)$$

holds (Conditions (a) and (b) constitute the condition for a circuit to be *fault-secure*);

(c) there exists at least one allowed transition $a-b$ or $b-a$ for which condition (10.2) or (10.4) is satisfied (the condition for a circuit to be *self-testing*).

As can be seen from this definition, a fault may be detected, not at the instant it occurs, but only in the next phase of circuit operation (for a two-phase discipline). This should not, however, be regarded as a violation of the total self-checking principle, as the notion of an “instant” or a “beat” of circuit operation should stem from such an operational unit as a full two-phase cycle.

With the use of Theorem 4.3 (indicability) we can now approach the total self-checking property for combinational circuits.

THEOREM 10.1. *Indicatable (aperiodic) combinational circuits are totally self-checking with respect to single and multiple stuck-at faults of circuit elements.*

Proof. Let there be a fault in some element of a circuit whose inputs X are indicated to the outputs Y . Let the fault be stuck-at-0, without loss of generality.

Consider two cases:

(i) If the faulty element does not have to switch during an allowed transition $a-b$, then condition (10.1) of Definition 10.1 is fulfilled. If this element has to switch from 0 to 1, then either $F_p(b) = F(a)$, or $F_p(b) = F(c)$ where $c \in [a, b]$ or $F_p(b) = h$, where $h \in [k, l]$, and according to Definition 4.3, we have $F(c) \notin L$, $F(a) \notin L$, $h \notin L$. However, in this case condition (10.2) of Definition 10.1 is fulfilled.

(ii) If the faulty element is essential, then there exists at least one allowed transition of the form $a-b$ or $b-a$ where this element should switch. We can assume, without loss of generality, that this transition is $a-b$ for the chosen fault (stuck-at-0). Then for any fault of such a type, there can be found an allowed transition such that $F_p(b) = F(a)$, or $F_p(b) = F_p(c)$, $F_p(b) = h$, where $h \in [k, l]$. Hence, $F_p(b) \notin L$ and condition (a) of Definition 10.1 will be fulfilled, thus meaning that the combinational circuit is totally self-checking.

Similar reasoning can be applied to the cases of a stuck-at-1 fault and multiple faults. *Q.e.d.*

Note. If there is no branching at circuit inputs, or if faults may only occur at branching points of input connections, as is usually the case when pin connections are broken, then stuck-at faults at circuit inputs can also be detected within the conditions of the above theorem.

10.2 Totally self-checking sequential machines

To define totally self-checking sequential machines (circuits with memory elements) we should generalize the indicatability notion for this class of circuits. This has been partly done, however, in Section 4.8 (Definition 4.7). In addition, a Mealy model machine will be considered as a composition of a Moore model machine and a combinational circuit. The general case of a specification for the machine will thus be the equation system $F(X, Y)$.

In formulating the definition for a self-checking state machine, we should bear in mind that, in case of a fault $p \in P$, an internal state transition $c-d$ cannot terminate, and hence, instead of d , the last to appear will be a symbol $\delta \in (c, d]$, while during the transition $d-c$ a symbol $\gamma \in (d, c]$ will be the last. Therefore, in case of a fault, the record $F(b, d)$ is replaced by $F_p(b, \delta)$ while the record $F(a, c)$ is replaced by $F_p(a, \gamma)$.

DEFINITION 10.2 An asynchronous Moore machine will be called *totally self-checking* with respect to faults in class P if for all $p \in P$, the following conditions are satisfied:

- (a) for every allowed transition $a-b$, either

$$F_p(b, \delta) = F(b, d) \quad (10.5)$$

or

$$F_p(b, \delta) \notin D \quad (10.6)$$

holds;

- (b) for every allowed transition $b-a$, either

$$F_p(a, \gamma) = F(a, c) \quad (10.7)$$

or

$$F_p(a, \gamma) \notin C \quad (10.8)$$

holds (Conditions a and b constitute the general condition for a machine to be *fault-secure*);

- (c) there exists at least one allowed transition of the form $a-b$ or $b-a$ for which the corresponding condition, either (10.6) or (10.8), is satisfied. (This is the condition for the machine to be *self-testing*.)

Note that in a machine any fault may manifest itself not “at the occurrence instant” but during the next operational phase in the two-phase discipline.

For aperiodic machines, we now present a statement that is analogous to Theorem 10.1.

STATEMENT 10.1. *Aperiodic state machines are totally self-checking with respect to single and multiple stuck-at faults of circuit elements.*

The proof follows directly from combining Definition 4.8 and Definition 10.2.

The fault detection mechanism is necessarily connected with the method for encoding input and output symbols by self-synchronizing codes that were discussed in Chapter 3. A time interval during which a circuit should complete a transition can be determined from calculating the operational speed of the circuit through those of its composing elements. Thus, the absence, after that time interval, of an output combination belonging to the class D (for a transition of the form $a-b$) or to the class C (for a transition of the form $b-a$) can be interpreted as the presence of faults in the circuit.

Both Theorem 10.1 and Statement 10.1 have fundamental importance. They establish a relationship between problems of engineering diagnostics and those of the theory of aperiodic automata. The key result of such a relationship is the possibility of using circuit analysis methods (see Section 4.3 and Chapter 8) for proving a circuit to be totally self-checking in the sense of Definitions 10.1 and 10.2. The results presented in this book contribute to the solution of certain problems of the theory of totally self-checking circuits, as well as to the problem of undetectability of faults in synchronization mechanisms. Conventional approaches usually imply that a checked circuit is first designed without regard for diagnostic requirements, and then the design is augmented, in a somewhat artificial way, with various mechanisms that ensure the checkability of the circuit, by splitting the sets of input and output values into classes. Finally, the circuit is connected to the checker which detects whether a given value belongs to one or the other class. By contrast, aperiodic design methodology ensures checkability by the circuit design itself, thus simplifying the methods for synthesizing both self-checking circuits and checkers.

10.3 Fault detection in autonomous circuits

The concept of totally self-checking asynchronous combinational circuits and state machines defined in Section 10.1 and 10.2 can now be extended to the case of specifying modelling circuits with the aid of the Muller model.

While a transition process in an *open-loop* (non-autonomous) circuit terminates in some deadlock state, a correct *autonomous* circuit should never enter deadlocks. A

stuck-at fault in a circuit given by a Muller model amounts to a fixation of a variable, associated with the faulty element, in an original system of equations.

EXAMPLE 10.1. Let a circuit be given by the system

$$z_1 = \bar{z}_3, \quad z_2 = \bar{z}_1, \quad z_3 = \bar{z}_2,$$

and its full transition diagram was shown in Fig. 8.4. When the fault $z_1 = 0$ occurs, the system assumes the form

$$z_1 = 0, \quad z_2 = \bar{z}_1, \quad z_3 = \bar{z}_2,$$

and its transition diagram will be of the form presented in Fig. 10.1.

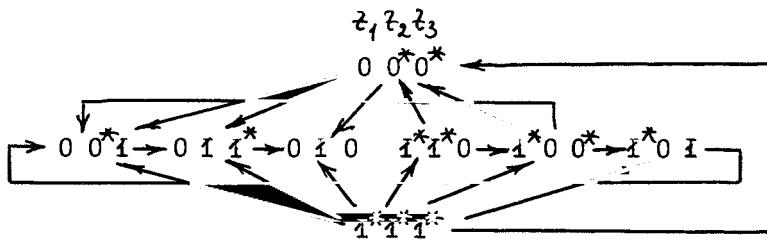


Figure 10.1. State-transition diagram for a circuit built as a three-inverter loop when one of the inverters has a stuck-at fault.

This fault has no effect upon the operation of elements z_2 and z_3 but since the 010 state is deadlock the fault will manifest itself after a series of changes of the circuit element's values. In this case, the 111-010 transition may require up to six changes.

Thus, a natural manifestation of a fault presents itself through the presence of deadlock states in a transition diagram of the faulty circuit.

To define a totally self-checking autonomous circuit we adopt the following notation.

$R(\alpha)$ is the set of operational states containing state α (in other words, the operational cycle with state α).

R is the set of infinite operational cycles of a circuit (contains all the states that belong to at least one of the sets of operational states in it, each of which is an infinite sequence of states).

T is the union of the sets of finite sets (containing deadlocks) of operational states of a fault-free and a faulty circuit (the latter is formed through transformation of a transition diagram of a fault-free circuit for a given fault (fixation of a variable)).

The sub-script p , as earlier, will denote corresponding notations for a faulty circuit. For a pair $R(\alpha), R(\beta) \in R$ we shall write $R(\alpha) \neq R(\beta)$ if these cycles do not coincide.

DEFINITION 10.3. An autonomous circuit with operational cycles in R is called *totally self-checking* with respect to faults in the class P if for all $p \in P$ the following conditions are satisfied:

(a) for every operational cycle $R(\alpha) \in R$, either

$$R_p(\alpha) = R(\alpha) \quad (10.9)$$

or

$$R_p(\alpha) \notin R \quad \& \quad R_p(\alpha) \in T \quad (10.10)$$

holds (the condition of a circuit to be *fault-secure*);

(b) there exists at least one operational cycle $R(\alpha) \in R$ for which condition (10.10) will be satisfied (this is the *self-testing* condition of the circuit).

Therefore, the fault-tolerance condition requires that either the operational cycles of fault-free and faulty circuits should be equal, or the operational cycle of the faulty circuit should not coincide with any infinite operational cycle of the corresponding fault-free circuit, and should, at the same time, be finite (contain a deadlock).

We now investigate the fault classes with respect to which semi-modular circuits are totally self-checking. In doing so, we need several more notations.

DEFINTION 10.4. A stuck-at fault p will be called

(i) *exitory* for the set R if, when it occurs, the circuit enters a state ω such that $\omega \notin T$ and $\omega \notin R$;

(ii) *substitutional* for the set R with operational cycles $R(\alpha)$ and $R(\beta)$ if there exists a state $\gamma \in R_p(\alpha)$ such that $\gamma \in R(\beta)$.

An exitory fault differs from a substitutional one in that it “knocks out” the circuit from the region where it has been analyzed. For a substitutional fault the state-transition diagram may contain some infinite cycles that have not, so far, been accounted for. Thus, an exitory fault may actually become substitutional.

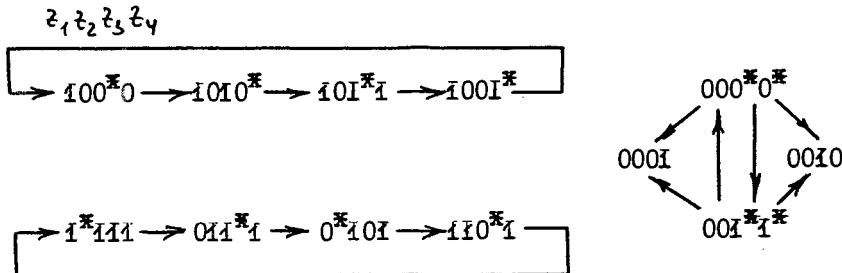


Figure 10.2. Illustration of substitutional and exitory faults.

EXAMPLE 10.2. Let the behaviour of a circuit be given by the state-transition diagram of Fig. 10.2 which corresponds to the following equation system

$$\begin{aligned} z_1 &= z_1 \bar{z}_2 \vee z_2 \bar{z}_3, & z_2 &= z_2, & z_3 &= z_1 z_2 \vee \bar{z}_4, \\ z_4 &= z_2 \vee z_1 z_3 \vee \bar{z}_1 \bar{z}_3. \end{aligned}$$

For $z_2 = 1$ and $z_2 = 0$, this system is reduced to

$$z_1 = \bar{z}_3, \quad z_2 = 1, \quad z_3 = z_1 \vee \bar{z}_4, \quad z_4 = 1$$

and

$$z_1 = z_1 \vee \bar{z}_3, \quad z_2 = 0, \quad z_3 = \bar{z}_4, \quad z_4 = z_1 z_3 \vee \bar{z}_1 \bar{z}_3,$$

respectively.

For $z_1 = z_2 = 0$, we obtain

$$z_1 = 0, \quad z_2 = 0, \quad z_3 = \bar{z}_4, \quad z_4 = \bar{z}_3.$$

For the set R of operational cycles $R(1000)$ and $R(1111)$, both stuck-at faults of z_2 are substitutional – for $z_2 = 1$, the circuit shifts from the $R(1000)$ operational cycle to the $R(1111)$ operational cycle, rather than to a deadlock, and for $z_1 = 0$, the operational cycles are shifted in the reverse order.

The $z_1 = z_2 = 0$ fault for these two operational cycles appear to be exitory because the circuit, in this case, leaves the set R but does not come to a deadlock.

It is clear that for exitory faults the $R_p(\alpha) \in T$ condition does not hold, nor does the condition $R_p(\alpha) \notin R$ and $R_p(\alpha) \in T$, for the substitutional ones.

Therefore, circuits with exitory and substitutional faults are generally not diagnosable.

It can be shown that in a semi-modular circuit having a fake equivalence class (see Definition 8.3) states α in $R(\alpha) \in R$ and β in $R(\beta) \in R$ that are reachable from states ϕ and ψ , for which we have $\phi \Rightarrow \alpha$ and $\psi \Rightarrow \beta$, can always be found and, furthermore, α and β do not belong to a fake class. A fake class is not a complete sequence. Hence, we may notice that if some part of the operational cycle $R(\alpha)$ forms a fake class in which variable z_s is excited but does not switch, then, according to Definition 10.3, the circuit with the $z_s = 1$ fault will not be totally self-checking. In fact, in the presence of that fault, states from the fake class may alternate infinitely, and the circuit will never reach a deadlock. Obviously, in this case, the fault is not detectable. Therefore, in the following we shall only consider semi-modular circuits with no fake classes.

Note that the presence of fake classes in state-transition diagrams accounts for the presence of such elements in a circuit whose inputs are isolated from the remaining elements, i.e. the elements whose outputs are closed to their inputs, or the elements which are generators of 0 or 1. The consideration of such circuits is not of any interest as they can, apparently, be simplified.

EXAMPLE 10.3. The circuit $z_1 = 1$, $z_2 = \bar{z}_3$, $z_3 = z_2$ contains the constant 1 generator, and its transition diagram, shown in Fig. 10.3, has a fake equivalence class of the form {000, 010, 011, 001}.

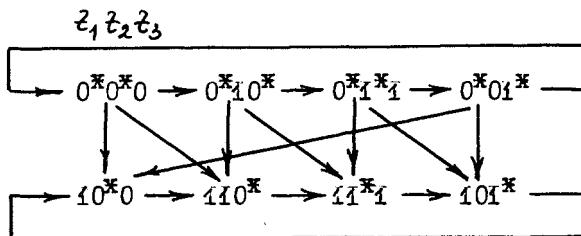


Figure 10.3. A circuit with a fake equivalence class.

THEOREM 10.2. An autonomous circuit with infinite operational cycles $R(\alpha), R(\beta), \dots, R(\lambda)$, which is semi-modular with respect to $\alpha, \beta, \dots, \lambda$, is totally self-checking with respect to non-exitory and non-substitutional for R , single and multiple, stuck-at faults of elements, if it contains no fake equivalence classes.

Proof. If the set R contains several operational cycles, then in some of them, say, in $R(\alpha)$, there can be found some non-live elements, i.e. elements which do not switch in this cycle. The stuck-at faults of such elements which do not coincide in their values with the signals on the outputs of these elements cannot manifest themselves in the $R(\alpha)$ cycle, and, hence, condition (10.8) will be fulfilled. If, by contrast, the element is live in $R(\alpha)$, then both non-exitory and non-substitutional faults will result in the situation that the signal will be constant at its output. Since the circuit is semi-modular, and the set of its operational states contains no fake classes, this set should have a state ϵ in which the only excited element is the faulty one. But, due to the fault, this element cannot switch, and thus the ϵ state will be deadlock. Hence, condition (10.9) of fault -tolerance will be fulfilled.

If the circuit has no element which is non-live in all operational cycles (such an element would be useless and should be removed from the circuit), condition (b) of Definition 10.3 (self-testability) will also be fulfilled. Thus, the circuit will be totally self-checking. *Q.e.d.*

EXAMPLE 10.4. Consider the circuit whose behaviour is described by the state-transition diagram of Fig. 10.4.

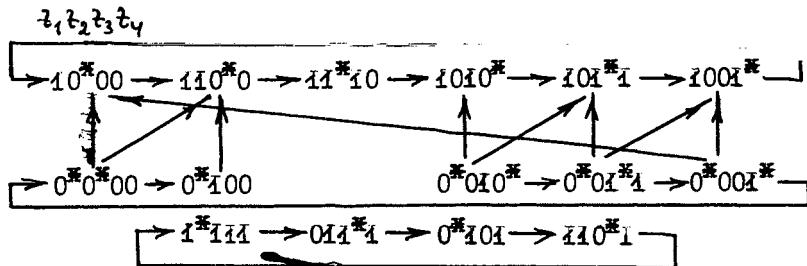


Figure 10.4. Example of a state-transition diagram with two operational cycles.

This diagram corresponds to the system of equations

$$z_1 = \bar{z}_2 \vee \bar{z}_3 \vee \bar{z}_4,$$

$$z_2 = z_2 z_4 \vee \bar{z}_3 \bar{z}_4,$$

$$z_3 = z_1 z_2 \vee z_3 \bar{z}_4,$$

$$z_4 = z_2 z_4 \vee \bar{z}_2 z_3.$$

Let $R(1000)$ and $R(1111)$ be operational cycles. For the $z_1 = 1$ fault the 1111 state will be deadlock (self-testability). The $z_1 = 0$ fault also manifests itself after $R(1111)$ – the deadlock state is 0101. The $z_1 = z_2 = 0$ is exitory for $R(1000)$ and for $R(1111)$, although having been given yet another cycle $R(0000)$, we can detect that fault because the circuit defined by the above equation will enter the deadlock state.

It should be pointed out that in the proof of Theorem 10.2 we have essentially used the fact of the semi-modularity of a correct circuit with respect to states in the R set. If the circuit is by no means semi-modular, then a stuck-at non-exitory and non-substitutional fault may be undetectable.

EXAMPLE 10.5. A fault in one of the inverters in the non-semi-modular circuit defined by the system of equations

$$z_1 = \bar{z}_3, \quad z_2 = \bar{z}_3, \quad z_3 = z_1 \vee z_2$$

does not make the circuit stop. In fact, the $z_1 = 0$ fault results in the diagram of Fig. 10.5 in which the operational cycle is infinite. The system in this case has the form

$$z_1 = 0, \quad z_2 = \bar{z}_3, \quad z_3 = z_2.$$

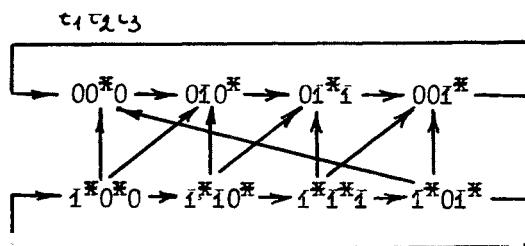


Figure 10.5. Example of a semi-modular circuit with undetectable stuck-at fault.

The analysis of diagnostic properties of autonomous circuits can, generally speaking, be carried out in two major directions. The first is checking a circuit for given faults (in the class of stuck-at faults) and given operational cycles. Such an analysis can use the same methods as those used for the ordinary analysis of the operational cycle, and is thus, relatively easy. The major goal of the analysis, in this case, is proving that some deadlock state will be reachable in a faulty circuit.

The second approach is somewhat similar to the complete analysis of a circuit. It involves finding all infinite operational cycles with respect to the states in which the circuit is semi-modular, and determining all finite operational cycles of the circuit.

Then, using the data obtained, we may elicit all exitory and substitutional faults. These types of faults are especially dangerous as, with respect to them, semi-modular circuits are not totally self-checking. Finding the fixations of variables which result in such faults allows the indication of the most "vulnerable" points of a circuit which should be "strengthened" by some technological techniques. The computational complexity of the required algorithms for such an approach is quite high. Techniques, more efficient than constructing a complete transition diagram and searching through all possible combinations of stuck-at faults of elements, have not yet been devised.

Even more complex, and less investigated, is the analysis of the detectability of faults that result in a change in the topology of a circuit, for example, bridging and breaking of wires, breaking of a wire after a branching point (a break before a branching point can amount to a stuck-at fault of the element output), and short-circuiting between inputs and outputs of elements. The most probable of such faults can also be studied by the techniques presented in Chapter 8, in much the same way as was proposed for stuck-at faults. For some of the faults indicated above, autonomous circuits may appear to be totally self-checking in certain cases.

10.4 Self-repair organization for aperiodic circuits

The *self-repair* (sometimes called self-recovery) process may be seen as a three-step procedure:

1. *Detect a fault* manifesting itself by an operational failure in circuit functioning.
2. *Localize a fault* by finding the place where the fault has occurred with a degree of precision, for example, to a module of a regular structure.
3. *Repair of a faulty circuit* by substituting a spare module for the faulty one.

The self-repair thus results in the restoration of the capability of a circuit to operate correctly. In order to resume the process that the circuit was executing before the occurrence of the fault, the circuit should be reset to the initial state, or into the latest monitored state, and the operation restarted.

It will be seen that the first of the above three steps, the fault detection, is realized by the techniques described earlier in this chapter. A result of such fault diagnosis will be the production of a fault signal $d = 1$.

Two methods can be suggested for self-repair organization. These methods are oriented towards systems and circuits having a regular structure. In the first method, when a fault occurs, a faulty module is replaced by a spare module. The same spare module can be used as a substitute for any faulty main module. This method is, therefore, called the *direct replacement redundancy* method. A circuit for organizing self-repair of single faults on the base of a common spare module is

shown in Fig. 10.6. This circuit can be used for a regular pipeline or path structure in which each i -th element is connected to the preceding $(i - 1)$ -th element and the succeeding $(i + 1)$ -th element. The block diagram can be easily generalized to arbitrary interconnections between elements of the structure.

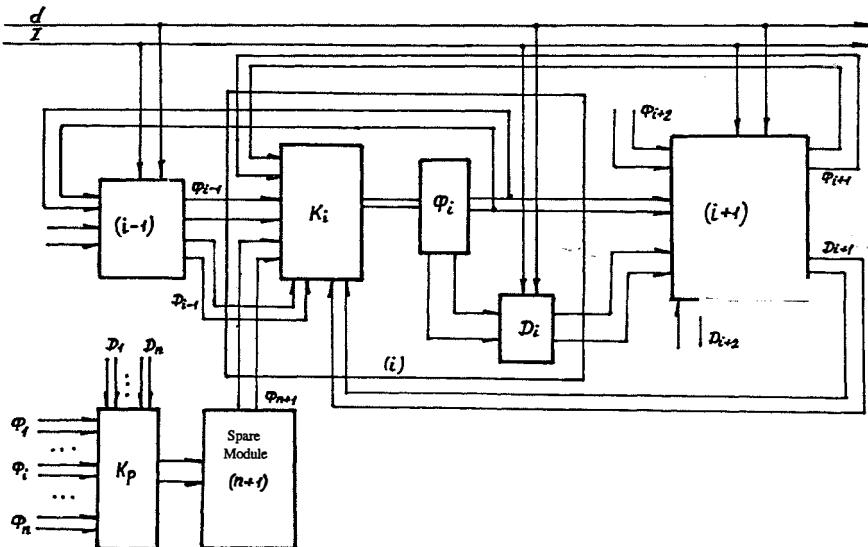


Figure 10.6. Organization of self-repair with direct replacement redundancy.

Each module consists of a functional unit Φ_i , a fault localization detector D_i and a multiplexer K_i that provides the interconnection between the inputs of the functional unit Φ_i and the outputs of a spare unit Φ_{n+1} instead of those of Φ_{i-1} if a fault has been localized in the $(i - 1)$ -th module , or instead of those of Φ_{i+1} , if a fault has been in the $(i + 1)$ -th module. The control of multiplexing is carried out by the outputs of detectors D_{i-1} and D_{i+1} . The repair also incorporates the provision of interconnections between the inputs of a spare module and the outputs of those modules which were linked with the inputs of a faulty module. This function is implemented by the K_p multiplexer at the inputs of the spare module. It is controlled by signals from detectors D_1, \dots, D_n indicating the information about fault localization. The correctness of self-repair organization is achieved due to the fact that the multiplexers are attached to the inputs of functional units. Therefore, any faults occurring in the i -th module, both in its functional part and its multiplexing part, can be cured by virtue of the adjacent modules.

Fig. 10.7 illustrates a circuit for the basic one-bit cell of a self-repairing binary counter. It contains a functional unit, the complementing flip-flop built of elements u_i , \check{u}_i , q_i , \check{q}_i , p_i , \check{p}_i , a localization detector consisting of a modulo 2 adder (for signals p_i and \check{p}_i) with output \bar{d}_i and a fault flag flip-flop (T_i , \check{T}_i). The multiplexer of the cell outputs is combined with the functional unit and is implemented directly by elements u_i and \check{u}_i . Each one-bit cell of the counter has one input connection \check{p}_{i-1} , the carry signal from the preceding cell. Hence, the multiplexer should implement one switch function $a_i = \check{T}_{i-1} \check{p}_{i-1} \vee T_{i-1} \check{p}_g$.

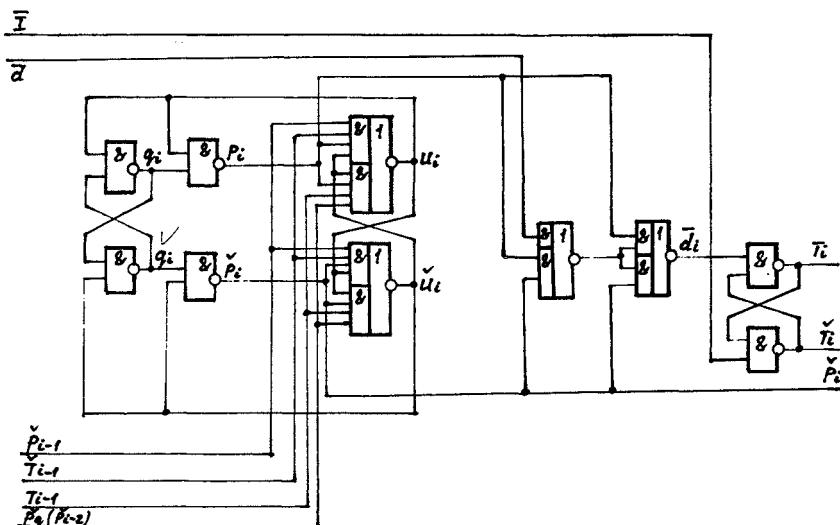


Figure 10.7. Cell circuit for self-repairing counter.

Hence,

$$u_i = \overline{p_i^{\checkmark} \dot{u}_i a_i} = \overline{p_i^{\checkmark} \dot{u}_i \overset{\checkmark}{T}_{i-1} \overset{\checkmark}{p}_{i-1}} \vee \overline{p_i^{\checkmark} \dot{u}_i T_{i-1} \overset{\checkmark}{p}_g}$$

and

$$\check{u}_i = \overline{u_i a \check{p}_i} = \overline{u_i^{\vee} T_{i-1} \check{p}_{i-1}} \vee u_i^{\vee} T_{i-1} \check{p}_e$$

The circuit for a spare one-bit cell consists of a complementing flip-flop and an input multiplexer realizing the function

$$a_g = T_1 \dot{p}_0 \vee T_2 \dot{p}_1 \vee \dots \vee T_n \dot{p}_{n-1}$$

where \dot{p}_0 is the input of the counter.

This circuit exhibits the following behaviour. If the counter is free from faults, the signal $\bar{d} = 1$ inhibits operation of (switching processes in) all localization detectors and each fault flag flip-flop (T_i, \dot{T}_i) is in the stable state (0, 1). Furthermore $a_i = \dot{p}_{i-1}$, as in an ordinary counter, and the input of a spare cell is locked ($a_g = 0$).

If a fault occurs in the i -th cell, a fault detection system activates the $\bar{d} = 0$ signal. This results in the switching of a local detector, $\bar{d}_i = 0$, and the fault flag flip-flop (T_i, \dot{T}_i) changes to the (1, 0) state. As a consequence, the multiplexing functions of the $(i+1)$ -th and the spare cells assume the forms $a_{i+1} = \dot{p}_g$ and $a_g = \dot{p}_{i-1}$ respectively. The repair process is thus carried out, which results in the situation when the spare one-bit cell with the \dot{p}_g output is introduced into the circuit instead of the i -th basic cell. If necessary, after the fault recovery, the (T_i, \dot{T}_i) flip-flop can be reset to its original state by a special reset signal $\bar{I} = 0$.

Note that if the i -th cell of the counter is faulty, then either $a_i = 0$ corresponds to $p_i = \dot{p}_i = 1$ or $a_i = 1$ to $p_i \neq \dot{p}_i$. The state of the spare cell before it is connected into the circuit, is $a_g = 0, p_g \neq \dot{p}_g$. If such a connection takes place when $a_i = 0$, then this cell does not change its state, and a_{i+1} immediately becomes equal to \dot{p}_g . If the connection is performed when $a_i = 1$, then the spare cell changes its state, and after this $a_{i+1} = 1$. Therefore, when being interconnected with the basic circuit, the spare cell has its operational phases in compliance with the phases of the basic cells of the counter. As a consequence, the transition process in the counter that has been interrupted by the fault, will successfully terminate, and after the completion indication signal, the signal \bar{d} will be reset to 1, which is the indication of the self-repair process completion.

It can be argued that the process is correct from the informational point of view because the data written in the spare cell may be different from what should be stored in the i -th cell. Therefore, after the completion of the self-repair the computations should be repeated.

The above methodology allows a self-repair to be enabled for any single conservative fault. Since localization detectors start their operation only after the appearance of a fault in a circuit (in either its functional or multiplexing part), then any single conservative faults in localization detectors have no effect on the circuit

operation. Hence, in the above technique none of the localization detector faults are self-repaired.

At the same time, any faulty setting (T_i, \check{T}_i) to the $(1, 0)$ state, with no fault in the i -th cell itself, may result in the connection of a spare cell into the circuit, in parallel with the normal transition processes in the counter, that may then be followed by a malfunction that cannot be detected. In other words, some single mutable faults in localization detectors may result in an incorrect operation. This difficulty can be eliminated if we implement the following switch function of a multiplexer $a_i = \check{d}' p_{i-1} \vee d' (\check{T}_{i-1} \check{p}_{i-1} \vee T_{i-1} \check{p}_g)$ where d', \check{d}' are the values of the fault flag flip-flop outputs which store the fact of producing the fault detection signal $\bar{d} = 1$. In such an implementation, any faults in localization detectors (single and multiple, conservative and mutable), provided that there are no faults in functional or multiplexing units of the same modules, have no effect on the correct operation of the functional part of the self-repairing counter.

Another method for self-repair organization is based on using the so-called *by-shift replacement redundancy* technique. In this method, when a fault in a circuit occurs, the faulty module is bypassed, or shunted, by some logical facilities in much the same way as a member of a list is deleted from that list, and the linear size of the structure is restored by the connection to it of a spare module. The block diagram for such an organization for the case of a regular linear circuit is shown in Fig. 10.8.

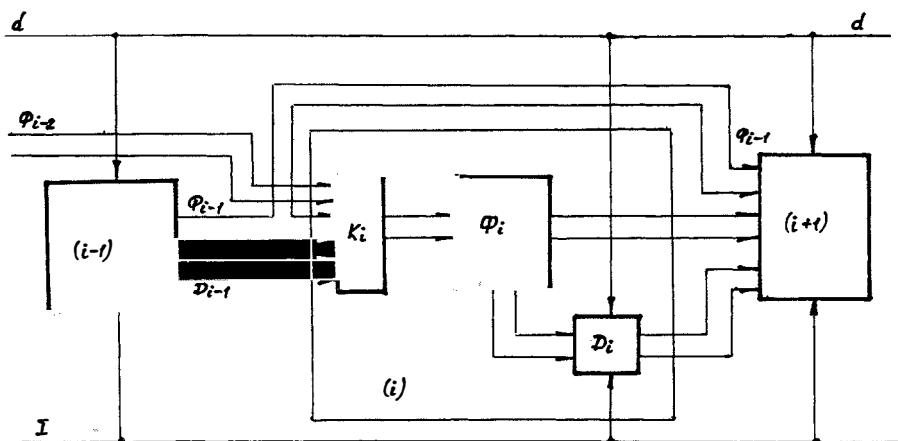


Figure 10.8. Organization of self-repair with by-shift replacement redundancy.

In normal operation, the inputs of each i -th module are linked only with the outputs of the $(i - 1)$ -th one. When a fault in the $(i + 1)$ -th module occurs, after the

localization of that fault, the multiplexer K_i connects the outputs of the $(i - 2)$ -th module to the inputs of the i -th module. For the example of the counter considered the K_i multiplexer implements a single function $a_i = \dot{T}_{i-1} \dot{p}_{i-1} \vee T_{i-1} \dot{\dot{p}}_{i-2}$. The circuit of a self-repairing counter with the by-shift replacement redundancy is similar to that of Fig. 10.7, with the slight difference that the signal $\dot{\dot{p}}_{i-2}$ rather than $\dot{\dot{p}}_g$ is applied to one of the inputs of the i -th cell.

To generalize the self-repairing method for by-shift replacement redundancy to a wider class of systems, we need several definitions.

1) A *regular structure* is assumed to be a structure with identical modules.

2) A regular structure is called *one-dimensional* if there exists an enumeration of modules such that for any pair of connected modules $a_i \rightarrow a_{i+k}$ or $a_i \leftarrow a_{i+k}$ there can be found a sequence of connected modules of the form $a_i \rightarrow a_{i+1} \rightarrow \dots \rightarrow a_{i+k-1} \rightarrow a_{i+k}$ or $a_i \leftarrow a_{i+1} \leftarrow \dots \leftarrow a_{i+k-1} \leftarrow a_{i+k}$.

3) A *chain* is a sequence of modules $A = \dots a_{i_1}, a_{i_2}, \dots, a_{i_k}, \dots$ such that every two adjacent modules $a_{i_l}, a_{i_{l+1}}$ of that sequence are connected to each other as $a_{i_l} \rightarrow a_{i_{l+1}}$ and separated from each other by the same number of modules (are at the same distance from each other) $|i_1 - i_2| = |i_2 - i_3| = \dots = |i_{k-1} - i_k|$.

4) A chain will be called *forward* if $i_l < i_{l+1}$, and *backward* if $i_l > i_{l+1}$.

5) The *action area of a module* $a_{i_l} \in A$ for the chain A is the set $\{a_j : i_l < j \leq i_{l+1}\}$, if A is a forward chain, and the set $\{a_j : i_{l+1} \leq j < i_l\}$, if A is a backward chain.

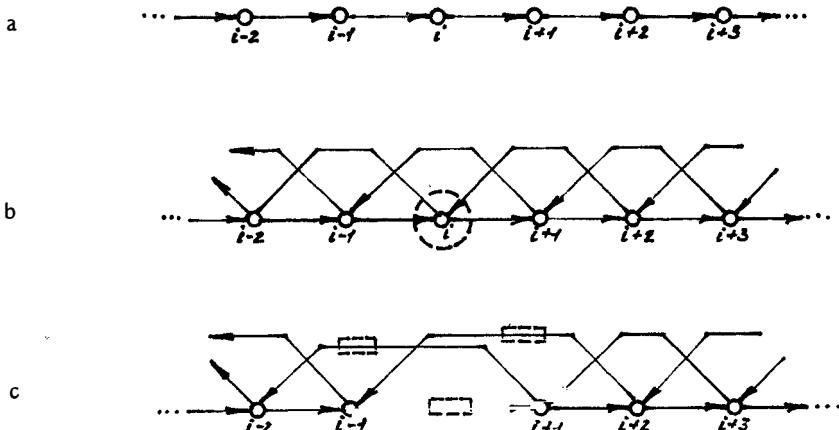


Figure 10.9(a) - (c). Possible reconfigurations in one-dimensional structures when a faulty element is deleted.

EXAMPLE 10.6. Fig. 10.9 illustrates examples of one-dimensional structures, the first of which is linear (see Fig. 10.9(a)). Fig. 10.9(b) involves three chains $A_1 = \dots a_{i-2}, a_{i-1}, a_i, a_{i+1}, a_{i+2}, \dots$; $A_2 = \dots a_{i+3}, a_{i+1}, a_{i-1}, \dots$ and $A_3 = \dots a_{i+2}, a_i, a_{i-2}, \dots$ where A_1 is a forward chain, and A_2 and A_3 are backward chains. The action area of a_i for A_1 is $\{a_{i+1}\}$, and for A_3 is $\{a_{i-1}, a_{i-2}\}$. The structure that can be obtained by the removal of a_i is shown in Fig. 10.9(c) where the dashed boxes stand for three new links.

The removal of a faulty module a_i during the self-repair process in the one-dimensional structure requires reconfiguration of the connections to the inputs of all modules belonging to the action area of the removed module a_i . This self-repair method can also be applied to other one-dimensional structures.

Figure 10.10(a) illustrates possible modifications of a two-dimensional regular structure for the case of the removal of a faulty module a_7 .

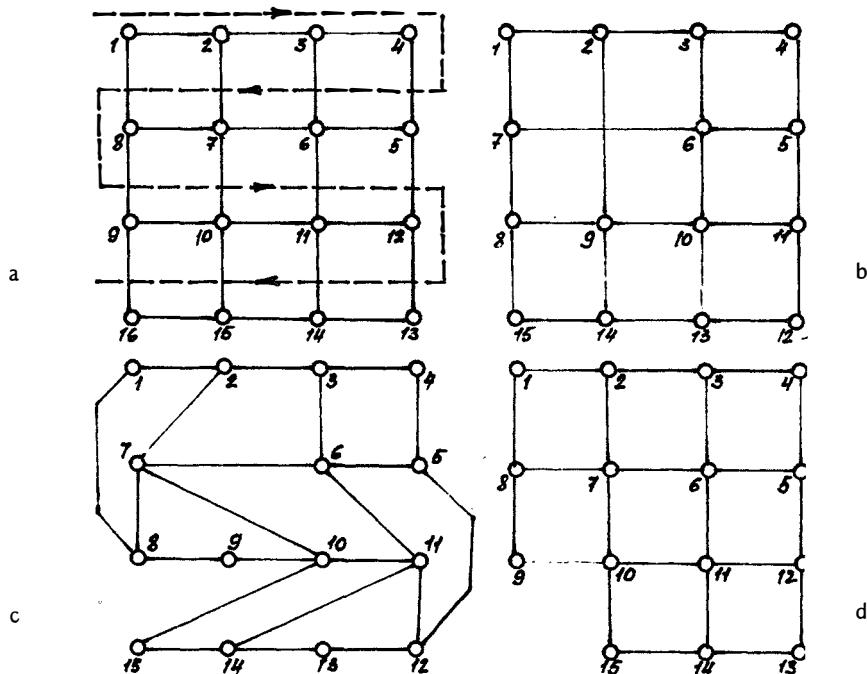


Figure 10.10(a) - (d). Possible reconfigurations in two-dimensional structures when a faulty element is deleted.

Figure 10.10(b) shows the result of a simple removal, whereas the removal, with preservation of the connection topology, is presented in Fig. 10.10(c). An equivalent representation of the last structure is shown in Fig. 10.10(d).

A way of choosing a particular reconfiguration method depends on the semantics of the computational structure. The self-repair of multiple faults is, of course, possible, although it leads to rather complicated constructions, and it is therefore, not constructive to present them here.

10.5 Reference notations

The concept of a totally self-checking circuit was proposed in [209]. Methods for the synthesis of such circuits are discussed in a large number of works, among which the most impressive are [160], [194], [196] and [302]. They contain the main definitions and properties of self-checking circuits. A survey of works in this field is presented in [157]. An example of using a parity-check encoding for constructing totally self-checking circuits can be found in [79] and [159]. The problem of synthesizing built-in checkers for combinational circuits whose outputs are encoded by the “ m out of n ” code are tackled in [194] and [261], while [156] and [196] are for the code with identifier (Berger's code). Similar encodings are used for synthesizing aperiodic combinational circuits [6].

The problems of constructing self-checking circuits with memory are, on the whole, less developed. The fundamental structural peculiarities of such circuits were first investigated in [302]. A formal definition of totally self-checking state machines, based on the results of [302] are given in [227] and refined in [222].

Particular cases of synthesis techniques for totally self-checking circuits were discussed in [2], [111], [112], [221], [226], [234], [252], [277], [289] and [294].

The most comprehensive presentation of problems related to self-checking (synchronous) circuits can be found in [130] and [157].

Attempts to synthesize asynchronous circuits in such a class were reported in [151] and [222], but the separation of solving the problems of hazard avoidance and those of fault detectability predetermined that the proposed solutions would be rather cumbersome. The fact that aperiodic (or semi-modular) circuits facilitate fault diagnosis is indicated in [6], [60], [120], [234] and [252]. The fault detection method is based on the engineering solution protected by patent certificates [35].

We see the use of aperiodic circuitry as a rather promising methodology in the construction of highly reliable fault-tolerant systems. Some general questions related to the design of such systems were discussed in [1], [67], [85], [118], [141], [186] and [188].

CHAPTER 11

TYPICAL EXAMPLES OF APERIODIC DESIGN MODULES

The vehicles were almost uninteresting to me, perhaps, because at the front armour of each of them the inventor, translucent from the inspiration, was tediously explaining the structure and purpose of his beloved creation. No-one seemed to be listening to these inventors and they, for their part, did not seem to be very keen to be heard.

A. Strugatzky, B. Strugatzky.

This chapter presents a discussion of various aperiodic circuit solutions for basic modules typically incorporated into any computer organization. The operation descriptions for these solutions are, as a rule, given in a short, informal manner. The aim is not a reduction of the length of the text, but a desire to expand the audience of potential readers to a large body of professional engineers. As before, the speed of the circuits described, will be estimated by the number of logical elements switching in series in both phases of circuit operation; the *mean switching time of an element will be denoted by T* . The complexity of circuit designs can be assessed in terms of the number of inputs and outputs of the composing elements. The choice of such metrics stems from the fact that it generally reflects the number of transistors required for the design when applied to the case of using an n -channel MOS technology. However, the corresponding assessment figures are no presented here, as they can easily be obtained by the reader.

Let us recall that *flip-flops built of pairs of arms may be denoted by a single letter, for example, by \dot{q} or by \hat{q} which helps the reader to be familiar with a particular transient state used in a given flip-flop, namely the double-one state or the double-zero state, respectively.*

The constructions of the most widely used flip-flops have already been presented in Section 4.4. The description of larger aperiodic modules starts from the *JK*-flip-flop whose prototype has gained such wide application that the authors simply could not pass it by.

11.1 The *JK*-flip-flop

A *JK*-flip-flop operates in the following way. When $J = K = 1$ it works as a *T*-flip-flop, and when $J \neq K$ it works as an *RS*-flip-flop where $J = S$ and $K = R$.

When $J = K = 0$ it stores the same value as has been assumed in the previous working phase.

The circuit for an aperiodic JK -flip-flop is shown in Fig. 11.1. It consists of the master flip-flop \dot{q} , a slave flip-flop \dot{y} , gates \dot{p}' and p , and the circuit which indicates the *transition process completion*.

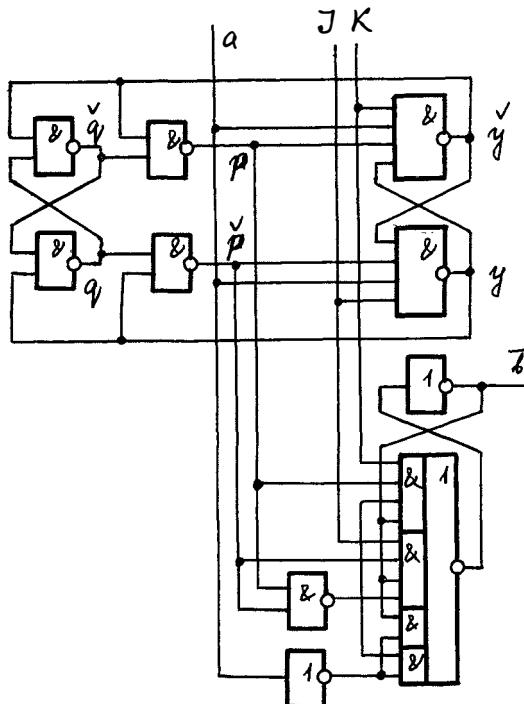


Figure 11.1. Aperiodic JK -flip-flop.

When the phase signal $a = 0$, the inputs J and K do not influence the state of the flip-flop elements, and in this phase, the values of J and K may be changed in any way. The indicator output is $\bar{b} = 1$.

In the working phase, which is initiated by the 0-1-transition of a , the flip-flop is receptive to the inputs J and K , and hence, their values must not change during the working phase. The latter is completed by the 1-0 transition of \bar{b} .

The operational cycle of the JK -flip-flop is $12T$ in the case when the flip-flop changes its value, and is $6T$ when it does not change.

11.2 Registers

Registers are the devices which are intended to receive and store combinations of binary values and, in some cases, to transform these values, for example, by shifting them. Registers may, in general, be constructed by using proper interconnections of any aperiodic flip-flops that have been considered already. The main differences between parallel registers are concerned with their different indication techniques.

The simplest variant of a parallel register can be built of RS-flip-flops of the type shown in Fig. 4.9(a) (without indicators \bar{b}). It has the common indicator which is built according to the idea of a parallel compression circuit (see Example 4.3) and whose inputs are the outputs q and \dot{q} of the RS-flip-flops. *The total length of the operational cycle of this register is $4T + \tau(n)$* where τ is the indicator's delay and n is the number of flip-flops in the register.

Using RS-flip-flops of the type of Fig. 4.10(a), we can also use the indicator with the following inherent function

$$\overline{b} = \overline{\overline{S}_1 y_1} \vee \overline{\overline{R}_1 \dot{y}_1} \vee \dots \vee \overline{\overline{S}_n y_n} \vee \overline{\overline{R}_n \dot{y}_n} \vee \overline{a}.$$

An alternative technique for constructing a register composed of the same flip-flops is shown in Fig. 11.2, in which the indicator implementation exploits both the idea of parallel compression and the collective responsibility principle (see 4.3.2).

When $a = 0$ all flip-flops of the register are idled, and, hence, $I_1 = I_2 = 0$. After a changes from 0 to 1 and a new input value is loaded into the register, its flip-flops become non-receptive to any input changes because they are cut off from the inputs and $I_1 = I_2 = 1$. After the completion of the transient process, b becomes equal to 1. The 1-0 transition of a causes the idling of the flip-flops to change to the all-one state, and then $I_1 = I_2 = 0$. This phase is ended by the 1-0 transition of b .

Consider a *serial register* realizing a multi-beat delay: the state of its j -th memory element is set equal to the value of the input signal which has entered the register j beats ago.

The serial register based on the circuit of Fig. 4.9(a) is shown in Fig. 11.3.

When $a = 0$ all flip-flops \dot{y}_i are idled while flip-flops \dot{q}_j are in the working state, and $\bar{b} = 1$. The change of a from 0 to 1 activates the sequential rewriting of values of flip-flops \dot{q}_j to flip-flops \dot{y}_j with subsequent idling of flip-flops \dot{q}_j . In the n -bit register all flip-flops \dot{y}_j will be in the working state in $2nT$ units of time while

all \check{q}_j will be idled. The idling of \check{q}_1 causes the 1-0 transition of \bar{b} thus completing the transient process in this phase. Similarly, the 1-0 transition of a initiates a sequential rewriting of the values of flip-flops \check{y}_{j-1} to flip-flops \check{q}_j with subsequent idling of the former ones. The idling of \check{y}_1 allows the writing of the input signal into \check{q}_1 that causes signal \bar{b} to change from 0 to 1 thereby ending the transient process.

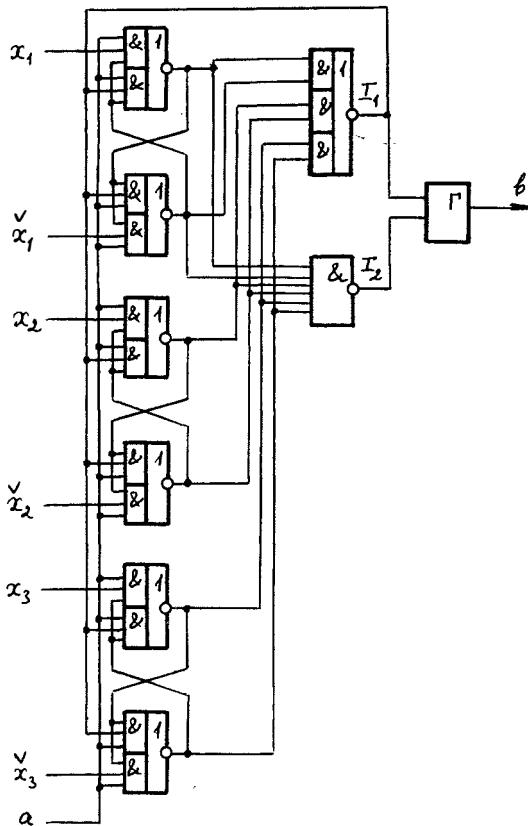


Figure 11.2. Parallel reception register with double-rail inputs.

The operational cycle of this register is $4nT + 2T$, due to this it can be called "slow" - all the elements in this circuit switch serially.

We may attempt to gain some parallelism in the process. Let us split the register into two parts, one of which will consist of r bits and the other of $n - r$ bits.

The phase signal will simultaneously activate transition processes in both parts of the register. A technique for combining these two parts is illustrated in Fig. 11.4.

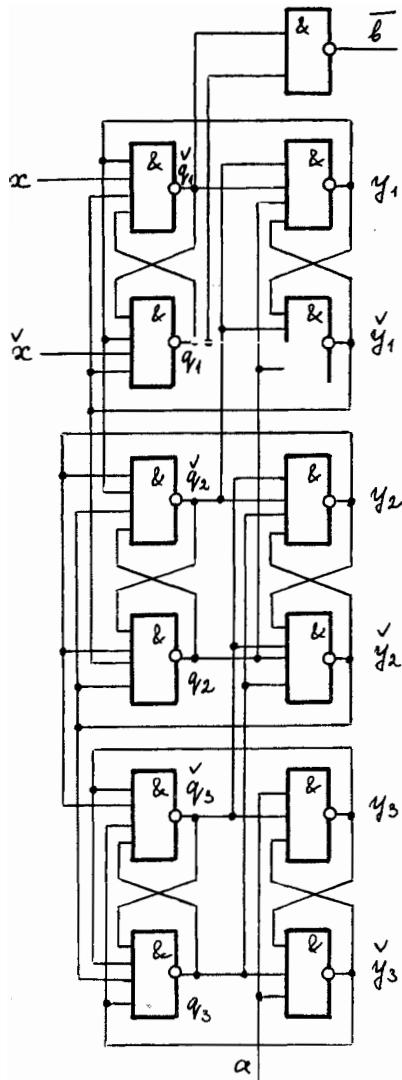


Figure 11.3. "Slow" serial register.

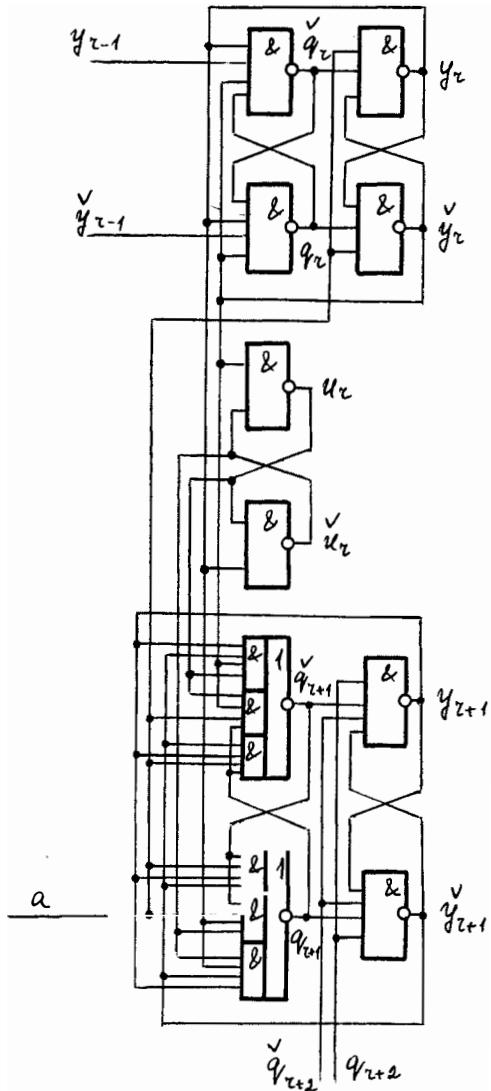


Figure 11.4. Parallelized write process implemented in a slow register.

The intermediate flip-flop \check{u}_r is used for storing the value of the output of one of the register parts until it is allowed to write the data into the input flip-flop of the other part of the register. When $a = 1$, after the completion of transition processes in this phase, the flip-flops \check{y}_r and \check{y}_{r+1} are in the working phase, and the intermediate flip-flop \check{u}_r copies the state of the output flip-flop of the first part of the register while all flip-flops \check{q}_r , including \check{q}_{r+1} , are idled. Both outputs of the flip-flops \check{q}_1 and \check{q}_{r+1} are connected to the inputs of the common indicator. The 1-0 transition of a initiates the idling of \check{y}_n and \check{y}_r , thus starting transition processes in both parts of the register. Since the state of the intermediate flip-flop \check{u}_r does not change in this phase, the transition process in the first part of the register ends in writing the data into the input flip-flop \check{q}_1 , and the process in the second part ends in the transition of the flip-flop \check{q}_{r+1} to the working phase. Note that the data transfer from one part to the other is possible only after the idling of the flip-flops \check{y}_{r+1} and \check{y}_r . When both \check{q}_1 and \check{q}_{r+1} come to the working state, the indicator changes its value, thus ending the transition process. When a returns to 1 it also activates processes in both parts at the same time. These processes end with the flip-flops \check{q}_1 and \check{q}_{r+1} being idled, which makes the indicator return to its original state. The latter provides evidence of the termination of the processes in the register.

When generalized to the case of splitting an n -bit register into n parts, this technique results in the serial register, two bits of which are presented in Fig. 11.5.

It should be noted that this circuit is not invariant to the time delay in applying the phase signal a to the inputs of different bits. Thus, if due to the limited fan-out capacity of the element producing the phase signal for the register, it is necessary to replace this element by a group, then the register will no longer be delay-insensitive. This is because certain combinations of the delays of the elements producing the phase signals for different bits of the register will be such that the latter operate incorrectly.

This disadvantage can be overcome by some complication of the circuit. Let a_q be a phase signal applied to flip-flops \check{q} , and a_y a phase signal applied to flip-flops \check{y} . Let a_y be taken from the output of one element, i.e. it arrives at the inputs of all flip-flops at the same time, and the signals a_q arriving from different elements are applied to flip-flops \check{q} .

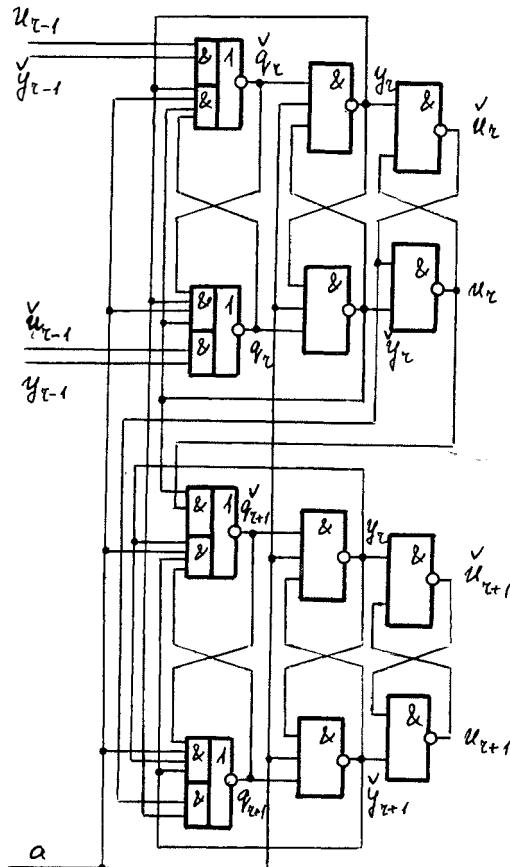


Figure 11.5. Serial register.

The serial register presented operates correctly if the change of a_q precedes the change of a_y . In fact, after the completion of the transition processes, the change of a_q does not change the state of the circuit elements, and the activation of processes in the register by the application of a_y will, therefore, be in such an already "prepared" circuit. It is clear that one solution to the indicated problem is to supply a_q to the indicator whose output will generate a_y . This solution, however, slows down the register. Another solution is based on the production of the signal a_{qj} of the individual indicator supplied for each flip-flop \dot{q}_j . The change of phase signal a_q is allowed immediately after the completion of a transition process in \dot{q} . In this case, for the indication of process completion in the whole register, it is sufficient to collect all signals a_{qj} at a common indicator.

Using the above serial registers, it is possible to construct *reversible registers*. The control over the direction of shift in aperiodic reversible registers can be suitably done by the double-rail signal. Such a register may incorporate two indicators, one for each direction of the shift.

11.3 Pipeline registers

A *pipeline register* is a serial register having specific dynamic properties that make it function as a data buffer between the source and the destination modules. Such a register consists of a number of serially connected *cells*, the first of which, the input head, is connected to the source, and the last, the output head, is connected to the destination as shown in Fig. 11.6.

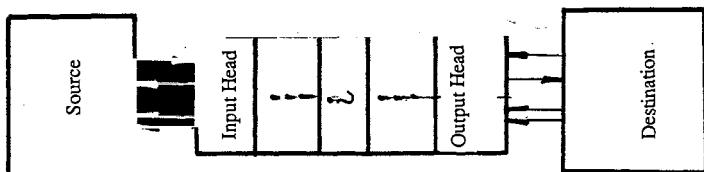


Figure 11.6. Pipeline register.

A data value produced by the source goes all the way through the register before it is consumed by the destination. Depending on the relative operation rates of the source and the destination, the pipeline register may either be increasingly filled with data, or emptied out. The interaction between the input head and the source and the output head and the destination, as well as the interaction between any pair of adjacent cells, is based on the handshake principle. The source and the destination may work independently of each other until the register is either totally emptied, or totally filled, with data.

Pipeline registers are characterized by such parameters as the *maximum number of data items* with respect to a given number n of register bits, and the *maximum throughput*, which is the reciprocal of the minimum time interval t between any two consecutive data items in any register cell.

11.3.1. NON-DENSE REGISTERS

The circuit for the register cell which is a three-state flip-flop is shown in Fig. 11.7(a). The operation of the cell can be defined by the state graph of Fig. 11.7(b). One of the graph vertices is associated with the absence of data in the cell, the idle state J_i , and two other vertices, the working states P_i^0 and P_i^1 , correspond to the binary values transferred through the cell, 0 and 1, respectively.

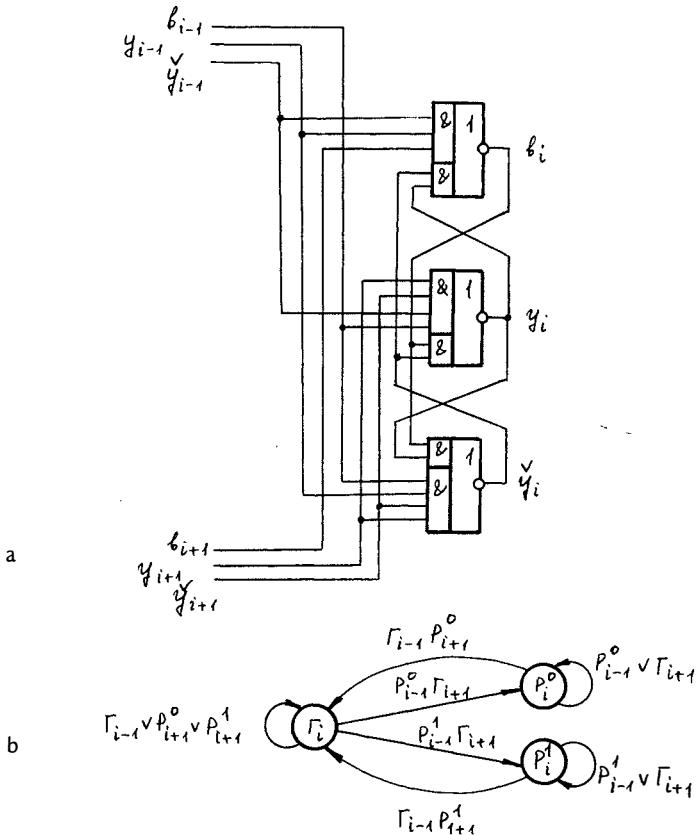


Figure 11.7(a) and (b). Cell for a non-dense pipeline register based on a three-state flip-flop (a), and (b) its state graph.

The maximum number of data items which may be entered into such a register is $\lfloor n/2 \rfloor$. In fact, if the source injects the sequence of data items into the input head, then with the destination stuck at, for example, \mathcal{J} -state, the register will eventually reach the state $\mathcal{J}_1 \mathcal{P}_2 \mathcal{J}_3 \mathcal{P}_4 \dots \mathcal{J}_{n-1} \mathcal{P}_n$ for even n , and $\mathcal{P}_1 \mathcal{J}_2 \mathcal{P}_3 \mathcal{J}_4 \dots \mathcal{P}_{n-1} \mathcal{J}_n$ for odd n . After that, the source will not be able to add any more items to the register. Since the pipeline register described may be only half-filled with data (there must be at least one cell in the idle state between any two cells in the working states), it is called the non-dense register. The maximum throughput of such a register is $1/(4t)$ where $t = 2T$. Indeed, the minimum interval between two consecutive data items is achieved when the state configuration for adjacent cells has the form $\mathcal{P}_i \mathcal{P}_{i+1} \mathcal{J}_{i+2} \mathcal{J}_{i+3} \mathcal{P}_{i+4} \mathcal{P}_{i+5} \mathcal{J}_{i+6} \mathcal{J}_{i+7}$ where the pair $\mathcal{P}_i \mathcal{P}_{i+1}$ corresponds to one and the same data item. In this case, the shift of the indicated configuration to one position

right is performed in time t , and the original configuration is re-established in $4t$. This implies that the register contains $\lceil n/4 \rceil$ or $\lfloor n/4 \rfloor - 1$ data items.

The state P_i^1 corresponds to the stable state, 110, of the flip-flop of Fig. 11.7(a), P_i^0 to 101, and J_i to 011. Thus, the presence of data in the cell is signalled by $b_i = 1$, and the absence of data (idling condition) is $y_i = \hat{y}_i = 1$.

More suitable, in practice, is the variant of a *non-dense pipeline register* built of NAND gates. The cell for such a register is presented in Fig. 11.8. For this register, $t = 2.5T$. As in the previous case, the cell is based on a three-state flip-flop whose states are:

- 011 – the cell stores the value of 1,
- 101 – the cell stores the value of 0,
- 110 – the cell contains no data.

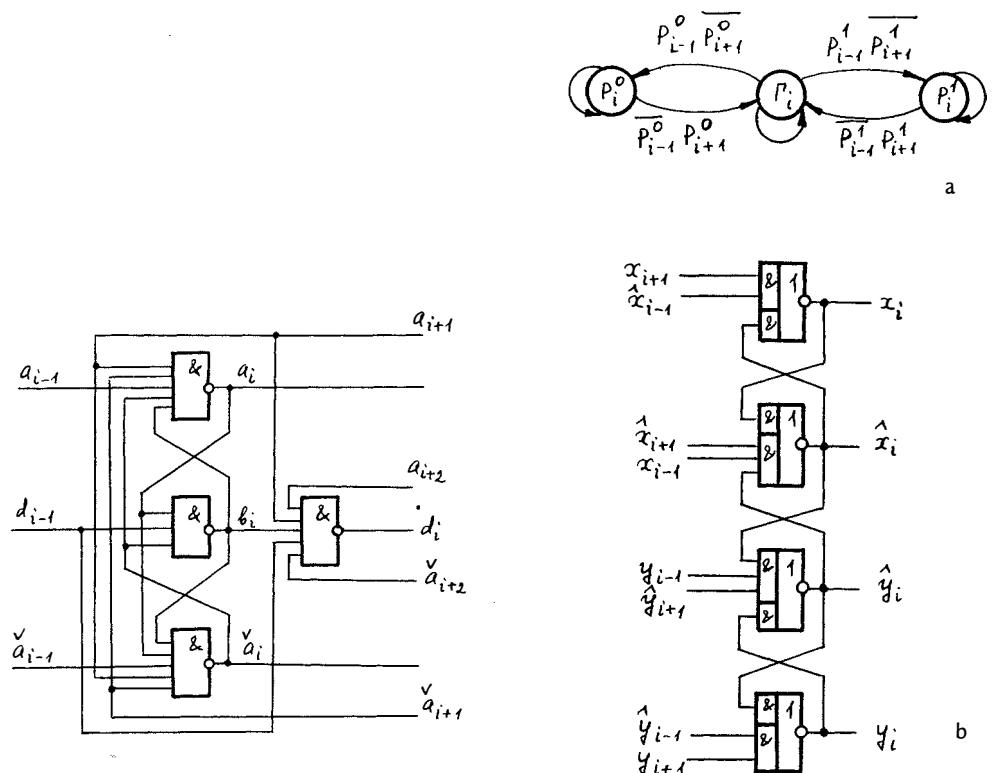


Figure 11.8. Cell for a non-dense pipeline register built of NAND elements.

Figure 11.9(a) and (b). State graph for a semi-dense pipeline register cell (a) and its implementation (b).

A data value can be written into the i -th cell if the $(i - 1)$ -th has the data, and the $(i + 1)$ -th is in the idle state, and, furthermore, the values of the outputs of additional elements in these cells satisfy $d_{i-1} = d_{i+1} = 1$. The writing process begins with the change of d_i from 1 to 0 in the i -th cell, and then the three-state flip-flop changes either to 011 or to 101. After this, the $(i - 1)$ -th cell is idled, i.e. its flip-flop comes to state 110, and then d_i returns to 1, thereby, allowing the data to be reloaded into the $(i + 1)$ -th cell.

11.3.2. SEMI-DENSE PIPELINE REGISTER

Two consecutive data items in a pipeline register may differ from each other in their values. Such items may then be stored in two adjacent cells without an idling “gap” between them. This idea lies at the heart of the circuit construction for the pipeline register with semi-dense data packing.

The state graph for a cell of the *semi-dense register* is shown in Fig. 11.9(a). Vertex \mathfrak{J}_i stands for the idle state whereas \mathfrak{P}_i^0 and \mathfrak{P}_i^1 are associated with the working states, the storing of 0 and 1, respectively. The corresponding circuit is shown in Fig. 11.9(b). It consists of two RS-flip-flops, \hat{x}_i and \hat{y}_i , and has four stable states assigned to the binary codes according to Table 11.1.

	x_i	\hat{x}_i	\hat{y}_i	y_i
\mathfrak{J}_{i_0}	0	1	1	0
\mathfrak{P}_i^1	1	0	1	0
\mathfrak{P}_i^0	0	1	0	1
—	1	0	0	1

Table 11.1.

The semi-dense pipeline register built of such cells has the following characteristics:

- the information capacity lies between $n/2$ and n items, and assuming that the incoming data rate is Bernoullian, its mean value is $3n/4$;
- the throughput of the register is between $1/(3t)$ and $1/(4t)$; the upper bound is achieved in the following state configuration for adjacent cells

$$\dots \mathcal{J}_i \mathcal{P}_{i+1}^\sigma \mathcal{P}_{i+2}^\sigma \mathcal{J}_{i+3} \mathcal{P}_{i+4}^\sigma \mathcal{P}_{i+5}^\sigma \mathcal{J}_{i+6} \mathcal{P}_{i+7}^\sigma \mathcal{P}_{i+8}^\sigma \dots ,$$

and the lower bound is achieved in the configuration

$$\dots \mathcal{J}_i \mathcal{J}_{i+1} \mathcal{P}_{i+2}^\sigma \mathcal{P}_{i+3}^\sigma \mathcal{J}_{i+4} \mathcal{J}_{i+5} \mathcal{P}_{i+6}^\sigma \mathcal{P}_{i+7}^\sigma \dots ;$$

the mean throughput value is $7/(24t)$.

From the fact that every memory cell in the above pipeline registers has several stable states, we can deduce the following.

When the power is applied to the circuit, the register cells assume arbitrary stable state values. Thus, we need a special reset procedure which can be done by clearing the register of data through the output head port. To do this, we should execute a multi-cycle data read operation. It is easy to check that this will definitely result in the situation when all the cells are in the state \mathcal{J}_i .

11.3.3. DENSE PIPELINE REGISTERS

A pipeline register circuit with *dense* data packing would be sensible if its complexity is not higher than that of a non-dense register having twice the number of cells. Sometimes such registers have even higher throughputs than non-dense ones.

The state graph for a variant of a *dense pipeline buffer* cell is shown in Fig. 11.10(a). According to this graph the cell has four states, namely \mathcal{P}_i^{00} , \mathcal{P}_i^{10} , \mathcal{P}_i^{01} , \mathcal{P}_i^{11} . The first super-script in this denotation corresponds to the value of a data bit written into the cell, and the second is a special label. The following designations are assumed in the graph

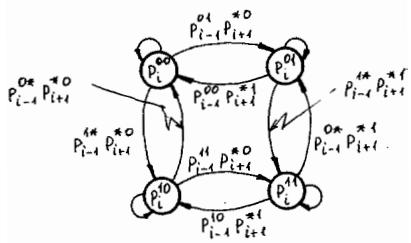
$$\mathcal{P}_{i-1}^* = \mathcal{P}_{i-1}^{\sigma 0} \vee \mathcal{P}_{i-1}^{\sigma 1}, \quad \mathcal{P}_{i+1}^{*\sigma} = \mathcal{P}_{i+1}^{0\sigma} \vee \mathcal{P}_{i+1}^{1\sigma}.$$

The cell does not have an idle state, i.e. the state in which there is no data in the cell. Hence, the register is supposed to be always loaded with some data, with one bit at least. In the latter case, all the cells are in the same state. Two different bits stored in adjacent cells are marked with different labels. Therefore, in the completely filled register, the states of any pair of adjacent cells have different labels.

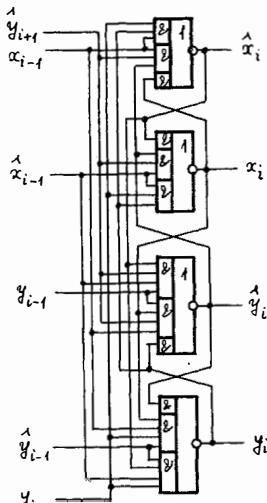
A specific feature of this cell is that each of its states may be both stable and transient, as the situation requires. For example, during the transition from \mathcal{P}_i^{00} to \mathcal{P}_i^{11} the cell goes through \mathcal{P}_i^{10} , which is, in this case, transient, whereas in the transition from \mathcal{P}_i^{11} to \mathcal{P}_i^{10} , the \mathcal{P}_i^{10} state is stable.

The circuit of the cell is shown in Fig. 11.10(b). As with the previous cell,

this one consists of two RS-flip-flops \hat{x}_i and \hat{y}_i , the first of which stores the data written into the cell, and the second stores the label.



a



b

Figure 11.10.(a) and (b). (a) State graph for a dense pipeline register cell, and (b) its implementation.

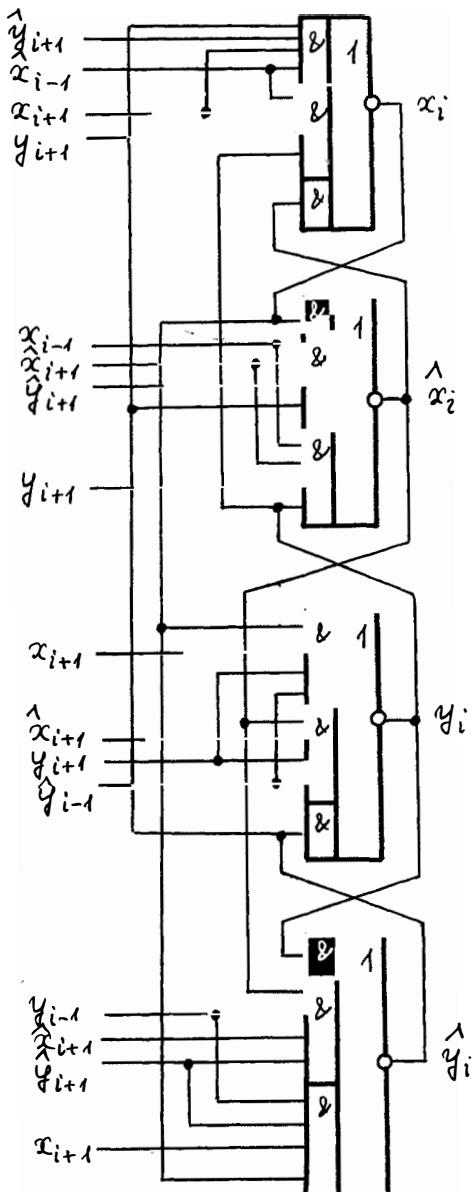


Figure 11.11. Cell for a dense pipeline register with better throughput (than cell of Fig. 11.10).

The cell states are encoded in accordance with Table 11.2.

	\hat{x}_i	x_i	\hat{y}_i	y_i
\mathcal{P}_i^{00}	1	0	1	0
\mathcal{P}_i^{10}	0	1	1	0
\mathcal{P}_i^{01}	1	0	0	1
\mathcal{P}_i^{11}	0	1	0	1

Table 11.2.

The *throughput of the dense pipeline register* built of such cells is between $1/(2t)$ and $1/(3t)$ where $t = 2T$. The upper bound is achieved when all data bits loaded into the register have the same value. The state configuration for adjacent cells will, in this case, be as follows.

$$\dots \mathcal{P}_i^{\sigma 0} \mathcal{P}_{i+1}^{\sigma 0} \mathcal{P}_{i+2}^{\sigma 1} \mathcal{P}_{i+3}^{\sigma 1} \mathcal{P}_{i+4}^{\sigma 0} \mathcal{P}_{i+5}^{\sigma 0} \mathcal{P}_{i+6}^{\sigma 1} \mathcal{P}_{i+7}^{\sigma 1} \dots .$$

The lower bound is achieved when all data bit values in the register alternate, which corresponds to the configuration

$$\dots \mathcal{P}_i^{00} \mathcal{P}_{i+1}^{00} \mathcal{P}_{i+2}^{01} \mathcal{P}_{i+3}^{11} \mathcal{P}_{i+4}^{11} \mathcal{P}_{i+5}^{10} \mathcal{P}_{i+6}^{00} \mathcal{P}_{i+7}^{00} \mathcal{P}_{i+8}^{01} \dots .$$

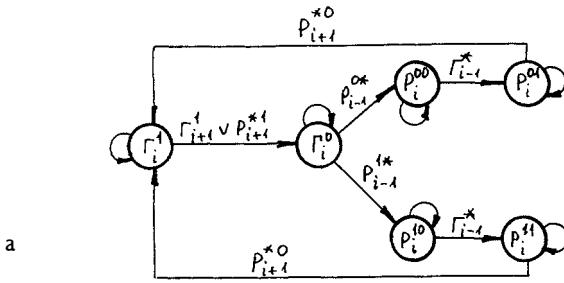
The mean throughput value is $5/(12t)$.

Another variant of a dense pipeline register can be obtained by adding two extra inputs in each cell. This slight complication increases by a factor of 1.5, the throughput of the register whose cell is shown in Fig. 11.11.

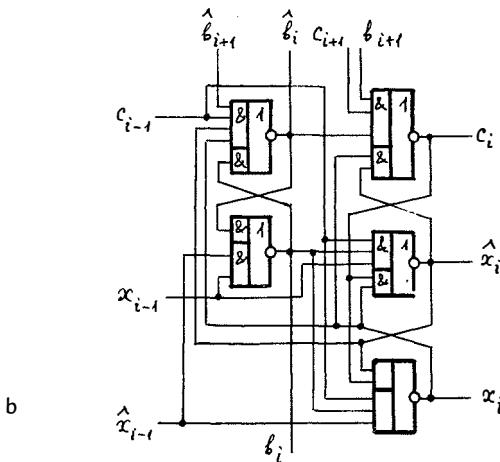
Even simpler cells can be constructed for dense registers as, for example, the one whose state graph and the circuit are presented in Fig. 11.12(a) and (b), respectively. The graph vertices correspond to the following six states: two idle states \mathcal{J}_i^0 and \mathcal{J}_i^1 , and four working states $\mathcal{P}_i^{00}, \mathcal{P}_i^{10}, \mathcal{P}_i^{01}, \mathcal{P}_i^{11}$. The state \mathcal{J}_i^0 is transient while the other states are stable. The denotations of the working states are the same as for the previous state graph of Fig. 11.10(a). The super-scripts of the idle states stand for the label values. The following designations are assumed in the graph

$$\mathcal{P}_{i-1}^{**} = \mathcal{P}_{i-1}^{00} \vee \mathcal{P}_{i-1}^{10} \vee \mathcal{P}_{i-1}^{01} \vee \mathcal{P}_{i-1}^{11}; \quad \mathcal{P}_{i-1}^{\sigma *} = \mathcal{P}_{i-1}^{\sigma 0} \vee \mathcal{P}_{i-1}^{\sigma 1};$$

$$\mathcal{P}_{i+1}^{*\sigma} = \mathcal{P}_{i+1}^{1\sigma} \vee \mathcal{P}_{i+1}^{0\sigma}; \quad \mathcal{J}_{i-1}^* = \mathcal{J}_{i-1}^0 \vee \mathcal{J}_{i-1}^1.$$



a



b

Figure 11.12(a) and (b). (a) State graph for dense pipeline register cell, and (b) its implementation using a flip-flop with separate inputs and a three-state flip-flop.

If the register is free from data, all of its cells are in the \mathcal{J}_i^1 state. When the register is filled with data, their states are either \mathcal{P}_i^{01} or \mathcal{P}_i^{11} . The cell contains two flip-flops one of which is a three-state and the other is an RS-flip-flop. The three-state flip-flop has a pair of stable states corresponding to the two different values of the data bit stored in the cell, and the third stable state which corresponds to the idle state of the cell. The RS-flip-flop is used for holding a label. The states of the cell are encoded as shown in Table 11.3.

The throughput of the dense pipeline register described is $1/(6t)$ where $t = 2T$.

	\hat{b}_i	b_i	c_i	\hat{x}_i	x_i
\mathcal{J}_i^1	1	0	0	1	1
\mathcal{J}_i^0	0	1	0	1	1
\mathcal{P}_i^{00}	0	1	1	1	0
\mathcal{P}_i^{10}	0	1	1	0	1
\mathcal{P}_i^{01}	1	0	1	1	0
\mathcal{P}_i^{11}	1	0	1	0	1

Table 11.3.

11.3.4. ONE-BYTE DENSE PIPELINE REGISTER

One of the approaches to constructing a pipeline register with each item of data equal to one byte is to use eight one-bit pipeline buffers in parallel. Synchronization of all input heads and output heads must obviously be provided for the data to be correctly delivered at its destination through the register. The alternative way is to synchronize these on-bit paths at every stage of the register.

Using the ideas on which the register of Fig. 11.12 is based, we can obtain the cell of a *one-byte dense buffer* which will be more economical than the interconnection of eight cells of one-bit registers. This is achieved through modifying the control circuit for eight paths in such a way as to implement it as a single, common, synchronizer, which provides synchronization for each bit. The control sub-unit of Fig. 11.12 can be separated by making identical the informational inputs of the three-state flip-flop, which is thus transformed into an RS-flip-flop, and this is equivalent to identifying the pairs of working states ($\mathcal{P}_i^{00}, \mathcal{P}_i^{10}$) and ($\mathcal{P}_i^{01}, \mathcal{P}_i^{11}$). Finally, the control circuit will have two working states and two idle states, one of which is transient. To obtain a cell for a one-byte dense pipeline register we now only need to interconnect an eight-bit parallel register with the control circuit shown in Fig. 11.13. *The throughput of such a register is 1/(12T).*

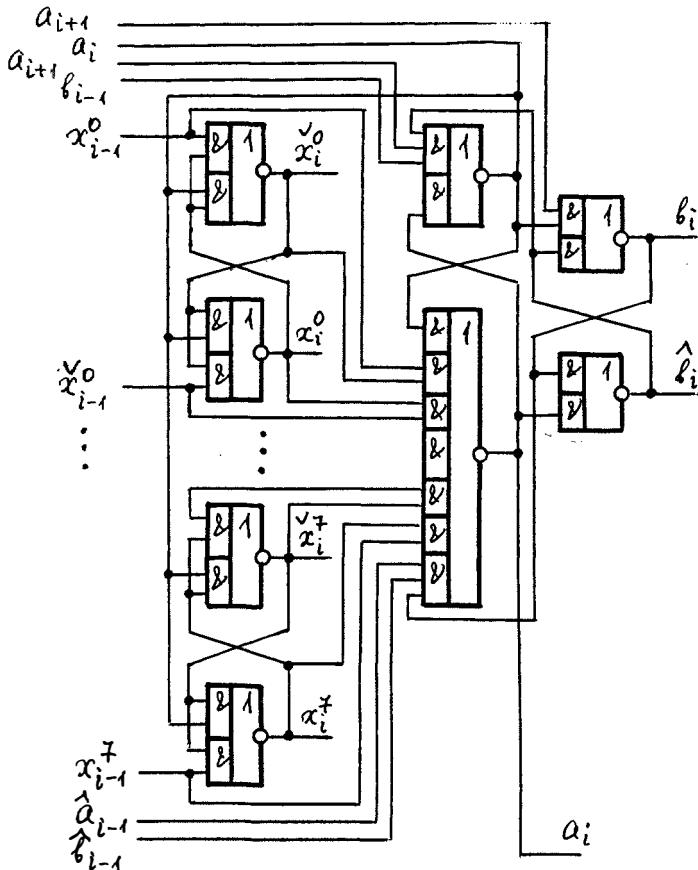


Figure 11.13. Cell for a byte-width dense pipeline register.

11.3.5. PIPELINE REGISTER WITH PARALLEL READ-WRITE AND THE STACK

In the above pipeline registers, the time during which every data item moves from the input head to the output head of the register is dependent on both the inherent delays of the composing logical elements and on how much the register is filled with data. Hence, it would be practically impossible to determine in which cell of the register a particular bit of the shifted code is, at a given time instant. On the other hand, the parallel reading of data from serial registers (among which are, to some extent, pipeline registers) requires that, at the time of reading, the i -th cell stores the i -th bit of shifted code. Consider a dense pipeline register connected via its input head to the data source which generates an ordered sequence of bits starting from the most significant bit. Then the i -th bit of code enters the i -th cell of the register after the $(i + 1)$ -th bit has been written into the $(i + 1)$ -th cell. This fact allows the moment

when the *data may be read from the register in a parallel way* to be indicated, starting from the last cell.

A cell for a pipeline register based on such a principle is shown in Fig. 11.14. It consists of two three-state flip-flops, the master flip-flop and the slave flip-flop.

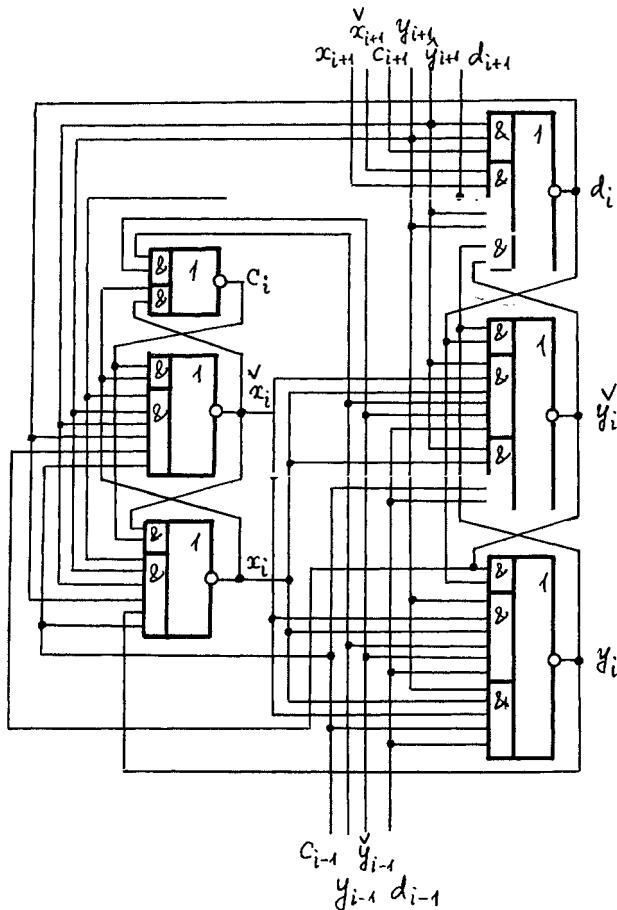


Figure 11.14. Cell for a pipeline register with parallel data read capability.

As in the case of the simplest (non-dense) pipeline register of Fig. 11.7, one state of each of the three flip-flops is associated with the situation when there is no data in the flip-flop, and the other two states are associated with holding the 0 and 1 values of data. The master flip-flops (the ones on the right in Fig. 11.14) constitute the pipeline register itself, which is analogous to that of Fig. 11.7(a). The data is read in parallel from outputs of the slave flip-flops (on the left in Fig. 11.14). While the slave flip-flops are empty, the register operates like a non-dense register. When the most significant bit of the shifted code enters the last cell of the register, the data in

this cell is reloaded from the master to the slave flip-flop. Now, if the cell before the last one is loaded with data, this data will obviously be the bit following the most significant bit. Therefore, even in this cell, the data will be reloaded for the master to the slave flip-flop. When a current data item enters the i -th cell, the indication that this item must be reloaded from the master flip-flop to the slave one, and thus be prepared for the parallel reading, is that the slave flip-flop of the next, $(i+1)$ -th cell, holds its own data item. The process continues until all the slave flip-flops hold data, after which the parallel reading may be carried out. The resetting to the original state is performed sequentially from the first cell to the last one, and as this happens, the data contained in the i -th cell is first idled in its master flip-flop (the necessary condition for this is the absence of data in the master flip-flop of the $(i-1)$ -th cell, and the presence of data in the slave flip-flop of the $(i-1)$ -th cell), and then in its slave flip-flop (the necessary condition for this is the absence of data in the master flip-flop of the $(i+1)$ -th cell).

The throughput of such a register in the ordinary shift mode is the same as for a non-dense pipeline, i.e. $1/(4t)$, whereas the informational capacity of the register having n cells is the same as for a dense pipeline, i.e. n .

The cell for a *pipeline buffer with parallel write capability* is shown in Fig. 11.15. As in the previous case, it consists of the master and the slave flip-flops. The distinction is that the slave flip-flop, the RS-one, is intended for holding a label, rather than for copying the data of the master flip-flop.

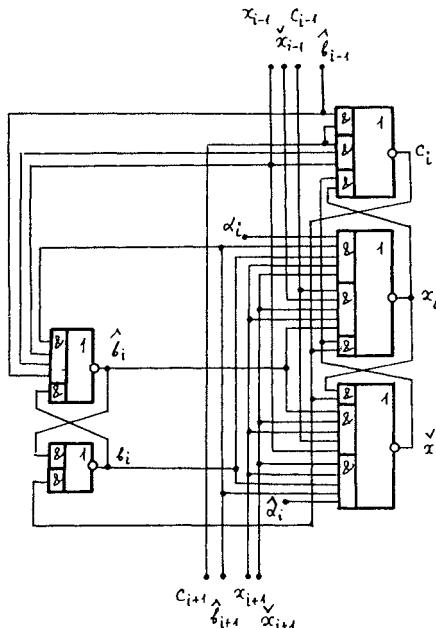


Figure 11.15. Cell for a pipeline register with parallel write capability.

In the parallel write mode, the data from the data lines enters the master flip-flop of the i -th cell only after the $(i + 1)$ -th cell changes from the write mode to the shift mode and the data in it is idled. As soon as the data is loaded into the master flip-flop of the i -th cell, a label, the so-called parallel write flag, in its slave flip-flop will be reset, i.e. the i -th cell is ready for the shift mode. Thus the process of changing to the shift mode in the register flows, sequentially, from the last cell to the first one. In this operation mode, this register, like the previous one, works in a similar way to the ordinary non-dense pipeline. The i -th cell goes back to the parallel write mode only after the $(i - 1)$ -th cell, and after the data in the master flip-flop of the i -th cell is idled. Thus, the return of the register to the write mode is also performed sequentially, but from the first cell to the last one. The throughput and the informational capacity of this buffer are the same as those of the ordinary non-dense pipeline.

Using the parallel read pipeline register as the basis, we can build a buffer with both the shift mode and the parallel read mode, as well as with the serial reading of data in reverse order to its writing, thereby realizing a *stack*. The cell for such a register is shown in Fig. 11.16. It differs from the prototype cell of Fig. 11.14 in having an additional RS-flip-flop for the co-ordination of the register operation modes.

In this register, the master flip-flops are first loaded with data, as was done in a parallel read pipeline. In so doing, as the register is filled with data, the additional flip-flops of all the cells, starting from the last one, are loaded with a label. Then the data is sequentially, starting from the first cell, reloaded from the master flip-flops to their respective slaves, from which it can be read in the same way as in the register which is composed of the cells of Fig. 11.14. As the data is written into the slave flip-flops, the labels in the additional flip-flops are reset (prior to which the data in the master flip-flops is idled). Now the data can be read, sequentially, from the chain of slave flip-flops, but in the reverse order, because the role of the output head is now performed by the slave flip-flop of the first cell. Thus, the additional flip-flops "switch" between two pipeline registers, which are connected to each other by parallel links. When the label is present in them, the register consisting of the master flip-flops operates, and the serial writing is carried out; when the label is absent, the operation is performed by the register consisting of the slave flip-flops with the serial reading of data. It should be noted that these operation modes may be combined in time.

The above register has the same features as its prototype, the register built of the cells of Fig. 11.14.

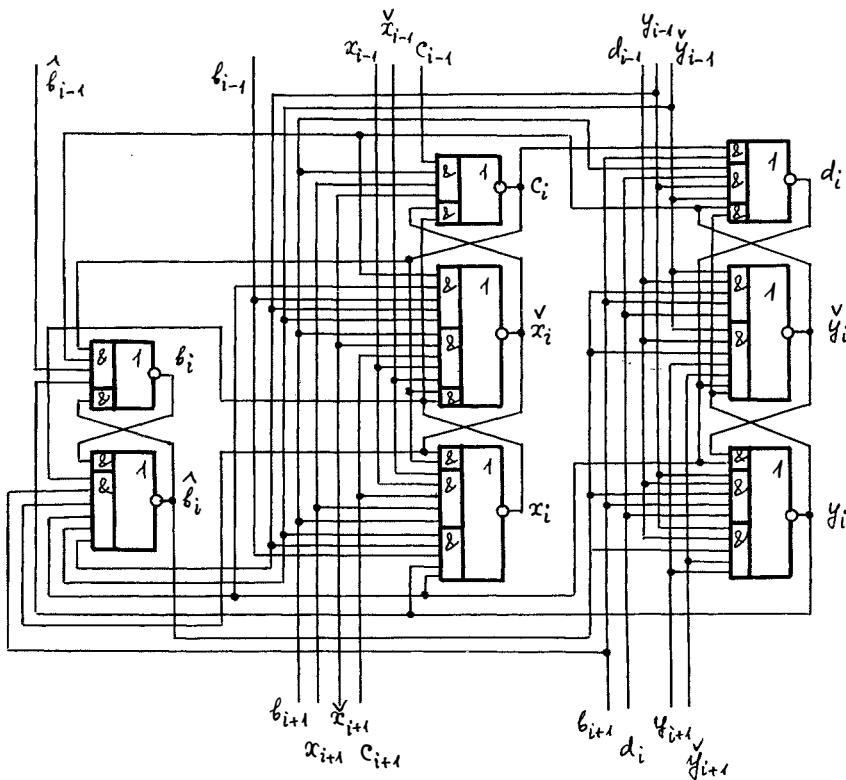


Figure 11.16. Cell for a stack.

11.3.6. REVERSIVE PIPELINE REGISTERS

One of the potential ways to expand the capabilities of pipeline buffers can be by introducing to them a mode of reversible shift of data. In this case, the pipeline register may be used as a basis for the stack memory in computing devices. The circuit for a cell of a *reversible pipeline register* is shown in Fig. 11.17.

It consists of a three-state flip-flop, one of the states of which, as before, corresponds to the absence of data in the cell, and the other two states stand for the 0 and 1 values in it. If the two pairs of cells which are left and right adjacent to the i -th cell are idled, then the data in the i -th cell is not shifted. The shift of data from the

i -th cell to the right one is performed if, on the left hand side of it, only one adjacent cell is idled. If on the left hand side of the i -th cell, three cells are idled, the shift of data is carried out to the left cell. The shift control in this register is done by the left most cell, which combines the functions of the input and the output heads of the register.

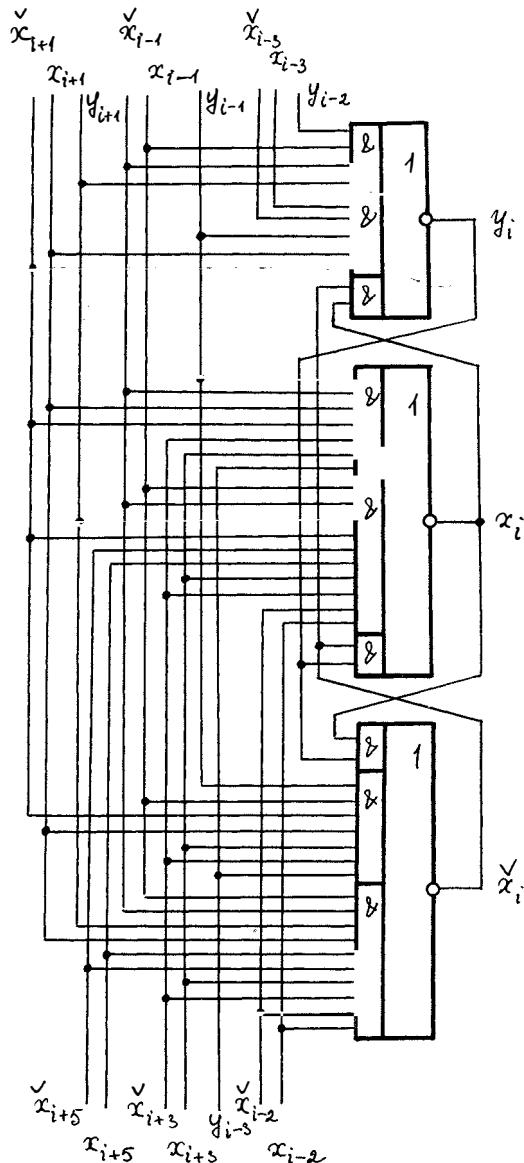


Figure 11.17. Cell for a non-dense reversible pipeline register.

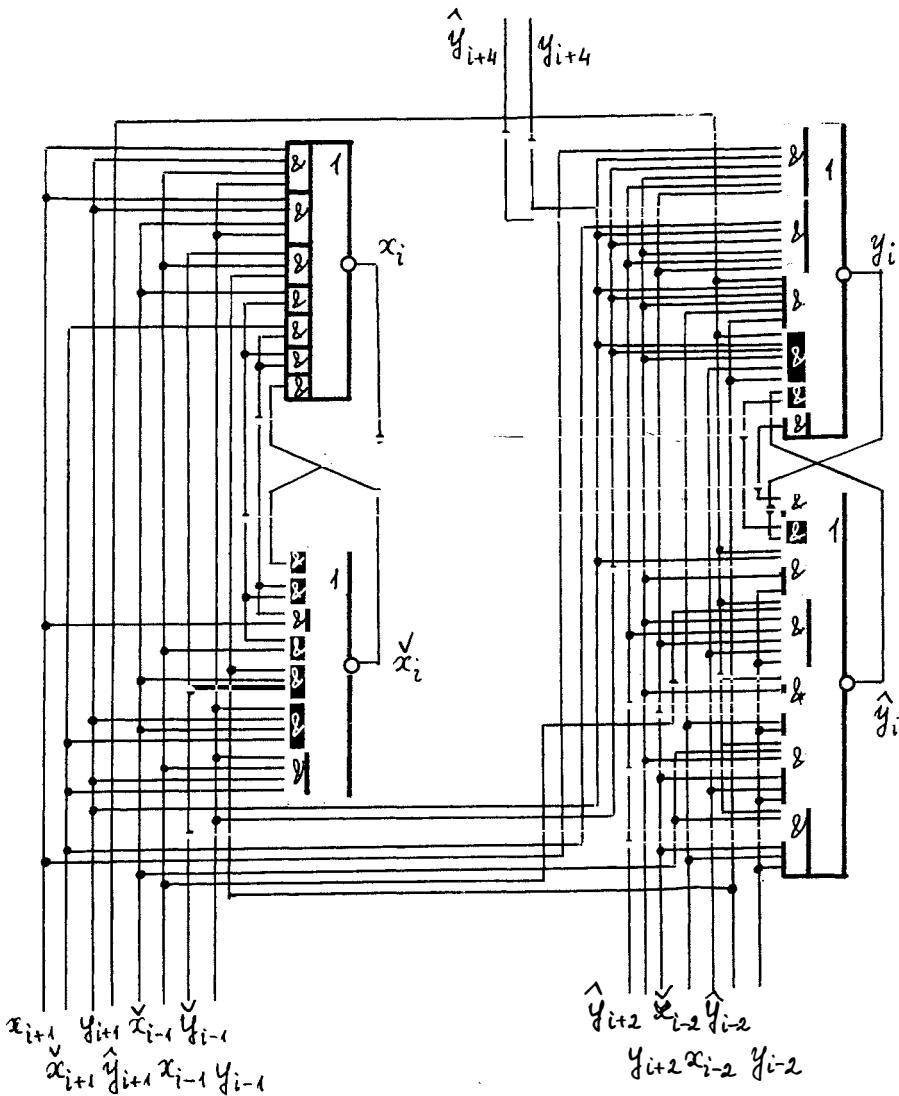


Figure 11.18. Cell for semi-dense reversible pipeline register.

The maximum informational capacity of the pipeline buffer consisting of the cells of Fig. 11.17 is $\lceil n/2 \rceil$. However, the maximum speed is provided when it is filled with data at the level of $\lceil n/3 \rceil$. It should also be noted that when the register described is used as a stack memory, it must be filled with no more than $\lceil n/3 \rceil$ items in order to be able to read all the data previously written in it. The shift of data in the

register to the one-bit position requires $6T$ units of time independent of the direction of the shift. Due to the fact that operations in adjacent groups can be overlapped in time, the frequency of operations is $1/(3T)$, from which we conclude that the throughput of the register is $1/(9T)$.

The main disadvantage of the above reversible pipeline is its high complexity with respect to rather low informational capacity. This results in relatively slow operation. This drawback can be eliminated, first of all, by raising its informational capacity through using a “denser” version of the pipeline. An example of a cell for such a *dense reversible pipeline buffer* is shown in Fig. 11.18. Its maximum capacity is n , which is twice that of the previous register, but the capacity for which the maximum speed performance and the ability to operate in stack mode can be achieved is only one-and-a-half times that of the previous buffer, i.e. $\lceil n/2 \rceil$. The one-bit position shift requires $6T$, as before. The frequency of the shift operation is also the same as before. However, due to the increased informational capacity of the register, it has a higher throughput, which is $1/(6T)$.

11.4 Converting single-rail signals into double-rail signals

11.4.1. PARALLEL REGISTER WITH SINGLE-RAIL INPUTS

The register shown in Fig 11.19 shows an implementation of the *parallel conversion of single-rail data into its double-rail form*. It can be used, for example, to transfer data from a single-rail data bus to the inputs of an aperiodic logic module.

Let a bus consisting of n lines be able to carry values of the n -bit code. The conversion register has, in addition, to n data inputs connected to the corresponding bus lines, the phase signal input a and the completion indication signal output b .

If $a = 0$, the register is in the idle state, in which

$$\dot{y}_i = y_i = 1, \quad z_i = 0, \quad 1 \leq i \leq n, \quad \bar{a} = 1, \quad \bar{\bar{a}} = u = b = 0,$$

independent of the state of the data inputs. After a change from 0 to 1, the registers are loaded with data. When the register indicators u and v have changed to 1, and provided that $\bar{\bar{a}}$ does likewise, the common output b mimics the change to 1 that indicates the completion of the switching processes in the register. After $b = 1$, any change of data on the bus lines will not affect the register because it stores the accepted data while $a = 1$. When signal a returns to 0, the register proceeds to the idling phase, which will result in the return of b from 1 to 0. After another data value is applied to the bus lines, the next working phase may be initiated.

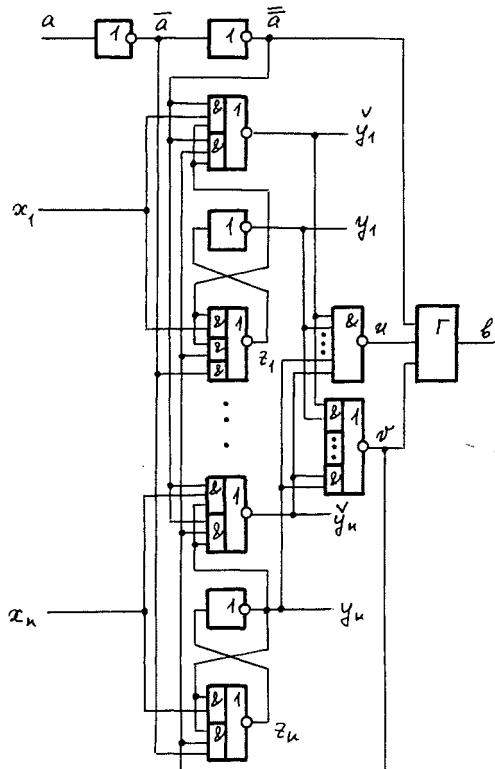


Figure 11.19. Parallel register with single-rail inputs.

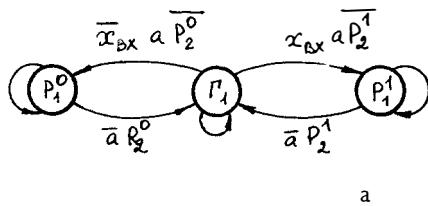
This design style can also be used to convert single-rail data to its double-rail format in both the input and the output heads of pipeline buffers.

11.4.2. INPUT AND OUTPUT HEADS OF PIPELINE REGISTERS

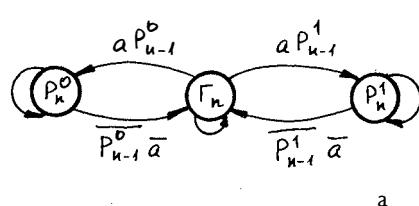
Pipeline registers operate with data signals in the double-rail representation, which is often not compliant with the interface of the source and the destination of the data. When such an interface requires single-rail signals to be used, the register can be supplied with special input and output heads whose circuits differ from those of the register cells.

Consider an example of a semi-dense pipeline buffer. The source is connected to the input head of the buffer by three wires, one of which is the data line, and the other two are phase signal \bar{a} and the indication b .

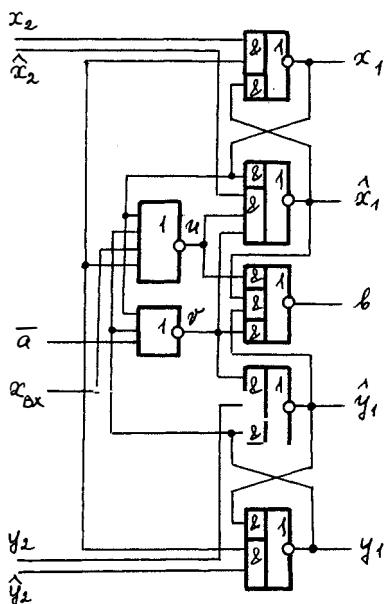
The state graph for the *input head* is shown in Fig.11.20(a). The circuit of this head is shown in Fig. 11.20(b).



a

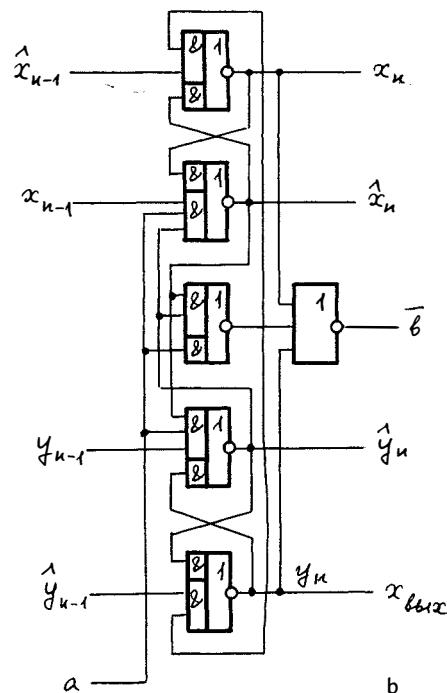


a



b

Figure 11.20(a) and (b). (a) State graph for the input head of a semi-dense pipeline register, and (b) its implementation.



a

Figure 11.21(a) and (b). (a) State graph for output head of a semi-dense pipeline register, and (b) its implementation.

It consists of seven elements, four of which form two RS-flip-flops \hat{x}_1 and \hat{y}_1 . The state encodings and their designations in the state graph remain the same as for the dense register cell of Fig. 11.9 (see Table 11.1). Note that, unlike that cell, the head does not have the connections between elements \hat{x}_1 and \hat{y}_1 which are necessary

for the cell to inhibit the transition to the fourth, forbidden, state. It is inherent in the input head that the situation causing such an undesired transition never occurs.

The head operates in the following way. When $\bar{a} = 1$ and $u = v = 0$, the head is disconnected from the data line x_{in} . If the head is in the idle state, i.e. the outputs of flip-flops \hat{x}_1 and \hat{y}_1 are set to the 0110 value (according to Table 11.1), then $b = 0$, and the head is ready for receiving a new value. After the new value of x_{in} is applied to the input, the change of \bar{a} from 1 to 0 starts the working phase in the head. This phase ends in b changing from 0 to 1, after which the input data value is allowed to be changed. Then the idle phase may be started by returning \bar{a} to 1. Now, as soon as the data value is reloaded from the head to the subsequent cell of the register, the head may change to the idle state, thereby, causing the 1-0 transition of b . This ends the idle phase of the head operation.

The state graph and the circuit for the *output head* for a semi-dense pipeline register are shown in Fig. 11.21(a) and (b), respectively. The output head, like the input one, is connected to the environment, the destination, by three wires, one of which is the data line, and the other two are used for the phase signal a and the indication signal \bar{b} . The head states are encoded and designated in the graph in the same manner as those for the semi-dense pipeline cell. The condition $a = 1$ implies that the destination is not yet ready to accept data, and $\bar{b} = 1$ means that the head is in the idle state, thus not ready to produce any data for the destination.

When data is sent to the outputs of the preceding cell, one of the flip-flops of the head switches and \bar{b} changes from 1 to 0, which signals, to the destination, the presence of data in the output head. If the destination is ready to accept data, it strobes the data from y_n and then initiates the idle phase in the head by changing a from 1 to 0. When $a = 0$, and if the preceding cell is either in the idle state or holds the data bit whose value is complement to the value held in the head, the head changes to its idle phase. This results in the change of \bar{b} from 0 to 1. Upon the signal $\bar{b} = 1$, the destination changes phase signal a from 0 to 1, which initiates the next working phase in the output head, etc.

The resetting of a semi-dense register to the initial state can be realized by connecting the \bar{b} signal of the output head to its input a . Now with the source halted, all the cells of the register will be brought to the idle state after a finite number of shifts.

11.5 Counters

Fig. 11.22 demonstrates a circuit for an *aperiodic counter* built of the flip-flops shown earlier in Fig. 4.12(b) (with symbolics in Fig. 4.12(c)). The counter is designed as a series of T-flip-flops with a common indicator.

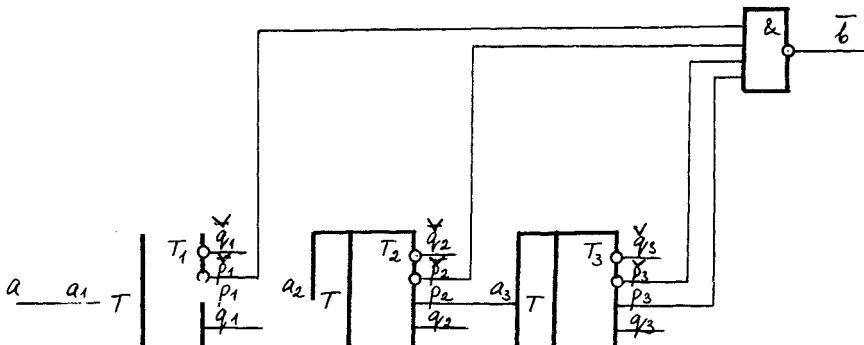


Figure 11.22. Aperiodic counter.

The maximum counter capacity of a n -bit counter is, of course, 2^n . The role of a carry signal in the j -th flip-flop is played by the output of the gate p_j because the transition process in each T-flip-flop terminates by changing the states of the gates p_j and \bar{p}_j , and the common indicator is thus implemented by the function

$$\bar{b} = \overline{\bar{p}_1 \dots \bar{p}_{n-1} p_n p_n}.$$

We now estimate the average carry propagation length. Half the number of cases have a carry length equal to 1. For a quarter of the cases, the length is 1, for an eighth part, is 2, etc. Hence, for $n \rightarrow \infty$, the average carry length is determined by the following sum of series:

$$G = (1/4)1 + (1/8)2 + (1/16)3 + \dots .$$

After rather obvious transformations we obtain $G = 1/2 + (1/2)G$ from which $G = 1$, i.e. on the average, two bits switch during one operation cycle. Cf. an analogous bound for an adder is $\log_2(0.8n)$.

Thus for the given counter, the average duration of the cycle is $12T + 2\tau_{\text{ind}}$ where the minimum is $6T + 2\tau_{\text{ind}}$ (in half the number of cases), and the maximum is $6nT + 2\tau_{\text{ind}}$ (only once in the whole operation cycle of the counter), where τ_{ind} is the indicator's delay.

Using the counter of Fig. 11.22 as the basis, we can build circuits with *arbitrary counter capacity*, other than 2^n . For example, the capacity 5 can be realized by the circuit of Fig. 11.23.

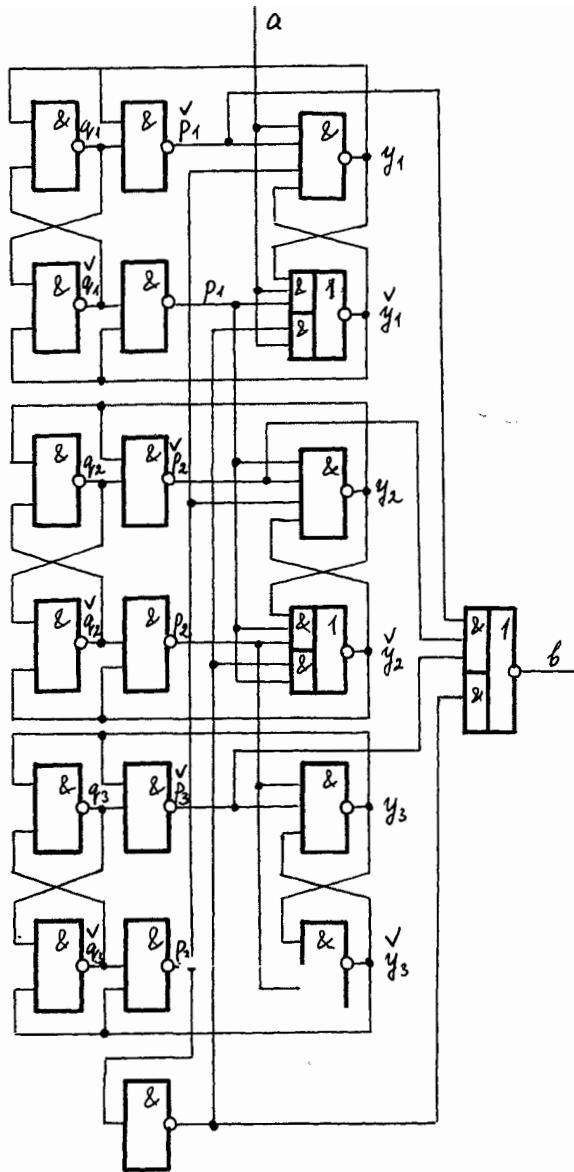


Figure 11.23. A scale of five counter.

Generally, after the occurrence of an overflow in the counter, the latter is not set to the all-zero state but to the state corresponding to $2^n - l$, by which any count

scale l can be achieved. The count scale may be defined by the externally preset double-rail value \overline{y}_j . Then the flip-flop \overline{y}_j is implemented according to the equations

$$y_j = \overline{\overline{y}_j p_j p_{j-1} p_n \vee p_{j-1} p_n \overline{y}_j}, \quad \overline{y}_j = \overline{y_j p_j p_{j-1} \vee p_{j-1} \bar{p}_n l_j},$$

$$1 \leq j \leq n, \quad p_0 = a,$$

and the inherent function for the indicator will be

$$\overline{b} = \overline{\overline{y}_1 \overline{y}_2 \dots \overline{y}_n \vee \bar{p}_n}.$$

Fig. 11.24 illustrates the circuit of a *decimal counter* which is based on the flip-flops of Fig. 4.12(b) and (c).

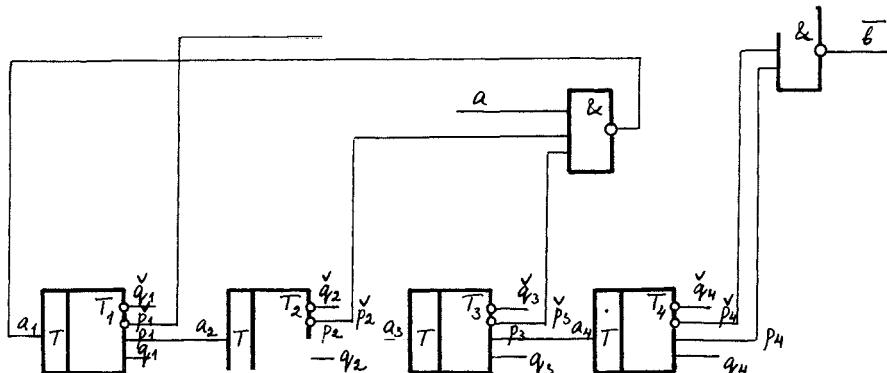


Figure 11.24. Decimal counter.

This counter works on the 1125 encoding scheme and is an example of the particular case where the appropriate binary values of the count scale are substituted into the above equations.

By connecting the output signal of the indicator of the counter with scale l to its complementing input, we obtain the circuit in which the l -times change of this signal will correspond to exactly one change of the output of any element in the most significant bit stage. Breaking the wire connecting the output of the chosen element to the inputs of the other elements of the counter, we obtain a “frequency multiplier” – to one change of the input of this circuit, we have l changes of the indicator's output signal. Now, using the interconnection theorem, we may insert such a circuit into the break of some connection in the most significant bit of another counter having, say,

the count scale m . As a result, we obtain a *scaler circuit* with count ratio m/l , i.e. with any given rational count scale.

An example of the *aperiodic counter with all-zero reset state* is shown in Fig. 11.25.

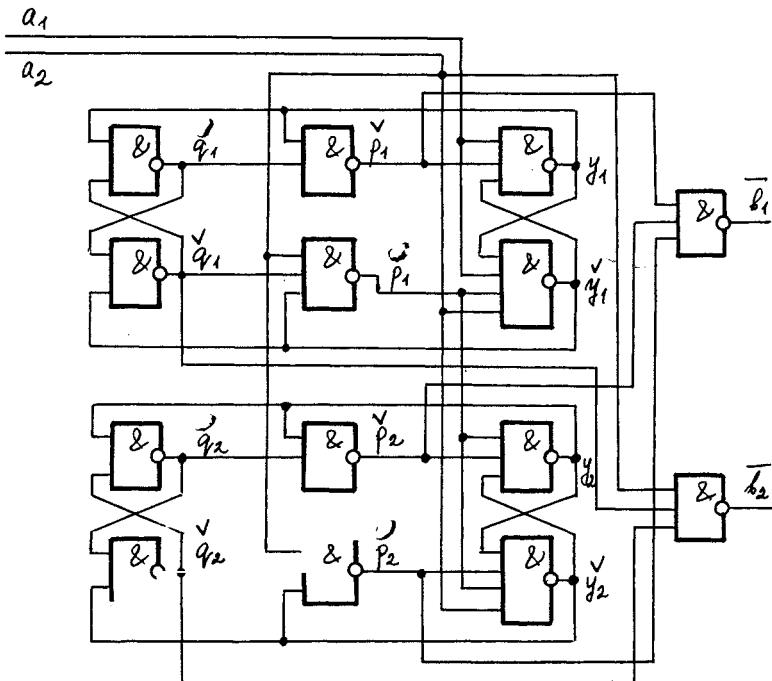


Figure 11.25. Counter with reset mode implementation.

In this circuit, a_1 is the complementing input, \bar{b}_1 is the first completion indicator output (for the counting mode), a_2 is the reset input, and \bar{b}_2 is the second completion indicator output (for the reset mode). When $a_1 = a_2 = 1$, the counter stores some data. The change over to either the counting or reset mode is initiated by the 1-0 transition of the associated signal a_1 or a_2 . The combination $a_1 = a_2 = 0$ is prohibited. The resetting completion is signalled by the change of \bar{b}_2 from 0 to 1. After that, we may change a_2 back to 1, and the indicator will respond with the 1-0 transitions of \bar{b}_2 , thereby identifying the completion of the second phase of the reset mode. The maximum length of the reset procedure is $6T$. In the counter mode, the counter operates in a way similar to that of Fig. 11.22.

We can simplify the organization of completion indication in the serial counter using the Muller theorem of interconnection of semi-modular circuits (see Section

6.2). Recall that the example of applying the theorem illustrated in Fig. 6.3 involved the serial counter in which the completion of transition processes in all of its bits was indicated through the elements of the first bit. By contrast, although the counter shown in Fig. 11.26 exploits the same indication principle, it has a higher speed due to the fact that in each bit, except for the last one, we have deleted two elements. The function of the deleted elements is now performed by the elements of the next bit because of having some extra inputs to the elements of the current bit stage (see Fig. 11.26).

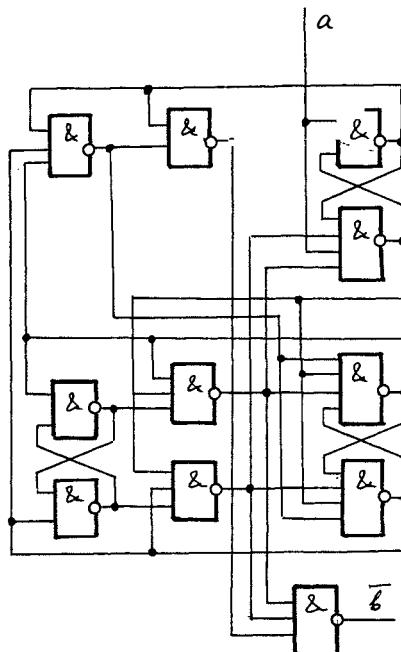


Figure 11.26. Counter obtained by applying the Muller theorem.

The above counters were built of the T-flip-flops of Fig. 4.12(b). However, an alternative can be the use of other aperiodic T-flip-flops, say that of Fig. 4.12(a). The counter circuit built of such flip-flops is shown in Fig. 11.27. The inherent function of the indicator has the form

$$\overline{b} = \overline{\overline{q}_n y_n} \vee \bigvee_{i=1}^n q_i \overline{y}_i .$$

The operational cycle of one bit here is $4T$, and since in this counter, as in the previous one, the average number of working bits in the cycle is two, the average cycle length for this counter is therefore $8T + 2\tau_{ind}$.

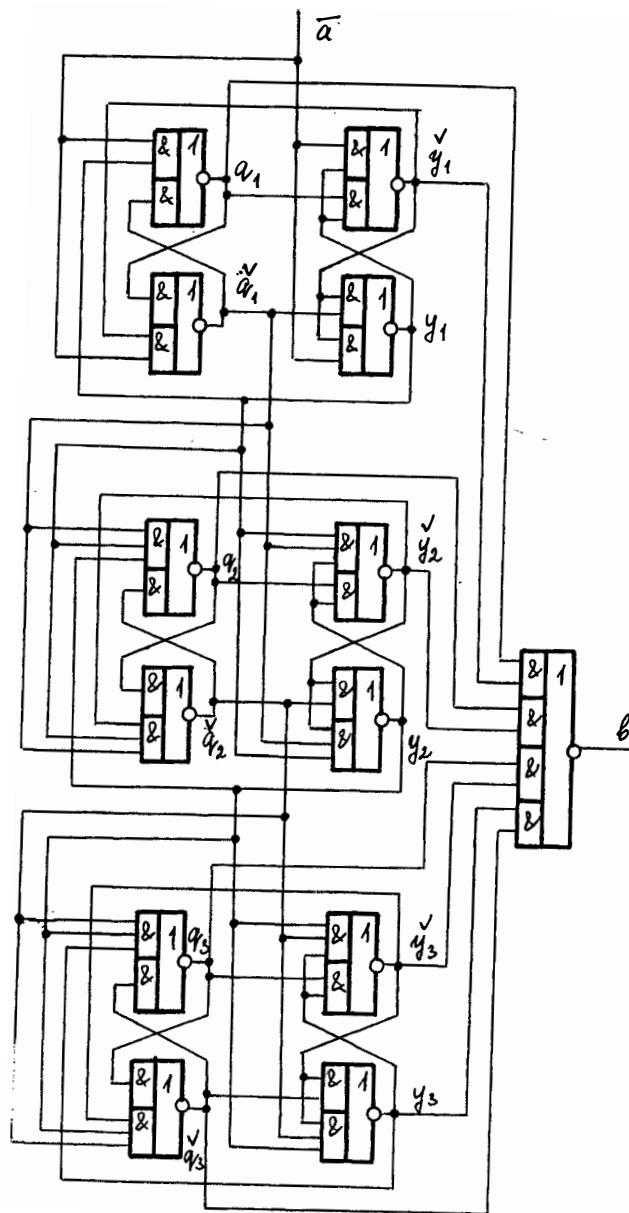


Figure 11.27. Counter based on the flip-flops of Fig. 4.12(b).

Aperiodic counters may also be constructed using the pipelination idea. The transition processes in different bit stages can be organized in a parallel way.

The operation of a *pipeline counter* can be briefly described as follows. The number of changes of the i -th bit signals must not exceed half the number of changes of the $(i - 1)$ -th bit signals. This requirement is satisfied if every bit has two working states P_i^0 and P_i^1 , and the i -th bit alternates between them whenever it comes from the working state. Furthermore, its transition to the working state only happens when the $(i - 1)$ -th bit is in the working state, say, P_{i-1}^1 . To remember the previous working state, each bit must have two idle states \mathcal{J}_i^0 and \mathcal{J}_i^1 .

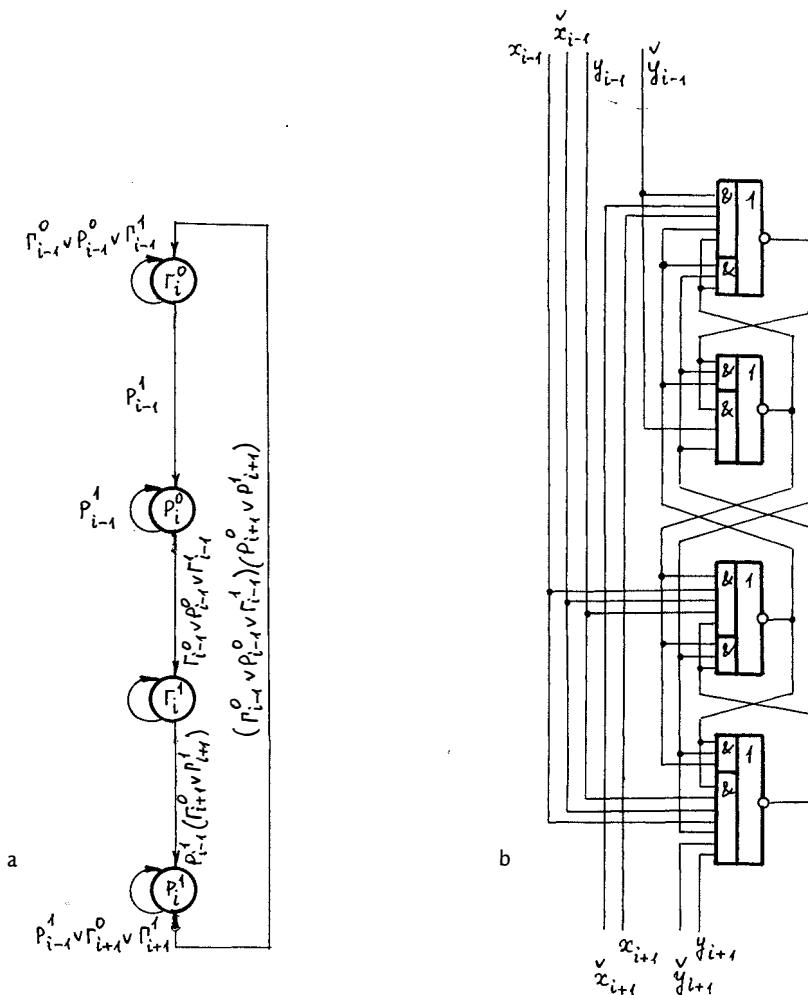


Figure 11.28(a) and (b). (a) State-graph for a pipeline counter cell, and (b) its implementation.

The state graph for the i -th bit of the pipeline counter is shown in Fig. 11.28(a), and its implementation is illustrated in Fig. 11.28(b). The values of outputs $x_i, \dot{x}_i, y_i, \dot{y}_i$ are associated with the state symbols by the following mapping: 0111- J_i^0 , 1011- J_i^1 , 1101- P_i^0 , 1110- P_i^1 .

The pipeline counter built on the basis of an ordinary serial counter is presented in Fig. 11.29.

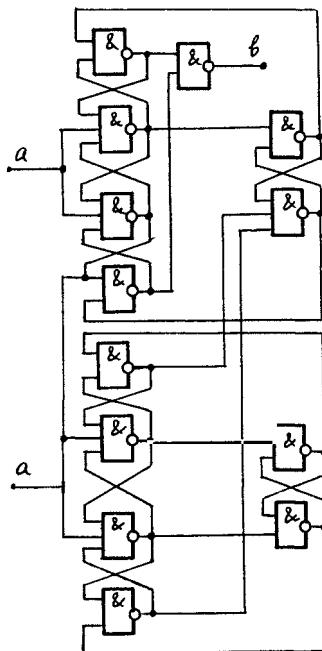


Figure 11.29. Pipeline counter.

We conclude this chapter with the note that the main advantage of pipeline devices, which makes their use so promising, is their high performance. The handful of examples shown in this chapter may be seen as the result of applying a design style based on the previously described modelling circuits.

11.6 Reference notations

All the circuits presented in this chapter are the original developments of the authors. The circuit modules that had been designed prior to 1975 were described earlier in [6]. The majority of the other circuits have been patented, and the interested reader may like to see some of them, [13] to [44].

The average carry propagation lengths for both aperiodic counters and adders have been obtained in [59].

EDITOR'S EPILOGUE

I do not know if Professor N is a great writer or not, but whoever reads his book through to the end, is undoubtedly a great reader.

From a review.

Dear Reader,

If you have reached this epilogue, we may assume that you have read all of this book. We are very grateful to you. But please do not imagine that you can now design every circuit in the best possible way. We came to such a conclusion after working for two years after the day this book was finished! This does not mean, however, that the book is in vain. It is the place where a theoretical foundation can be found that enables us to design correct circuits with the many positive features presented in the introduction, and then justified in subsequent chapters. It points out some universal techniques for the analysis and synthesis of circuits whose behaviour is invariant to the delays of logical elements.

We should, however, bear in mind that, because of the universality, such techniques appear to be good in all cases, but may be bad in each particular case. The design process is a process of creative work, whose prerequisites are: theoretical knowledge, design experience and the skill of the designer. If any of these components are lacking, or are underestimated, the whole design labour may be lost. We are now encouraged to conclude the book with a discussion of some considerations that seem to be rather important.

The "classical" methods of logical synthesis are known to be constrained by the so-called combinatorial explosion. The reported asymptotic bounds seem to impede all efforts to construct circuits consisting of more than two or three dozen Boolean variables. However, in practice, we often work with devices whose behaviour is defined by hundreds of variables. Why does the theory not work here? The reason is that, in practice, we never design circuits in general, but normally deal with a rather restricted class, which can be characterized as "circuits that are met in practice". The combinatorial explosion, in the classical sense, is also extended to the technique of specifying the behaviour of a circuit. If we are able to specify such a behaviour, in some language, then we may expect that the implementation will be of the same order of complexity as the original specification. This methodologically general concept brings us to the idea of modular synthesis as an obvious consequence of modular specification. The notion of the "circuits that are met in practice" is also a rather obvious consequence of "practically met specifications", and, no doubt, such specifications *always* have a modular structure. The languages for specifying

co-operating and communicating asynchronous processes are the languages that express algorithms of the interaction and co-ordination of modules which are the matched modules. The construction of such languages, studying their properties, and developing methods for analyzing the consistency of the specification, constitute the main objective of theory. The efficiency and conciseness of the specification always results from the design experience and inventiveness of the designer. Theoretical methods for optimizing the specification, with their intrinsic exponential complexities, would again be effective only in combination with an experienced and skilled designer.

The next important design step is the conversion of a high-level specification into a circuit implementation. For structured design methodology, this implies the direct *translation* of a modular description of an asynchronous process into the representation of a related modelling circuit. The correspondence established between language constructs and circuit fragments allows us to note a rather obvious issue – the complexity of a modelling circuit is linearly related to that of the original specification. This fact has recently given rise to some efficient synthesis methods, based on signal graphs, that do not involve the introduction of additional memory variables into the circuit. Although the latter seems quite an arguable statement, since any device must have an internal memory, it does, however, conceal the fact that the memory elements are implicitly implemented within the circuit cells which are the language construct equivalents.

Further development of this idea has led us to the concept of an *auto-correct* implementation⁽¹⁾. Let us refer to an example. Assume that the system's behaviour is given by a non-safe Petri net. The insertion of *one* extra wire between each pair of adjacent cells, modelling adjacent places in the Petri net, or, speaking in terms of a transistor level, introducing one (two for CMOS technology) more transistor, we can pipeline the circuit, thereby making a non-safe net be implemented by a safe circuit. Therefore, a special selection of building blocks, or cells, may help in the removal of some flaws in the original specification, without any verification or correction. In addition, the modification may significantly increase the specification dimensions.

There is, however, one type of incorrectness which cannot be obviated, either by the implementation, or by the changing of the specification. This is because it originates from the interaction between a described mechanism and the environment. This is the arbitration phenomenon. In the last few years, work done by the group represented by the authors, as well as the work of some other researchers, has given rise to certain new facts in understanding the physical foundations of electronic arbitration. The oscillatory anomaly has been proved as a potentially possible phenomenon, not only on the logical, but also on the physical level. Furthermore,

(1) See Recent publications, [31].

while the meta-stability is curable at the circuit level, the situation with the oscillatory anomaly is essentially more difficult. The search for the reduction of the probability of entering the oscillation mode must be done at the layout level, and furthermore, this problem requires deeper investigation. What can be asserted now, is that *the change over from synchronous circuits to delay-insensitive ones, significantly reduces the overall probability of the occurrence of arbitration conditions in a device.*

Notwithstanding what has been said above, the reader will not find a definitive answer to the question which may deeply interest him (her): whether the logic designer should unconditionally give preference to synchronous or to delay-insensitive circuits? Rather, we quote, "Scriptur ad narrandum, non ad probandum" ("One writes to narrate, not to prove" – Quintillian, Learning the Oratory Art⁽¹⁾). We did not plan to find an answer to the above question, but rather sought to find some way to shed light on the status and capabilities of speed-independent circuit theory and the theory of aperiodic automata, as applied to the problem of asynchronous process control, or, if you prefer, to the problem of modular synthesis in the class of delay-insensitive circuits. However, in conclusion, it may be useful to present some comparative considerations bearing on different circuit design principles.

As has already been noted in the introduction, the last few years have been marked by increasing interest in the research and use of self-timed circuits and systems. The self-timing problem has been included in major R and D programmes in the fields of VLSI design. Nevertheless, the discussion of the prospects of using various synchronization techniques is still under way. What is the major argument against the wide-spread use of self-timed circuits? This argument, or better say prejudice, is the feeling that self-timed circuits are much more complex than their synchronous counterparts. The change-over from the synchronous to the asynchronous principle is now, reportedly, assessed as a doubling of the circuit dimension. Unfortunately, the slogan "Do not save silicon", which used to be heard among LSI designers, appears inconsistent. In fact, the "watchful guard" which prevents the growth of the integration scale, is the yield, which roughly speaking, is exponentially related to the area of the fabricated circuit. For highly complex circuits, the yield may fall to 0.5 percent. So, in this case, if we reduce the circuit complexity by just 5 percent, the yield will increase by 50 percent, and a reduction of 10 percent in the complexity increases the yield by 70 percent. Hence, what can be said about doubled complexity?

This is, however, an issue where the situation has been radically changed over the last two years. Let us, first of all, refer to Fig. 11.26. This figure presents a circuit for the counter that was obtained from the well-known "Harvard circuit" by using the theory of delay-insensitive circuits. Having this circuit implemented in the

(1) The original is *Institutio Oratoria*.

standard TTL-Schottky gate array (by four-input NANDs), we obtain a 20 percent saving even for ordinary synchronous implementation. The example is unique, but, at the same time, encouraging.

Our research group has been working, for more than fifteen years, in the field of the theory of aperiodic state machines and circuits that are delay-independent. During these years, we have been able to reduce the complexity of circuits, particularly, the building blocks. This can be seen by comparing the circuits presented in the book "Aperiodic Automata" [6], published in 1976, and those in this book. The last couple of years have been phenomenal, but unfortunately, we could not manage to include the results of the research in this book. The main progress has been due to the change-over from gate level to transistor level.

Now, what do we mean by progress in the field of constructing building blocks?

Firstly, it is the implementation of logical functions. As is clear from the text, the circuits which are speed-independent require the double-rail (in a few cases, the self-synchronizing with logarithmic increase) representation of signals, which has, to a great extent, determined the doubling of their complexity as compared with conventional circuits. It has, however, been shown that, when using a two-phase discipline with spacer, the double-rail implementation in a CMOS basis does not result in a doubling of the number of transistors as against a clocked single-rail logic. This is because of the fact that CMOS-circuits, even in their single-rail format, contain channels of both direct and complemented conductivity.

Secondly, it is the implementation of indicators. Aperiodic circuits have special paths for the indication of transition process completion. These paths are usually implemented using Γ -flip-flops. Since a Γ -flip-flop contains the AND term, the number of possible inputs is limited in it, but the necessity of using either pyramids of bounded Γ -flip-flops, or the parallel compression structure, will lead to an increase in the size of the circuitry and in the delay of the indication system. The latter may seriously affect the positive feature of the ability to operate with real element delays, which is inherent in aperiodic circuits. The two-way conductivity of the MOS transistor allows the construction of a circuit for a two-layer input with a practically unbounded number of inputs, and with a silicon expenditure of four transistors per indicated input.

Thirdly, the finding of such classes of devices that can be implemented in an aperiodic circuit with a negligibly small increase in hardware compared to a synchronous implementation. For counters, this was known as far back as 1974. Now solutions of the same kind have been found for memory units.

As aperiodic circuits give certain margins above conventional logic, these margins, of course, must be paid for. In these circumstances, the price is the circuit complexity and the design complexity. The latest results, however, give some hope

that this price will not be excessive, or, at least, will be commensurable with what is gained by their use.

It should be noted that it was precisely the absence of advanced methods for efficient design work that led the American researchers in the self-timing area into the search for compromises. What can now be found as self-timed circuitry in American literature, say in the book by Mead and Conway "Introduction to VLSI Systems", or in the proceedings of annual conferences on VLSI held in Caltech, cannot, however, be regarded as delay-insensitive implementations. These designs are correct and self-timed only with respect to a given ratio between element delays. No doubt, the idea of constructing circuits with limited delay-independence is quite productive, and, in many cases, results in certain simplifications (though, not major ones); this idea requires special study. However, when abandoning total delay-independence, we also depart from a very important, perhaps the most important, feature of the delay-insensitive circuits which is their total self-checking capability with respect to stuck-at faults.

The problem of fault detection and fault recovery, i.e. the self-repair problem, is increasingly assuming paramount importance. This is not only because of the need for greater reliability in circuits, but also, from the viewpoint of the integration scale growth, the benefit of automatic recovery from technological faults, and thereby, corresponding yield enhancement.

At present there is a strong trend towards creating wafer-scale integrated systems; the systems on a wafer can be of a size up to 100cm^2 . The reported manufacturing strategies for such circuits are chiefly based on the off-line testing of the wafer, the detection of faulty components, and the laser-based or high-current-based reprogramming. No wonder these technologies are so difficult and expensive! By contrast, self-repairing circuits open up an alternative way where the fault recovery may be seen as continuous over the whole circuit life-cycle, including assurance of reliability in maintenance. Such an approach would, however, be expedient only if the complexity of fault localization and reconfiguration circuitry is not too great. It is hoped that the combination of using delay-insensitive circuits and organizing on-line redundancy will eventually do a good job. There have, recently, been some promising results in this field.

Another important issue is the problem of interfacing and dealing with long intra-chip wires. Note that a one-millimetre wire takes an area equal to that of several gates, and having a double-rail interconnection would be rather expensive. An attractive opportunity is given by the idea of using a three-state bus line for the data transfer, which carries an acknowledgement to the signal along the same wire as the signal.

All these problems will be the subject of discussion in a forthcoming book which is already in preparation. We hope that our work on the manuscript and the usual publishing routine will not take more than five years.

We also express our hope that the reader's involvement in the problems outlined in this book will initiate him (her) into the fascinating world of delay-insensitive circuits and aperiodic machines as well as encourage him (her) to make a contribution to the areas of both theory and practice in the design of such circuits.

Leningrad
February 1986.

V. Varshavsky.

REFERENCES

Articles behave in the same way as the population, except that they apparently need a dozen to produce one more article, whereas among people a pair would be enough.

D. Price.

SPECIAL NOTE ON REFERENCES

The list of references contains a large number of works which are in Russian. The comment “(in translation)” at the end of the reference means that translations in English are published in USA, UK and elsewhere.

References 13 to 44 are patent specifications and some of these may also be in English.

The “mapping” between the various Soviet journals and the English translations is as follows:

1. Avtomatika i Telemekhanika (USSR) – Automation and Remote Control (USA).
2. Izvestiya Akademii Nauk SSSR. Tekhnicheskaya Kibernetika (USSR) – Soviet Journal of Computer and Systems Science (USA).
3. Avtomatika i Vychislitel'naya Tekhnika (USSR) – Automatic Control and Computer Sciences (USA).
4. Kibernetika (USSR) – Cybernetics (USA).
5. Problemy Peredachi Informatzii (USSR) – Problems of Information Transmission (USA).
6. Elektronnoe Modelirovaniye (USSR) – Electronic Modelling (UK).
7. Programmirovaniye (USSR) – Programming and Computer Software (USA).
8. Problemy Kibernetiki (USSR) – Problems of Cybernetics (USA).

REFERENCES

1. Avizienis, A. : ‘Fault Tolerance: The Survival Attribute of Digital Systems’, *Proc. IEEE*, 66 (10) (October 1978), 1109-1125.

2. Aksanova, G.N., Sogomonyan, E.S. : 'The Design of Self-checking Built-in Checkers for Sequential Machines with Memory', *Avtomatika i Telemekhanika* (USSR), (7) (1975), 132-142 (in translation).
3. Ambartzumyan, A.A., Potekhin, A.I. : 'Canonical Implementations for an Asynchronous State Machine', *Avtomatika i Telemekhanika* (USSR), (10) (1977), 122-131 (in translation).
4. Varshavsky, V.I., Kishinevsky, M.A., Taubin, A.R., Tzirlin, B.S. : 'Analysis of Asynchronous Logical Circuits. I. The Reachability Problem and Speed-Independent Circuits. II. The Reachability of Operational States and Effects of Wire Delays', *Izvestiya Akademii Nauk SSSR. Tekhnicheskaya Kibernetika* (USSR), (3) (1982), 137-149; (4) (1982) 84-97 (in translation).
5. Unger, S.H. : *Asynchronous Sequential Switching Circuits*. Wiley Interscience, New York, 1969.
6. Varshavsky, V.I. et al. : *Aperiodic Automata*, Nauka, Moscow, 1976 (in Russian).
7. Artukhov, V.J., Kopeikin, G.A., Shalyto, A.A. : *Adjustable Modules for Logical Control Devices*. Energoisdat, Leningrad, 1981 (in Russian).
8. Varshavsky, V.I., Marakhovsky, V.B., Peschansky, V.A., Rosenblum, L.Ya., Taubin, A.R. : "Asynchronous Interfaces: Information Encoding, Organization", Research Report Preprint, Institute of Social and Economic Problems, Academy of Sciences, USSR, Leningrad, 1981 (in Russian).
9. Varshavsky, V.I., Marakhovsky, V.B., Peschansky, V.A., Rosenblum, L.Ya.. : 'Asynchronous Processes. I. Basic Definitions and Interpretation. II. Composition and Matching', *Izvestiya Akademii Nauk SSSR. Tekhnicheskaya Kibernetika* (USSR), (4) (1980), 137-1492; (5) (1980) 138-143 (in translation).
10. Askerov, Ch.I., Gamidov, V.V. : *Equivalent Representations of Discrete Devices*. Energiya, Moscow, 1978 (in Russian).
11. Astatnovsky, A.G. : *Aperiodic Computing Devices*. Ph. D. Thesis, Leningrad Electrical Engineering Institute, Leningrad, 1975 (in Russian).
12. Aho, A.V., Hopcroft, J.E., Ullman., J.D. : *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass. 1974.
13. Varshavsky, V.I. et al. : 'Flip-flop for Transition Process Completion Indication', USSR Patent Certificate No. 425318, *The Inventions Bulletin* (15), 1974.
14. Varshavsky, V.I. et al. : 'Asynchronous Adder', USSR Patent Certificate No. 491949, *The Inventions Bulletin* (42), 1975.
15. Astanovsky, A.G. et al. : 'Asynchronous Shift Register', USSR Patent Certificate No. 528612, *The Inventions Bulletin* (34), 1976.

16. Astanovsky, A.G. et al. : 'Asynchronous Counter', USSR Patent Certificate No. 532963, *The Inventions Bulletin* (39), 1976.
17. Astanovsky, A.G. et al. : 'Shift Register', USSR Patent Certificate No. 548892, *The Inventions Bulletin* (8), 1978.
18. Astanovsky, A.G. et al. : 'Complementing Flip-flop', USSR Patent Certificate No. 558380, *The Inventions Bulletin* (18), 1977.
19. Varshavsky, V.I. et al. : 'Universal Logic Module', USSR Patent Certificate No. 561182, *The Inventions Bulletin* (21), 1977.
20. Varshavsky, V.I. et al. : 'Serial Counter', USSR Patent Certificate No. 561298, *The Inventions Bulletin* (2), 1977.
21. Varshavsky, V.I. et al. : 'Parallel One-Phase Register', USSR Patent Certificate No. 583480, *The Inventions Bulletin* (45), 1977.
22. Varshavsky, V.I. et al. : 'Device for Monitoring Transition Processes in Asynchronous Logical Units', USSR Patent Certificate No. 598081, *The Inventions Bulletin* (10), 1978.
23. Varshavsky, V.I. et al. : 'Serial Counter', USSR Patent Certificate No. 618853, *The Inventions Bulletin* (29), 1978.
24. Varshavsky, V.I. et al. : 'Device for Monitoring Transition Processes in Logical Units', USSR Patent Certificate No. 658561, *The Inventions Bulletin* (15), 1979.
25. Varshavsky, V.I. et al. : 'Memory Cell for Buffer Register', USSR Patent Certificate No. 661606, *The Inventions Bulletin* (17), 1979.
26. Varshavsky, V.I. et al. : 'Serial Counter', USSR Patent Certificate No. 706934, *The Inventions Bulletin* (48), 1979.
27. Varshavsky, V.I. et al. : 'Asynchronous Distributor Cell', USSR Patent Certificate No. 718940, *The Inventions Bulletin* (8), 1980.
28. Varshavsky, V.I. et al. : 'One-Beat Shift Register', USSR Patent Certificate No. 723683, *The Inventions Bulletin* (11), 1980.
29. Varshavsky, V.I. et al. : 'Asynchronous Shift Register', USSR Patent Certificate No. 728161, *The Inventions Bulletin* (14), 1980.
30. Varshavsky, V.I. et al. : 'Buffer Memory Cell', USSR Patent Certificate No. 756479, *The Inventions Bulletin* (30), 1980.
31. Varshavsky, V.I. et al. : 'Reversible Buffer Shift Register', USSR Patent Certificate No. 780045, *The Inventions Bulletin* (42), 1980.
32. Varshavsky, V.I. et al. : 'Shift Register', USSR Patent Certificate No. 799009, *The Inventions Bulletin* (3), 1981.
33. Tzirlin, B.S., : 'Memory Cell for Buffer Register', USSR Patent Certificate No. 799010, *The Inventions Bulletin* (3), 1981.
34. Derbunovich, L.V., Shatillo, V.V. : 'Combinational Adder', USSR Patent Certificate No. 800992, *The Inventions Bulletin* (4), 1981.

35. Varshavsky, V.I. et al. : 'Device for Monitoring a Matched State Machine', USSR Patent Certificate No. 807307, *The Inventions Bulletin* (7), 1981.
36. Brailovsky, G.S. : 'Flip-flop Device', USSR Patent Certificate No. 807490, *The Inventions Bulletin* (7), 1981.
37. Varshavsky, V.I. et al. : 'Count Scaler Device', USSR Patent Certificate No. 834929, *The Inventions Bulletin* (20), 1981.
38. Tzirlin, B.S. : 'Reversible Buffer Shift Register', USSR Patent Certificate No. 841050, *The Inventions Bulletin* (23), 1981.
39. Varshavsky, V.I. et al. : 'Counter', USSR Patent Certificate No. 913604, *The Inventions Bulletin* (10), 1982.
40. Varshavsky, V.I. et al. : 'Asynchronous Distributor Cell', USSR Patent Certificate No. 924899, *The Inventions Bulletin* (16), 1982.
41. Tzirlin, B.S. : 'Memory Cell for Buffer Register', USSR Patent Certificate No. 928417, *The Inventions Bulletin* (16), 1982.
42. Brailovsky, G.S. : 'Aperiodic Impulse Device', USSR Patent Certificate No. 940283, *The Inventions Bulletin* (24), 1982.
43. Brailovsky, G.S. : 'Γ-Flip-flop', USSR Patent Certificate No. 945960, *The Inventions Bulletin* (27), 1982.
44. Varshavsky, V.I. et al. : 'Serial Counter', USSR Patent Certificate No. 953737, *The Inventions Bulletin* (31), 1982.
45. Bandman, O.L. : 'The Synthesis of Asynchronous Micro-program Control of Parallel Processes', *Kibernetika* (USSR), (1) (1980), 42-47 (in translation).
46. Baranov, S.I. : *The Synthesis of Micro-program Sequential Circuits (Flow-charts and State-machines)*. Energiya, Leningrad, 1979 (in Russian).
47. Birkhoff, G. : *Lattice Theory*. Third Edition, American Mathematical Society, Providence, R.I., 1967.
48. Blokh, A.Sh. : *The Synthesis of Switching Circuits*. Nauka i Tekhnika, Minsk, USSR, 1966 (in Russian).
49. Bukreev, I.N., Mansurov, B.M., Goryachev, V.I. : *Micro-electronic Circuits for Digital Devices*. Soviet Radio, Moscow, 1975 (in Russian).
50. Butakov, E.A. : *Methods for the Design of Relay Devices Based on Threshold Elements*. Energiya, Moscow, 1970 (in Russian).
51. Butrimenko, A.V. : *Development and Maintenance of Computer Networks*. Finansy i Statistika, Moscow, 1981 (in Russian).
52. Varshavsky, V.I. : 'Aperiodic Machines with Self-timing'. In: *Discrete Systems: Proc. IFAC Symposium, Riga 1974*, 1, Zinatne, Riga, USSR, (1974) pp. 9-25.
53. Varshavsky, V.I. : 'Modular Synthesis in the Class of Aperiodic Circuits'. In: *A Theory of Discrete Control Devices*. Nauka, Moscow, 1982, pp. 152-159 (in Russian).

54. Varshavsky, V.I. : *Collective Behaviour of Automata*. Nauka, Moscow, 1973. German translation: Kollektives Verhalten von Automaten, Akademie Verlag, Berlin, 1978.
55. Varshavsky, V.I., Kishinevsky, M.A. : 'Anomalous Behaviour of Logical Circuits and the Arbitration Problem', *Avtomatika i Telemekhanika* (USSR), (1) (1982), 123-131 (in translation).
56. Varshavsky, V.I., Rosenblum, L.Ya. : 'A Dynamic Stereotype of Terminal Device and Aperiodic State Machines'. In: *Robotics*, Mashinostroenie, Leningrad, 1979, pp. 18-22 (in Russian).
57. Varshavsky, V.I., Rosenblum, L.Ya. : *Methods for Hazard Avoidance in Asynchronous Circuits: A Tutorial*. Leningrad Electrical Engineering Institute, Leningrad, 1978 (in Russian).
58. Varshavsky, V.I., Rosenblum, L.Ya. : 'On the Minimization of Pyramid Circuits of Majority Elements', *Izvestiya Akademii Nauk SSSR, Tekhnicheskaya Kibernetika* (USSR), (3) (1964), 24-29 (in translation).
59. Varshavsky, V.I., Rosenblum, L.Ya., Starodubtzev, N.A. : 'On the Mean Time for Carry Signal Production in Aperiodic Counter and Adder circuits', *Avtomatika i Vychislitel'naya Tekhnika* (USSR), (3) (1975), 88-90 (in translation).
60. Varshavsky, V.I., Rosenblum, L.Ya. , Taubin, A.R. : 'Totally Self-checking Asynchronous Combinational Circuits and the Indicatability Property', *Avtomatika i Telemekhanika* (USSR), (5) (1982), 138-146 (in translation).
61. Varshavsky, V.I., Rosenblum, L.Ya., Timokhin, V.I. : 'A Protocol Model for the Interaction in Interactive Systems'. In: *Interactive Systems: Proceedings of Third School Seminar*. Metzniereba, Tbilisi, USSR, 1(1981), pp. 8-16 (in Russian).
62. Varshavsky, V.I., Rosenblum, L.Ya., Tzirlin, B.S. : 'Aperiodic Implementation of Operator Charts'. In: *Optimization in the Design of Discrete Devices. Proc. of the Seminar, LDNTP*, Leningrad, 1976, pp. 34-42 (in Russian).
63. Varshavsky, V.I., Rosenblum, L.Ya., Tzirlin, B.S. : 'On the Composition of Aperiodic Circuits', *Izvestiya Akademii Nauk SSSR. Tekhnicheskaya Kibernetika* (USSR), (1) (1980), 206-210 (in translation).
64. Varshavsky, V.I., Rosenblum, L.Ya., Tzirlin, B.S.: 'Functional Completeness in the Class of Speed-independent Circuits. Part.I', *Kibernetika* (USSR), (2) (1981), 12-15 (in translation).
65. Varshavsky, V.I., Rosenblum, L.Ya., Tzirlin, B.S.: 'Functional Completeness in the Class of Speed-independent Circuits. Part.II', *Kibernetika* (USSR), (1) (1982), 86-88 (in translation).

66. Vizirev, I.S. : ‘Self-timed Asynchronous Digital Circuits’. *Izvestiya VMEI “Lenin”* (Bulgaria), **34** (11) (1976), 419-425.
67. Volkov, A.F., Vedeshenkov, V.A., Zenkin, V.D. : *Automatic Fault Detection in Computers*. Soviet Radio, Moscow, 1965 (in Russian).
68. Astanovsky, A.G., Rosenblum, L.Ya., Starodubtzev, N.A., et al : ‘Computation and Control Structures Based on Aperiodic Self-timed Machines’. In: *Discrete Systems: Proc. IFAC Symposium, Riga 1974*, **3**, Zinatne, Riga, USSR, (1974) pp. 69-78.
69. Gavrilov, M.A. : ‘The Design of Relay Devices and Finite State Machines in a Modular Way’, *Izvestiya Akademii Nauk SSSR. Tekhnicheskaya Kibernetika* (USSR), (3) (1963), 13-27 (in translation).
70. Gavrilov, M.A. : ‘Today’s Problems in the Theory of Discrete Devices’. In: *Computer-aided Design of Discrete Control Devices*. Nauka, Moscow, 1980, pp. 3-30 (in Russian).
71. Gavrilov, M.A. : *The Theory of Relay-contact Circuits. Analysis and Synthesis of Relay-contact Circuit Structure*. USSR Academy of Sciences Press, Moscow, Leningrad, 1950 (in Russian).
72. Gavrilov, M.A., Devyatkov, V.V., Pupyrev, E.I. : *The Logical Design of Discrete State Machines*. Nauka, Moscow, 1977 (in Russian).
73. Gilbert, E.N. : ‘Synchronization of Binary Messages’, *IRE Transactions, IT-6* (4) (1960), 470-477.
74. Glushkov, V.M. : *The Synthesis of Digital Automata*. Phismathgis, Moscow, 1962 (in Russian).
75. Glushkov, V.M., Tzetlin, G.E., Yuschenko, E.L. : *Algebra, Languages, Programming*. Naukova Dumka, Kiev, USSR, 1974 (in Russian).
76. Golovkin, B.A. : ‘Methods and Tools for Parallel Data Processing’. In: *The Theory of Probability. Mathematical Statistics. Theoretical Cybernetics*. VINITI, Moscow, **17** (1979), 85-193 (in Russian).
76. Golovkin, B.A. : *Parallel Computer Systems*. Nauka, Moscow, 1980 (in Russian).
78. Gorbatov, V.A. : *A Semantic Theory of Automata Design*. Energiya, Moscow, 1979 (in Russian).
79. Gorozhin, A.D., Krainov, K.S. : ‘The Design of Totally Self-checking Combinational Circuits with the Use of Poynomial Forms’, *Avtomatika i Telemekhanika* (USSR), (12) (1979), 159-166 (in translation).
80. Goryashko, A.P. : *Logical Circuits and Real Constraints*. Energoatomizdat, Moscow, 1982 (in Russian).
81. Grätzer, G. : *General Lattice Theory*. Birkhäuser Verlag, Basel, 1978.
82. Gurtovzev, A.L., Petrenko, A.F., Chapenko, V.P. : *The Logical Design of Automatic Devices*. Zinatne, Riga, 1978 (in Russian).

83. Dennis, J.B., Fossin, J.E., Linderman, J.P. : 'Data Flow Charts'. In: *The Theory of Programming. Part II*. Computer Centre of Siberian Division of USSR Academy of Sciences, Novosibirsk, 1972, pp. 7-43.
84. Yablonsky, S.V., Lupanov, O.B. (Eds.) : *Discrete Mathematics and Mathematical Problems of Cybernetics 1*. Nauka, Moscow, 1974 (in Russian).
85. Domanitzky, S.M. : *The Design of Reliable Logical Devices*. Energiya, Moscow, 1971 (in Russian).
86. D'yachenko, V.F., Lazarev, V.G., Savvin, G.G. : *The Control in Communication Networks*. Nauka, Moscow, 1967 (in Russian).
87. Evreinov, E.V. : *Homogeneous Computer Systems, Structures and Environments*. Radio i Svyaz, Moscow, 1981 (in Russian).
88. Zakrevsky, A.D. : *Algorithms for the Synthesis of Discrete Automata*. Nauka, Moscow, 1971 (in Russian).
89. Zakrevsky, A.D. : *The Logical Synthesis of Cascade Circuits*. Nauka, Moscow, 1981 (in Russian).
90. Zakharov, V.N. : *Automata with Distributed Memory*. Energiya, Moscow, 1975 (in Russian).
91. Zakharov, V.N., Pospelov, D.A., Khazatzky, V.E. : *Control Systems. Specification. Design. Implementation*, Energiya, Moscow, 1982 (in Russian).
92. Zlatanov, P.S. : 'The Implementation of a Finite State Machine that is Independent of Element Switching Speed'. In: *Electronics and Modelling*, 13. Naukova Dumka, Kiev, USSR, 1976 (in Russian).
93. Gorelikov, N.I., et al. : *The Interface for Programmable Instrumentations in Computer-aided Experiment Systems*. Nauka, Moscow, 1981 (in Russian).
94. Kalachev, V.A., Kravchenko, A.V. : 'Towards Optimized Circuits of Double-rail Logic'. In: *Homogeneous Digital Computer and Integration Structures 9*, Taganrog, USSR, 1978, pp. 68-69 (in Russian).
95. Kalachev, V.A., Kravchenko, A.V. : 'The Problem of Checking Computer structures and Double-rail Logic' *Proc. of Fifth Seminar on Applied Aspects of Automata Theory*. Varna, 1979, pp. 556-562 (in Russian).
96. Karp, R.M., Miller, R.E. : 'Parallel Program Schemata'. *Journal of Computer and System Science* 3 (4) (May 1969), 167-195.
97. Kishinevsky, M.A. : *Implementation and Analysis of Aperiodic Circuits*. Ph.D. Thesis, Leningrad Electrical Engineering Institute, Leningrad, 1982 (in Russian).
98. Kishinevsky, M.A., Taubin, A.R., Tzirlin, B.S. : 'Modelling Methods for Analysis of Dialogue Interaction'. In: *Interactive Systems: Proc. of IV School Seminar*, Metzniereba, Tbilisi, USSR, 1(1982), pp. 8-16 (in Russian).

99. Kishinevsky, M.A., Taubin, A.R., Tzirlin, B.S. : 'Petri Nets and the Analysis of Switching Circuits', *Kibernetika* (USSR) (4) (1982), 114-117 (in translation).
100. Kibrinsky, N.E., Trakhtenbrot, B.A. : *Introduction to the Theory of Finite State Machines*. Phismathgiz, Moscow, 1962 (in Russian).
101. Kotov, V.E. : *Petri Nets*. Nauka, Moscow, 1984 (in Russian).
102. Kotov, V.E. : 'The Theory of Parallel Programming: Applied Aspects', *Kibernetika* (USSR), (1) (1974), 1-16, (2) (1974), 1-18 (in translation).
103. Kotov, V.E. : 'Algebra of Regular Petri Nets', *Kibernetika* (USSR), (5) (1980), 10-18 (in translation).
104. Kotov, V.E., Narinyany, A.S. : 'Asynchronous Computing Processes over Memory', *Kibernetika* (USSR), (3) (1966), 64-71 (in translation).
105. Kuznetsov, O.P. : 'On Asynchronous Logical Networks', *Problemy Perekadchi Informatzii* (USSR) 9 (1961), 103-115 (in translation).
106. Kuznetsov, O.P., Adelson-Velskey, G.M. : *Discrete Mathematics for Engineers*. Energiya, Moscow, 1980 (in Russian).
107. Lazarev, V.G., Piil', E.I. : *The Synthesis of Asynchronous Finite State Machines*. Nauka, Moscow, 1964 (in Russian).
107. Lazarev, V.G., Piil', E.I. : *The Synthesis of Control State Machines*. Energiya, Moscow, 1978 (in Russian).
109. Aizerman, M.A., Gusev, L.A., Rosenoer, L.I., et al. : *Logic, Automata, Algorithms*. Phismathgiz, Moscow, 1963 (in Russian).
110. Lyapunov, A.A. : 'On Logical Program Schemata'. *Problemy Kibernetiki* (USSR), 1 (1958), 46-71 (in translation).
111. Maznev, V.I. : 'On the Synthesis of Self-testing 1/p-testers', *Avtomatika i Telemekhanika*, (USSR), (9) (1978), 142-145 (in translation).
112. Maznev, V.I. : 'The Synthesis of Totally Self-checking Sequential Circuits', *Avtomatika i Telemekhanika*, (USSR), (9) (1977), 167-175 (in translation).
113. Glushkov, V.M., Kapitonova, Yu.V., Letichevsky, A.A., et al. : 'Macro-pipeline Computations of Functions over Data Structures', *Kibernetika* (USSR), (4) (1981), 13-21 (in translation).
114. Naumov, B.N. (Ed.) : *Small Computers and their Applications*. Statistika, Moscow, 1980 (in Russian)
115. Marchuk, G.I., Kotov, V.E. : *Modular Asynchronous Expandable System (The Concept)*. Pt. I. *Rationale and Main Trends for System Architecture Development*; Pt. II. *Main Principles and Features*. Computer Centre of Siberian Division of Academy of Sciences of USSR, Novosibirsk, Preprints Nos. 86, 87, 1978 (in Russian).
116. Mazda, F.F. : *Integrated Circuits : Technology and Applications*. Cambridge University Press, Cambridge, 1978.

117. Melikhov, A.N. : *Directed graphs and Finite State Machines*. Nauka, Moscow, 1971 (in Russian).
118. Escary, J.D., Proschau, F. : *The Reliability of Coherent Systems, Redundancy Techniques for Computer Systems*. Spartan books, Washington, D.C., 1963.
119. Bandman, O.L. (Ed.) : *Methods for Parallel Micro-programming*. Nauka, Novosibirsk, 1981 (in Russian).
120. Miller, R.E. : *Switching Theory II. Sequential Circuits and Machines*. John Wiley and Sons, New York, 1965.
121. Minsky, M. : *Computation : Finite and Infinite Machines*. Prentice-Hall, Englewood Cliffs, New York, 1967.
122. Mikheev, V.M. : 'About Sets Containing a Maximum Number of Pairs of Pairwise Incomparable Boolean Vectors', *Problemy Kibernetiki* (USSR), 2 (1959), 69-71 (in translation).
123. Varshavsky, V.I., Marakhovsky, V.B., Peschansky, V.A., Rosenblum, L.Ya. : 'A Model of an Asynchronous State Machine with Direct Transitions of Input and Output Values', *Avtomatika i Telemekhanika* (USSR), (11) (1980), 117-123 (in translation).
124. Enslow, P.H. (Ed.). : *Multi-processors and Parallel Processing*. Wiley-Interscience, New York, 1974.
125. Narinyany, A.S. : 'The Theory of Parallel Programming Formal Models', *Kibernetika* (USSR) (3) (1974), 1-15; (5), 1-14 (in translation).
126. Nikolenko, V.N. : 'On Transformations of Asynchronous Logical Circuits', *Kibernetika* (USSR) (5) (1978), 6-8 (in translation).
127. Nikolenko, V.N., Chebotarev, A.N. : 'On the Implementation of Non-deterministic State Machines'. In: *Design Methods for Hardware and Software*. Kiev, 1979, pp. 3-10 (in Russian).
128. Nilsson, N.J. : *Problem-solving Methods in Artificial Intelligence*. McGraw-Hill, New York, 1971.
129. Varshavsky, V.I., Marakhovsky, V.B., Peschansky, V.A., Rosenblum, L.Ya. : 'On the Implementability of an Asynchronous Interface Using a Self-synchronizing Code with Identifier', *Avtomatika i Vychislitel'naya Tekhnika* (USSR), (5) (1981), 84-88 (in translation).
130. Parkhomenko, P.P., Sogomonyan, E.S. : *Fundamentals of Engineering Diagnostics : Diagnosis Algorithm Optimization, Hardware Tools*. Energiya, Moscow, 1981 (in Russian).
131. Peterson, J.L. : 'Computation Sequence Sets', *Jounal of Computer and System Sciences*, 13 (August 1976), 1-24.
132. Plaks, T.P. : 'The Synthesis of Parallel Programs Using Computation Models', *Programmirovanie* (USSR), (4) (1977), 55-63 (in translation).

133. Popov, E.V. : *Communication with Computer in Natural Language*. Nauka, Moscow, 1982 (in Russian).
134. Popov, E.V., Firdman, G.R. : *Algorithmic Fundamentals of Intelligent Robots and Artificial Intelligence*. Nauka, Moscow, 1982 (in Russian).
135. Pospelov, G.S., Pospelov, D.A. : 'Artificial Intelligence'. *Vestnik Akademii Nauk SSSR* (USSR), (10) (1978), 26-36 (in Russian).
136. Pospelov, D.A. : *Introduction to the Theory of Computer Systems*. Soviet Radio, Moscow, 1972 (in Russian).
137. Pospelov, D.A. : *Logical Methods for Circuit Analysis and Synthesis*. Energiya, Moscow, 1974 (in Russian).
138. Pospelov, D.A. : *Logic-linguistic Models in Control Systems*. Energoizdat, Moscow, 1981 (in Russian).
139. Prangishvilli, I.V., Stetzura, G.G. : *Micro-processor Systems*. Nauka, Moscow, 1980 (in Russian).
140. Maiorov, S.A. (Ed.) : *The Design of Micro-processor Digital Devices*. Soviet Radio, Moscow, 1977 (in Russian).
141. Putintzev, N.D. : *Hardware Testing for Control Digital Computers*. Soviet Radio, Moscow, 1966 (in Russian).
142. Riordan, J. : *An Introduction to Combinatorial Analysis*. John Wiley and Sons, New York, 1958.
143. Roginsky, V.N. : *Fundamentals of Discrete Systems : The Statics and Dynamics of Discrete Machines*. Svyaz, 1975 (in Russian).
144. Rosenblum, L.Ya. : 'Petri Nets', *Izvestiya Akademii Nauk SSSR, Tekhnicheskaya Kibernetika* (USSR), (5) (1983), 12-40 (in translation).
145. Rosenblum, L.Ya., Tzirlin, B.S. : 'On the Implementation of Asynchronous Matched Circuits'. In: *Problems of the Theory of Computer Design and Data Processing System Design*. Kiev, 1976, pp. 52-59 (in Russian).
146. Rosenblum, L.Ya., Yakovlev, A.V. : 'Modelling Duologues'. In: *Duologue in Computer-aided Systems*. MDNTP, Moscow, 1981, pp. 118-125 (in Russian).
147. Rosenblum, L.Ya., Yakovlev, A.V. : 'The Model of Controlled Protocol'. In: *Packet Switching Computer Networks I. Proc. of Second All-Union Conference*. Riga, 1981, pp. 62-66 (in Russian).
148. Rotanov, S.V. : 'An Approach to Protocol Validation', *Avtomatika i Vychislitel'naya Tekhnika* (USSR) (4) (1982), 17-19 (in translation).
149. Sagalovich, Yu. : *State Encoding and Reliability of Sequential Machines*. Svyaz, Moscow, 1975 (in Russian).
150. Sagalovich, Yu. : 'Totally Dividing Systems', *Problemy Peredachi Informatzii* (USSR), 2 (1982), 74-82 (in translation).

151. Sapozhnikov, V.V., Sapozhnikov, V.V. Jr. : 'The Synthesis of Totally Self-testing Asynchronous Automata', *Avtomatika i Telemekhanika* (USSR) (1) (1978), 54-166 (in translation).
152. Sellers, F.F. Jr. : *Error Detecting Logic for Digital Computers*. McGraw-Hill, New York, 1968.
153. Zakrevsky, A.D. (Ed.) : *The Synthesis of Asynchronous State Machines with Computer-aided Tools*. Nauka i Tekhnika, Minsk, 1975 (in Russian).
154. Cypser, R.J. : *Communications Architecture for Distributed Systems*. Addison-Wesley, Reading, MA, 1978.
155. Scarlett, J.A. : *Transistor-Transistor Logic and its Interconnections : A Practical Guide to Micro-electronic Circuits*. Van Nostrand Reinhold Company, London, 1977.
156. Slabakov, E.V. : 'The Synthesis of Totally Self-checking Devices Using the Method of Splitting Inputs into Independent Groups', *Electronnoe Modelirovanie* (USSR) (4) (1980), 39-43 (in translation).
157. Slabakov, E.V. : 'Self-checking Computers and Systems', *Avtomatika i Telemekhanika* (USSR) (11) (1981), 147-167 (in translation).
158. Slagle, J.R. : *Artificial Intelligence : the Heuristic Programming Approach*. McGraw-Hill, New York, 1971.
159. Sogomonyan, E.S. : 'The Design of Self-checking Built-in Checkers for Combinational Circuits', *Avtomatika i Telemekhanika* (USSR) (2) (1974), 121-143 (in translation).
160. Sogomonyan, E.S. : 'The Design of One-output Self-checking Checkers', *Elektronnoe Modelirovanie* (USSR) (4) (1980), 26-31 (in translation).
161. Starodubtzev, N.A. : 'Autonomous Antitonus Sequential Circuits. I. Definitions and Interpretation. II. Cyclogrammes and Their Properties. III. Minimization. IV. Complexity Bounds', *Izvestiya Akademii Nauk SSSR. Tekhnicheskaya Kibernetika* (USSR) (4) (1981), 155-162; (5) (1981), 87-93; (6) (1981), 82-86; (1) (1982), 124-130 (in translation).
162. Starodubtzev, N.A. : *On the Organization of Control in Aperiodic Computer Devices*. Ph.D. Thesis, Leningrad Electrical Engineering Institute, Leningrad, 1975 (in Russian).
163. Starodubtzev, N.A. : *The Synthesis of Control Circuits of Parallel Computer Systems*. Nauka, Leningrad Division, Leningrad, 1984 (in Russian).
164. Tahl, A.A., Yuditzky, S.I. : 'Hierarchy and Parallelism in Petri Nets. I Complex Petri Nets. II Complex Automata Petri Nets with Concurrency', *Avtomatika i Telemekhanika* (USSR) (7) (1982), 113-122; (9) (1982), 83-89 (in translation).
165. Taubin, A.R. : *The Analysis of Aperiodic Circuits*. Ph.D. Thesis, Leningrad Electrical Engineering Institute, Leningrad, 1981 (in Russian).

166. Trakhtenbrot, B.A., Bardzin, Y.M. : *Finite Automata*. North-Holland, Amsterdam, 1973.
167. Tyugu, E.H. : 'Problem Solving Using Computer Models'. *Journal Vyshislitel'noi Matematiki i Matematicheskoi Physiki (USSR)* **10** (3) (1970), 716-733 (in translation).
168. Winston, P.H. : *Artificial Intelligence*. Addison-Wesley, Reading MA, 1977.
169. Utkin, A.A. : *The Analysis of Logical Networks and Techniques of Boolean Computations*. Nauka i Tekhnika, Minsk, 1979 (in Russian).
170. Fet, Ya.I. : *Parallel Processors for Control Systems*. Energoizdat, Moscow, 1981 (in Russian).
171. Finkelstein, R.L. : *Circuit Design Aspects for Construction of Aperiodic Computer and Control Devices* Ph.D. Thesis, LITMO, Leningrad, 1975.
172. Faure, R., Kaufmann, A., Denis-Papin, M. : *Mathématiques Nouvelles 1*. Dunod, Paris, 1964.
173. Friedman, A.D., Menon, P.R. : *Theory and Design of Switching Circuits*. Computer Science Press, Inc., New York, 1975.
174. Khazanov, B.I. *Interfaces of Instrumentation Systems*. Energiya, Moscow, 1979.
175. Zemanek, G. : 'Sequential Asynchronous Logic'. In: *Theory of Finite and Stochastic Automata: Proc. IFAC Symposium*. Nauka, 1965, pp. 232-245.
176. Tzirlin, B.S. : 'Algebra of Asynchronous Logical Circuits'. *Kibernetika (USSR)* (1) (1984), 16-20 (in translation).
177. Tzirlin, B.S. : *Algebra and Analysis of Asynchronous Logical Circuits*. Research Report Preprint, Institute of Social and Economic Problems, Academy of Sciences of USSR, Leningrad, 1981 (in Russian).
178. Tzirlin, B.S. : 'On Implementation Bases for Speed-Independent Circuits', *Izvestiya Akademii Nauk SSSR. Tekhnicheskaya Kibernetika (USSR)* (3) (1981), 121-126.
179. Tzirlin, B.S. : *On the Synthesis of Aperiodic Circuits*. Ph.D. Thesis, LIAP, Leningrad, 1976 (in Russian).
180. Tzirlin, B.S. : 'On the Implementation of Asynchronous Logical Circuits', *Kibernetika (USSR)* (5) (1981), 136 (in translation).
181. Chebotarev, A.N. : 'The Analysis of Asynchronous Logical Circuits', *Kibernetika (USSR)* (6) (1980), 14-23 (in translation).
182. Chebotarev, A.N. : 'Decomposition of Asynchronous Logical Circuits. Parts I and II', *Kibernetika (USSR)* (2) (1978), 1-9; (4) (1978), 6-9 (in translation).
183. Chebotarev, A.N. : 'Hazards in Asynchronous Logical Circuits', *Kibernetika (USSR)* (4) (1976), 8-11 (in translation).
184. Chebotarev, A.N. : 'Circuits and Automata. Parts I and II', *Kibernetika (USSR)* (5) (1976), 5-9; (5) (1979), 9-14 (in translation).

185. Chebotarev, A.N., Nikolenko, V.N. : 'Simple Decompositions of Asynchronous Circuits', *Kibernetika (USSR)* (6) (1979), 51-55 (in translation).
186. Chang, H.Y., Manning, E., Metze, G. : *Fault Diagnosis in Digital Systems*. Wiley-Interscience, New York, 1970.
187. Sholomov, L.A. *Fundamentals of Theory and Discrete Logical Computer Devices*. Nauka, Moscow, 1980 (in Russian).
188. Scherbakov, N.S. : *Self-correcting Discrete Devices*. Mashinostroenie, Moscow, 1975 (in Russian).
189. Yuditzky, S.A., Tagaevskaya, A.A., Efremova, T.K. : *The Design of Discrete Automatic Systems*. Mashinostroenie, Moscow, 1980 (in Russian).
190. Yakubaitis, E.A. : *Computer Network Architecture*. Statistika, Moscow, 1980 (in Russian).
191. Yakubaitis, E.A. : *The Synthesis of Asynchronous Finite State Machines*. Zinatne, Riga, 1970 (in Russian).
192. Yakovlev, A.V. : *The Design and Implementation of Communication Protocols in Asynchronous Backplane Interfaces*. Ph.D. Thesis, Leningrad Electrical Engineering Institute, Leningrad, 1982 (in Russian).
193. Akers, S.B. : 'A Truth Table Method for the Synthesis of Combinational Logic', *IRE Trans. on Electronic Computers*, EC-10 (4) (1961), 604-615.
194. Anderson, D.A., Metze, G. : 'Design of Totally Self-checking Check Circuits for m -out-of- n Codes'. *IEEE Trans. on Computers*, C-22, (3) (1973), 263-269.
195. Armstrong, D.B., Friedman, A.D., Menon, P.R. : 'Design of Asynchronous Circuits Assuming Unbounded Gate Delay', *IEEE Trans. on Computers*, C-18, (12) (1969), 1110-1120.
196. Ashjace, M., Reddy, S.M. : 'On Totally Self-checking Checkers for Separable Codes'. In: *International Symposium on Fault-tolerant Computing*. S.1, 1976, pp. 151-156.
197. Blanchard, M., Cavarroc, J.C., Gillon, J., et al. : *Rapport final du contrat DGRST H.7.2912 / DERA*. Tuillet, 1973.
198. Baer, J.L. 'A Survey of Some Theoretical Aspects of Multi-processing'. *Computing Surveys* 5 (1) (1973), 31-80.
199. Bartky, W.S., Muller, D.E. : *An Illiac Program for Simulating the Behaviour of Asynchronous Logical Circuits and Detecting Undesirable Race Conditions*. File Report No. 221, University of Illinois, Digital Computer Laboratory. (Presented at ACM National Meeting, Houston, Texas, June 1957).
200. Berger, J.M. : 'A Note on an Error Detection Code for Assymmetric Channels', *Information and Control* 4 (1) (1961), 68-73.

201. Bochman, G.V. : 'A General Transition Model for Protocols and Communication Services', *IEEE Trans. on Communications*, COM-28 (4) (1980), 643-650.
202. Bochman, G.V. : 'Finite-state Description of Communication Protocols'. *Proc. of Computer Network Protocols Symposium*. Liege, Belgium, 1978, pp. F3-1 - F3-11.
203. Bochman, G.V., Sunshine, C.A. : 'Formal Methods in Protocols Design', *IEEE Trans. on Communications*, COM-28 (4) (1980), 624-631.
204. Brand, D., Zafiropulo, P. : 'Synthesis of Protocols for an Unlimtied Number of Processes'. *Proc. of Computer Network Protocols Symposium*. Gaithersburgh, MD, 1980, pp. 29-40.
205. Bredeson, J.G., Hulina, P.T. : Generation of a Clock Pulse for Asynchronous Sequential Machines to Eliminate Critical Races'. *IEEE Trans. on Computers*, C-20 (2) (1971), 187-188.
206. Bruno, J., Altman, S.M. " A Theory of Asynchronous Control Networks". *IEEE Trans. on Computers*, C-20 (6) (1971), 629-638.
207. Bryant, R.E. : *Report on the Workshop on Self-timed Systems*. MIT Laboratory for Computer Science, Techn. Memo No. 166, Cambridge, MA, 1980.
208. Brzozowski, J.A., Yoeli, M. : *Digital Networks*. Prentice-Hall, Englewood Cliffs, New York, 1976.
209. Carter, W.C., Schneider, P.R. : 'Design of Dynamically Checked Computers'. In: *Information Processing '68 : Proc. IFIP Congress*, 2, Edinburgh 1968, pp. 878-883.
210. Catt, I. : 'Time Loss through Gating of Asynchronous Logic Signal Pulses'. *IEEE Trans. on Electronic Computers*, EC-15 (1) (1966), 108-111.
211. Cavarroc, J.C., Blanchard, M., Gillon, J. : 'An Approach to the Modular Design of Industrial Switching Systems'. In: *Proc. of International Symposium on Discrete Systems, Riga* 1974, 3 (1974), pp. 93-102.
212. Chaney, T.J. : 'Comments on 'A Note on Synchronizer and Interlock Maloperation'', *IEEE Trans. on Computers*, C-28 (10) (1979), 802-804.
213. Chaney, T.J. : *The Synchronizer 'Glitch' Problem*. Computer Systems Laboratory, TR No. 47, Washington University, St. Louis, 1974.
214. Chaney, T.J., Molnar, C. : 'Anomalous Behaviour of Synchronizer and Arbiter Circuits', *IEEE Trans. on Computers*, C-22 (4) (1973), 421-422.
215. Chaney, T.J., Ornstein, S.M., Littlefield, W.M. : 'Beware the Synchronizer'. In: *COMPCON -72: Proc. of the IEEE Computer Conference*. San Francisco, 1972.

216. Chaney, T.J., Rosenberger, F.U. : 'Characterization and Scaling of MOS Flip-flop Performance in Synchronizer Applications'. *Proc. of the Caltech Conference on VLSI*, Pasadena, 1979, pp. 357-374.
217. Corsini, P. : 'An n -user Asynchronous Arbiter', *Electronic Letters* 11 (1) (1975), 1-2.
218. Corsini, P. : 'Self-synchronizing Asynchronous Arbiter', *Digital Processes* 1 (1) (1975), 67-73.
219. Courans, G.T., Wann, D.F. : 'Theoretical and Experimental Behaviour of Asynchronous Flip-flop Operating in Meta-stable Region', *IEEE Trans. on Computers*, C-24 (6) (1975), 604-616.
220. Danthine, A. : 'Protocol Representation with Finite-state Models'. *IEEE Trans. on Communications*, COM-28 (4) (1980), 632-643.
221. David, R. : 'Totally Self-checking 1-out-of-3 Checker', *IEEE Trans. on Computers*, C-27 (6) (1978), 570-572.
222. David, R.. Thevenod-Fosse, P. : 'Design of Totally Self-checking Asynchronous Modular Circuits', *Journal of Design Automation and Fault-Tolerant Computing* 4 (1978), 271-287.
223. Day, J., Sunshine, C. : 'A Bibliography on the Formal Specification and Verification of Computer Network Protocols', *Computer Communications Review* 9 (4) (1979), 23-39.
224. Dennis, J.B. : 'First Version of Data Flow Procedure Language', *Lecture Notes in Computer Science* (19) (1974), 362-376.
225. Dennis, J.B. : 'Modular Asynchronous Control Structures for a High Performance Computer', *Records of the Project MAC Congress on Concurrent Systems and Parallel Computation*, ACM, New York, 1970, pp. 55-80.
226. Cook, R.W., Sisson, W.H., Storey, Y.F., Toy, W.H. : 'Design of Self-checking Micro-program Control', *IEEE Trans. on Computers*, C-22 (3) (1973) 255-263.
227. Diaz, M.: 'Design of Totally Self-checking and Fail-safe Sequential Machines', *Proc. of International Symposium on Fault-Tolerant Computing* , Urbana, 1974, pp. 3.19-3.24.
228. Elinean, G., Werner, W. : 'A New JK-flip-flop for Synchronizers', *IEEE Trans. on Computers*, C-26 (12) (1977) 1277-1279.
229. Frazer, W.D., Muller, D.E. : 'A Method for Factoring the Action of Asynchronous Circuits'. *Proc. of First Annual AIEE Symposium on Switching Circuit Theory and Logical Design* S-134 (October 1961), 246-249.
230. Friedman, A.D. : 'Feedback in Asynchronous Sequential Circuits', *IEEE Trans. on Electronic Computers*, EC-15 (5) (1966) 740-749.

231. Friedman, A.D., Menon, P.R. : 'Synthesis of Asynchronous Sequential Circuits with Multiple-input Changes', *IEEE Trans. on Computers* **C-17** (6) (1968) 559-566.
232. Friedman, A.D., Menon, P.R. : 'Systems of Asynchronously Operating Modules', *IEEE Trans. on Computers* **C-20** (1) (1971) 100-104.
233. Garcia, M.H. : 'Hardware Implementation of Communication Protocols: A Formal Approach', *Proc. Seventh symposium on Computer Architecture* (1980), 253-263.
234. Geffroy, J., Diaz, M. : 'Unified Approach to the Study of Self-checking Systems', *Digital Processes* (3) (1977), 289-306.
235. Gilbert, E.N. : 'Lattice Theoretic Properties of Frontal Switching Functions', *J. Mathematical Physics* **33** (1) (1954), 57-67.
236. Gioffi, G. : 'Autotesting Speed-independent Sequential Circuits', *IEEE Trans. on Computers* **C-27** (1) (1978) 90-94.
237. Grabowski, J. : 'On the Analysis of Switching Circuits by Means of Petri Nets', *Elektronische Informations-Verarbeitung und Kybernetik* **14** (1978), 611-617.
238. Hack, M. : 'Analysis of Production Schemata by Petri Nets', *Computer Structure Group, TR-94, Project MAC*, MIT, Cambridge, MA, 1972.
239. Hack, M. : 'Decidability Questions for Petri Nets', *Computer Structure Group, TR-161, Project MAC*, MIT, Cambridge, MA, 1972.
240. Hackbart, R.R., Dietmeyer, D.L. : 'The Avoidance and Elimination of Function Hazards in Asynchronous Sequential Circuits', *IEEE Trans. on Computers* **C-20** (2) (1971) 184-189.
241. Harangozo, J. " 'Protocol Definition with Formal Grammars'. *Proc. Computer Networks Protocols Symposium*. Liege, Belgium 1978, pp. F6-1 - F6-10.
242. Hojberg, K.S. : 'An Asynchronous Arbiter Resolves Resource Allocation Conflicts on a Random Priority Basis', *Computer Design* **16** (8) (1977), 120-123.
243. Holt, A.W., Commoner, F. : 'Events and Conditions', *Records of Project MAC Conference on Concurrent Systems and Parallel Computation*, New York, 1970, pp. 3-52.
244. *IEEE Transactions on Communications*, Special Issue on Computer Network Architectures and Protocols, **COM-28** (4) (1980), 624-661.
245. Jones, N.D., Landweber, L.H., Lien, Y.E. : 'Complexity of some Problems on Petri Nets', *Theoretical Computer Science* **4** (3) (1977), 277-299.
246. Jotwani, N.D., Jump, J.R. : 'Top-down Design in the Context of Parallel Program', *Information and Control* **40** (3) (1979), 241-257.
247. Jump, J.R., Thiagarajan, P.S. : 'On the Interconnection of Asynchronous Control Structures', *J. ACM* **22** (4) (1975), 596-612.

248. Keller, R.M. : 'A Fundamental Theorem of Asynchronous Parallel Computation'. *Lecture Notes in Computer Science* (24) (1975), 102-112.
249. Keller, R.M. : "Toward a Theory of Universal Speed-independent Modules", *IEEE Trans. on Computers* C-23 (1) (1974) 21-33.
250. Kinniment, D.J., Edwards, D.B.G. : 'Circuit Technology in a Large Computer System', *The Radio and Electronic Engineer* 43 (7) (1973), 435-441.
251. Kinniment, D.J., Woods, J.V. : 'Synchronization and Arbitration Circuits in Digital Systems', *Proc. IEE* 123 (10) (1976), 961-966.
252. Ko, D.C., Brener, M. : 'Self-checking of Multi-output Combinatorial Circuits Using Extended Parity Technique', *J. Design Automation and Fault-tolerant Computing* 2 (1) (1978), 29-62.
253. Kwong, Y.S. : 'On Reduction of Asynchronous Systems', *Theoretical Computer Science* 5 (1977), 25-50.
254. Landweber, L.H., Robertson, E.L. : 'Properties of Conflict-free and Persistent Petri nets', *J. ACM* 25 (3) (1978), 253-304.
255. Le Moli, G. : A Theory of Colloquies', *Alta Frequenza XLII* (10) (1976), 223E-230E.
256. Littlefield, W.M., Chaney, T.J. : 'The Glitch Phenomenon', *Computer Systems Laboratory, Techn. Memo No. 10*. Washington University, St Louis, MO, 1966.
257. Mago, G. : 'Realization Methods for Asynchronous Sequential Circuits', *IEEE Trans. on Computers* C-20 (3) (1971) 290-297.
258. Marino, L.R. : 'General Theory of Metastable Operation', *IEEE Trans. on Computers* C-30 (2) (1981) 107-116.
259. Marino, L.R. : 'The Effect of Asynchronous Inputs on Sequential Circuit Reliability', *IEEE Trans. on Computers* C-26 (11) (1977) 1082-1090.
260. Commoner, F., Holt, A.W., Even, S., Pnueli, A. : 'Marked Directed Graphs', *J. Computer and Systems Science* 5 (5) (1971), 511-523.
261. Marouf, M.A., Friedman, A.D. : 'Efficient Design of Self-checking Checker for any m -out-of- n Code'. *IEEE Trans. on Computers* C-27 (6) (1978) 482-490.
262. Mayne, D., Moore, R. : 'Minimize Computer 'Crashes'', *Electronic Design* 22 (9) (1974), 168-172.
263. Mayr, E. : 'An Effective Representation of the Reachability Set of Persistent Petri Nets', *Laboratory for Computer Science, Techn. Memo No. 188*. MIT, Cambridge, MA, 1981.
264. Mayr, E. : 'Persistiance of Vector Replacement Systems is Decidable', *Laboratory for Computer Science, Techn. Memo No. 189*. MIT, Cambridge, MA, 1981. (Also in: *Acta Informatica* 15 (3) (1981), 308-318.)

265. Merlin, P.M. : 'A Methodology for the Design and Implementation of Communication Protocols', *IEEE Trans. on Communications COM-24* (6) (1976) 614-621.
266. Merlin, P.M. : 'Specification and Validation of Protocols', *IEEE Trans. on Communications COM-27* (11) (1979) 1671-1680.
267. Merlin, P.M., Farber, D.J. : 'Recoverability of Communication Protocols. Implications of a Theoretical Study', *IEEE Trans. on Communications COM-24* (9) (1976) 1036-1046.
268. Mikami, Y. : 'Glitchless TTL Arbiter Selects First of Two Inputs', *Electronics*, **50** (12) (1977), 136.
269. Misunas, D. : 'Petri Nets and Speed-independent Design', *Comm. ACM* **16** (8) (1973), 474-481.
270. Muller, D.E. : 'Asynchronous Logic and Application to Information Processing'. *Proc. Symposium on the Application of Switching Theory in Space Technology*. Stanford University Press, March 1962, pp. 289-297.
271. Muller, D.E. : *Lecture Notes on Asynchronous Circuit Theory*. Spring 1961, Digital Computer Laboratory, University of Illinois, Urbana.
272. Muller, D.E., Bartky, W.S. : 'A Theory of Asynchronous Circuits'. *Proc. International Symposium on the Theory of Switching*. Volume **29** of the Annals of the Computation Laboratory of Harvard University, Harvard University Press, 1959, pp. 204-243.
273. Muller, D.E., Bartky, W.S. : 'A Theory of Asynchronous Circuits'. *Reports Nos. 75 and 78*. Digital Computer Laboratory, University of Illinois, 1956 and 1957.
274. Nakamura, T., Utsunomiya, K. : 'On a Universal Design Procedure to Realize the Semi-modular State Transition Graph', *Digital Processes* **3** (3) (1977), 237-257.
275. Net Theory and Applications. *Lecture Notes in Computer Science* (84), Springer-Verlag, Berlin, 1980.
276. Nozaki, A. : 'Hazard Analysis of Asynchronous Circuits in Muller-Bartky's Sense', *J. Computer and Systems Science* **13** (2) (1976), 161-171.
277. Ozguner, F. : 'Design of Totally Self-checking Asynchronous and Synchronous Sequential Machines'. *Proc. International Symposium on Fault-tolerant Computing*, 1977. Los Angeles, CA, 1977, pp. 89-97.
278. Patil, S.S. : 'An Asynchronous Logic Array', *Computer Structure Group, Memo-62, Project MAC*. MIT, Cambridge, MA, 1975.
279. Patil, S.S. : 'Circuit Implementation of Petri Nets', *Computer Structure Group, Memo-73, Project MAC*. MIT, Cambridge, MA, 1972.
280. Patil, S.S., Dennis, J.B. : 'The Description and Realization of Digital Systems', *RAIRO* (1) (1973), 55-69.

281. Pearce, R.C., Field, J.A., Little, W.D. : 'Asynchronous Arbiter Module', *IEEE Trans. on Computers C-24* (9) (1975), 931-932.
282. Pechoucek, M. : 'Anomalous Response Times of Input Synchronizers', *IEEE Trans. on Computers C-25* (2) (1976), 133-139.
283. Peterson, J.L. : *Petri Net Theory and the Modelling of Systems*. Prentice-Hall, New York, 1981.
284. Peterson, J.L. : 'Petri Nets', *Computing Surveys* 9 (3) (1977), 223-252.
285. Petri, C.A. : 'Kommunikation mit Automaten', *Schriften für des Rheinisch-Westfälischen Institute für Instrumentalrechnung und Mathematik, Universität Bonn*, Heft 2, Bonn, 1962.
286. Petriu, E. : 'N-channel Asynchronous Arbiter Resolves Resource Allocation Conflicts', *Computer Design* 19 (8) (1980), 126-132.
287. Plummer, W.W. : 'Asynchronous Arbiters', *IEEE Trans. on Computers C-21* (1) (1972), 37-42.
288. Priese, L. : 'An Automata Theoretical Approach to Concurrency', *Digital Systems Laboratory, Helsinki University of Technology, Otaniemi*. Series B (12) (1980).
289. Reddy, S.M. : 'A Note on Self-checking Checkers'. *IEEE Trans. on Computers C-23* (10) (1974), 1100-1102.
290. Sechovsky, H., Jura, S. : 'An Asynchronous S-I Arbiter in the Form of a Hardware Control Module'. *Proc. AFIPS Conference*. New York, 1976, pp. 777-782.
291. Seitz, C.L. : 'Self-timed VLSI Systems'. *Proc. Caltech Conference on VLSI*. Pasadena, CA, 1979, pp. 345-355.
292. Semantics of Concurrent Computation. Proc. International Symposium, Evian, 1979, *Lecture Notes in Computer Science* (70). Springer-Verlag, Berlin 1979.
293. Sifakis, J. : 'A Unified Approach for Studying the Properties of a Transition System', *Theoretical Computer Science* 18 (3) (1982), 227-258.
294. Smith, J.E., Metze, G. : 'Strongly Fault-secure Logic Networks', *IEEE Trans. on Computers C-27* (6) (1978), 491-499.
295. Starke, P.M. : *Petri Netze*. Veb. Deutscher Verlag der Wissenschaften, Berlin, 1980.
296. Strom, B.I. : 'Proof of the Equivalent Realizability of a Time-bounded Arbiter and a Runt-free Inertial Delay'. *Proc. Sixth Annual IEEE Symposium on Computer Architecture*. New York, 1979, pp. 178-181.
297. Stucki, M.J., Cox, J.R. : 'Synchronization Strategies'. *Proc. Caltech Conference on VLSI*. Pasadena, CA, 1979, pp. 375-393.
298. Sunshine, C.A. : 'Survey of Protocol Definition and Verification Techniques'. *Proc. of Computer Network Protocols Symposium*. Liege, Belgium, 1978, pp. F1-1 - F1-4.

299. Andre, C., Diaz, M., Girault, G., Sifakis, J. : 'Survey of French Research and Applications Based on Petri Nets'. In: *Lecture Notes in Computer Science* (84) (1980), 321-345. Springer Verlag, Berlin.
300. Thayse, A., Davio, M. : 'Boolean Differential Calculus and Its Application to Switching Theory', *IEEE Trans. on Computers* C-22 (4) (1973), 409-420.
301. Gotterez, G., Blanchard., Gillon, G., et al. : 'The Simulation of a Switching System's Requirements'. *Proc. IFAC Symposium on Discrete Systems 1974, Riga* 3, pp. 103-112.
302. Tohma, Y., Ohyma, Y., Sokai, R. : 'The Realization of Fail-safe Sequential Machines by Using an R -out-of- H Code', *IEEE Trans. on Computers* C-20 (11) (1971), 1270-1275.
303. Zafiropulo, P., West, C.H., Rudin, H., Cowan, D. : 'Towards Analyzing and Synthesizing of Protocols', *IEEE Trans. on Communications* COM-28 (4) (1980), 651-661.
304. Tracey, J.H. : 'Internal State Assignment for Asynchronous Sequential Machines', *IEEE Trans. on Electronic Computers* EC-15 (4) (1966), 551-560.
305. Valette, R. : 'Analysis of Petri Nets by Stepwise Refinements', *J. Computer and Systems Science* 18 (1) (1979), 35-46.
306. Valette, R., Diaz, M. : 'Top-down Formal Specification and Verification of Parallel Control Systems', *Digital Processes* (4) (1978), 181-199.
307. Varshavsky, V.I., Rosenblum, L.Ya. : 'Dead-beat Automata and Asynchronous Parallel Process Control'. In: *Preprints of the First IFAC-IFIP Symposium. SOCOCO-76, Tallin, 1976*, pp. 161-164.
308. Warshall, S. : 'A Theorem on Boolean Matrices', *J. ACM* 9 (9) (1962), 11-12.
309. West, C.H. : 'An Automated Technique of Communication Protocol Validation', *IEEE Trans. on Communications* COM-26 (8) (1978), 1271-1275.
310. Wormald, E.G. : 'A Note on Synchronizer or Interlock Maloperation', *IEEE Trans. on Computers* C-26 (3) (1977), 317-318.
311. Wormald, E.G. : 'Support for Chaney's "Comments on a Note on Synchronizer or Interlock Maloperation"', *IEEE Trans. on Computers* C-28 (10) (1979), 804.
312. Yoeli, M., Rinon, S. : 'Applications of Ternary Algebra to the Study of Static Hazards', *J. ACM* 11 (1) (1964), 84-97.
313. Zafiropulo, P. : 'Protocol Validation by Duologue-matrix Analysis', *IEEE Trans. on Communications* COM-26 (8) (1978), 1187-1194.

314. Zafiropulo, P., Rudin, H., Cowan, D. : 'Towards Synthesizing Asynchronous Two-process Interactions'. *Proc. Computer Network Protocols Symposium.* Gaithersburgh, MD, 1979, pp. 169-175.
315. Zissos, D., Duncan, F.G. : 'Micro-processor Interfaces', *Electr. Letters* 12 (23) (1976), 624-625.

ADDITIONAL REFERENCE

316. Findlay, J.N. : 'Time: A Treatment of Some Puzzles'. *The Australasian Journal of Psychology and Philosophy*, XIX, (1941), pp. 225-227.
Reprinted in Flew, A.G.N. : *Logic and Language Series* 1. (Oxford, 1951), pp. 45-47, and in Smart, J.J.C : *Problems of Time and Place*. Macmillan, New York, 1976, pp. 346-348. (These references quoted by Lucas, J.R. : *A Treatise on Time and Space*. Methuen. London, 1973.)

RECENT PUBLICATIONS ON SELF-TIMING

1. Varshavsky, V.I., et al. : 'Functional Completeness within the Class of Semi-modular Circuits', *Izvestiya Akademii Nauk SSSR. Tekhnicheskaya Kibernetika* (USSR) (4) (1985), 77-90 (in translation).
2. Varshavsky, V.I., et al. : 'The Self-timed Design Principle and Interface Models for VLSI Systems', *Avtomatika i Vychislitel'naya Tekhnika* (USSR) (3) (1985), 86-93 (in translation).
3. Varshavsky, V.I., et al. : 'The Implementation and Analysis of TRIMOSBUS, a Self-timed Interface', *Avtomatika i Vychislitel'naya Tekhnika* (USSR) (4) (1985), 83-90 (in translation).
4. Varshavsky, V.I., et al. : *Asynchronous Interfaces. A Tutorial Text*. Leningrad Electrical Engineering Institute, Leningrad, 1984 (in Russian).
5. Varshavsky, V.I., et al. : 'An Approach to Reliable Hardware Implementation of Physical Layer Protocols', *Avtomatika i Vychislitel'naya Tekhnika* (USSR) (6) (1986), 76-81 (in translation).
6. Varshavsky, V.I., et al. : 'Translation of a Physical Layer Protocol Specification into Self-timed Logical Circuits', *Avtomatika i Vychislitel'naya Tekhnika* (USSR) (5) (1987), 82-88 (in translation).
7. Varshavsky, V.I. : 'Hardware Support of Parallel Asynchronous Processes'. *Helsinki University of Technology, Digital Systems Laboratory, Series A : Research Reports* (2) (September 1987).
8. Varshavsky, V.I., et al. : 'Models for Specifying and Analyzing Processes in Asynchronous Circuits', *Izvestiya Akademii Nauk SSSR. Tekhnicheskaya Kibernetika* (USSR) (2) (1988), 171-190 (in translation).

9. Starodubtzev, N.A. : 'Asynchronous Processes and Antitonus Control Circuits. I., II., III.', *Izvestiya Akademii Nauk SSSR. Tekhnicheskaya Kibernetika* (USSR) (1, 4, 6) (1985), 10-19, 83-92, 115-122 (in translation).
10. Barney, C. : 'Logic Designers Toss out the Clock', *Electronics* (December 9, 1985), 42-45.
11. Barton, E.E. 'Non-Metric Design Methodology for VLSI'. In: *VLSI-81* (Ed. J.P. Gray), pp. 25-34. Academic Press, London.
12. Black, D.L. : 'On the Existence of Delay-insensitive Fair Arbiters: Trace Theory and Its Limitations', *Distributed Computing* 1 (4) (1986), 205-225.
13. Bochmann, G.V. : 'Hardware Specification with Temporal Logic: An Example', *IEEE Trans. on Computers* C-31 (3) (1982), 223-231.
14. Carter, T.M., Davis, A., Hayes, A.B., et al. : 'Transforming an ADA Program Unit to Silicon and Testing it in an ADA Environment', *Spring Compcon 84*. San Francisco, CA, 1984, pp. 448-455.
15. Kelem, S.H. : 'A Method for the Automatic Translation of Algorithms from a High-level Language into Self-timed Integrated Circuits', *IEEE Circuits and Devices Magazine* 1 (2) (1985), 17-19, 44.
16. Lister, P.F., Alhelvani, A.M. : 'Design Methodology for Self-timed VLSI Systems', *IEE Proc.* 132 (Pt. E) (1) (1985), 25-32.
17. Malachi, Y., Owicki. : 'Temporal Specification of Self-timed Systems'. In: *Proc. Carnegie-Mellon University Conference on VLSI Systems and Computers*, Pittsburgh, PA, 1981, pp. 203-212.
18. Martin, A.J. : 'The Design of a Self-timed Circuit for Distributed Mutual Exclusion'. In: *Proc. Chapel-Hill Conference on VLSI* 1985, pp. 245-260.
19. Martin, A.J. : 'Compiling Communicating Processes into Delay-insensitive VLSI Circuits', *Distributed Computing* 1 (4) (1986), 226-234.
20. Mishra, B., Clarke, E.M. : 'Hierarchical Verification of Asynchronous Circuits Using Temporal Logic', *Theoretical Computer Science* 38 (1985), 269-291.
21. Molnar, C.E. et al : 'Synthesis of Delay-insensitive Modules'. In: *Proc. Chapel-Hill Conference on VLSI* 1985, pp. 67-86.
22. Holaar, L.A. : "Direct Implementation of Asynchronous Control Units", *IEEE Trans. on Computers* C-31 (12) (1982), 1133-1141.
23. Frank, E.H., Sproul, R.F. : 'A Self-timed Static RAM'. In: *Proc. 3rd. Caltech Conference on VLSI* Pasadena, CA, 1983, pp. 274-284.
24. Rem, M., Snepscheut, J.L.A. van de. : 'Trace Theory and the Definition of Hierarchical Components'. In: *Proc. 3rd. Caltech Conference on VLSI* Pasadena, CA, 1983, pp. 225-239.
25. Rem, M. : 'Concurrent Computations and VLSI Circuits'. In: *Control Flow and Data Flow : Concepts of Distributed Programming*., (Ed. M. Broy), Springer, Berlin, 1985, pp. 399-437.

26. Snepscheut, J.L.A. van de. : *Trace Theory and VLSI Design*. Lecture Notes in Computer Science (200). Springer-Verlag, Berlin 1985.
27. Rosenblum, L.Ya., Yakovlev, A.V. : 'Signal Graphs: from Self-timed to Timed'. In: *Proc. International Workshop on Timed Petri Nets*. Torino, Italy, 1985, IEEE Computer Society Press, New York, 1985, pp. 199-207.
28. Chu, T.- A. : 'On the Models for Designing VLSI Asynchronous Digital Systems', *Integration, the VLSI Journal* (4) (1986), 99-113.
29. Udding, J.T. : 'A Formal Model for Defining and Classifying Delay-insensitive Circuits and Systems', *Distributed Computing* 1 (4) (1986), 197-204.
30. Yakovlev, A.V. : 'Designing Self-timed Systems', *VLSI Systems Design* (9), (1985), 70-90.
31. Varshavsky, V.I., Tiuasanen, M. : 'Hardware Support of Concurrent Process Interaction and Synchronization: on the Principle of Auto-correct Implementation'. *Helsinki University of Technology, Digital Systems Laboratory, Series A: Research Reports* (1988).

SUBJECT INDEX

Our words are giants when they are agianst us,
and dwarfs when we need their help.

W. Collins

Abstract synthesis, 43
Acknowledge, 114, 117
Acknowledgement, 232
see also “Request-acknowledge”
discipline
Acknowledgement time, bounded, 292
Action area, of module, 326-327
Acquiring access, 296
Adapter, 219-222
 interfacing, 215
Adder, modulo 2, 108, 113, 118
 one-bit, 75
Algebra, Boolean, 212-213
 homological, 211
 of circuits, 203-214
Alphabet, input, 39,65
 output, 39, 65
Allocators, 296-298
Analysis, 243-244, 279-280
 algorithms of, 245, 259-260, 278-279
 classification, 252-258
 complete, 257
 functional of asynchronous circuits, 243-244
 of marked graphs, 279
 of Petri nets, 279
 of speed-independent circuits, 280
 reachability, 244-252
Anomalies in circuit behaviour, 282

Anomalous behaviour, 3, 9, 134, 281
Anomalous modes, of operation 134
Anomalous state, 305-307
Anomaly, dragging in one of edges, 282
instable transition process, 282
meta-stability, 9, 281-2822, 288-292, 307-308
oscillatory, 9, 281-282, 286-288, 307-308
Aperiodic automaton, 10, 64
Arbitration, 9, 281-282, 307-308
 conditions, 3
 effect, 3
 facilities, 8
 phenomenon, 9, 282
 problem, 281
Arbiter, 3, 9, 132-134, 282-285, 292, 307-308
 bounded, 292, 300-307
 centralized, 295
 decentralized, 295
 distributed, 295
 multi-channel, 295, 308
 of k -th rank, 284
 reliable, 3
 self-diagnosing, 297
 unbounded, 292
Arc, incoming, 30,36
 of directed graph, 15, 33
 of marked graph, 30

- Arc (continued)
 - of parallel asynchronous flow chart, 36
 - outgoing, 30, 36
 - of signal graph, 32
- Assumption of delays, 65
- Asynchronous design requirements, 43
- Asynchronous pipeline, 11
- Asynchrony, 1
- Atomicity of semaphores, 134
- Auto-correct implementation concept, 365
- Automata equations, 109
- Average carry propagation length, 356
- Average gate delay (T), 90
- Bars, 25, *see also* Transitions
- Basis, 7, 140
 - functional, 7, 141
 - limited, 165
 - logical, 140
 - unconstrained, 7
- Bifurcator, 8, 30, 38, 126, 170
- Bits, of code, basic, 53
 - additional, 53
- Boolean algebra, 212-213
- Branch, conditional, 8, 36, 126, 183
- Breaking of wires, 321
 - after branching point, 321
- Bridging, 321
- Buffering properties of pipeline modelling circuits, 177
- Built-in error checking device, 122, *see also* Checker
- Bus, 223
 - input, 223
 - output, 223
- CAD products, 279
- Canonical techniques, 85
- Causal semantics, 2
- CCC lines, 223
- CCD lines, 223
- Cell, David, 166, 174, 187, 263
 - pipeline register, 362
 - register, 336-352
 - stack, 348
- Central control unit (CCU), 222
- Chain, 326
 - backward, 326
 - forward, 326
- Channel processor (CP), 220
- Checker, 69
 - built-in, 310, 328
 - CI, 71
 - DRC, 70
 - OBC, 71
 - of single-rail signals, 74
 - pyramid network, 74
 - spacer, 69
- Circles, 25, *see also* Places
- Circuit, 75
 - A_k^- , 284
 - antitone-self-dependent, 286
 - antitonous linear, 188
 - aperiodic, 6, 64, 74, 85, 113
 - decisive advantage of, 309
 - asynchronous, 113, 243
 - classes of, 140
 - functional analysis of, 243
 - asynchronous logical, 284
 - autonomous, 314
 - built-in checker, 310, 328
 - built of functional elements, 6
 - combinational, 64, 74, 112, 311
 - synthesis, 74-88
 - delay-independent, 6, 7

Circuit (continued)
 distributive, 7, 140, 159,
 162-164, 244, 252, 256
 with respect to state, 256
 equivalent to circuit, 267
 expansion of, 266
 fault-secure, 313, 316
 independent of wire delays, 280
 indicatable, 77-85, 312
 initialized, 272-273
 insensitive to set of wires, 268
 strictly, 268
 loop control, 130-132
 modelling, 7, 114, 127
 mono-transient, 201
 Muller, 6, 8
 Muller model for, 36
 multiple use, 8, 127-129, 187
 non-antitone-self-dependent, 286
 non-self-dependent, 286, 308
 open loop, 314
 “parallel compression”, 84
 parallel-sequential, 252, 256,
 262-263, 273
 with respect to state, 256
 perfect, 201, *see also* Perfect
 implementation
 pipeline, 177
 dense, 176
 scalar, with *m/b* count ratio, 359
 self-checking, 310
 self-dependent, 286
 self-inversible, 208
 self-testing, 313, 316
 self-timed, 64, 114
 semi-modular, 7, 115, 140,
 142-143, 165-175, 244,
 252-254, 260, 264-265, 277
 with respect to state, 253

Circuit (continued)
 sensitive to wires, 270
 strictly, 270
 shift of, 249
 simple, 149, 151
 speed-independent, 6, 140, 188,
 252-253
 synchronous, 244, 328
 ternary, 288
 totally self-checking, 9, 140,
 310-311
 autonomous, 316
 with respect to stuck-at faults,
 9, 311
 totally sequential, 140, 159-164,
 244, 252, 256
 with respect to state, 256
 with nodal states, 262-262
 Circuit equivalence, 267
 Circuit indicator, *see* Indicator
 Circuit membership within class, 252
 Circuit modelling of control flow,
 114, *see also* Modelling circuit
 Circuit sensitivity to wire delays, 271
 Classes of code combinations, 53,
see also Combinations
 v-classes, 53
 w-classes, 53
 Classification of speed-independent
 circuits, 140, 188, 252
 Clock, 2-4
 “Clock beat”, 3
 Clocked systems, 3
 Closure, of process, 195
 Code, auto-synchronous, 63
 balanced, 57, 63
 optimally (OBC), 57-61, 63,
 237-239
 Berger, 63, 328

- Code (continued)
 direct transition, 48-51, 63, 109-110
 double-phase, *see* two-phase
 double-rail (DRC), 52-53, 63, 70
 m-out-of-n, 328
 non-balanced, 239
 positional binary, 55
 self-synchronizing (SSC), 6, 48, 69, 232
 two-phase, 51-52
 with equal weights, *see* balanced
 with identifier (CI), 53-57, 61-63, 233-237, 328
- Combination, 44
 adjacent, 45
 code, 47
 comparable, 45
 compulsory, 56, 73, 237
 “don’t care”, 35, 237
 extra (auxiliary), 56, 73, 237
 incomparable, 79, 82
 spacer, 52
 working, 52
- Communication channels, 43
- Comparator, 294
- Comparing flip-flop state with values of set-reset inputs, 90
- Complement, of circuit, 206
 of function, 86
- Complementing input, 102-103
- Completeness, functional, 7, 140-164, 188
 in limited bases, 165-175
- Complexity, of analysis algorithms, 278
- Complexity bounds, 279
- Compliance, 1
- Component, 24, 33, 40, 190, *see also* Variable
 input, 24, 40, 190
 internal, 24, 40, 190
 output, 24, 40, 190
- Composition of asynchronous processes, 189, 193
 parallel, 195
 sequential, 194
- Concurrency, 1
- Concurrent operators, 182
- Condition, acknowledgement set, 284
 acknowledgement reset, 285
 acknowledgement stability, 285
 arbitration, 3
 meta-stability, 282
- Conflicts, 8
- Conjunction, of circuits, 211
- Constituent of unity, of combination, 44
- Control lines, 231
- Control structures of parallel programs, 280
- Controller, 232
- Counter, 355-363
 aperiodic, 355, 363
 two-bit, 199, 214
 decimal, 358
 obtained by using Muller theorem, 360
 pipeline, 362-363
 scale of five, 357
 scale of two, 163
 self-repairing, 323
 with all-zero reset state, 359
 with arbitrary counter capacity, 356

- Counter (continued)
 with reset mode implementation,
 359
- Covering problem, 50
- “Cutting of the inputs”, 91
- Cycle, 261
- Cyclo-diagrams, 114
- Cyclogram language, 188
- Data transfer, 231
 byte, 238
 byte-width, 233
 half-byte, 237
- Deadlock, 243, 277, 280
 freedom from, 277, 280
- Decoder, 232
- Decomposing into control part and
 operational part, 106
- Decomposition, of a system of
 Boolean equations, 141
- Delay, 3, 300
 built-in, 66, 113, 294
 character of, 65-66
 element, 6, 264
 ideal, 301-302, 308
 inertial, 65, 301, 308
 safe, 303
 perfect, 65, 301
 wire, 65, 264
 non-zero, 264
- Delay values, 303
- Dense information packing, 178
- Derivative, Boolean, 80, 268-269
 partial, 80
- Detonancy rank, 150
- Diagram, Hasse, 152, 212
 Muller, *see* state-transition
 signal change, 152
 state-transition, 4, 7, 33
 controlled, 33
- Diagram (continued)
 distributive, 33-34
 semi-modular, 33-34
 totally sequential, 33-34
 transition, *see* state-transition
- Difference, Boolean, of function,
 see Derivative
- Dijkstra semaphores, 134
- Discipline, of signal changing, 106
 “request-acknowledge”, 64
 two-phase, 8, 65
 with spacer, 82
- “Done” signal, 199
- Dragging in one of edges, 282
- ECL-technology, 264
- Element, 245, *see also* Gate
 AND, 73
 AND-OR, 73
 AND-OR-NOT, 7, 87, 128,
 144-148, 175, 305
 C, 112
 David (David's), 119, 121, 182,
 264, *see also* David cell
 delay, 114-116, 127
 delayless decision, 198-199
 excited, 36, 115, 142, 319
 faulty, 310, 314, 319
 functional, 6, 170
 idle, 36, 43, 141
 in equilibrium state, 142
 inherent function of, 142-148,
 161, 164, 169, 246
 majority function, 301-302
 mean switching time of, 329
 memory, 102
 NAND, 7, 138, 159, 164,
 167-169, 338
 NOR, 7, 96, 159

- Element (continued)
 of modelling circuit, 115
 OR, 73
 rectangular hysteresis (RHE), 293
 scaled delay, 66
 2 input NAND (2-NAND), 7, 165
 2 input NOR (2-NOR), 7
 2-NAND-2, 168-169, 175
 2-NOR-2, 175
- Elements, interacting, 4
 of control system, 2
 transition times in, 2
- Element cycle, 275
- Element delay, 6-7, 64, 111, 115, 160, 232, 310
 assumption of, 65, 198
- Element fault, 309-310
- Encoder, 232
- Encoding, 47
 by changes, *see* differential differential, 61-62, 232
 parity-check, 328
- Encoding discipline, 48
- Encoding scheme, (1125), 358
- Engineering diagnostics, 112, 314
- Environment, 2, 65, 113
- Equivalence class, 18, 41, 252
 closed, 252
 fake, 252, 318
 final, 18
 initial, 18
- Event, 4, 25
 discrete, 5
 of Petri net, 25
 disabled, 26
 enabled, 26
 firing (occurrence) of, 26
 live, 27
- Example of degenerate state machine, 90
- Excitation functions of variable, 150
- Expansion, of circuit with respect to set of wires, 266
- Extensions, of Boolean function, normal, 290
- External synchronization principle, 3
- Factor-set, 18
- Fan-in, 148, 165
- Fan-out, 148, 165
- Fault, 309
 conservative, 310
 exitory, 316
 mutable, 310
 parametric, 2
 stuck-at, 9, 309
 multiple, 314
 single, 314
 substitutional, 316
- Fault detection, 314, 321
- Fault diagnosis, 309
- Fault flag flip-flop, 323
- Fault localization detector, 322-325
- Finite automaton, 2 *see also* Finite state machine
- Firing, of event, 26
 of set of events, 26
 of vertex, 30
- Flip-flop, 7-9
 aperiodic, 89-98
 complementing (T), 8, 96-98, 102, 113, 163, 355-356
 delay (D), 8, 94-96, 98, 113
 Harvard, 97
 hysteresis (Γ), 69, 87, 113, 160, 172, 176
 JK, 329-330

Flip-flop (continued)
 with separated inputs (RS),
 89-94, 100, 329, 331, 341, 343,
 354

Flow charts, 216
 parallel asynchronous, *see*
 Parallel asynchronous flow chart

Focus, of detonant transition, 173

Fork, 38

Form, complemented, 45, 86
 direct, 45
 minimum, 85
 normal (canonical), 86
 conjunctive (CNF), 85
 disjunctive (DNF), 85
 orthogonal, 86
 reduced DNF, 147

French research on Petri nets, 41

“Frequency multiplier”, 358

Function, antitonous, 46, 78-79, 81, 208
 antitonous in input variables, 144
 in transition, 46
 in variable, 46
 Boolean, 7, 74, 88, 141, 197, 290
 Boolean difference of, 80
 Boolean ternary (BT), 288
 carry, 75, 83
 characteristic, Boolean, 246-247
 decisive, 271-272
 delay element, 114
 derivative of, 80
 equilibrium, 271
 excitation, 105, 122, 150, 165-166, 231, 271, 291-292
 guard, 137-140
 incidence, 25
 indicator, 104

Function (continued)
 inherent, 36, 75, 91, 141-148, 161, 164, 169, 175, 276, 310
 initial marking, 25
 isotonous, 46, 78-79, 81, 208
 in transition, 46
 in variable, 46
 logical, 36, 144, 209
 majority, 71
 monotonous, 46
 negative, 46
 of output responses, 39
 of state transitions, 39
 output, 39
 positive, 46
 reset, 122
 sensitivity, 269
 shift control, 165
 switching, 102, 105
 double-rail, 103
 ternary (T), 288-290
 transition, 39, 102, 105, 107
 unate, 46

Functional unit, 322-326

Gate, *see also* Element
 AND, 70, 84-85, 118
 AND-OR-NOT, 93
 average delay of (T), 90
 NAND, 86, 89, 119, 164, 175
 NOR, 86
 OR, 84-85, 118

Gate delay, *see* Element delay

Gate switching time, mean, 329

GCC lines, 223

GCD lines, 223

“Global” behaviour, 273

“Global” property, 254

“Go” signal, 199

- Graph. bipartite, 25
 directed, 15
 marked, 30, 41
 live, 30
 safe, 30
 mono-chromatic, 31
 signal, 30-32, 65, 70, 235,
 298-300
- Group control unit (GCU), 222
- Groups of intersections, largest, 50
- Hazards, 111, 113
 functional, 75, 111
 logical, 75, 151
- Head, of pipeline register, 336,
 353-355
 input, 353
 output, 355
- Identification lines, 231
- Identification, semantic, 217
- Identifier, 53
- Implementation, aperiodic, 64, 76,
 89, 98
 canonical, 85, 98-106, 112
 compliant, *see* matched
 correct, 141, 159
 in functional basis, 7, 140, 165
 matched, 68, 86-88, 109-110
 of combinational circuits, 74
 based on “collective
 responsibility”, 87
 crossed, 86
 hysteresis-flip-flop-based, 87
 minimum form, 85
 orthogonal form, 86
 standard, 85
 two-channel, *see* crossed
 of distributive and totally
 sequential circuits, 159
- Implementation (continued)
 of finite state machines, aperiodic,
 64
 canonical, 98-106, 112
 using flip-flops with separated
 inputs, 100-102
 with delay flip-flops, 98-100
 with direct transitions,
 109-110
 with multiple phase signals,
 106
 of parallel asynchronous flow
 charts, 124, 187
 guard-based, 137-140
 of standard fragments,
 124-127
 of semi-modular circuits, perfect,
 144-149, 159
 in limited basis of functional
 elements, 170
 perfect, 144-149, 159
 AND-OR-NOT-based, 144
 double-rail, 144
 two-channel, 144
 two-phase, 65-69
 Implementability problem, 7,
 140-141, *see also* Completeness,
 functional
 “Incorrect” loop implementation,
 132
- Indicability, 77-85, 111-112
 circuit, 77
- Indicability analysis, 77, 112
- Indication of transient process
 completion, 69, 113
- Indicator, 69
- Information lines, 231
- Initiator, 13-14, 16
- Initiator, set of, 13, 16

Input code checker, 232
 Input converter, 232
 Input discipline, 302
 Inputs indicated to outputs, 77
 Inputs translated to outputs, 78
 in one step, 78
 Instable transition process, 282
 Interconnection theorem, *see* Muller theorem
 Integrated circuits (ICs), high speed, 3
 technologically available, 73
 very large scale, 3
 Interface, 10, 215
 asynchronous, 231
 invariant to skew, 232
 self-synchronizing, 10, 231
 T2, 222-231
 Interface board, 222
 Interface layers, 215, *see also* Layer
 Interface organization, 215-216
 Interpretation, 5, 12, 24, 28, 41
 model, 24, 28, 33, 42
 semantic, 24, 32-33, 42
 Intersection, 50
 Intersection of circuits, 204
 Inverse, of circuit, 206
 Inverter, 197

 Join, 38
 Josephson junction technology, 264

 Latch, 307
 Lattice, 34
 distributive, 34, 208, 211, 256
 semi-modular, 34
 totally ordered, 34
 Lattice theory, 41

Layer, 259
 electrical, 215
 logical, 215
 mechanical, 215
 multiple, 259
 reference, 260
 width of, 260
 Length, of transition phase, 90, *see also* Operational cycle
 operating cycle, 92-98
 Lines, control, 223, 231
 data, 223
 identification, 231
 information, 231
 Liveness, 277, 280
 of marked graphs, 30
 of Petri nets, 27
 Localization, of faults, 321
 Localization detector, 323
 Logic, double-rail, 63
 Loop, 8, 130, 136, 184
 simple, 274
 Lower bound, 150

 MAC Project, 41
 Machine, aperiodic, 8, 64, 111
 asynchronous, 8, 39-40
 clocked, 112
 compliant, 68, 112
 control, 106-107
 fault-secure, 313
 finite-state, 2, 39, 64-65
 indicatable, 111
 matched, 68, 112
 Mealy, 111
 Moore, 111
 operational, 106,
 self-testing, 313

- Machine (continued)
 self-timed, 111
 synchronous, 64, 112
 totally self-checking, 313-314,
 328
- Marker, *see* Token
- Marking, 25
 conflict, 27
 deadlock, 27
 initial, 25
 initiator, 28
 live, 27
 of marked graph, 30
 of Petri net, *see* Marking
 reachable, 27
 safe, 28
- Match-maker, *see* Matching process
- Matching, 1
 of asynchronous processes, 215
- Maximum number of data items, 336
- Maximum throughput, 336
- Merger, 8, 37, 126
- Meta-model, 5, 12, 24
- Meta-stability, 3, *see also* Anomaly,
 meta-stability
- Minimal length of code, 58
- Minimization algorithm, 35
- Minimum interval, between two data
 items, 337
- Model, 12
 Huffman, 64
 mathematical, 12
 Mealy, 11
 Moore, 111
 Muller, 33-36, 41, 244, 278-279
 initialized, 35
 object, 12
 stuck-at fault, 309
- Modelling, of control flow, 114, *see*
also Circuit modelling
 of Petri nets, 115-123
 condition-based, 119-123
 event-based, 115-118
 of parallel asynchronous flow
 charts, 124-140
- Muller theorem 8, 197, 214
 generalization of, 198-203
- Multiplexer, 322-326
- Multiply used section, 185
- Necessity, logical, 14
- Non-persistence, localized, 186
- Notation, *see* Representation
- Occurrence of event, 26, 115, *see*
also Firing
- Operating cycle, 92-97
 length of, 92, 331
- Operational cycle, of JK-flip-flop,
 330
 of register, 331, 332
- Operator, multiply-used, of PAFC,
 35, 125
 multiply-used, 127
 $P(P')$, 246
 $Q(Q')$, 246
 single-used, 126
- Operator vertex, *see* Operator of
 PAFC
- Output converter, 232
- Output value changed, 36
- Parallel asynchronous flow-chart
 (PAFC), 5, 7, 36-39, 41, 124
 circuit modelling of, 124-140
 elementary, 124

- PAFC (continued)
 non-repetitive, 127
 safe, 38
 standard, 124
 well-formed, 124
- Parallel compression, 74, 84, 331
- Parallel conversion, single-rail data
 into double-rail form, 352
- Parallelism, 1
- Parbegin, 38
- Parend, 38
- Partition, blocks, 48
 π , 48
- Periferal devices (PDs), 222
- Persistence of Petri net, 9, 27
- Petri net, 5, 25, 41, 115, 177, 273
 circuit, 273, 276
 conjunctive, 276-277
 disjunctive, 276-277
- finite state machine, 32
- live, 27
- persistent, 7, 27, 180
- pipeline, 177-187
- pipelinized, 29
- reconvergence, 182
- safe, 7, 28, 180
- simple, 117, 180
 pipeline analogues of, 180
- unbounded, 29
- Phase, 1
 idle, 8, 65
 store, 91
 transition, 90
 working, 8, 65
 write, 91
- Phase signal, 68, *see also* Phase variable
- Pipeline, *see* Pipeline circuit,
 Pipeline process, Pipeline register
- Pipeline analogues, of Petri net
 fragments, 180
- Pipelining, process, 11, 177, *see also* Pipelinization
- Pipelinization, of conditional branch, 183
 of loops, 184
 of multiply used sections, 185
 of parallel fragments, 182
- Places, of Petri net, 25
- Possibility, logical, 14
- Principle, "request-acknowledge", 197, *see also*, "Request-acknowledge" discipline
- Procedure, canonical for synthesis of automaton, 8
 for representing aperiodic machine, 8
- Process, 13
 asynchronous (AP), 5, 12-24, 41
 controlled, 19
 descriptive, 13
 effective, 17
 matched, 217
 matching, 219
 pipeline, 21, 175-187
 reinstated, 189-198, 216
 simple, 20
 autonomous, 13
 transition, 330
- Projection, of process, 218
 of state, 267
 of state sequence, 267
- Protocol, 216, 218, 225, 242
 communication, 216, 225, 242, 280
 of matching, 218
 of simple asynchronous process, 20

- Pyramid network of spacer checkers, 74
- Races, critical, 244, 310
- Reachability, analysis of, 244-252, 279
- backward, 245
 - forward, 245
 - immediate, 36, 245
 - neighbour, *see* stepwise
 - stepwise 245
- Reachability problem, 244, 280
- Reconfiguration, 326-328
- Rectangular hysteresis element (RHE), 293
- Receiver, 232
- Receiver converter, 234, *see also*
- Input converter
- Reception, 236
- Reduction, of non-reinstated process, 191-193
- process, 190
- reposition of, 192
- Redundancy, code, *see* of code system
- of code system, 59, 63
- replacement, 321
- by-shift, 325
 - direct, 321
- Region, detonant, 150
- equichronic, 3
 - excitation, 150
- Register, parallel, 331
- with double-rail inputs, 332
 - with single-rail inputs, 352-353
- Register (continued)
- pipeline, 336
 - dense, 340-345
 - one-byte, 344
 - throughput of, 344
 - throughput of, 342-343
 - input head of, 353
 - non-dense, 336-339
 - output head of, 353
 - reversible, 349-352
 - dense, 352
 - non-dense, 350
 - semi-dense, 351
 - semi-dense, 339-340
 - with parallel reading, 345-346
 - with parallel writing, 346
 - reversible, 336
 - serial, 331, 335
 - “slow”, 332-333
- Regular structure, 326
- one-dimensional, 326
 - two-dimensional, 327
- Relation, concurrency (||), 153
- \mathcal{F} , 13
 - equivalence, 18
 - M , 14
 - “immediately followed by”, *see* of immediate sequence
 - of immediate sequence (F), 16
 - of immediate reachability, 36
 - Q , 20
- reachability, 245, *see also*
- Reachability
- strong precedence (\vdash), 152
 - weak precedence (\rightarrow), 152
- Relaxation time, 302

- Repair, of faulty circuit, 321
 Reposition, 20, 40, 65
 of asynchronous process, 21
 complete, 21
 non-trivial, 21
 trivial, 21
 partial, 21
 of reduction, 192
 Representation, balanced, 239
 differential, 241
 non-balanced, 239
 orthogonal, 86
 positional binary, 55
 Request, 64, 114, 197
 Resource allocation, 297-300
 Resource, common (shared), 282, 296
 Resource handling, 296
 Response, 66, *see also* Acknowledge and Acknowledgement
 Resultant, 13-14, 16
 Resultants, set of, 13, 16
 Schmidt trigger, 293, 308
 Self-checking, 9, 310-311
 with respect to stuck-at faults, 9
 total, 311
 Self-recovery, *see* Self-repair
 Self-repair, 10, 321-328
 Self-synchronization, 4
 Self-timed design, 64
 Self-timing, 4
 Semantics, of conditions and events, 24
 of system processes, 5
 operational, 24
 Semi-modularity, 34, *see also*
 Semi-modular diagrams and
 Semi-modular circuits
 Sequence, of situations, 17
 of states, complete, 261
 stationary, 271
 Server, 220
 Set of events, 25
 firing of, 26
 jointly enabled, 26
 Set of states, 2, 39, 65, 246
 of circuits, 246
 immediate predecessor, 246
 immediate successor, 246
 Sensitivity, 269-273
 Shift of circuit, 249-252
 Short circuiting, 321
 Signal distribution network, 166, *see also* Token shifter
 Situation, 13, 28, 32-33
 Situations, set of, 13, 16
 structured, 22
 structuring of, 22-24
 Skew, 10, 216, 232
 Skew phenomenon, 232, 242
 Spacer, 52
 all-one (*e*), 52, 69
 all-zero (*i*), 52, 69
 Spacer byte, 225
 Stack, 348
 Stack-at fault model, 309
 State, 1, 111
 adjacent, 36, 286
 allowed, 257,
 anomalous, 305-307
 bifurcant, 255
 conflict, 248, 253-254
 critical, 269
 cumulative, 34
 deadlock, 252
 detonant, 255
 discrete, 1

- State (continued)
 forbidden, 257
 full, 4
 hammock, 255
 idle, 65
 immediately following, 36
 immediately preceding, 36
 immediate successor, 245
 initial, 258, 272
 internal, 39, 65
 lower bound, 150
 meta-stable, 288, 292
 minimum, of excitation region, 150
 nodal, 261-263
 of circuit, 36
 of element, 142, *see* Element state
 of environment, 66
 of machine, 66
 operational, 258
 projection of, 267
 reachable, 245
 immediately, 36, 245
 stepwise, 245
 strictly critical, 269
 structured, 5, *see also* Structured situations
 transient, 90, 329
 all-one (double-one), 201, 329
 all-zero (double-zero), 201, 329
 working, 65
 1-conflict, 248, 254
 Steady state response, 39
 Strict sensitivity, 270
 Structured synthesis, 43
 Structuring, 5, 22, 190
 Sub-cube, of transition, 44
 internal, 44
 Sub-net, 274
 Sub-set, of allowed states, 257
 of forbidden states, 257
 Sub-state, circuit, 284
 meta-stable, 288
 stable, 284
 Symbol, 39-40, 51-52
 empty, 52
 input, 39
 output, 39
 working, 52
 Synchro-primitives, 134
 Synchronizer, 3, 8, 30, 38, 126, 173, 304
 bounded, 304
 in marked graph, 30
 in parallel asynchronous flow chart, 38
 Synchronization, 2-4
 Synchronization mechanisms, 2
 Synchronous system, 3, 4
 System, 1
 code, 47
 completely separating, 63
 System (continued)
 compliant (matched) parallel asynchronous, 2
 control, 1, 6
 functionally complete in class, 141
 of Boolean equations, 35, 141
 of constituents, 208-211
 minimum, 209
 of inherent functions (SIF), 75
 T2 addressing, 225
 T2 interface operations, 225
 control, 225
 exchange, 225
 read, 225

- T2 interface protocol specification, 225-228
- Table, switching, 41
truth, 35, 228
- Takeover, excitation function term, 151, *see also* term
term, 151, 273-274
complex, 154-155
simple, 154-155
- Taxogram notation, 188
- Term, excitation function, 151
transition constant, 44
transition variation, 45
- Theory, automata, 41, 189
lattice, 41
of protocols, 216, 242
of speed-independent circuits, 6
of state machines and sequential networks, 41
Petri nets, 41
- Threshold, filtration, 302
pass, 303
- Time, 4
- Timing diagrams, 114
- Token, 25
- Token shifter, 166
- Trace, of asynchronous process, 17, 19
- Tracey method, 63
- Transient (transition) process
completion, 113, 330
- Transition, 1, 4, 20, 25, 32
adjacent, 45
allowed, 47, 49
comparable, 45
concurrent, 153
detonant, 150, 170
direct, 48, 109-111
- Transition (continued)
input, for transition (for region), 154
of Petri net, 25
phase, 1
regular, 45
synchronizing, for detonant transition, 175
- Transition systems, 279
- Transition table, 109
- Transitions, of Petri net, 25
- Translatability, 78
- Translation of process description into circuit, 365
- Transmitter, 232
- Transmitter converter 236, *see also* Output converter
- Transmission, 236
- Union, of circuits, 204
- Universal speed-independent modules, 188
- Variable, acknowledgement, 287
active, 269
analogue, 1
antitone-self-dependent, 286
binary, 33
continuous, 1
control, 5
data, 5
decreasing, 78
double-rail, 103
excited, 33
idle, 33
in equilibrium, 33
increasing, 78
input, of wire, 265

- Variable (continued)
 - internal, 103
 - meta-stable, 288
 - non-antitone-self-dependent, 286
 - non-self-dependent, 208, 286
 - passive, 269
 - phase, 5
 - ternary, 288
- Verification of data transfer
 - protocols, 280
- Vertex, of marked graph, 30
 - bifurcator, 30
 - enabled, 30
 - synchronizer, 30
- of parallel asynchronous flow chart, 36-39, 125-127
 - bifurcator, 36, 38, 126
 - conditional branch, 36, 126
 - merger, 36, 37, 126
 - operator, 26, 126
 - synchronizer, 36, 38, 126
 - of signal graph, 32
- Vertices, of directed graph, 15, 33
- VLSI circuits, 10, 264
- Width of layer, 260
- Wire, 265
 - circuit, 265
 - intermodular, 280
 - intra-modular, 280
- Wired-OR merging, 296, 300