S. P. Vingron

Switching Theory

Insight through Predicate Logic

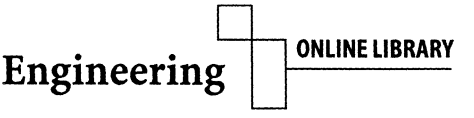


Springer

S.P. Vingron

Switching Theory

Springer-Verlag Berlin Heidelberg GmbH



http://www.springer.de/engine/

Shimon P. Vingron

Switching Theory

Insight through Predicate Logic

With 323 Figures



Springer

Dr. Shimon P. Vingron
formerly head of the Department of Theoretical Systems Analysis,
Institute for Information Processing, Austrian Academy of Sciences, Vienna
Bärenkogelweg 21
2371 Hinterbrühl
Austria

ISBN 978-3-642-07318-2

Library of Congress Cataloging-in-Publication-Data
Vingron, Shimon Peter, 1936–
Switching theory : insight through predicate logic / Shimon Peter Vingron.
p. cm.
ISBN 978-3-642-07318-2 ISBN 978-3-662-10174-2 (eBook)
DOI 10.1007/978-3-662-10174-2
1. Switching theory. 2. Predicate (Logic) I. Title.
TK7868.S9V56 2004
621.3815'372--dc22

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitations, broadcasting, reproduction on microfilm or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag Berlin Heidelberg GmbH.
Violations are liable for prosecution under the German Copyright Law.

http://www.springer.de

© Springer-Verlag Berlin Heidelberg 2004
Originally published by Springer-Verlag Berlin Heidelberg New York in 2004
Softcover reprint of the hardcover 1st edition 2004

The use of general descriptive names, registered names trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Typesetting: camera-ready by author
Cover design: deblik Berlin
Printed on acid free paper 62/3020/M - 5 4 3 2 1 0

Preface

About the subject. Switching Theory is an applied science used in analysing, designing, and testing switching circuits irrespective of the technology in which they are realised. Its origin goes back to **Shannon**'s [1938] remarkable paper in which he introduced *'propositional logic'* to describe switching circuits. This was the first time that propositional logic was applied to an engineering problem. The method proved successful but was restricted to circuits that had no memorising ability, so-called *combinational circuits*. It was **Huffman** [1954] who, in his doctor's thesis, showed how to analyse and design circuits that had a memorising ability, so-called *sequential circuits*. At the heart of his breakthrough was a simple way of specifying a sequential circuit by a table he called a *'flow table'*. It enabled a sequential circuit to be reduced to a number of combinational circuits, each of which could then be handled by Shannon's methods.

About this book. Just as propositional logic was superseded in mathematics by **predicate logic**, developed by **Frege** [1879], this is also done here for Switching Theory. You will notice that with the use of predicate logic proofs and derivations become possible that hitherto remained out of reach.

Huffman's flow table is used to develop a new tree representation of a sequential circuit, a so-called **word-recognition tree**, by which a sequential circuit is *not* reduced to a number of combinational circuits but, rather, calculated directly as interconnected **latches** (memory circuits). This approach required a new look at the theory of latches. But most importantly, the formidable problems connected with the so-called *'encoding of the internal states'* of an asynchronous sequential circuit are bypassed.

Pointing out what's new. The many new concepts, proofs, and methods incorporated into this book are not explicitly listed. Those *chapters* marked by an **asterisk** (*) present new concepts. Those *sections* marked by an **asterisk** (*) at least *contain* new concepts, proofs, or methods.

Whom the book is for. The book is written for the theoretically inclined practising engineer working in the field of digital circuit design, for the scientist concerned with developing methods and algorithms for designing digital circuits, and for the student feeling the need to supplement a course in Switching Theory or Logic Design. While Division One is an undergraduate-level introduction, the remaining parts constitute a sound basis for a graduate course in Switching Theory or Logic Design.

Expressing my gratitude. When I retired, my wife, Dora, told me she didn't want me hanging around the house not knowing what to do with myself. So I went into my study, and took up the research (interrupted more than a decade earlier) on what was to become this book. This project, that I had thought would take two to possibly three years, engulfed me for eight. During these years, my wife unfailingly encouraged me in the many difficult phases of the work. My gratitude is only surpassed by my love to her.

Deep thanks are due to my friend Dr. Oliver König who worked through the whole manuscript and whose formidable and creative criticism I simultaneously feared and awaited. Many major improvements are his due.

In many ways, this book is a tribute to my former teacher Prof. emeritus Dr. Karl Heinz Fasol.

A closing remark. I should consider it a compliment if this book were to stimulate comment, or argument, or contradiction. I should consider it a success if it were to help start a process of questioning established views in a subject which, although having lost some of its former vigour, has lost nothing of its importance.

It is not given to you to complete the task,
but you are not freed from starting.

Rabbi Tarphon
Ethics of the Fathers ii/21

Contents

Division One Fundamental Concepts

1 Outline and Basics 3

1.1 Specifying a Circuit in Plain Prose 3
1.2 Diagram of Successive Events 4
1.3 Table of Asserted Events *[1] 5
1.4 Logic Variables and Logic Formulas * 7
1.5 Drawing the Circuits 9
1.6 Sequential Circuits * 10
1.7 Verifying a Circuit * 12

2 Switching Devices 14

2.1 Pneumatic Valves 14
2.2 Electric Relays 18
2.3 CMOS Transistor Circuits 23

3 Functions 27

3.1 Ordered Pairs 27
3.2 Speaking of Functions... 31
3.3 Switching Functions 33

4 Logic Functions and Gates 36

4.1 Elementary Switching Functions 36
4.2 Positive versus Negative Logic * 37
4.3 Elementary Logic Functions * 38
4.4 The Basic Gates * 40
4.5 Derived Gates * 43

5 Synthesis and Duality 46

5.1 Minterm Functions & Minterms * 46
5.2 Maxterm Functions & Maxterms * 49
5.3 Synthesis via Partial Outputs * 50
5.4 Duality * 53

6 Karnaugh Maps 57

6.1 Speaking of Sets... 57
6.2 Introducing the Karnaugh Map * 59
6.3 Karnaugh Maps for multiple Inputs 62
6.4 Karnaugh Sets * 64

[1] Those Sections or Chapters marked by an **asterisk** (*) contain new concepts, proofs, or methods.

7 Utilising Karnaugh Maps **66**
7.1 Specifying Switching Circuits in K-maps * 66
7.2 Obtaining Disjunctive Formulas 67
7.3 Obtaining Conjunctive Formulas * 68
7.4 Logically Equivalent Expressions * 70
7.5 Logical Implications * 72
7.6 K-maps of Dual Functions * 74

Division Two Logic

8 Tautologies **79**
8.1 Logic Expressions 79
8.2 Truth Tables 81
8.3 Speaking of Tautologies... 82
8.4 Replacement versus Substitution 85
8.5 Logic Reasoning 86

9 Propositional Logic **89**
9.1 Axiomatic Approach to Propositional Logic 89
9.2 Complementation 91
9.3 IMPLICATION and NEGATION 93
9.4 DeMorgan's Theorems 94
9.5 Commutativity of AND and OR 96
9.6 Logic Implications of IMPLICATIONS 97
9.7 Formulas with a single Variable 99

10 Summary of Theorems **101**
10.1 Commutative and Associative Laws * 101
10.2 Single-Variable Formulas 103
10.3 Distributive Laws * 104
10.4 Generalised DeMorgan Theorems 106
10.5 Basic Theorems on AND, OR and NOT 109

11 Algebraic Proofs **110**
11.1 Min- and Maxterms are Complementary * 110
11.2 Disjunction of all Minterms * 111
11.3 Conjunction of two Mintermms * 113
11.4 Maxterm as the Disjunction of Minterms * 114
11.5 Minterm AND/OR Maxterm * 115
11.6 Solving a System of Logic Equations * 117
11.7 DeMorgan on DeMorgan 118

12 On Predicate Logic **121**
12.1 Inside a Proposition 121
12.2 Symbolic Notation of Propositions 122

12.3 The Switching Algebra Connection * 124
12.4 Quantifiers 124
12.5 Quantification and Replication * 126
12.6 Free and Bound Variables 126

13 Predicate Logic 129

13.1 Axioms and Rules of Switching Algebra * 129
13.2 Theorems on Identity 132
13.3 Theorems on Quantification 135

Division Three Combinational Circuits

14 Canonical Normal Forms 141

14.1 An Overview 141
14.2 Direct Derivation of Normal Forms * 144
14.3 Shannon's Expansion Theorems * 147
14.4 Shannon's Expansion to Normal Forms 149

15 Shegalkin Normal Form 151

15.1 An Overview 151
15.2 Developing the Shegalkin Polynomial * 153
15.3 Shegalkin Coefficients 155
15.4 On Combinations of Input Variables 156
15.5 Dual Shegalkin Polynomial 157
15.6 Necessary and Sufficient Connectives 160

16 Synthesis Examples 162

16.1 Multiplexers and Demultiplexers 162
16.2 Binary Coded Decimal Digits 163
16.3 Priority Decoders 165
16.4 Comparators 166

17 Concepts Old and New 171

17.1 Multiple Events * 171
17.2 Karnaugh Sets Revisited * 172
17.3 Generalised Minterms * 173
17.4 Generalised Maxterms * 175
17.5 Partitions and Equivalents 176
17.6 Prime Sets 177
17.7 Covers 177
17.8 Inclusions and Exclusions * 179
17.9 Evaluation Formulas * 180

18 Minimisation Preliminaries 182

18.1 Aspects of Minimisation 182
18.2 Incomplete Specification 183

18.3 Prime Sets that cover Don't Cares 185
18.4 Evaluation Formulas for Don't Cares * 188
18.5 Minimisation by K-maps 189
18.6 Algebraic Minimisation 190

19 Minimisation—an Excerp 193
19.1 On Adjacent K-Sets and their Consensus * 193
19.2 Formalising Adjacency and Consensus 195
19.3 Finding the Full Cover * 198
19.4 Finding Minimal Covers * 201
19.5 Minimisation considering Don't Cares * 204

20 Reduced Karnaugh Maps 207
20.1 Reducing the Dimension of a K-map 207
20.2 K-maps within K-maps * 209
20.3 Evaluating Reduced K-maps * 211
20.4 Specification by Map-Entered Variables * 215

21 NOT-AND and NOT-OR 219
21.1 Miscellaneous Theorems 219
21.2 Two-Level Circuits 221
21.3 Multilevel NOT-AND Circuits 222
21.4 Multilevel NOT-OR Circuits 228

22 Composition of Circuits * 231
22.1 The Basic Concept 231
22.2 Catenation 233
22.3 Choosing a Generic Function 234
22.4 Hinge Sets 236
22.5 Composing a Circuit: Example 1 236
22.6 Composing a Circuit: Example 2 239
22.7 Composing a Circuit: Example 3 239

23 Hazards 242
23.1 Hazards on Stationary Signals 242
23.2 Single-Input Hazards 243
23.3 Gates and Delays 245
23.4 Detecting Single-Input Hazards 246
23.5 Hazard-Detecting Formulas 248
23.6 Hazards in a NOT-AND Circuit 249
23.7 Hazards in a Composed Circuit 251

Division Four Latches

24 Memorising by Feedback 255
24.1 What is Memorisation? 255
24.2 The Feedback Model of Memory 258

24.3 Transient versus Stationary Behaviour 261
24.4 Latches without Feedback 263
24.5 Is a Delay in the Feedback necessary? * 264

25 Basic Theory of Latches 267
25.1 What is a Latch? * 267
25.2 Specifying Latches in Reduced K-maps * 269
25.3 Evaluation Formulas and Generic Latches * 270
25.4 An Evaluation Example 272
25.5 Identical and Isomorphic Latches * 273
25.6 Inverting a Sample Memory-Circuit 275
25.7 General Negation of Latches * 277

26 Optimised Latches 279
26.1 Designing Minimised Latches * 279
26.2 A Minimised Latch—an Example 280
26.3 Designing Hazard-Free Latches * 282
26.4 A Hazard-Free Latch—an Example 285

27 Elementary Latches 288
27.1 Classification of Elementary Latches 288
27.2 Two Systems of Symbolic Latches * 291
27.3 Symbols with Complementary Outputs * 293
27.4 Symbolic Latches with a Common Event * 294
27.5 Standard Symbols for Elementary Latches 297

28 Composition of Latches * 300
28.1 Catenation Diagram for Latches 300
28.2 Composing the D-Latch using a SR-Latch 302
28.3 Checking for a Hazard-Free Solution 303
28.4 Avoiding Undesirable Input-Events 305

Division Five Sequential Circuits with Continuously Read Inputs

29 Automata and Programs * 311
29.1 Continuously reading Automata 311
29.2 Developing an Example Flow Table 313
29.3 The Word-Recognition Tree 318
29.4 Evaluating a Sequential Circuit 320

30 Word-Recognition Tables * 325
30.1 Developing Word-Recognition Tables 325
30.2 Primitive Determinateness 327
30.3 General Determinateness 330

30.4 Evaluation and Partial Mappings 332
30.5 Testing for General Determinateness 333

31 Catenation Model * 335

31.1 The Moore and Mealy Models 335
31.2 Catenation Model — Feedback and Delays 337
31.3 Moore and Mealy Flow Tables 340
31.4 Merging Moore and Mealy Flow Tables 341
31.5 From Mealy to Moore Flow Tables 344
31.6 The Composite Word-Recognition Table 345

32 Toggle Circuits 349

32.1 T-Flipflop — its Feedback Design 349
32.2 T-Flipflop — its Composition 352
32.3 JK-Flipflop — its Feedback Design 355
32.4 JK-Flipflop — its Composition 357

33 Triggering and Synchronising 360

33.1 Triggering 360
33.2 Sampling 363
33.3 Synchronising 366

34 Verifying a Logic Design * 370

34.1 A Verification-DSE for the D-Latch 370
34.2 A Verification-DSE for a given Example 376
34.3 Grafting a Verification Tree 379

35 Discussing Huffman's Theory 385

35.1 Basics of Huffman's Synthesis Procedure * 386
35.2 Delay and Feedback 387
35.3 State Encoding and Circuit Realisation 389
35.4 Problems with Arbitrary Binary Encodings 392
35.5 Race Analysis in the Catenation Model * 394

36 State Encoding Techniques 398

36.1 Introducing Multiple State-Transitions 399
36.2 Adding Unstable States Intuitively 401
36.3 On the Number of Adjacent Encodings 402
36.4 One-Hot Code 403
36.5 Safe but Slow standardised Encodings 404
36.6 Safe and Fast standardised Encodings 408
36.7 Number of Required State Variables 410

Bibliography 411

Index 415

Division One

Fundamental Concepts

1 **Outline and Basics** 3 (11 pages)
2 **Switching Devices** 14 (13 pages)
3 **Functions** 27 (9 pages)
4 **Logic Functions and Gates** 36 (10 pages)
5 **Synthesis and Duality** 46 (11 pages)
6 **Karnaugh Maps** 57 (9 pages)
7 **Utilising Karnaugh Maps** 66 (11 pages)

Apart from using Chapter 1 as a general introduction to the book, these first chapters serve to motivate later developments.

Logic variables, being explained as so-called *propositional forms*, indicate the need to use **predicate logic** (as opposed to **propositional logic** conventionally advocated) in describing and designing switching circuits. This approach is tentatively and partially introduced in Sections 1.3 and 1.4, and first tested in Chapter 2.

The mathematical description of a (combinational) switching circuit is a **function**. Every theoretical aspect of a switching circuit is derived from this concept. Chapter 4 gives a first impression of this approach, while Chapter 3 is only a reminder of the concept of function (in case it is needed). In a somewhat intuitive way, Chapters 4 and 5 combine concepts of predicate logic with that of a function.

Karnaugh maps are introduced as early as possible (Chapter 6) as they are the most vivid descriptive and analytical tool we have in switching algebra. The Karnaugh map, introduced in a formal way in Section 6.2, unites the concepts of function, predicate logic, and set theory. The concept of the **Karnaugh set**, introduced in Section 6.4, turns out to be the intuitive basis of many theoretical developments in subsequent chapters of the book. The various basic uses to which a Karnaugh map can be put, are covered in Chapter 7.

Chapter 1

Outline and Basics

The **synthesis** of a switching circuit — i.e., its design or development — is started by stating or specifying what the circuit is supposed to do. We first briefly discuss the cumbersome plain prose specification, but it quickly becomes clear that more refined methods are needed. In fact one often employs specialised specification methods which are adapted to the type of problem under consideration. For instance, **combinational circuits** — the name used for circuits that are said to have *no memorising capability* — are advantageously specified in a **table of asserted events** (introduced in Section 3), or in so-called **Karnaugh maps** (discussed in Chapter 6). These two methods have neither the means of depicting the flow of time nor a succession of events, either of which is needed to specify a circuit *with memorising capabilities* — a **sequential circuit**.

The **diagram of successive events** developed in Section 2 is based on the *flow of time*, and is arguably the most intuitive, albeit not the most powerful, means of specifying sequential circuits. Far more powerful, but not nearly as intuitive, is the **word-recognition tree**, touched on in Section 6. This representation depicts the *succession of events* that specify the behaviour of a sequential circuit (not the flow of time, as in a diagram of successive events). With the advent of the word-recognition tree it became possible to develop a sequence of input events by which to test sequential circuits.

Instrumental to the understanding of all which follows in subsequent chapters is the first acquaintance with **propositional forms** in Section 3. In Section 4 they are referred to as **logic variables** and used in developing **logic formulas**. Such logic formulas are exact descriptions of the input-output behaviour of combinational circuits.

1.1 Specifying a Circuit in Plain Prose

Let us employ a running example of a switching circuit to introduce some basic concepts. As formal methods for describing a circuit's behaviour are, as yet, lacking, we shall do the obvious, and use plain prose: *Imagine a*

windowless room with two doors and a light switch at each door. Every time someone passes through a door (to become the sole occupant of the room, or to leave it empty) he brings the switch next to that door from its current to its alternative position. By doing so, the room's lights go on if they were off, and vice versa, go off if they were on. Thus, the idea is: whenever someone enters the room he turns the electric lights on, and then turns them off when he leaves the room.

There are a few things to note in the above specification. Firstly, care was taken to state the number of inputs and outputs to the circuit: Each of the light switches provides one of the two input signals, whereas the single output signal is the condition of lighting in the room. Secondly, each of the circuit's signals occurs in either of two distinctly different states: The contacts of the switches are either open or closed, the room's lights either on or off. Signals with this property are called **binary signals**. Usually the two alternative states of a binary signal are referred to by the integers 0 and 1. The 0 is most frequently associated with the lower energy level of a binary signal, the 1 with the higher energy level. If the energy level of a signal's state is irrelevant (as with the switches of our example), one is free to associate 0 and 1 with the signal's states in any way convenient. Less obvious aspects of the above specification are: The wording of the specification is **redundant**. For instance, the last sentence of the specification *can be left away without loosing information.* For human beings redundancy is helpful in grasping things more quickly, and often more thoroughly. Formal systems (on the other hand) need no redundancy (whatsoever). Finally, the above *specification is complete*, meaning that it contains all the information needed to design the circuit; you will soon see how this statement can be verified.

1.2 Diagram of Successive Events

The next step to take, is to introduce a somewhat more formal way of describing a switching circuit. In this book we shall make extensive use of a representation called a **Diagram of Successive Events (DSE)**, which simply shows *how the value of each input and output signal changes in the course of time.* The DSE for our running example is shown in Fig. 1–1. This being our first acquaintance with a DSE, we shall interpret each of the time intervals T_1 to T_5 of the example DSE according to the plain prose specification given above.

T_1 When you open a door to enter the room, both light switches will be in a certain (let us say original) state to which we arbitrarily assign the integer 0. The lights will be off, and we again choose the integer 0 to describe this state.

T_2 Entering the room, you activate, say, switch 1 bringing it into its alternative state which is represented by the integer 1. Activating switch 1 lets the lights go on which is represented by 1.

T_3 You walk through the room leaving it via the opposite door and bringing switch 2 into its alternative state (a value of 1 for switch 2) to turn the lights off as you leave (letting the value for the lights drop to zero).

T_4 To turn the lights on, a person entering the room tips the switch at hand (say switch 1).

T_5 When leaving the room by the opposite door, switch 2 will be brought to its alternative state to turn the lights off.

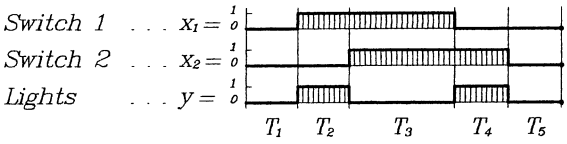


Fig. 1–1 Diagram of successive events (DSE)

The vertical lines in Fig. 1–1, which part the time intervals, still need some commenting on. The question is, where are vertical lines drawn? *A vertical line is drawn when at least one input variable changes its value, and only then* (note that no reference is made to the value or state of the output signal). The output of a (well behaved) switching circuit may only change if the value of at least one input variable changes. But not every change of an input value necessitates a change of the output value. As no values change between vertical lines (neither those of input signals nor those of the output signal) it becomes irrelevant how far apart the vertical lines of a DSE are drawn, and whether they are drawn equally far apart or not.

The time intervals of a DSE are better described by the concept of an **event**. An **input event** *comprises the values of all input signals (or input variables) for a time interval in which these values remain constant.* When considering circuits with more than one output, one also uses the concept of an **output event**. When referring to only one output, we shall normally not use the term output event. Rather, we shall simply speak of the value of the output variable. The **Diagram of Successive Events (DSE)** *can now be said to describe the successive occurrence of input events together with the output values these input events initiate.*

1.3 Table of Asserted Events *

The simplest switching circuits, the *combinational circuits*, can be described completely by a **table of asserted events** (TAE). To understand such tables, first consider how we express the value of any input x_i. The value that any input may have is, of course, either 0 or 1. Assuming x_i to have the value 0 is written $x_i = 0$, assuming x_i to be 1 is written $x_i = 1$. Note that

writing $x_i = 0$ does not mean that x_i really *is* 0. It only means that we are *considering as a possibility* or *supposing* x_i to be 0. If we really wanted to know the value of x_i we would have to *measure* its value. We could then substitute the measured value for x_i in $x_i = 0$ obtaining either 0=0 or 1=0, as the case may be. These expressions are **propositions** that are, respectively, **true** (denoted as **1**) or **false** (**0**). The term $x_i = 0$, on the other hand, is called a **propositional form**, and acts as a **logic variable**. Assuming each input x_i to have a specified value, and stating the associated value of the output y, provides us with a single column of the table of asserted events.

T_5			
T_1	T_4	T_2	T_3
x_1=0	x_1=0	x_1=1	x_1=1
x_2=0	x_2=1	x_2=0	x_2=1
y=0	y=1	y=1	y=0

a) Explicit form

	T_5			
	T_1	T_4	T_2	T_3
x_1=	0	0	1	1
x_2=	0	1	0	1
y=	0	1	1	0

b) Brief form

Fig. 1–2 Table of asserted events (TAE)

Let us develop the TAE of Fig. 1–2a from the DSE of Fig. 1–1. Assume that the bottom row of Fig. 1–2a, the row containing the output values, has not yet been filled in, so that this row is still empty. It is our intention to fill in the output values for one column after the other. In the first column we claim that $x_1 = 0$ and $x_2 = 0$. This assertion occurs in the time intervals T_1 and T_5 of the DSE, and the corresponding output is 0 in both cases. This asserted output value of 0 is entered into the first column of the TAE of Fig. 1–2a. You carry on in a like manner by choosing the next input event in the TAE for which to search in the DSE.

It is characteristic of a **combinational circuit**, that the output value is the same in all time intervals of a DSE that have an identical succession of input values. We would run into trouble when entering output values into a TAE if not all output values for a given input event in a DSE were the same (imagine that the output value in the time interval T_1 of Fig. 1–1 were 0, and that of T_5 were 1). A switching circuit described by such a diagram of successive events is called a **sequential circuit**. You can describe a sequential circuit in a DSE but you cannot do so in a TAE.

For clarity (i.e., to concentrate better on the information — the 0s and 1s — of the table), as well as to reduce the amount of writing necessary, we usually use the TAE in its *brief form*, shown in Fig. 1–2b, in which the variables have been *'extracted'* from the columns of each row. The 0s and 1s in such a table are the *asserted values* of the inputs and outputs. The succession of 0s and 1s of a single column that correspond to the input variables $x_1, x_2, ..., x_n$ is called an **asserted input event**.

To draw a **table of all** *input* **events** — see Fig. 1–3 — you might like to follow this procedure. If you draw the table horizontally it will have as

many rows as the circuit has input variables, say n. Assign x_1 to the top row, x_2 to the next one, and so on. The number of columns (one for each input event e) is 2^n. Thus, for 2 inputs you will need 4, for 3 inputs 8, and for 4 inputs 16 columns. Now, do not try to fill in one column after the other with zeroes and ones. Filling in one row after the other is much faster, and leaves you with a natural order of input events if you use the following method. In the top row, the one for x_1, you fill the first half of the columns with zeroes, the second half with ones. In the second row, the one for x_2, you fill the first quarter of columns with zeroes, the second with ones, the third again with zeroes, and the fourth again with ones. In the third row you again halve the number of columns, filling in one eighth after the other, first with zeroes and then with ones. In the last row you will be writing zeroes and ones alternatively. Finally, to get a complete TAE, add as many rows as you have output variables and fill in the zeroes and ones as appropriate to the circuit you wish to describe. If a table of all events is designed as just explained you can number the input events, i.e. the columns, from left to right, starting with 0 and ending with $2^n - 1$. These numbers are the decimal representation of the input events.

$e\ldots$	0	1	2	3
$x_1=$	0	0	1	1
$x_2=$	0	1	0	1

$e\ldots$	0	1	2	3	4	5	6	7
$x_1=$	0	0	0	0	1	1	1	1
$x_2=$	0	0	1	1	0	0	1	1
$x_3=$	0	1	0	1	0	1	0	1

Fig. 1–3 Tables of asserted input events

A great advantage of the table of asserted events is that it enables us to recognise when a circuit is **completely specified**; this is the case when all cells of the row in which the values of the output variable are to be entered actually do contain either a 1 or a 0, that is, are not left empty. If a circuit specification does not tell you what output value is initiated by each of the input events, it is said to be **incomplete**. Most technical problems — thankfully — lead to incompletely specified circuits. Those input events for which the problem does not clearly state an output value are called **don't care events**. Our running example, as already mentioned, is completely specified.

1.4 Logic Variables and Logic Formulas *

The table of asserted events (TAE) is a tool of major importance as its analysis leads directly to the introduction of *logic variables*, and to the *logic treatment of switching circuits*. So, let us take a closer look at the TAE of

our running example, repeated below. It states when the output y is 0 and when it is 1. This is expressed stringently in the sentence next to the TAE.

x_1=0	x_1=0	x_1=1	x_1=1
x_2=0	x_2=1	x_2=0	x_2=1
y=0	y=1	y=1	y=0

The output variable y is 1
if and only if
x_1 is 0 AND x_2 is 1
OR
x_1 is 1 AND x_2 is 0

Employing the symbols

$\Leftrightarrow$ for *logically equivalent* or *if and only if* (**iff**),
$\wedge$ for AND, and
$\vee$ for OR

the above sentence can be written as:

$$(y = 1) \Leftrightarrow \big((x_1 = 0) \wedge (x_2 = 1)\big) \vee \big((x_1 = 1) \wedge (x_2 = 0)\big). \tag{1.1}$$

This rather cumbersome expression is greatly simplified by introducing the following **logic variables** X_i, $\overline{X}_i$, Y, and $\overline{Y}$ as a shorthand notation for the expressions written within the inner parentheses, i.e. defining

$$\begin{aligned} X_i &\overset{df}{\Leftrightarrow} x_i = 1, & \overline{X}_i &\overset{df}{\Leftrightarrow} x_i = 0 \\ Y &\overset{df}{\Leftrightarrow} y = 1, & \overline{Y} &\overset{df}{\Leftrightarrow} y = 0 \end{aligned} \tag{1.2}$$

Rewriting (1.1) with the help of the logic variables X_i, $\overline{X}_i$, Y, and $\overline{Y}$ leads to what is called a **logic formula**:

$$Y \Leftrightarrow (\overline{X}_1 \wedge X_2) \vee (X_1 \wedge \overline{X}_2). \tag{1.3}$$

Colloquially, the symbol $\Leftrightarrow$, called **logic equivalence**, refers to its left and right sides being equal from the point of view of logic, allowing us — among other things — to use the left-hand and right-hand expressions interchangeably in logic formulas (substituting one by the other). For a detailed discussion of logic equivalence see Chapter 8.

Alternatively, we could have used the TAE to state precisely when the output variable y is 0, writing this in analogy to (1.1) as

$$(y = 0) \Leftrightarrow \big((x_1 = 0) \wedge (x_2 = 0)\big) \vee \big((x_1 = 1) \wedge (x_2 = 1)\big), \tag{1.4}$$

which, with (1.2), may be written as the logic formula

$$\overline{Y} \Leftrightarrow (\overline{X}_1 \wedge \overline{X}_2) \vee (X_1 \wedge X_2). \tag{1.5}$$

The two formulas (1.3) and (1.5) are said to be **complementary**.

1.5 Drawing the Circuits

A logic circuit is comprised of **gates**, these being graphic symbols representing logic connectives (see Fig. 1–4). *Logic formulas, as (1.3), state which gates to use and how to connect them. To transform a logic formula into a circuit consisting of gates, you draw an AND gate for each $\wedge$-connective of the formula, and an OR-gate for each $\vee$-connective. The arguments of the $\wedge$- and $\vee$-connectives in the logic formula are taken as inputs to the associated gates, this ensuring the correct connection between the gates.* Some people like to draw a circuit starting with the innermost parentheses of a formula, thus drawing the circuit from the inputs to the output. Others prefer to start with the outermost parentheses, thus drawing the circuit from the output to the inputs. Use whichever method you feel more comfortable with.

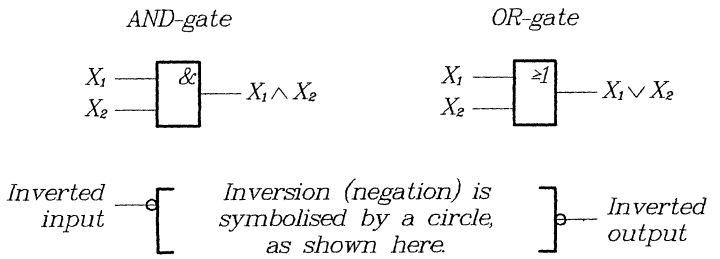


Fig. 1–4 Gates as graphical symbols for logical connectives

To invert an input signal you draw a small circle where the input signal enters the gate. If you want to invert an output signal, you draw a small circle where the output signal leaves the gate.

The logic circuits of Fig. 1–5 all represent the same formula, i.e. (1.3). Fig. 1–5b demonstrates that you can stretch a gate to a desired size, while Fig. 1–5c shows that gates may be densely packed. The output signal, by the way, need not be positioned in the middle of a gate.

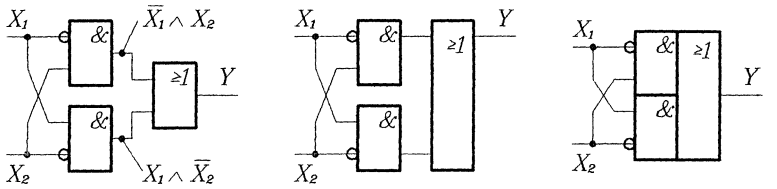


Fig. 1–5 Drawing the circuit for $Y \Leftrightarrow (\overline{X}_1 \wedge X_2) \vee (X_1 \wedge \overline{X}_2)$

To fully understand these gates, we need to know precisely how their output signals react to the input signals. This is shown in Fig. 1–6 for the NEGATION, AND gate, and the OR gates. Thick input or output lines represent TRUTH (i.e. **1**), thin lines indicate FALSITY (i.e. **0**).

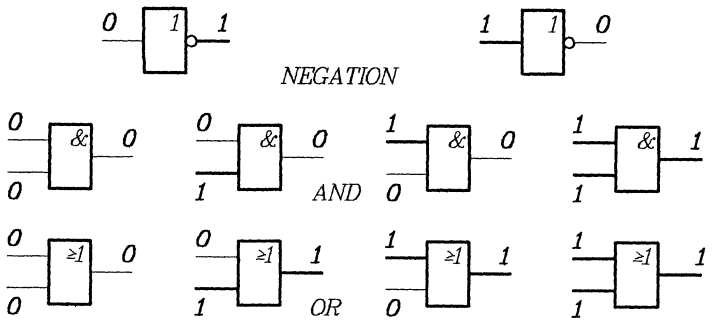


Fig. 1–6 Explaining the behaviour of NEGATION, AND, and OR

1.6 Sequential Circuits *

The combinational circuits, discussed so far, are circuits of the simplest structure inasmuch as they have no *memory*. This means that any given input event, whenever it occurs in a sequence of input events, always causes the same output value.

Sequential circuits, on the other hand, do have a memory, this letting the circuit memorise certain *sequences of input events* together with their associated output values. The simplest of these circuits are elementary memory circuits, so-called *latches*. Latches are so important because they are used as building blocks (together with gates) in creating sequential circuits. In other words, latches and combinations of their outputs realise the memory of every sequential circuit.

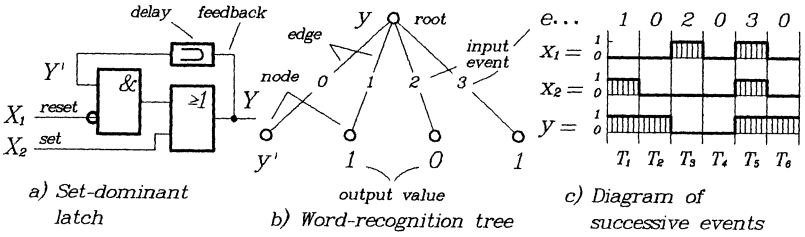


Fig. 1–7 Set-dominant latch: An example latch

To get a first impression of latches, and of how they differ from combinational circuits, take a look at the logic circuit of Fig. 1–7a of an example latch. As with combinational circuits it is constructed of gates. Yet, contrary to combinational circuits, you will notice that the output signal Y is *fed back, delayed, and used as an input Y' to the circuit.*

The behaviour of the latch is shown in the diagram of successive events of Fig. 1–7c. This diagram shows very nicely that when both input variables x_1 and x_2 are zero — i.e., during the time intervals T_2, T_4, and T_6 — the output y is *either* 0 *or* 1. But, it is nevertheless possible to say what value the output variable y will have when both inputs become zero. You will notice that the input event (0,0), the values for (x_1, x_2), *retains or memorises the previous output value.* An input event having this ability is called a **memorising input event.**

The diagram of successive events (Fig. 1–7c) gives rise to a formal specification called a **word-recognition tree** (Fig. 1–7b). Such trees are always drawn with their *root* uppermost, letting them grow downward, i.e., in the same direction in which we add further lines when writing. You will notice that the leaves (the end-nodes) assigned to the input events 1, 2, and 3 carry the same output values as stated in the DSE. But, as soon as the input event becomes 0 (and from then on, while it is 0) the output value y must always take on the same value as its time delayed version y', this being assigned to the leaf.

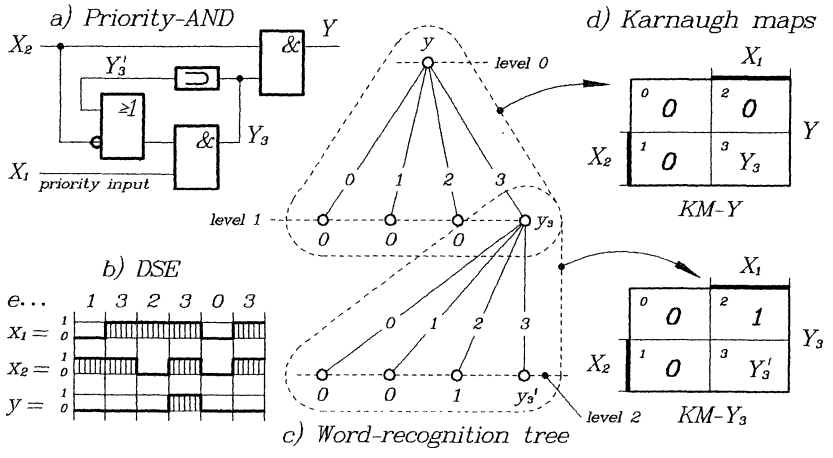


Fig. 1–8 Priority-AND: An example sequential circuit

Fig. 1–8 shows an example of a very simple sequential circuit known among some as a 'priority-AND'. As expressed in the DSE, the output y is 1 when both inputs x_1 and x_2 are 1, but only if x_1 has become 1 *prior* to x_2. Contrary to a pure memory circuit — the word-recogniton tree of which *never grows beyond the first level* — the word-recognition tree of any other

sequential circuit *grows to two or more levels.* The so-called **Karnaugh maps** of Fig. 1–8d are developed from the word-recognition tree of Fig. 1–8c as will be shown in a later chapter. Such Karnaugh maps are the backbone of circuit design and will be discussed at length.

It seems intuitively obvious that the *height* (the number of levels) of a word-recognition tree expresses the longest sequence of input events that a sequential circuit can memorise, and thus act upon. The word-recognition tree is the formal expression of the fact that a *finite* sequence of input events determines the behaviour of a sequential circuit. Finding such finite sequences lies at the heart of *circuit verification.*

1.7 Verifying a Circuit *

Verifying a circuit is a process in which you apply *appropriate* voltages to the input lines, and check whether the *correct* voltage appears on the output line. This is feasible for combinational circuits by verifying one column after another of a table of asserted events. Applying voltages to input lines is algebraically expressed as *substituting* the values 0 or 1 for the input variables x_i. We use the symbol $\hookleftarrow$ (read as '*... is substituted by...*') for **substitution.** When substituting 0 for x_i, the logic *variable* $X_i \Leftrightarrow x_i = 1$ turns into a false *proposition* (when substituting 1, we obtain a true proposition):

$$X_i \Leftrightarrow x_i = 1 \overset{x_i \hookleftarrow 0}{\Longleftrightarrow} 0 = 1 \Leftrightarrow \mathbf{0}, \qquad X_i \Leftrightarrow x_i = 1 \overset{x_i \hookleftarrow 1}{\Longleftrightarrow} 1 = 1 \Leftrightarrow \mathbf{1}.$$

These logic values can be used to substitute for logic variables X_i of logic formulas and of circuits comprised of gates. We shall see how the latter is done for the sequential circuit Fig. 1–8a called a priority-AND. But, first a remark on the verification of *sequential* circuits.

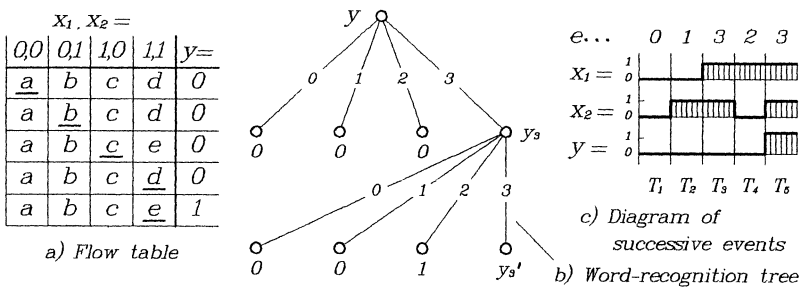


Fig. 1–9 Three specifications of the priority-AND of Fig. 1–8

For any circuit you want to verify you must decide on the sequence of input events you want to apply to the circuit and document such a sequence as a diagram of successive events. But such a diagram does not lend itself to

a formal design method. Circuit specifications, such as the *flow table* or the *word-recognition tree*, see Fig. 1–9, which *do* lead to formal synthesis methods are on the other hand not easily transformed to diagrams of successive events as needed for testing. In fact, this has only been possible for the word-recognition tree. As it has also been shown that a flow table (by far the most common specification method) can be transformed into a word-recognition tree, you can also develop a table of successive events from a flow table.

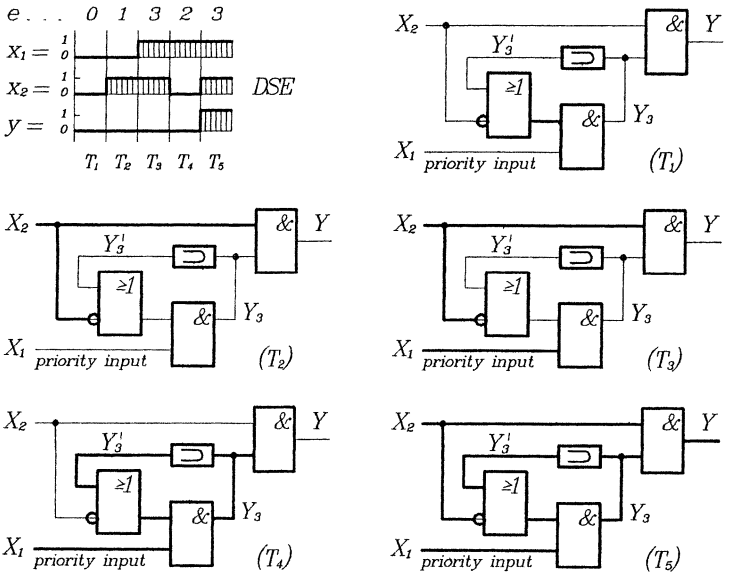


Fig. 1–10 Verifying the priority-AND of Fig. 1–8

The DSE used to test a sequential circuit is modified to avoid that values of two (or more) input variables change simultaneously. Verifying the priority-AND of Fig. 1–8 is shown in Fig. 1–10. The circuit has been reproduced for each time period, allowing us to apply one input event after another, and to see how the signals propagate. The kind of logic analysis demonstrated in Fig. 1–10 is nowadays done with the help of computer programs.

Chapter 2

Switching Devices

Logic gates, as introduced in the previous chapter, represent the connectives used in logic formulas. Assume you have drawn a circuit diagram using logic gates, and you now want to build the circuit using switching devices of a given technology (pneumatic, electric, or electronic). To be able to do so you need to know how to translate each logic symbol into a symbol or a collection of symbols of the technology you intend to employ. This chapter is restricted to discussing symbols used for pneumatic valves, electric relays, and CMOS transistors, and to showing how they correspond to logic gates.

In general, any quantifiable physical property (such as voltage, pressure, force, etc.) can be used as a signal. To be independent of the actual physical property of any *binary* signal (and its unit of measurement, such as Volt, psi, Newton, etc.) it is convenient and customary to map its two values to the *integers* 0 and 1. If nothing is said to the contrary, the smaller numerical value of a signal is mapped to 0, while the larger numerical value is mapped to 1, and these two integers are not assigned any unit of measurement. In this book the lower case letters x and y (most often with indices) shall always be used as variables whose range is $\{0, 1\}$ (denoted as $x, y \in \{0, 1\}$, and meaning that the value of x and that of y is — in any given moment — either 0 or 1), and referred to as **numeric variables**. These variables, x and y, are used to denote the input and output signals of actual switching devices and their symbols.

2.1 Pneumatic Valves

One of the most common valves is the *spool-and-sleeve* design, the principle of which is shown in Fig. 2–1. A *zero* signal refers to atmospheric pressure, a *one* signal to a pressure of say 90 psi (pounds per square inch). If the pilot pressure is 0 the valve is said to be non-actuated and a spring presses the spool in the rightmost position. Actuating the valve by applying a 1-signal to the pilot chamber causes the spool to be pressed to the left (against the force of the spring). In the non-actuated valve you will notice that the output port and the exhaust port are connected while the air supply is blocked. In

the actuated valve the output is connected to the air supply and the exhaust is blocked. Let us refer to this valve as a **normally open valve** (or **NO-valve**, for short), this naming convention being borrowed from the realm of electric relays. *Normal* refers to the pilot pressure x being 0 (the valve is not actuated), whereas *open* refers to the path between the air supply and the output being *interrupted.* The standard symbol for the NO-valve is that of Fig. 2-1(a2). The rectangle to which the spring is attached (in our case, the left rectangle) shows the port connections in the non-actuated (the *normal*) case, the right rectangle depicts the port connections in the actuated case.

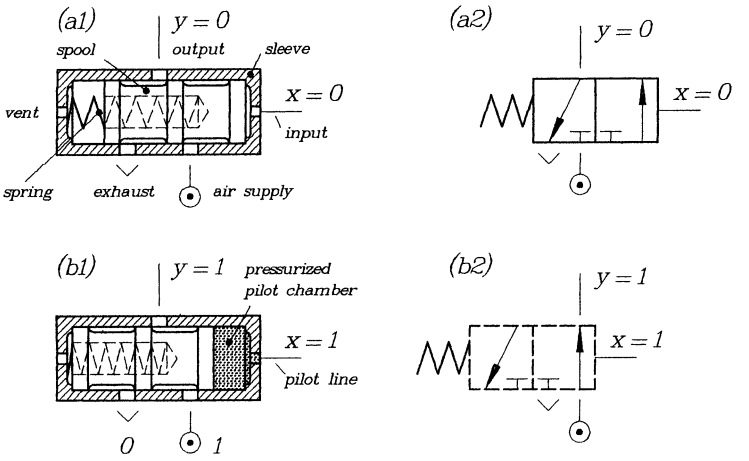


Fig. 2–1 A Normally Open valve

I took the liberty of drawing the *controlled lines* — exhaust, air supply, and output — as being detached from the symbol's rectangles to better show how the symbol is to be understood in the actuated case: Think of the controlled lines as being stationary and the rectangles being pushed to the left by the pilot pressure, as indicated in Fig. 2–1(b2). When the valve is not actuated, the spring pushes the rectangles back into the position shown in Fig. 2–1(a2). In actual use, the controlled lines are always drawn attached to the rectangle (the one with the spring), and *the symbol is always drawn in the* non-actuated *mode.* It is *never* drawn in the actuated mode indicated in Fig. 2–1(b2).

An alternate behaviour to the NO-valve is obtained if the air supply and the exhaust are interchanged as shown in Fig. 2–2a. When the valve is *not* actuated the output is connected to the air supply while actuating the valve connects the output to the exhaust. This behaviour is described by either of the symbols of Fig. 2–2b,c, and the valve is called a **normally closed valve** or **NC-valve**.

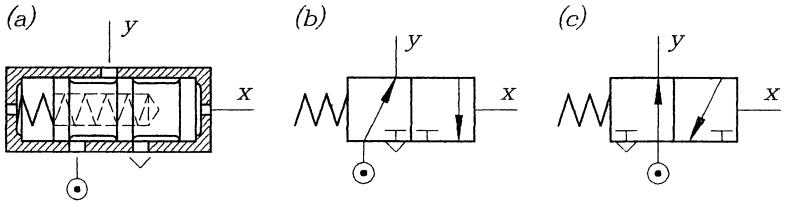


Fig. 2–2 A Normally Closed valve

To analyse the behaviour of a spool-and-sleeve valve (Fig. 2–3) it suffices to work with the valve symbol. The input-output behaviour of such a valve is best documented in a table of events as shown in Fig. 2–3(a2) and (b2).

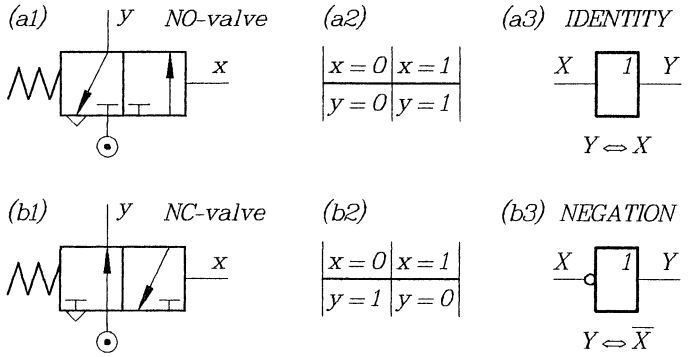


Fig. 2–3 Spool-and-sleeve valves as power amplifiers

According to the TAE of Fig. 2–3(a2), 'the output y is 1 iff the input x is 1', or (in falling back on Section 1.4)

$$y = 1 \Leftrightarrow x = 1,$$
$$Y \Leftrightarrow X,$$

this function being called the IDENTITY, as the output Y is always identical to the input X. The symbol for the gate is shown in Fig. 2–3(a3). You will probably not be alone in asking what good the IDENTITY function is anyway, as its output is always the same as its input. Frankly, *the IDENTITY function serves no logical purpose.* But, it is of real technical importance as a *power amplifier.* In pneumatics, power is determined as the product of flow times pressure. A low-power pilot-signal can control a high-power air-supply to the output. Such a switch is called **active.**

The behaviour of the NC-valve, on the other hand, is fully documented in the TAE of Fig. 2–3(b2), and from this follows

$$y = 1 \Leftrightarrow x = 0,$$
$$Y \Leftrightarrow \overline{X}.$$

The above function is called a NEGATION, the output always having the complementary value to the input.

The air supply of a spool-and-sleeve valve can be replaced by an input *signal*, as indicated in Fig. 2–4. The difference is that the air supply is *always* 1 while a signal can *vary*, meaning it can be either 0 or 1.

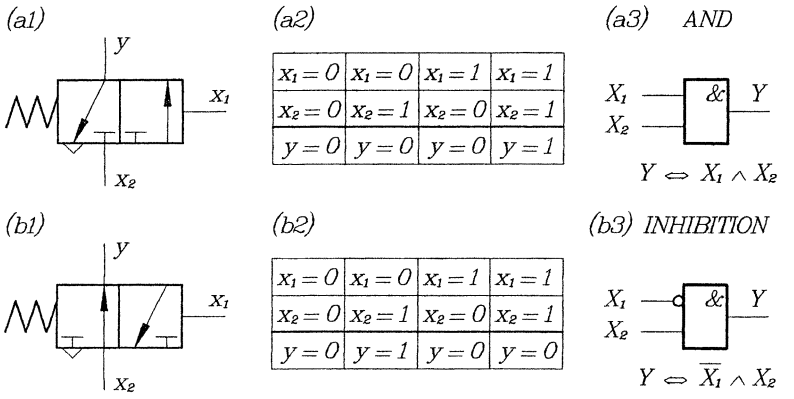


Fig. 2–4 Spool-and-sleeve valves as logic switches

You will hopefully be able to recognise the valves' behaviour as being documented in the two TAEs. The functions associated with these TAEs are called AND and INHIBITION. The latter name stems from the fact that x_1 *inhibits* y, meaning that, when $x_1 = 1$, the output y is blocked. It is probably routine to you by now to express the TAEs of Figs. 2–4(a2) and (b2) in the following formal manner:

$$y = 1 \Leftrightarrow x_1 = 1 \text{ AND } x_2 = 1 \qquad y = 1 \Leftrightarrow x_1 = 0 \text{ AND } x_2 = 1$$
$$Y \Leftrightarrow X_1 \wedge X_2 \qquad Y \Leftrightarrow \overline{X}_1 \wedge X_2$$

In the above usages the spool-and-sleeve valve cannot realise the logical OR function, a function no design engineer would like to do without. A cheap and effective realisation of the OR function is the shuttle valve of Fig. 2–5a. The ball in the design shown will always seal the input of lower pressure, connecting the output with the input of higher pressure. This valve is called a **passive** valve as the air of the output signal is derived from an input line, not from an air supply. This is a drawback, as is the fact that the valve does not have an exhaust of its own; the output must be exhausted via an input line. The standard symbol for this valve (Fig. 2–5b) is obviously inspired by the design shown in Fig. 2–5a. The TAE describing the valve's behaviour marks the valve as a realisation of the OR function.

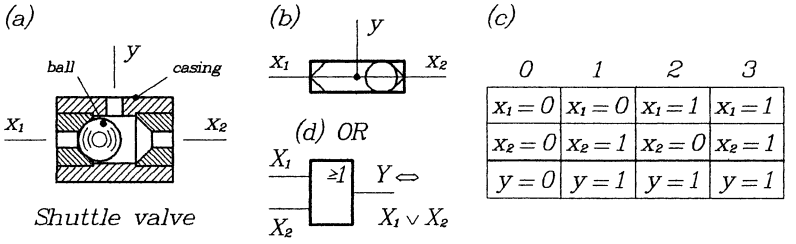


Fig. 2–5 A shuttle valve to realise the OR function

To see that the TAE of Fig. 2–5c does describe the OR function, read the table row wise (and not column wise, as we have been doing up till now). First note that 'if $x_1 = 1$ then $y = 1$' (this covering columns 2 and 3), and then that 'if $x_2 = 1$ then $y = 1$' (which covers columns 1 and 3). Thus we reason that '$y = 1$ iff $x_1 = 1$ OR $x_2 = 1$', and write this as

$$y = 1 \Leftrightarrow x_1 = 1 \text{ OR } x_2 = 1,$$
$$Y \Leftrightarrow X_1 \vee X_2.$$

We are now in a good position to transform a given logic circuit into one consisting of valve symbols. Such a transition is shown in Fig. 2–6 for the running example of Chapter 1. Of course this example is only used to demonstrate a technique, and has nothing to do with the original problem of switching lights on or off. To replace the INHIBITIONs of Fig. 2–6a by NC-valves, note the correspondence laid down in Fig. 2–4.

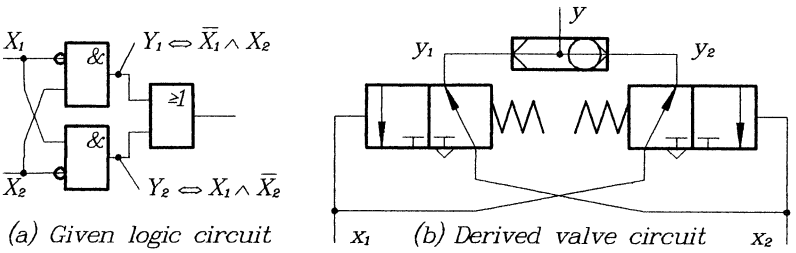


Fig. 2–6 A valve realisation of a logic circuit

2.2 Electric Relays

In its simplest form, an electric relay (see Fig. 2–7) consists of a coil, a plunger, and a contact that is actuated by the plunger. When currant flows

through the coil its magnetic field attracts the plunger, thereby closing a *normally open contact* (Fig. 2–7(a1)) and opening a *normally closed contact* (Fig. 2–7(b1)). Interrupting the currant to the coil allows a spring to return the plunger, and thus the contacts, to their original positions.

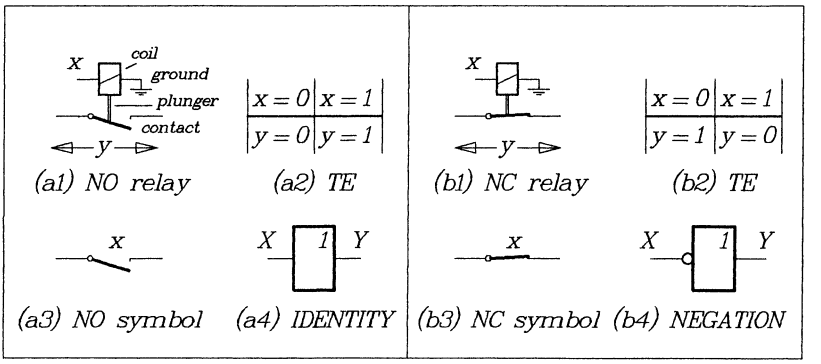


Fig. 2–7 The electric relay: principle, IO-behaviour, symbol

Every switching device must have one or more inputs, and one or more outputs. Let us consider what the input and output signals of a relay are. It seems clear that there is only *one* **input** signal, namely the voltage x applied to the coil of the relay. The **output** signal y is best defined directly as a binary numeric signal $y \in \{0,1\}$ that is 0 if the contact is *open*, and is 1 if the contact is *closed*; it is commonly called **transmission**. We can now formulate the input-output behaviour of a NO-relay as in the TAE of Fig. 2–7(a2), and of a NC-relay as in the TAE of Fig. 2–7(b2). Using these TAEs to state unequivocally when the output y is 1 shows clearly that a relay with one contact is either an IDENTITY or a NEGATION

$$y = 1 \Leftrightarrow x = 1, \qquad y = 1 \Leftrightarrow x = 0,$$
$$Y \Leftrightarrow X, \qquad Y \Leftrightarrow \overline{X},$$

allowing us to use the appropriate gate — Figs. 2–7(a4) or (b4) — to describe the relay. The standard **electric symbols** for relays — Figs. 2–7(a3) and (b3) — are rather unusual as they obscure the fact that not only a switching device, but also its symbol, should have a well defined input and an equally well defined output. In the symbol you *must* state the name of the input variable — in our case x — and whether the relay is an IDENTITY (draw a NO-contact) or a NEGATION (draw a NC-contact).

More often than not, a relay has multiple contacts, all operated simultaneously by the plunger (Fig. 2–8a). Each contact is assigned an individual standard electric symbol (Fig. 2–8b), and each such symbol may be used and positioned independently of the others. The gate representation of the relay is shown in Fig. 2–8c.

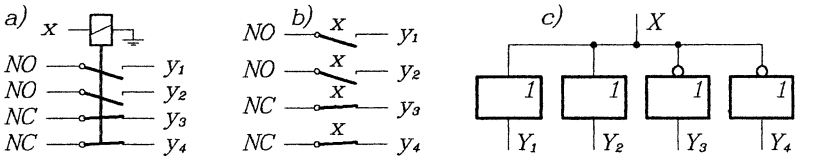
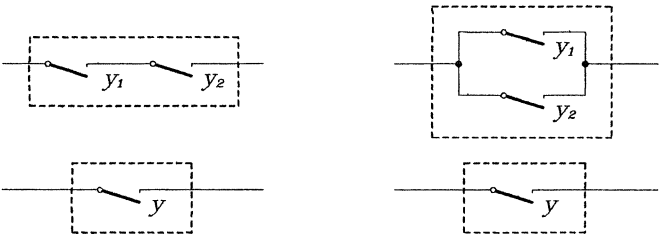


Fig. 2–8 A relay with multiple contacts

It is obvious, after what has been said above, that a single relay does not realise AND or OR functions. *The* AND *and* OR *functions are realised in the way relay* contacts *are wired together or interconnected.* Contacts — and I am speaking *exclusively* of contacts, not of complete relays — may be interconnected in series or in parallel, as pictured in Fig. 2–9. In each case we conceive a surrogate contact y with the same transmission as the interconnected contacts. In particular, the transmission y of the surrogate contact is defined as follows.



a) Contacts in series *b) Contacts in parallel*

Fig. 2–9 Connecting contacts in series and parallel

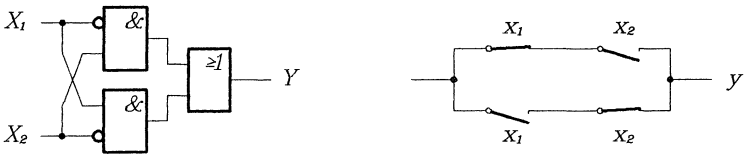
Contacts y_1 and y_2 in series:

$$y = 1 \overset{df}{\Leftrightarrow} y_1 = 1 \text{ AND } y_2 = 1,$$
$$Y \overset{df}{\Leftrightarrow} Y_1 \wedge Y_2. \tag{2.1}$$

Contacts y_1 and y_2 in parallel:

$$y = 1 \overset{df}{\Leftrightarrow} y_1 = 1 \text{ OR } y_2 = 1,$$
$$Y \overset{df}{\Leftrightarrow} Y_1 \vee Y_2. \tag{2.2}$$

The contacts in Fig. 2–9 with the transmissions y_1 and y_2 can be the contacts of NO-relays or NC-relays, as required. Again, take the logic circuit (Fig. 2–10a) of the running example of Chapter 1. The AND gates lead to the upper and lower serial paths in Fig. 2–10b, while the OR gate is equivalent to the parallel wiring of the two serial paths.



a) Given logic circuit *b) Derived relay circuit*

Fig. 2–10 Realising a given logic circuit with electric relays

A more complicated logic circuit is sometimes quite awkward to transform into a relay network, especially as there are a number of ways to do so. Let me demonstrate this with the monstrous invention of Fig. 2–11.

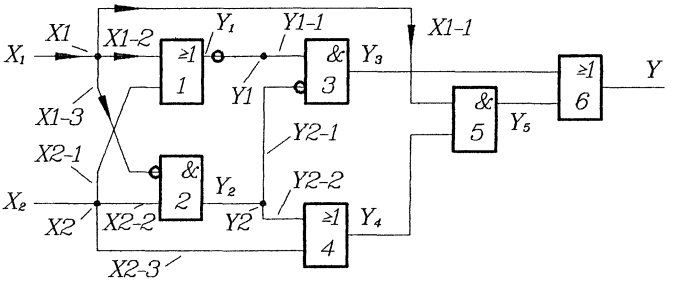


Fig. 2–11 A logic circuit to be realised with electric relays

A foolproof method for transforming a logic circuit into a relay circuit is to **mimic the logic gates** as shown in Fig. 2–12a. Each gate is realised by a relay, the gate output representing the coil of the relay; the contacts of the relay are commonly named with the same letter as the coil. This sparse comment should suffice to let you understand how Fig. 2–12a was obtained. This method uses far too many relays and should be avoided on this and other grounds.

The circuit of Fig. 2–12b has a **minimised number of relays** — four, all in all. The number of relays is found by counting the number of *nodes* in the given logic circuit — ($X1$, $X2$, $Y1$ and $Y2$). It is practical to distinguish between two kinds of nodes that I shall call *positive* and *negative*. A **positive node** is a junction comprised of one input lead originating from a *non-inverted* input, or from a gate with a *non-inverted* output, and two or more output leads that connect to inverted or non-inverted inputs of gates. A **negative node**, on the other hand, is a junction comprised of one input lead originating from an *inverted* input, or from a gate with an *inverted* output, and *one* or more output leads that connect to inverted or non-inverted inputs of gates. Each node is associated with the coil of a relay; each output lead from the node is associated with *one and only one* contact of the relay. The contact — in the case of a positive node — is normally open (NO) *iff*

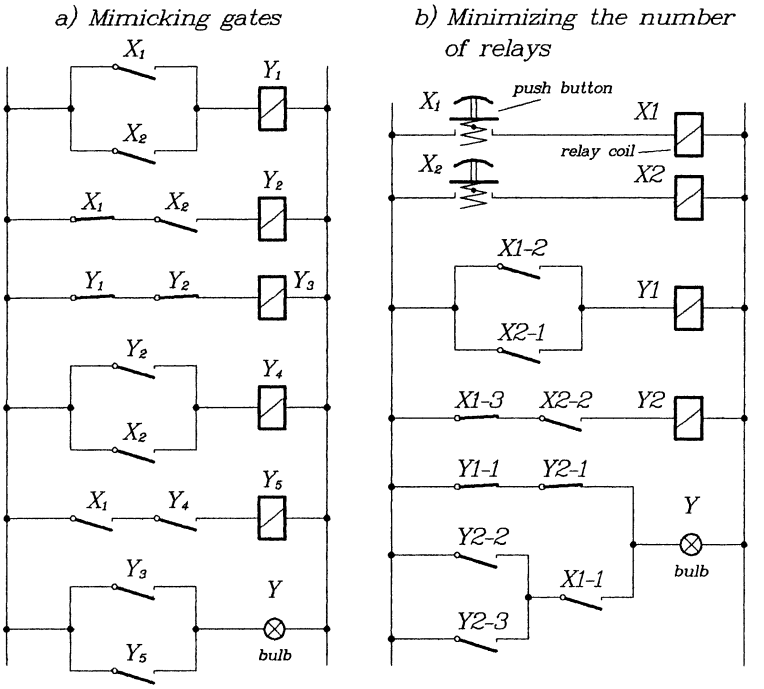


Fig. 2–12 Two methods of realising a logic circuit with electric relays

the lead's input to the gate to which it connects is non-inverted. In the case of a negative node the contact is normally closed (NC) *iff* the lead's input to the gate to which it connects is non-inverted. One thing to note about negative nodes with a *single* output lead is that you introduce the node as a *virtual* node, as was done for node $Y1$.

$x_1 = 0$	$x_1 = 0$	$x_1 = 1$	$x_1 = 1$
$x_2 = 0$	$x_2 = 1$	$x_2 = 0$	$x_2 = 1$
$y = 1$	$y = 0$	$y = 0$	$y = 1$

a)

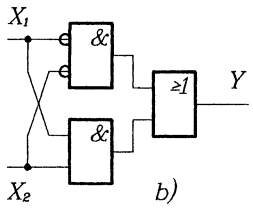


Fig. 2–13 Analysis of the previous circuit

To compare the relay circuits of Figs. 2–11a,b you must either drop the two push buttons from Fig. 2–12b (together with the associated relay X_1 and X_2) or add them to Fig. 2–12a. When adding push buttons to

Fig. 2–12a, mimicking gates needs seven relays as compared to four relays when minimising the number of relays.

You might like to know why I called the logic circuit of Fig. 2–11a monstrous. The only reason is that the circuit can be reduced to that shown in Fig. 2–13b, as can be deduced from its TAE of Fig. 2–13a which (as you are invited to verify) describes the IO-behaviour of the circuit of Fig. 2–11a.

2.3 CMOS Transistor Circuits

The transistor was invented in 1948 by John Bardeen, Walter Brattain and William Shockley and soon replaced the vacuum tube as a cheap amplifier. We shall take a look at only one transistor device, of a large proliferation, namely the CMOS technology. The transistors used are **insulated-gate field-effect transistors (IG-FET)** — Fig. 2–14 — more commonly called **MOS** for the sequence of the layers of material used in their design — **metal-oxide-silicon**. There are two types of these transistors, according to the polarity (**n**egative or **p**ositive) of the electric charges used to create a conducting channel between the *drain* and the *source* regions: *n-channel MOS* (**NMOS**) and *p-channel MOS* (**PMOS**). Digital circuits that employ both these transistor types are called **complementary MOS (CMOS)**, and have the advantage of needing only a single level of supply voltage.

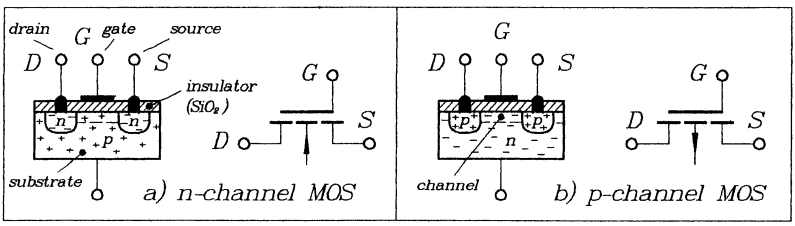


Fig. 2–14 Principle of the n-channel and p-channel MOS

For current to be able to flow between the drain and the source region, you need to *create a conducting channel* in the silicon between these regions. This is achieved by applying the proper voltage between the gate and the substrate. In an n-channel MOS you obtain a conducting channel by applying a more positive potential to the gate than to the substrate (the voltage from gate to substrate is positive), thus attracting negative charges to the gate while repulsing positive charges. In a p-channel MOS the conducting channel is obtained by making the voltage from substrate to gate positive, thus attracting positive charges to the gate while repulsing negative charges from the gate. *In actual usage, the source is always connected to the substrate.* To obtain a channel, the density of source-type charges must be enhanced giving these transistors the name **enhancement type transistors**.

The simplest CMOS circuit is the **inverter** of Fig. 2–15. Its principle of operation is basic to CMOS technology. If you understand how the inverter works, you almost automatically understand the next two gates. The idea behind the inverter is shown in the relay circuit of Fig. 2–15a: The two serial contacts are always in a state of alternate transmission — when one is closed, the other is open. When the top switch, the NC-contact, is closed and the bottom switch, the NO-contact, open, the full supply voltage is fed through to the output y. *Vice versa*, the output y is zero when the top switch is open and the bottom one closed.

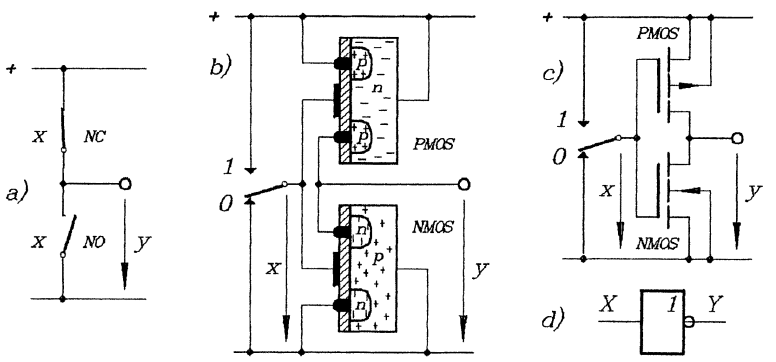


Fig. 2–15 The CMOS inverter

The relay circuit translates directly to the transistor circuit of Fig. 2–15b, the NC-contact being replaced by a PMOS, the NO-contact by an NMOS. In the circuit as drawn, the gates of both transistors are at ground potential dew to the way the switch governing the gate potential is set. As the substrate of the PMOS is connected to high potential a conducting p-channel is created. The NMOS transistor, on the other hand, is non-conducting as it has 0 voltage between gate and substrate, both being connected to ground. The voltage measurable at the output y is thus high. If the switch is brought to the alternate position, in which the potential at the gates of both transistors is high, the PMOS becomes non-conducting while the NMOS becomes conducting. The output voltage y is thus brought to zero. The actual circuit, of course, is always drawn using MOS symbols, giving you Fig. 2–15c. As the input and output of this circuit are always complementary, it is an inverter or a negation, the symbol for which is shown in Fig. 2–15d.

There are two successful ways, of four trivial ones, to generalise the principle inverter circuit of Fig. 2–15a: One of these is shown in Fig. 2–16a, the other in Fig. 2–17a, and both lead to circuits superbly adapted to CMOS technology.

The first generalisation of the inverter of Fig. 2–15a is achieved by substituting the single NC-contact by two serial NC-contacts, and substituting the single NO-contact by two parallel NO-contacts — this leading to Fig. 2–16a.

The equivalent CMOS circuit is obtained by substituting PMOS transistors for NC-contacts, and NMOS transistors for NO-contacts. As with the inverter of Fig. 2–15c, note that in Fig. 2–16b the sources and substrates of the PMOS transistors are connected to or towards high potential, while the sources and substrates of the NMOS transistors are connected to low potential.

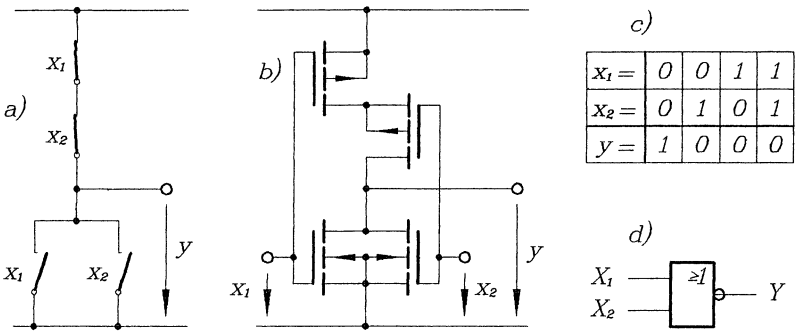


Fig. 2–16 The CMOS NOR-circuit

The simplest way to analyse the circuit of Fig. 2–16 is to develop the TAE shown in Fig. 2–16c. You start out with the table of input events, leaving the bottom row, the one for the output y, empty. Then, for each input event, you deduce from the behaviour of the circuit the value for the output y, entering the outputs into the bottom row. Considering the result, you will notice that inverting each output value leads to the **or** function. In other words, the TAE of Fig. 2–16c is that of a **not-or** function, usually referred to as the **nor** function. The logic symbol used for this function is shown in Fig. 2–16d; it is simply an OR gate with an inverted output.

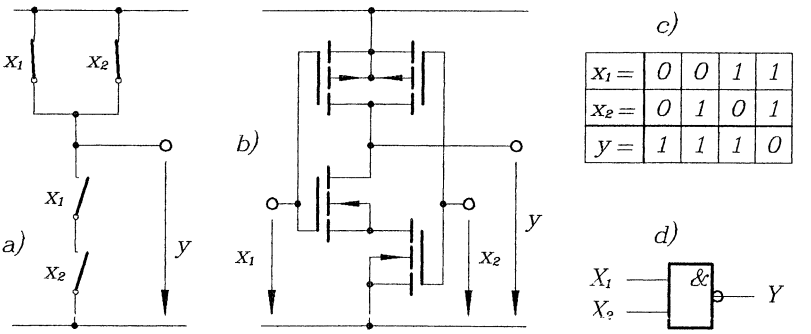


Fig. 2–17 The CMOS NAND-circuit

The second generalisation of the inverter leads to Fig. 2–17a and that to Fig. 2–17b. The analysis needed to obtain the TAE of Fig. 2–17c is easy so that I again leave the details to you. The result, as you will have noticed, is the inversion of the **and** function, i.e. is a **not-and** function, and is thus referred to as the **nand** function, the logic symbol for which is shown in Fig. 2–17d.

Of course the CMOS circuits discussed here are only a small excerpt of the circuits used. But they are basic and absolutely essential to the understanding of the techniques employed. One of the most important omissions is a discussion of circuits with feedback, circuits that lead to the realisation of memory devices — so-called *latches* and *flip-flops*. These circuits are discussed, together with their theory, in later chapters.

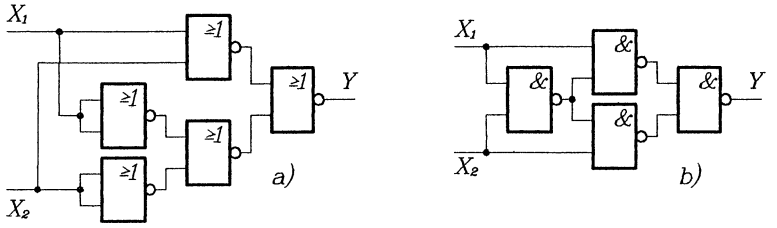


Fig. 2–18 NOR and NAND versions of our running example

The logic building blocks of CMOS technology are NAND-, NOR- and NEGATION-gates. These, therefore, one wishes to use when realising logic circuits in CMOS technology. To get an impression of pure NOR- and NAND-design, take a look at Fig. 2–18. These circuits are equivalent to that of Fig. 1–4 of our running example of Chapter 1.Transforming larger AND, OR and NOT based circuits into NOR or NAND circuits can be quite demanding.

Chapter 3

Functions

When reading through this book, you will frequently be confronted with various kinds of *functions* and *relations*. Not all interested readers have grown up with the same concept of function, so it seems worth while to take a short time off to ponder the concept. According to your background take a more thorough, or merely a cursory look at this chapter. But do make sure you understand the concept of function as it is used in this book. The concept of function is based on that of the *ordered pair* this, therefore, being discussed first.

3.1 Ordered Pairs

Nowadays, mathematical concepts are explained in terms of sets. Quite informally, a **set** $\mathcal{S}$ is a collection of real or abstract things — we call them the *elements* s of the set, writing this as $s \in \mathcal{S}$ (which is read as 's is an element of $\mathcal{S}$'). The sets relevant to switching algebra are all finite, i.e., they have a finite number of elements. As a trivial example take the set of input events for three input variables (see Fig. 1–3) and defined by listing them (within curly braces):

$$\{(0,0,0),(0,0,1),(0,1,0),(0,1,1),(1,0,0),(1,0,1),(1,1,0),(1,1,1)\}.$$

Sometimes it is impracticable to list the elements of a finite set, as when trying to list *'those who have read this book'*. Thus, it has become customary to specify sets predominantly by saying what attributes the elements must have in common. For example, the mathematician might write

$$\mathcal{R} \stackrel{df}{=} \{\, r \mid r \text{ has read this book}\}. \tag{3.1}$$

Although the above notation is the one most common it has the formal drawback of not being formulated in terms the $\in$-relationship, as is the case in the following way of expressing R:

$$r \in \mathcal{R} \Leftrightarrow (r \text{ has read this book}) \tag{3.2}$$

meaning: $\mathcal{R}$ consists exactly of all those elements r for which the defining expression (written to the right of $\Leftrightarrow$) is true.

The **equality of two sets** is explained by the *material* EQUIVALENCE ($\leftrightarrow$, see Fig. 4–3):

$$\mathcal{S} = \mathcal{T} \stackrel{df}{\Leftrightarrow} \text{For all } x\text{: } (x \in \mathcal{S} \leftrightarrow x \in \mathcal{T}), \tag{3.3}$$

stating that two sets are equal *iff* they have the same elements. This implies that the sequence in which the elements are written is irrelevant ($\{0,1,2,3\} = \{2,3,1,0\}$), and that two sets are the same even if certain elements are repeated ($\{0,1,2,3\} = \{2,3,2,2,1,0,1\}$).

The **number of elements** (the cardinal number) of a finite set $\mathcal{S}$ is denoted as $\#\mathcal{S}$. Stating that $\{0,1,2,3\}$ has four elements would thus be written as $\#\{0,1,2,3\} = 4$. The **cardinal number** of a *finite set* is the number you stop at when having counted all the elements of the set.

Contrary to the just mentioned examples, it is certainly advisable to abide by the sequence of integers and their possible duplicity when dialling a telephone number. Such *ordered collections* of elements are written between *parentheses* (not between *braces*, as when specifying sets). Here, a few examples, primarily to set forth the Latin based naming conventions leading to the expression 'n-tuple':

(a,b)	ordered pair	2 ordered elements
(a,b,c)	ordered triple	3 ordered elements
(a,b,a,b)	ordered quadruple	4 ordered elements
$(4,4,3,9,3)$	ordered quintuple	5 ordered elements
$(a_1,a_2,a_3,a_4,a_5,a_1)$	ordered sextuple	6 ordered elements
$(a_7,a_6,a_5,a_4,a_3,a_2,a_1)$	ordered septuple	7 ordered elements
$(a_1,a_2,\ldots,a_{n-1},a_n)$	ordered **n-tuple**	n ordered elements

Of course, a mathematician would define an **ordered pair** (a_1,a_2) on the basis of sets, for instance as $(a_1,a_2) \stackrel{df}{=} \{\{a_1\},\{a_1,a_2\}\}$. And from this he would prove that for two ordered pairs (a_1,a_2) and (a_1',a_2') to be equal, equally positioned elements must be equal, and would write this formally as

$$(a_1,a_2) = (a_1',a_2') \Leftrightarrow (a_1 = a_1') \wedge (a_2 = a_2'). \tag{3.4}$$

Generalising, the **ordered n-tuple** lends itself comfortably to the recursive definition $(a_1,a_2,\ldots,a_n) \stackrel{df}{=} ((a_1,a_2,\ldots,a_{n-1}),a_n)$, again, the question of when two n-tuples are equal, being the actual matter of interest:

$$\begin{aligned}(a_1,a_2,\ldots,a_n) = (a_1',a_2',\ldots,a_n') \Leftrightarrow \\ (a_1 = a_1') \wedge (a_2 = a_2') \wedge \ldots \wedge (a_n = a_n').\end{aligned} \tag{3.5}$$

One obvious interpretation of the ordered pair is to view its elements as the Cartesian co-ordinates of a point in a plane. All the points of a given

rectangular area $L \cdot W$, of length L and width W, would represent all the ordered pairs (l, w), with $l \leq L$ and $w \leq W$. Extrapolating, all the points of the volume $L \cdot W \cdot H$ of a rectangular parallelepiped of height H represent all ordered triples (l, w, h), with $h \leq H$. Taking these examples as a lead, and calling the *set of all n-tuples* a **Cartesian product**

$$\begin{aligned} &\mathcal{A}_1 \times \mathcal{A}_2 \times \ldots \times \mathcal{A}_n \stackrel{df}{\Leftrightarrow} \\ &\quad \{(a_1, a_2, \ldots, a_n) \,|\, a_1 \in \mathcal{A}_1 \wedge a_2 \in \mathcal{A}_2 \wedge \ldots \wedge a_n \in \mathcal{A}_n\}, \end{aligned} \tag{3.6}$$

the number of elements of the Cartesian product can be shown to be

$$\#(\mathcal{A}_1 \times \mathcal{A}_2 \times \ldots \times \mathcal{A}_n) \Leftrightarrow (\#\mathcal{A}_1).(\#\mathcal{A}_2).\cdots.(\#\mathcal{A}_n). \tag{3.7}$$

We shall mainly use the ordered n-tuple when working with input events $(e_1, e_2, \ldots, e_n)$. Of course, each input value is either 0 or 1

$$e_1, e_2, \ldots, e_n \ \in \ \{0, 1\}, \tag{3.8}$$

allowing us to rewrite (3.6) as

$$\{0,1\}^n \Leftrightarrow \left\{(e_1, e_2, \ldots, e_n) \,|\, e_1 \in \{0,1\} \wedge e_2 \in \{0,1\} \wedge \ldots \wedge e_n \in \{0,1\}\right\} \tag{3.9}$$

in which $\{0,1\}^n$ is defined as

$$\{0,1\}^n \stackrel{df}{=} \underbrace{\{0,1\} \times \{0,1\} \times \ldots \times \{0,1\}}_{n \text{ times}}. \tag{3.10}$$

The right side of (3.9), and thus the n-fold Cartesian product $\{0,1\}^n$ itself, is the **set of all input events**. Noting that the set $\{0,1\}$ has 2 elements $(\#\{0,1\} = 2)$, we can rewrite (3.7) to state the **number of input events**:

$$\#(\{0,1\}^n) = \underbrace{2.2.\cdots.2}_{n \text{ times}} = 2^n. \tag{3.11}$$

$(e_1, e_2, \cdots, e_n)$ is also the notation of an n-dimensional vector **e**:

$$\mathbf{e} \stackrel{df}{=} (e_1, e_2, \ldots, e_n), \tag{3.12}$$

while in the form $(e_1, e_2, \ldots, e_n)_2$ the n-tuple is also used as the binary (base 2) number representation of the decimal number $(e)_{10}$. Given the binary number $(e_1, e_2, \ldots, e_n)_2$ — for which (3.8) holds — you calculate its decimal value $(e)_{10}$ by employing the formula

$$(e)_{10} = \sum_{i=1}^{n} e_i.2^{n-i} = e_1.2^{n-1} + e_2.2^{n-2} + \ \ldots \ + e_{n-1}.2^1 + e_n.2^0. \tag{3.13}$$

	0	1	⋯	e	⋯	2^n-2	2^n-1	
$x_1=$	0	0	⋯	e_1	⋯	1	1	2^{n-1}
$x_2=$	0	0	⋯	e_2	⋯	1	1	2^{n-2}
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
$x_{n-1}=$	0	0	⋯	e_{n-1}	⋯	1	1	2^1
$x_n=$	0	1	⋯	e_n	⋯	0	1	2^0

Fig. 3–1 Assigning a decimal equivalent e to each input event $(e_1, \ldots, e_n)$

By this formula the given binary number $(1010)_2$ is transformed into its decimal equivalent $(10)_{10}$:

$$1.2^3 + 0.2^2 + 1.2^1 + 0.2^0 = 8 + 0 + 2 + 0 = (10)_{10}.$$

When referring to decimal numbers we usually omit the number base (the index 10), writing e instead of $(e)_{10}$. To better understand (3.13), take a look at Fig. 3–1 in which each column contains a binary number $(e_1, e_2, \ldots, e_n)_2$. The *weights* 2^{n-i} of each variable e_i are written to the right of the table, while the decimal equivalent e is written atop each column.

If, on the other hand, you want to find the binary equivalent $(e_1, \ldots, e_n)_2$ of a given decimal number e you can use the algorithm demonstrated in the example of Fig. 3–2. Say, the given decimal number is 10. To find the binary equivalent, you start by drawing a table as that of Fig. 3–2a. The calculations to be done are shown in Fig. 3–2b: You divide 10 by 2, obtaining the quotient 5 and a remainder of 0. Now, halve 5, obtaining a new quotient of 2, and a remainder of 1. You continue halving each quotient, obtaining a new one together with a remainder 0 or 1. The process terminates when you obtain a quotient of zero (in which case the remainder will always be 1).

10 | 2

(a)

10 | 2
5 | 0
2 | 1
1 | 0
0 | 1

(b)

10 | 2	
5 | 0	... e_n
2 | 1	... e_{n-1}
1 | 0	... e_{n-2}
0 | 1	... e_{n-3}

(c)

Fig. 3–2 Finding the binary equivalent $(e_1, e_2, \ldots, e_n)_2$ of a given decimal number e

The remainders $(e_{n-3}, e_{n-2}, e_{n-1}, e_n)$ in the example of Fig. 3–2c are the rightmost digits of a binary number $(e_1, e_2, \ldots, e_{n-3}, e_{n-2}, e_{n-1}, e_n)_2$

that has 4 or more binary digits, with all binary digits to the left of e_{n-3}, if there are any, being zero. Thus, a binary equivalent of $(10)_{10}$ is, e.g., $(1010)_2$ or $(00001010)_2$.

3.2 Speaking of Functions...

It is not uncommon for the notion of *function* to be grasped only intuitively and thus to be obscure. A *function* is nothing as fuzzy as a *dependence* of one value, say y, upon another, say x, nor is it a *rule* by which values are *assigned* one to another. Likewise, it is neither a *formula*, nor a *table*, nor is it a *graph*.

The just mentioned concepts represent only certain aspects of a function but none of these concepts are comprehensive. Let us develop a comprehensive notion of function, doing so in small steps. Remember that mathematical concepts are, nowadays, explained in terms of sets. So, the first thing to note is: *A function is a set.* Of course, a function cannot be just any set, its elements must be quite special so as to represent the notions of dependence, rule, formula or the like, thus: *A function is a set of ordered pairs.* Clearly, the intention is that the ordered pair, say (x, y), represents the values obtained, e.g., by a formula, table, or graph; but this intention must be stated expressly. We first of all say what values can be used in our ordered pair (x, y) by specifying two sets, say $\mathcal{X}$ and $\mathcal{Y}$, and requiring the elements x and y to be taken from these sets: *A function is a set of ordered pairs* (x, y) *in which* $x \in \mathcal{X}$ *and* $y \in \mathcal{Y}$.

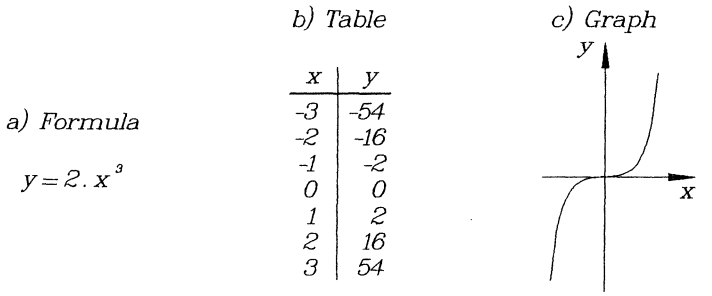


Fig. 3–3 Neither a *formula*, nor a *table* nor a *graph* is a *function*

This is still far too vaguc to avoid misinterpretation. Compare, for instance, the table and the graph of Fig. 3–3: they cannot represent the same function as the table contains far less values than the graph does. Any two representations must contain *all* arguments x of a given set $\mathcal{X}$ leading to: *A function*

is a set of ordered pairs (x,y) *in which* each *element* $x \in \mathcal{X}$ *is a first element of* (x,y) *while* $y \in \mathcal{Y}$. There is one more restriction to consider. We want functions, as contrasted to *relations*, to provide us with one and only one value for any given argument. An example to the contrary, that is, an example for a relation is provided by $y = \sqrt{x}$ where each argument x gives us *two* values for y, e.g., $\sqrt{4} = +2$ and $\sqrt{4} = -2$. This leads to: *A function is a set of ordered pairs* (x,y) *in which* each *element* $x \in \mathcal{X}$ *occurs exactly once as a first element of* (x,y) *while* $y \in \mathcal{Y}$.

For better or for worse, there is a long-standing tradition among mathematicians to formulate definitions as concisely as possible. Thus, you will usually find the above definition formulated something like this: *A* **function** *is a set of ordered pairs* (x,y) *such that for each* $x \in \mathcal{X}$ *there exists exactly one such ordered pair with* $y \in \mathcal{Y}$. A common and helpful way to visualise a function is a *mapping* as in the example of Fig. 3–4a. In general, $\mathcal{X}$ is called the **domain** and $\mathcal{Y}$ the **co-domain** of the function f. Each element (x,y) of the function f is represented by an *arrow* that points from x to y, i.e., *maps* x to y. This mapping of the elements of the domain $\mathcal{X}$ into the elements of the co-domain $\mathcal{Y}$ is usually symbolically denoted as $\mathcal{X} \overset{f}{\mapsto} \mathcal{Y}$ or as $f : \mathcal{X} \mapsto \mathcal{Y}$, the symbol $\mapsto$ intentionally being reminiscent of the arrows in Fig. 3–4. It is a long established tradition to denote the value y of a function f as $f(x)$, i.e., $f(x) \overset{df}{=} y$, or $y = f(x) \overset{df}{\Leftrightarrow} (x,y) \in f$. Do note that the definition of a function does not require *all* elements y of the co-domain $\mathcal{Y}$ to be second elements of the ordered pairs $(x,y) \in f$. Those $y \in \mathcal{Y}$ for which $(x,y) \in f$ is true comprise the **range** of the function f.

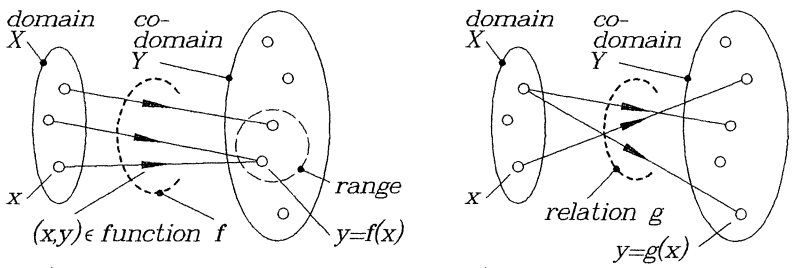


Fig. 3–4 Picturing functions and relations as mappings

Without any further proof, let me state the formula by which to calculate the **number** φ **of functions** that can exist from a domain $\mathcal{X}$ into a co-domain $\mathcal{Y}$. For mnemonic reasons let δ be the number of elements of the domain, and κ the number of elements of the co-domain, allowing us to write

$$\delta \overset{df}{=} \#\mathcal{X}, \qquad \kappa \overset{df}{=} \#\mathcal{Y}.$$

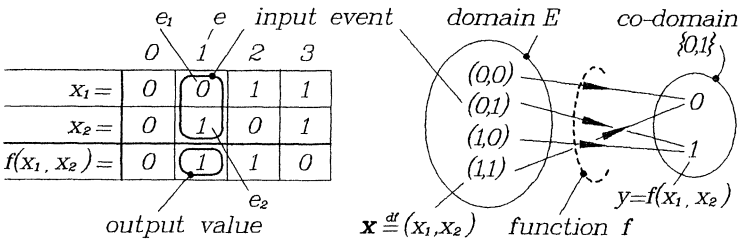
$$\varphi = \kappa^{\delta}. \tag{3.14}$$

To define a certain function, you need to specify three sets: the domain $\mathcal{X}$, the co-domain $\mathcal{Y}$, and the set f of all ordered pairs (x, y) with $x \in \mathcal{X}$ and $y \in \mathcal{Y}$. Usually f will be specified in the form of a formula, a table, or a graph as in Fig. 3–3, but any specification that is complete and unambiguous is valid. Likewise, two functions $f : \mathcal{X} \mapsto \mathcal{Y}$ and $g : \mathcal{R} \mapsto \mathcal{S}$ are equal *iff* their domains are equal, $\mathcal{X} = \mathcal{R}$, and their co-domains, $\mathcal{Y} = \mathcal{S}$, and both sets f and g consist of the same ordered pairs $(x, y) \in f$ and $(r, s) \in g$, i.e., $f = g$.

The mapping of Fig. 3–4b is not that of a function as not all elements of the domain $\mathcal{X}$ are starting points of arrows, and there is one $x \in \mathcal{X}$ from which more than one arrow emerges.

3.3 Switching Functions

As mentioned, switching *circuits* (meaning the technical realisation consisting of, say, transistors, relays, or pneumatic valves) are usually divided into two categories: combinational circuits (having no memorising ability), and sequential circuits (that are able to memorise certain sequences of input events). Mathematically, combinational circuits are *functions* and we call them **switching functions**. They lend themselves very comfortably to logical manipulations or calculations. The mathematical description of a sequential circuit, on the other hand, is a *relation*, and not a function. I.e., there exists at least one input event that can invoke an output value of 0 and of 1 (of course not at the same time). Evaluating relations is quite a problem. Since Huffman, it is usually done by breaking a sequential circuit down into a number of combinational circuits. So, in this section we shall take a first look at switching functions, these being the mathematical description of combinational circuits and, indirectly, of sequential circuits.



a) Table of events b) Mapping of a function

Fig. 3–5 Two visualisations of an elementary switching function

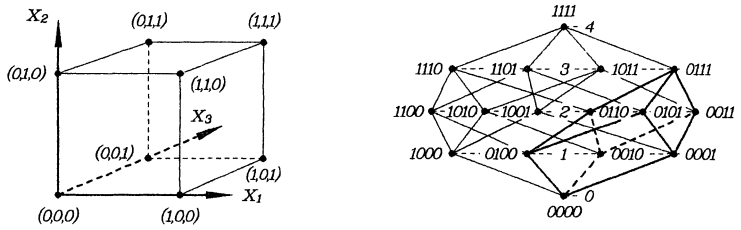
Most frequently, switching circuits have *two or more* input variables. On the other hand, the picture of a function given in Fig. 3–4a tends to

evoke the impression that the definition of a function is restricted to a single 'argument', i.e., input variable. This is wrong for the simple fact that the domain in the definition of a function can be any set, for instance, the set $\{0,1\}^n$ of all input events $\mathbf{x} \stackrel{df}{=} (x_1, x_2, \cdots, x_n)$. With this in mind, we define a **switching function** f as a function from the set $\{0,1\}^n$ of input events into the set $\{0,1\}$ of output values. A switching function is thus any function from $\{0,1\}^n$ into $\{0,1\}$:

$$f : \{0,1\}^n \mapsto \{0,1\}. \tag{3.15}$$

For this general switching function (albeit restricted to a single output), the number φ of functions can be calculated by (3.14) and (3.11):

$$\varphi = \kappa^\delta = \{\kappa = 2, \quad \delta = 2^n\} = 2^{(2^n)}. \tag{3.16}$$



a) Three-dimensional cube

b) Four-dimensional cube

Fig. 3–6 Three- and four-dimensional n-cubes

Those switching functions with two input variables are called **elementary switching functions**, defined as

$$f : \{0,1\}^2 \mapsto \{0,1\}, \tag{3.17}$$

and of which there are, according to (3.16), $2^{(2^2)} = 2^4 = 16$. Fig. 3–5 shows two different representations — the table of asserted events and the mapping — of one and the same elementary switching function. You will usually work with the table of asserted events. But at this stage it is good practice to note carefully the correspondence between the two representations. Each possible combination of 0s and 1s in the bottom row of the TAE corresponds to one of the 16 elementary switching functions, just as each arrangement of the arrows in the mapping corresponds to a function.

The **n-cube** is an altogether different representation of switching functions. The elements of the Cartesian product $\{0,1\}^n$ can be visualised as the

nodes of a n-cube. The three-dimensional case shown in Fig. 3–6a is a cube, and it is this cube that lends its name to all other cases.

The (planar) organisation of a 4-cube, as shown in Fig. 3–6b, centres around the concepts of *Hamming weight* h_w and *Hamming distance* h_d. The **Hamming weight** h_w of an input event refers to the number of its 1s. The **Hamming distance** h_d of two input events is the difference of their Hamming weights. The 4-cube is organised into levels. The input events of each level have the same Hamming weight, this being the number given the level. The edges connect input events whose Hamming distance is 1.

Chapter 4

Logic Functions and Gates

As we have just seen in the last chapter, there are 16 elementary switching functions — functions with two inputs and one output. Each of the elementary switching functions corresponds to an (elementary) logic function. The logic functions are traditionally given names such as AND, OR, IMPLICATION, etc. that, in this book, are written in capital letters. The switching functions themselves are given the same names as the logic functions — e.g., **and, or, implication** etc. — only that they are written in lowercase boldface. In this chapter we discuss all the elementary switching and logic functions, and how they correspond to one another. But, apart from the elementary logic functions (that have two inputs), we shall discuss the functions AND, OR, XOR, and EQUIVALENCE for which gates with two and more inputs exist, thus laying the basis for drawing logic circuits.

4.1 Elementary Switching Functions

The 16 elementary switching functions are summarised in the table of Fig. 4–1 where they are defined in the first column. Each of the 16 output rows y_r contains a possible combination of output values 0 and 1. They are ordered in such a way that their decimal equivalencies increase from top to bottom row. This is achieved by filling in the columns (not the rows) in analogy to Section 1.3.

A horizontal line in the middle of the table divides the outputs into two groups. You will notice that two functions, y_r and y_{15-r}, mirrored across this line, have complementary output values. In logic terms, one such function is the negation of the other.

The connectives or function symbols shown in the central column of Fig. 4–1 were introduced by **Iverson** [1962] and are used in the computer language APL, a language well adapted, among other things, to logical problems. In the context of this book, I took the liberty of restricting the range of the variables of the connectives to the set $\{\mathbf{0}, \mathbf{1}\}$ of truth values, allowing us to write the formulas as logic formulas.

Definition $x_1 = 0\ 0\ 1\ 1$ $x_2 = 0\ 1\ 0\ 1$	Iverson's notation	Name of the switching function
$y_0 = 0\ 0\ 0\ 0$	$Y_0 \Leftrightarrow 0$	**zero**
$y_1 = 0\ 0\ 0\ 1$	$Y_1 \Leftrightarrow X_1 \wedge X_2$	**and**
$y_2 = 0\ 0\ 1\ 0$	$Y_2 \Leftrightarrow X_1 > X_2$	**inhibition**
$y_3 = 0\ 0\ 1\ 1$	$Y_3 \Leftrightarrow X_1$	**identity**
$y_4 = 0\ 1\ 0\ 0$	$Y_4 \Leftrightarrow X_1 < X_2$	*transposed* **inhibition**
$y_5 = 0\ 1\ 0\ 1$	$Y_5 \Leftrightarrow X_2$	**identity**
$y_6 = 0\ 1\ 1\ 0$	$Y_6 \Leftrightarrow X_1 \neq X_2$	**xor**
$y_7 = 0\ 1\ 1\ 1$	$Y_7 \Leftrightarrow X_1 \vee X_2$	**or**
$y_8 = 1\ 0\ 0\ 0$	$Y_8 \Leftrightarrow X_1 \tilde{\vee} X_2$	**nor**
$y_9 = 1\ 0\ 0\ 1$	$Y_9 \Leftrightarrow X_1 = X_2$	**equivalence**
$y_{10} = 1\ 0\ 1\ 0$	$Y_{10} \Leftrightarrow \sim X_2$	**negation, not**
$y_{11} = 1\ 0\ 1\ 1$	$Y_{11} \Leftrightarrow X_1 \geq X_2$	*transposed* **implication**
$y_{12} = 1\ 1\ 0\ 0$	$Y_{12} \Leftrightarrow \sim X_1$	**negation, not**
$y_{13} = 1\ 1\ 0\ 1$	$Y_{13} \Leftrightarrow X_1 \leq X_2$	**implication**
$y_{14} = 1\ 1\ 1\ 0$	$Y_{14} \Leftrightarrow X_1 \tilde{\wedge} X_2$	**nand**
$y_{15} = 1\ 1\ 1\ 1$	$Y_{15} \Leftrightarrow 1$	**one**

Fig. 4–1 The 16 elementary switching functions

When used in an informal manner, Iverson's notation is mnemonically convenient for the connectives $<, \leq, =, \geq, >, \neq$ that compare two values with each other. *Informal*, in this context, means that we shan't distinguish between the truth value **1** and the number 1, and between **0** and 0. *Comparing* values, as in $X_1 > X_2$, means *asking* whether the value of X_1 is larger than that of X_2; the answer to the question is assigned to the appropriate output variable. Let us demonstrate this, using the table for $Y_2 \Leftrightarrow X_1 > X_2$. Substituting the input events $(0, 0)$ for (X_1, X_2) lets us write $Y_2 \Leftrightarrow X_1 > X_2$ as $\mathbf{0} \Leftrightarrow 0 > 0$, or as $0 \Leftrightarrow 0 > 0$, if you prefer. Similarly we obtain $0 \Leftrightarrow 0 > 1$, $1 \Leftrightarrow 1 > 0$ and $0 \Leftrightarrow 1 > 1$ for the remaining input events of the **inhibition**.

4.2 Positive versus Negative Logic *

One of the basic concepts of this presentation is the strict distinction between logic variables $X_i, \overline{X}_i, Y_i, \overline{Y}_i$ and their numeric counterparts x_i and y_i as

expressed in

$$\left.\begin{array}{ll} X_i \overset{df}{\Leftrightarrow} x_i = 1, & \overline{X}_i \overset{df}{\Leftrightarrow} x_i = 0, \\ Y \overset{df}{\Leftrightarrow} y = 1, & \overline{Y} \overset{df}{\Leftrightarrow} y = 0. \end{array}\right\} \tag{1.2}$$

The symbol $\overset{df}{\Leftrightarrow}$ is used to state that the variable to its left is an abbreviation for the expression written to its right. But, you should note that there are two ways to define X_i — either as $x_i = 1$ or as $x_i = 0$. These two possibilities let us define a **positive logic variable** P together with an associated **negative logic variable** Q,

$$P \overset{df}{\Leftrightarrow} p = 1, \qquad Q \overset{df}{\Leftrightarrow} p = 0, \tag{4.1a,b}$$

in which $p \in \{0,1\}$ is a binary numeric variable. Note that both logic variables, P and Q, refer to the same numeric variable p. For this reason the positive logic variable P and its associated negative logic variable Q are complementary or negated variables:

$$P \Leftrightarrow \overline{Q}, \qquad Q \Leftrightarrow \overline{P}. \tag{4.2}$$

Using what we call **positive logic** implies the exclusive use of positive logic variables, while **negative logic** refers to the exclusive use of negative logic variables. It is common practice to use positive logic, not negative logic; *however, the two are never mixed.* Whenever negative logic *is* used, the fact is stated explicitly.

Although there is no pertinent reason to use positive logic, it is right that it ensures that equally named switching and logic functions have similarly looking mappings — which is obviously intended. On the other hand, it is advisable to use negative logic when employing switching devices the outputs of which react to a *drop* or *decrease* in the energy (i.e., voltage, pressure and the like) of an input signal.

4.3 Elementary Logic Functions *

While a *switching function* $f : \{0,1\}^n \mapsto \{0,1\}$ maps n-tuples of the *numeric values* 0 and 1 into the set $\{0,1\}$, a **logic function** F maps n-tuples of the *truth values* FALSITY (for which we write **0**) and TRUTH (denoted as **1**) into the set $\{\mathbf{0},\mathbf{1}\}$ of truth values. Thus, every function F of the form

$$F : \{\mathbf{0},\mathbf{1}\}^n \mapsto \{\mathbf{0},\mathbf{1}\} \tag{4.3}$$

is a logic function, whereas functions of the form

$$F : \{\mathbf{0},\mathbf{1}\}^2 \mapsto \{\mathbf{0},\mathbf{1}\} \tag{4.4}$$

$x_1=0$	$x_1=0$	$x_1=1$	$x_1=1$
$x_2=0$	$x_2=1$	$x_2=0$	$x_2=1$

(a)

$\overline{X}_1$	$\overline{X}_1$	X_1	X_1
$\overline{X}_2$	X_2	$\overline{X}_2$	X_2

(b)

$X_1 \leftrightarrow \mathbf{0}$	$X_1 \leftrightarrow \mathbf{0}$	$X_1 \leftrightarrow \mathbf{1}$	$X_1 \leftrightarrow \mathbf{1}$
$X_2 \leftrightarrow \mathbf{0}$	$X_2 \leftrightarrow \mathbf{1}$	$X_2 \leftrightarrow \mathbf{0}$	$X_2 \leftrightarrow \mathbf{1}$

(c)

Fig. 4–2 From numeric input variables (a) to logic input variables (c)

are called **elementary logic function**, i.e., functions that, at most, have two input variables.

The association between switching functions f and logic functions F is, you might think, given by virtue of positive logic variables $X_i \overset{df}{\Leftrightarrow} x_i = 1$ and $\overline{X}_i \overset{df}{\Leftrightarrow} x_i = 0$. But with these formulas alone, you would only be able to transform the table of input events of Fig. 4–2a into the table of Fig. 4–2b. To complete the analogy between switching and logic functions, we need the following logic formulas which are stated without proof.

$$A \leftrightarrow \mathbf{1} \Leftrightarrow A, \tag{4.5}$$

$$A \leftrightarrow \mathbf{0} \Leftrightarrow \overline{A}. \tag{4.6}$$

They lead directly from Fig. 4–2b to Fig. 4–2c. Now, comparing the tables of Figs. 4–2a and 4–2c, you will easily notice the important correspondence between the equally positioned symbols of the quadruples $(x_i, =, 0, 1)$ and $(X_i, \leftrightarrow, \mathbf{0}, \mathbf{1})$.

Of course the same sort of reasoning as above leads to the correspondence between $(y_i, =, 0, 1)$ and $(Y_i, \leftrightarrow, \mathbf{0}, \mathbf{1})$ allowing us to write the first column of Fig. 4–1 in the logic form shown in the first column of Fig. 4–3.

The middle column of this table presents connectives commonly used in switching algebra. The exceptions are the INHIBITIONs, NOR and NAND for which Iverson's notations have been chosen. This was done for the INHIBITIONs because there is no generally accepted notation for these functions. In the case of NOR and NAND Iverson's notation was chosen because it is mnemonically superior to the respective notations of Sheffer $(X_1 \downarrow X_2 \overset{df}{\Leftrightarrow} X_1 \tilde{\vee} X_2)$ and Nicod $(X_1|X_2 \overset{df}{\Leftrightarrow} X_1 \tilde{\wedge} X_2)$ sometimes used. In switching algebra the negation of a variable X is commonly denoted as $\overline{X}$ whereas in logic the negation would preferably be written as $\neg X$, the two notations thus being equivalent:

$$\neg X \overset{df}{\Leftrightarrow} \overline{X}. \tag{4.7}$$

Definition $X_1 \leftrightarrow$ **0 0 1 1** $X_2 \leftrightarrow$ **0 1 0 1**	Logic notation	Name of the logic function
$Y_0 \leftrightarrow$ **0 0 0 0**	$Y_0 \Leftrightarrow \mathbf{0}$	CONTRADICTION, FALSITY
$Y_1 \leftrightarrow$ **0 0 0 1**	$Y_1 \Leftrightarrow X_1 \wedge X_2$	CONJUNCTION, AND
$Y_2 \leftrightarrow$ **0 0 1 0**	$Y_2 \Leftrightarrow X_1 > X_2$	INHIBITION
$Y_3 \leftrightarrow$ **0 0 1 1**	$Y_3 \Leftrightarrow X_1$	IDENTITY
$Y_4 \leftrightarrow$ **0 1 0 0**	$Y_4 \Leftrightarrow X_1 < X_2$	*(transposed)* INHIBITION
$Y_5 \leftrightarrow$ **0 1 0 1**	$Y_5 \Leftrightarrow X_2$	IDENTITY
$Y_6 \leftrightarrow$ **0 1 1 0**	$Y_6 \Leftrightarrow X_1 \oplus X_2$	ANTIVALENCE, XOR
$Y_7 \leftrightarrow$ **0 1 1 1**	$Y_7 \Leftrightarrow X_1 \vee X_2$	DISJUNCTION, OR
$Y_8 \leftrightarrow$ **1 0 0 0**	$Y_8 \Leftrightarrow X_1 \tilde{\vee} X_2$	NOR, Sheffer function
$Y_9 \leftrightarrow$ **1 0 0 1**	$Y_9 \Leftrightarrow X_1 \leftrightarrow X_2$	(material) EQUIVALENCE
$Y_{10} \leftrightarrow$ **1 0 1 0**	$Y_{10} \Leftrightarrow \overline{X}_2$	NEGATION, NOT
$Y_{11} \leftrightarrow$ **1 0 1 1**	$Y_{11} \Leftrightarrow X_1 \leftarrow X_2$	*(transposed)* IMPLICATION
$Y_{12} \leftrightarrow$ **1 1 0 0**	$Y_{12} \Leftrightarrow \overline{X}_1$	NEGATION, NOT
$Y_{13} \leftrightarrow$ **1 1 0 1**	$Y_{13} \Leftrightarrow X_1 \rightarrow X_2$	(material) IMPLICATION
$Y_{14} \leftrightarrow$ **1 1 1 0**	$Y_{14} \Leftrightarrow X_1 \tilde{\wedge} X_2$	NAND, Nicod function
$Y_{15} \leftrightarrow$ **1 1 1 1**	$Y_{15} \Leftrightarrow \mathbf{1}$	TAUTOLOGY, TRUTH

Fig. 4–3 Table of the elementary logic functions

4.4 The Basic Gates *

Gates, unique graphic symbols for logic functions, were introduced in Section 1.5. Not every logic function has a gate assigned to it. Those gates that do exist are called **basic**, hereby emphasising that all other logic functions are represented by graphic symbols comprised of basic gates. If a gate has one input it is called **monadic**, if it is has two inputs it is called **dyadic**.

There are two monadic gates, IDENTITY and NEGATION. The symbol for the IDENTITY gate is a rectangle with a 1 inscribed (Fig. 4–4a). The symbol for the NEGATION is a *circle* drawn over a lead carrying the signal to be negated. The circle marking negation is never drawn anywhere along a lead, it is always drawn where a lead enters or leaves a rectangular gate symbol (e.g., Fig. 4–4b,c). The *inverter* of Fig. 4–4b,c is often itself called NEGATION.

As the output of IDENTITY is always logically identical with its input, there is clearly no *logical* necessity for employing a switching device that realises this function. The only validation for such a device is that all technical realisations of an IDENTITY are **amplifiers**, these being needed to boost weakened signals. Examples are the spool-and-sleeve valve of Fig. 2–3a and the normally open (NO) relay of Fig. 2–7a. You can find some examples of inverters in previous chapters. A valve realisation is discussed in Fig. 2–3b, a relay realisation is shown in Fig. 2–7b, while a CMOS NEGATION is explained in Fig. 2–14. Of necessity, the NEGATION or inverter is an amplifier.

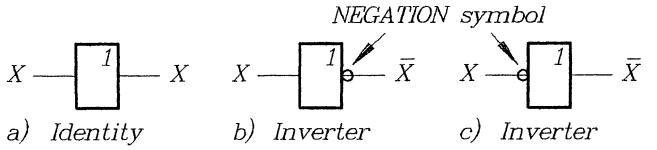


Fig. 4–4 The monadic gates

Only four dyadic gates exist, called AND, OR, XOR, and EQUIVALENCE (Fig. 4–5), each representing the equally named logic functions. But, why are gates only defined for these four functions? For a gate to be of practical use, it must represent a function that is **associative** and **commutative**, and only the above mentioned functions have both these properties. We shall look into this later. As for the rather curious choice of inscriptions (&, ≥1, =1, =) distinguishing the gates, I'm afraid you'll just have to accept them — they are international standard as declared by the International Electrotechnical Commission (IEC 117-15).

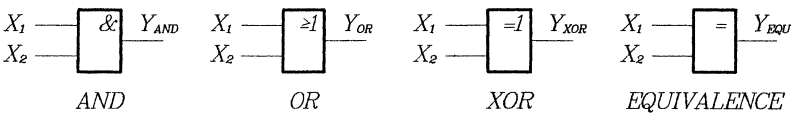
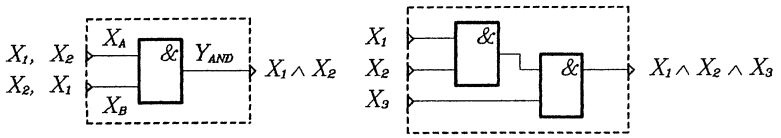


Fig. 4–5 The dyadic gates

Commutativity allows you to interchange the input signals to a gate as you like without being able to notice a change in the behaviour of the output value. For instance, assigning (X_1, X_2) to (X_A, X_B) in Fig. 4–6a will give the same result as assigning (X_2, X_1) to (X_A, X_B).

Associativity permits you to *cascade* identical gate types thus creating a multiple inputs gate of that type. For instance, cascading two dual input AND gates, as shown in Fig. 4–6b, leads to a triple input AND gate. Its inputs are again commutative and associative.



a) Commutativity *b) Associativity*

Fig. 4–6 On commutativity and associativity

Due to the properties of commutativity and associativity, the above four functions, with the outputs

$$Y_{AND} \Leftrightarrow X_1 \wedge X_2 \wedge \cdots \wedge X_n, \tag{4.8}$$

$$Y_{OR} \Leftrightarrow X_1 \vee X_2 \vee \cdots \vee X_n, \tag{4.9}$$

$$Y_{XOR} \Leftrightarrow X_1 \oplus X_2 \oplus \cdots \oplus X_n, \tag{4.10}$$

$$Y_{EQU} \Leftrightarrow X_1 \leftrightarrow X_2 \leftrightarrow \cdots \leftrightarrow X_n, \tag{4.11}$$

can be assigned gates with multiple inputs (see Fig. 4–7). An overview of when these functions are **1** is given in the following statements.

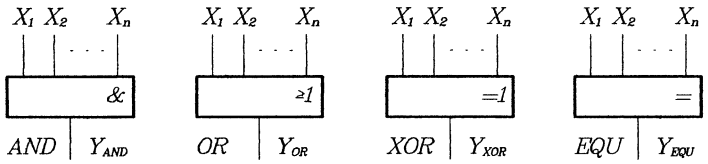


Fig. 4–7 Gates with multiple inputs

AND: Y_{AND} is **1** *iff* all inputs are **1**.
OR: Y_{OR} is **1** *iff* at least one input is **1**.
XOR: Y_{XOR} is **1** *iff* an odd number of the inputs are **1**.
EQU: For an **even** number of inputs (taking 0 as even):
Y_{EQU} is **1** *iff* an even number of the inputs are **1**.
For an **odd** number of inputs:
Y_{EQU} is **1** *iff* an odd number of the inputs are **1**.

These statements are illustrated in the tables of Fig. 4–8 for 3 and for 4 input variables. You will possibly want to compare the rows for Y_{XOR} and Y_{EQU} more carefully. From the above statements, saying when XOR and EQUIVALENCE are **1**, we deduce that their outputs are inverted for functions with

an even number of input variables, and are logically equivalent for functions with an odd number of input variables:

$$\text{For an } \textbf{even} \text{ number of inputs:} \quad Y_{XOR} \Leftrightarrow \overline{Y}_{EQU} \tag{14.12a}$$

$$\text{For an } \textbf{odd} \text{ number of inputs:} \quad Y_{XOR} \Leftrightarrow Y_{EQU} \tag{14.12b}$$

This rather inconsistent behaviour of Y_{EQU} usually lets us work with the XOR function instead of the EQUIVALENCE function.

$X_1 \leftrightarrow$	0	0	0	0	1	1	1	1							
$X_2 \leftrightarrow$	0	0	1	1	0	0	1	1							
$X_3 \leftrightarrow$	0	1	0	1	0	1	0	1							
$Y_{AND} \leftrightarrow$	0	0	0	0	0	0	0	1							
$Y_{OR} \leftrightarrow$	0	1	1	1	1	1	1	1							
$Y_{XOR} \leftrightarrow$	0	1	1	0	1	0	0	1							
$Y_{EQU} \leftrightarrow$	0	1	1	0	1	0	0	1							

$X_1 \leftrightarrow$	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
$X_2 \leftrightarrow$	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
$X_3 \leftrightarrow$	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
$X_4 \leftrightarrow$	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
$Y_{AND} \leftrightarrow$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
$Y_{OR} \leftrightarrow$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$Y_{XOR} \leftrightarrow$	0	1	1	0	1	0	0	1	1	0	0	1	0	1	1	0
$Y_{EQU} \leftrightarrow$	1	0	0	1	0	1	1	0	0	1	1	0	1	0	0	1

Fig. 4–8 AND, OR, XOR and EQUIVALENCE for 3 and 4 inputs

4.5 Derived Gates

The fact that the elementary logic functions INHIBITION, IMPLICATION, NOR, and NAND don't have their own characteristic gates makes it necessary to express these functions as combinations of other gates, e.g., of AND, OR , and NOT (although other combinations of gates are possible, and not uncommon). To facilitate this we need (4.13) which we now develop. Substituting $\overline{A}$ for A in $A \leftrightarrow \mathbf{1} \Leftrightarrow A$ (4.5) we obtain $\overline{A} \leftrightarrow \mathbf{1} \Leftrightarrow \overline{A}$ allowing us to replace $\overline{A}$ of $A \leftrightarrow \mathbf{0} \Leftrightarrow \overline{A}$ (4.6) by $\overline{A} \leftrightarrow \mathbf{1}$ (and also to write the formula for $\overline{A}$):

$$A \leftrightarrow \mathbf{0} \Leftrightarrow \overline{A} \leftrightarrow \mathbf{1}, \qquad A \leftrightarrow \mathbf{1} \Leftrightarrow \overline{A} \leftrightarrow \mathbf{0}. \tag{4.13}$$

We start by expressing the INHIBITION as an AND function — with the use of (4.13) — as shown in Fig. 4–9.

$X_1 \leftrightarrow$	0	0	1	1					$X_1 \leftrightarrow$	0	0	1	1		
$X_2 \leftrightarrow$	0	1	0	1	$\cdots$ (4.13) $\cdots$				$\overline{X}_2 \leftrightarrow$	1	0	1	0		
$Y_{INH} \leftrightarrow$	0	0	1	0					$Y_{INH} \leftrightarrow$	0	0	1	0		
a)									b)						

Fig. 4–9 Interpreting the INHIBITION via AND

The INHIBITION of Fig. 4–9a is taken from the table of Fig. 4–3. To be able to express the single output value of **1** via AND, both input variables of the third column would have to be **1**. This is easy to achieve by applying (4.13) to X_2, the result being shown in Fig. 4–9b. As the output of an AND function is **1** *iff* all inputs are true, we can now write:

$$Y_{INH} \Leftrightarrow X_1 > X_2 \Leftrightarrow X_1 \wedge \overline{X}_2. \tag{4.14}$$

$X_1 \leftrightarrow$	**0**	**0**	**1**	**1**	$\cdots$ (4.13) $\cdots$	$\overline{X}_1 \leftrightarrow$	**1**	**1**	**0**	**0**
$X_2 \leftrightarrow$	**0**	**1**	**0**	**1**		$X_2 \leftrightarrow$	**0**	**1**	**0**	**1**
$Y_{IMP} \leftrightarrow$	**1**	**1**	**0**	**1**		$Y_{IMP} \leftrightarrow$	**1**	**1**	**0**	**1**
a)						b)				

Fig. 4–10 Interpreting the IMPLICATION via OR

The three **1**-values of the output variable of Fig. 4–10a of the IMPLICATION cannot be described by an OR function because the OR function requires all output **1**s (and only these) to be caused by at least one input **1**. This is achieved by applying (4.13) to X_1, thus obtaining the table of Fig. 4–10b. For this table we can write the output as an OR function:

$$Y_{IMP} \Leftrightarrow X_1 \rightarrow X_2 \Leftrightarrow \overline{X}_1 \vee X_2. \tag{4.15}$$

If you wish to employ the same method to the NOR and NAND functions follow the reasoning laid down in the table of Fig. 4–11b allowing us to express these two functions as

$$Y_{NOR} \Leftrightarrow \overline{X_1 \vee X_2}, \qquad Y_{NAND} \Leftrightarrow \overline{X_1 \wedge X_2}. \tag{4.16}$$

$X_1 \leftrightarrow$	**0**	**0**	**1**	**1**		$X_1 \leftrightarrow$	**0**	**0**	**1**	**1**
$X_2 \leftrightarrow$	**0**	**1**	**0**	**1**		$X_2 \leftrightarrow$	**0**	**1**	**0**	**1**
$Y_{NOR} \leftrightarrow$	**1**	**0**	**0**	**0**	$\cdots$ (4.13) $\cdots$	$\overline{Y}_{NOR} \leftrightarrow$	**0**	**1**	**1**	**1**
$Y_{NAND} \leftrightarrow$	**1**	**1**	**1**	**0**	$\cdots$ (4.13) $\cdots$	$\overline{Y}_{NAND} \leftrightarrow$	**0**	**0**	**0**	**1**
a)						b)				

Fig. 4–11 Interpreting NOR and NAND as NOT-OR and NOT-AND

The functions INHIBITION, IMPLICATION, NOR, and NAND can be represented by AND, OR and NOT according to the equations (4.14), (4.15), and (4.16). These formulas, on their part, lead to the gate representation of Fig. 4-12.

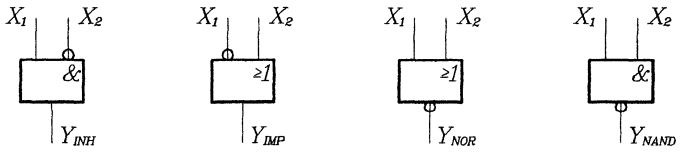


Fig. 4–12 INHIBITION, IMPLICATION, NOR, and NAND

Chapter 5

Synthesis and Duality

As shown in Section 1.4, a *logic formula* can be developed from a *table of asserted events* (TAE) that states exactly when the value of the output variable y of a circuit is 1, and when it is 0. In general, the logic formulas thus developed consist of constituent terms called **minterms** and **maxterms**. These are basic to switching algebra because they are used in the synthesis of all kinds of circuits, combinational and sequential. In accordance with their importance they are introduced in a general and quite formal manner contrasting with the usual perfunctory way of defining them which is inadequate for use in formal proofs (such as in Chapter 11).

Here, minterms and maxterms are used in an *intuitive method* to obtain logic formulas characteristic of a specified circuit. A formalisation of this intuitive approach will have to wait till we have accumulated greater knowledge of formal logic (Chapter 14). The intuitive method of designing circuits is used to develop two alternative formulas satisfying one and the same specification. A certain symmetry between these formulas refers to a property we call **duality**, this being discussed in the last section.

5.1 Minterm Functions & Minterms *

When originally introduced, *minterms* and *maxterms* were the names given the conjunctions $\overline{X}_1 \wedge \overline{X}_2$, $\overline{X}_1 \wedge X_2$, $X_1 \wedge \overline{X}_2$, $X_1 \wedge X_2$, and disjunctions $X_1 \vee X_2$, $X_1 \vee \overline{X}_2$, $\overline{X}_1 \vee X_2$, $\overline{X}_1 \vee \overline{X}_2$, respectively. Their importance stemmed from the observation that it was always possible to write the logical formula of a combinational switching circuit by either exclusively using minterms, or exclusively using maxterms.

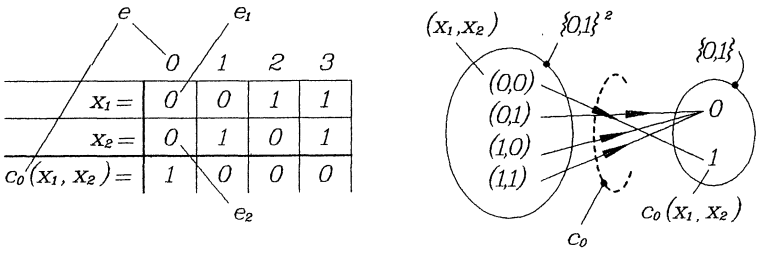
To introduce minterms quite generally, we need the concept of a *minterm function*. A minterm function, denoted as c_e, is a switching function. It is characteristic for the TAE defining the minterm function that the output value of only a single column is 1; the decimal equivalent of the input event of this column is the index of c_e. Fig. 5–1 gives an example of a minterm function.

More formally, and going directly to n inputs, for which we use vector notation as in

$$\mathbf{x} \stackrel{df}{=} (x_1, x_2, \ldots, x_n), \qquad \mathbf{e} \stackrel{df}{=} (e_1, e_2, \ldots, e_n), \tag{3.12}$$

we define a **minterm function** $c_e : \{0,1\}^n \mapsto \{0,1\}$ *as a switching function whose output* $c_e(\mathbf{x})$ *is* 1 iff *the measurable input event* $\mathbf{x}$ *equals the asserted input event* $\mathbf{e}$ *this being the binary equivalent of the index* e*, i.e.*

$$c_e(\mathbf{x}) = 1 \stackrel{df}{\Leftrightarrow} \mathbf{x} = \mathbf{e} \quad \text{with} \quad e = \sum_{i=1}^{n} e_i.2^{n-i} \quad \text{and} \quad e_i \in \{0,1\}. \tag{5.1}$$



a) Table of events *b) Mapping*

Fig. 5–1 Defining the minterm function c_0

The **minterm** $C_e(\mathbf{x})$ is simply an abbreviation for the assertion that the output variable $c_e(\mathbf{x})$ is 1

$$C_e(\mathbf{x}) \stackrel{df}{\Leftrightarrow} c_e(\mathbf{x}) = 1. \tag{5.2}$$

Inserting this abbreviation $C_e(\mathbf{x})$ into the definition (5.1) of the minterm function c_e

$$C_e(\mathbf{x}) \Leftrightarrow \mathbf{x} = \mathbf{e} \quad \text{with} \quad e = \sum_{i=1}^{n} e_i.2^{n-i} \quad \text{and} \quad e_i \in \{0,1\} \tag{5.3}$$

allows us to interpret a minterm *as the assertion that the input event* $\mathbf{x}$ *measured is (or equals) the binary equivalent* $\mathbf{e}$ *of the index* e. Whenever this is true, we can substitute $\mathbf{e}$ for $\mathbf{x}$ in the above formula, showing that for this case (and only this case) the minterm is true

$$C_e(\mathbf{e}) \Leftrightarrow \mathbf{e} = \mathbf{e} \Leftrightarrow \mathbf{1}.$$

Remembering that, according to (3.5), two ordered n-tuples are equal *iff* equally positioned elements are equal

$$\mathbf{x} = \mathbf{e} \Leftrightarrow (x_1 = e_1) \wedge (x_2 = e_2) \wedge \ldots \wedge (x_n = e_n) \Leftrightarrow \bigwedge_{i=1}^{n} (x_i = e_i)$$

enables us to write (5.3) as

$$C_e(\mathbf{x}) \Leftrightarrow \bigwedge_{i=1}^{n} (x_i = e_i) \quad \text{with} \quad e = \sum_{i=1}^{n} e_i.2^{n-i} \quad \text{and} \quad e_i \in \{0,1\}. \qquad (5.4)$$

This **general minterm formula** is the principle result on minterms. A simple example of its use is shown in Fig. 5–2 where all minterms $C_e(x_1, x_2)$ of two inputs are calculated. Note the convention of dropping the $\wedge$-connective (writing AB instead of $A \wedge B$) when desired.

e	0	1	2	3
x_1=	0	0	1	1
x_2=	0	1	0	1
$c_0(x_1,x_2)$=	1	0	0	0
$c_1(x_1,x_2)$=	0	1	0	0
$c_2(x_1,x_2)$=	0	0	1	0
$c_3(x_1,x_2)$=	0	0	0	1

$$\ldots C_0(x_1,x_2) \Leftrightarrow (x_1 = 0) \wedge (x_2 = 0) \Leftrightarrow \overline{X}_1\overline{X}_2$$
$$\ldots C_1(x_1,x_2) \Leftrightarrow (x_1 = 0) \wedge (x_2 = 1) \Leftrightarrow \overline{X}_1 X_2$$
$$\ldots C_2(x_1,x_2) \Leftrightarrow (x_1 = 1) \wedge (x_2 = 0) \Leftrightarrow X_1\overline{X}_2$$
$$\ldots C_3(x_1,x_2) \Leftrightarrow (x_1 = 1) \wedge (x_2 = 1) \Leftrightarrow X_1 X_2$$

Fig. 5–2 Reading minterms from the table of events

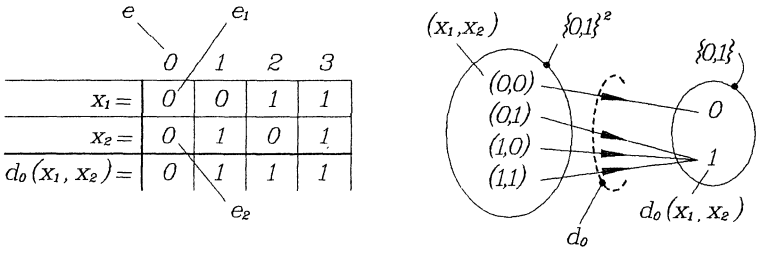
To calculate, say, $C_{26}(x_1, \ldots, x_6)$ you first employ the algorithm given at the end of Section 3.1 to obtain $(26)_{10} = (011010)_2$. These binary digits are the values of $(e_1, \ldots, e_6)$ allowing us to write

$$C_{26}(x_1, \ldots, x_6) \Leftrightarrow$$
$$(x_1 = 0) \wedge (x_2 = 1) \wedge (x_3 = 1) \wedge (x_4 = 0) \wedge (x_5 = 1) \wedge (x_6 = 0) \Leftrightarrow$$
$$\overline{X}_1 X_2 X_3 \overline{X}_4 X_5 \overline{X}_6.$$

The **mnemonic result** is: For a given column e of the *table of asserted input events* the minterm $C_e(\mathbf{x})$ is the conjunction of all logic variables, inverted or not-inverted; they are inverted (i.e., $\overline{X}_i$) when their input variables (x_i) are asserted to be *zero* ($x_i = 0$), and not-inverted (X_i) when their input variables are asserted to be *one* ($x_i = 1$). *This mnemonic result provides the standard procedure for writing minterms.*

5.2 Maxterm Functions & Maxterms*

The maxterm function d_0 of Fig. 5–3 is clearly the complementary function to c_0, shown in Fig. 5–1, as their output values $d_0(x_1, x_2)$ and $c_0(x_1, x_2)$ are complementary for every given input event (x_1, x_2).



a) Table of events *b) Mapping*

Fig. 5–3 Defining the maxterm function d_0

Taking Figs. 5–3 as a lead, a **maxterm function** $d_e : \{0,1\}^n \mapsto \{0,1\}$ is defined *as a switching function whose output* $d_e(\mathbf{x})$ *is* 0 iff *the measurable input event* $\mathbf{x}$ *equals the asserted input event* $\mathbf{e}$ *this being the binary equivalent of the index* e, *i.e.*

$$d_e(\mathbf{x}) = 0 \overset{df}{\Leftrightarrow} \mathbf{x} = \mathbf{e} \qquad \text{with} \qquad e = \sum_{i=1}^{n} e_i . 2^{n-i}. \tag{5.5}$$

In contrast to the way the maxterm function is defined, i.e., by stating when its output $d_e(\mathbf{x})$ is *zero*, the **maxterm** $D_e(\mathbf{x})$ is an abbreviation for the output $d_e(\mathbf{x})$ being *one* (due to the use of positive logic); the output $d_e(\mathbf{x})$ being zero is the complement $\overline{D}_e(\mathbf{x})$ of the maxterm

$$D_e(\mathbf{x}) \overset{df}{\Leftrightarrow} d_e(\mathbf{x}) = 1, \qquad \overline{D}_e(\mathbf{x}) \Leftrightarrow d_e(\mathbf{x}) = 0. \tag{5.6}$$

With the latter of these two formulas, (5.5) can be written as

$$\overline{D}_e(\mathbf{x}) \Leftrightarrow \mathbf{x} = \mathbf{e}$$

which, when inverting each side individually, becomes

$$D_e(\mathbf{x}) \Leftrightarrow \mathbf{x} \neq \mathbf{e}. \tag{5.7}$$

As ordered pairs are equal when all equally positioned elements are equal, they — i.e., the ordered pairs — are unequal when at least one pair of equally positioned elements is unequal:

$$\mathbf{x} \neq \mathbf{e} \Leftrightarrow (x_1 \neq e_1) \vee (x_2 \neq e_2) \vee \ldots \vee (x_n \neq e_n) \Leftrightarrow \bigvee_{i=1}^{n} (x_i \neq e_i). \tag{5.8}$$

Taking into account that all variables stand for either zero or one — i.e., $x_i, e_i \in \{0,1\}$ — it is easy to verify that

$$x_i \neq e_i \Leftrightarrow x_i = (1 - e_i). \tag{5.9}$$

Using the two previous formulas, allows us to rewrite (5.7) as

$$D_e(\mathbf{x}) \Leftrightarrow \bigvee_{i=1}^{n} (x_i = 1 - e_i) \qquad \text{with} \qquad e = \sum_{i=1}^{n} e_i.2^{n-i}. \tag{5.10}$$

This **general maxterm formula** is the principle result on maxterms. A simple example of its use is shown in Fig. 5–4 where all maxterms $D_e(x_1, x_2)$ of two inputs are calculated.

e	0	1	2	3	
x_1=	0	0	1	1	
x_2=	0	1	0	1	
$d_0(x_1,x_2)$=	0	1	1	1	$\ldots D_0(x_1,x_2) \Leftrightarrow (x_1 = 1) \vee (x_2 = 1) \Leftrightarrow X_1 \vee X_2$
$d_1(x_1,x_2)$=	1	0	1	1	$\ldots D_1(x_1,x_2) \Leftrightarrow (x_1 = 1) \vee (x_2 = 0) \Leftrightarrow X_1 \vee \overline{X}_2$
$d_2(x_1,x_2)$=	1	1	0	1	$\ldots D_2(x_1,x_2) \Leftrightarrow (x_1 = 0) \vee (x_2 = 1) \Leftrightarrow \overline{X}_1 \vee X_2$
$d_3(x_1,x_2)$=	1	1	1	0	$\ldots D_3(x_1,x_2) \Leftrightarrow (x_1 = 0) \vee (x_2 = 0) \Leftrightarrow \overline{X}_1 \vee \overline{X}_2$

Fig. 5–4 Reading maxterms from the table of events

The **mnemonic result** is: For a given column e of the *table of asserted input events* the maxterm $D_e(\mathbf{x})$ is the disjunction of all logic variables, inverted or not-inverted; they are inverted (i.e., $\overline{X}_i$) when their input variables (x_i) are asserted to be *one* ($x_i = 1$), and not-inverted (X_i) when their input variables are asserted to be *zero* ($x_i = 0$). *This mnemonic result provides the standard procedure for writing maxterms.*

5.3 Synthesis via Partial Outputs*

This section presents an *intuitive* approach to developing logic formulas of switching functions. It could be called the **method of partial outputs**, as the output variables $c_e(x_1, x_2)$ or $d_e(x_1, x_2)$ of either minterm or maxterm functions play the role of partial or interim outputs that can be 'combined' (here, intentionally being vague, for the moment) to create the actual output.

Using minterms as partial outputs

In the first example, we develop a logic formula for the **xor**. This function is extracted from the table of Fig. 4–1 and repeated in the top four rows (for e, x_1, x_2 and y_6) of Fig. 5–5.

For each column e in which the output variable y_6 is 1, we introduce an output variable $c_e(x_1, x_2)$ of a minterm function c_e, each variable being called a partial output, as it coincides with only part of the desired result for y_6. Next, to obtain y_6 from the partial outputs $c_e(x_1, x_2)$ — to 'combine' these partial outputs — we argue that

$$(y_6 = 1) \Leftrightarrow \big(c_1(x_1, x_2) = 1\big) \text{ OR } \big(c_2(x_1, x_2) = 1\big),$$

from which we obtain the logical formula for the **xor**:

$$Y_6 \Leftrightarrow C_1(x_1, x_2) \vee C_2(x_1, x_2) \Leftrightarrow \overline{X}_1 X_2 \vee X_1 \overline{X}_2. \tag{5.11}$$

e	0	1	2	3	
x_1=	0	0	1	1	
x_2=	0	1	0	1	
y_6=	0	1	1	0	
$c_1(x_1,x_2)$=	0	1	0	0	$\ldots C_1(x_1,x_2) \Leftrightarrow \overline{X}_1 X_2$
$c_2(x_1,x_2)$=	0	0	1	0	$\ldots C_2(x_1,x_2) \Leftrightarrow X_1 \overline{X}_2$
y_6=	0	1	1	0	$\ldots Y_6(x_1,x_2) \Leftrightarrow C_1 \vee C_2 \Leftrightarrow \overline{X}_1 X_2 \vee X_1 \overline{X}_2$

Fig. 5–5 Evaluating the **xor** by minterms

Formulas of this kind are called **disjunctions of minterms** — (DMin) formulas — or **disjunctive normal forms** (DNFs). The expression *disjunctive* is used to point out that the output Y is the result of a disjunction (in this case, the disjunction of minterms). A *normal form* refers to a standardised method that always provides the same unique result for a given application.

The **procedure to obtain a DNF** might, informally, be summarised as follows: *The logical output variable Y of a DNF is written as the disjunction of those minterms C_e for which the switching function is 1.*

A second example, for the **equivalence** y_9, is given in Fig. 5–6.

e	0	1	2	3	
x_1=	0	0	1	1	
x_2=	0	1	0	1	
y_9=	1	0	0	1	
$c_0(x_1,x_2)$=	1	0	0	0	$\ldots C_0(x_1,x_2) \Leftrightarrow \overline{X}_1 \overline{X}_2$
$c_3(x_1,x_2)$=	0	0	0	1	$\ldots C_3(x_1,x_2) \Leftrightarrow X_1 X_2$
y_9=	1	0	0	1	$\ldots Y_9(x_1,x_2) \Leftrightarrow C_0 \vee C_3 \Leftrightarrow \overline{X}_1 \overline{X}_2 \vee X_1 X_2$

Fig. 5–6 Evaluating the **equivalence** by minterms

The logical formula obtained for the **equivalence**, Fig. 5–6, is

$$Y_9 \Leftrightarrow C_0(x_1, x_2) \vee C_3(x_1, x_2) \Leftrightarrow \overline{X}_1 \overline{X}_2 \vee X_1 X_2. \tag{5.12}$$

The logic circuits for (5.11) and (5.12) are shown in Fig. 5–7, the intention being to point out the partial outputs.

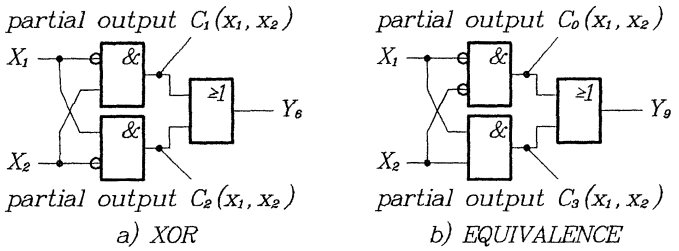


Fig. 5–7 Disjunctive logic circuits for XOR and EQUIVALENCE

Note that circuits are frequently named according to their output gate. The output gates in Fig. 5–7 being disjunctions, the circuits are referred to as disjunctive circuits.

Using maxterms as partial outputs

Using the same two functions as above, the **xor** and **equivalence**, and starting with the former, we now develop logic formulas based on maxterms. For each column e in Fig. 5–8 in which the output variable y_6 is *zero* (0), we introduce an output variable $d_e(x_1, x_2)$ of a maxterm function d_e, each variable again being called a partial output.

e	0	1	2	3	
x_1=	0	0	1	1	
x_2=	0	1	0	1	
y_6=	0	1	1	0	
$d_0(x_1,x_2)$=	0	1	1	1	$\ldots D_0(x_1,x_2) \Leftrightarrow (X_1 \vee X_2)$
$d_3(x_1,x_2)$=	1	1	1	0	$\ldots D_3(x_1,x_2) \Leftrightarrow (\overline{X}_1 \vee \overline{X}_2)$
y_6=	0	1	1	0	$\ldots Y_6(x_1,x_2) \Leftrightarrow D_0 \wedge D_3 \Leftrightarrow (X_1 \vee X_2)\,(\overline{X}_1 \vee \overline{X}_2)$

Fig. 5–8 Evaluating the **xor** by maxterms

Next, to obtain y_6 from the partial outputs $d_e(x_1, x_2)$, we argue that

$$(y_6 = 1) \Leftrightarrow \big(d_0(x_1, x_2) = 1\big) \text{ AND } \big(d_3(x_1, x_2) = 1\big),$$

from which we obtain the logical formula for the **xor**:

$$Y_6 \Leftrightarrow D_0(x_1, x_2) \wedge D_3(x_1, x_2) \Leftrightarrow (X_1 \vee X_2)\,(\overline{X}_1 \vee \overline{X}_2). \tag{5.13}$$

Proceeding similarly for the **equivalence** leads to Fig. 5–9.

e	0	1	2	3	
x_1=	0	0	1	1	
x_2=	0	1	0	1	
y_9=	1	0	0	1	
$d_1(x_1,x_2)$=	1	0	1	1	$\ldots D_1(x_1,x_2) \Leftrightarrow (X_1 \vee \overline{X}_2)$
$d_2(x_1,x_2)$=	1	1	0	1	$\ldots D_2(x_1,x_2) \Leftrightarrow (\overline{X}_1 \vee X_2)$
y_9=	1	0	0	1	$\ldots Y_9(x_1,x_2) \Leftrightarrow D_1 \wedge D_2 \Leftrightarrow (X_1 \vee \overline{X}_2)\,(\overline{X}_1 \vee X_2)$

Fig. 5–9 Evaluating the **equivalence** by maxterms

The logical formula obtained for the **equivalence** (as shown in Fig. 5–9) is

$$Y_9(x_1,x_2) \Leftrightarrow D_1(x_1,x_2) \wedge D_2(x_1,x_2) \Leftrightarrow (X_1 \vee \overline{X}_2)\,(\overline{X}_1 \vee X_2) \qquad (5.14)$$

whereas the logic circuits for (5.13) and (5.14) are shown in Fig. 5–10.

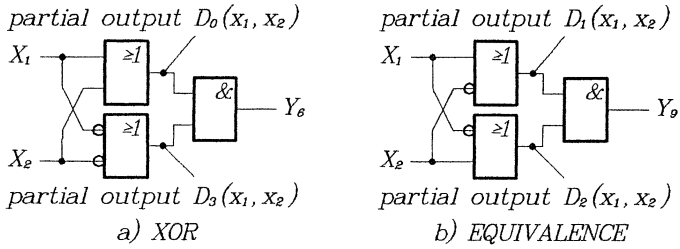


Fig. 5–10 Conjunctive logic circuits for XOR and EQUIVALENCE

The output gates in Fig. 5–10 being conjunctions, the circuits are referred to as conjunctive circuits. **The procedure to obtain a CNF — conjunctive normal form** — might, informally, be summarised thus: *The logical output variable Y of a CNF is written as the conjunction of those maxterms D_e for which the switching function is 0.*

5.4 Duality *

In Chapter 4 we used *positive* logic for the input and output variables, specifically, $X_i \Leftrightarrow x_i = 1$, $\overline{X}_i \Leftrightarrow x_i = 0$, $Y_i \Leftrightarrow y_i = 1$, $\overline{Y}_i \Leftrightarrow y_i = 0$, together with the formulas $X_i \Leftrightarrow X_i \leftrightarrow \mathbf{1}$, $\overline{X}_i \Leftrightarrow X_i \leftrightarrow \mathbf{0}$ and $Y_i \Leftrightarrow Y_i \leftrightarrow \mathbf{1}$, $\overline{Y}_i \Leftrightarrow Y_i \leftrightarrow \mathbf{0}$, (4.5) and (4.6), to transform the elementary *switching* functions of Fig. 4–1 into the elementary *logic* functions of Fig. 4–3. These logic functions

are repeated in Fig. 5–11 in the column marked 'Positive logic'. On the other hand, if you use *negative* logic — $X_i \Leftrightarrow x_i = 0$, $\overline{X}_i \Leftrightarrow x_i = 1$, $Y_i \Leftrightarrow y_i = 0$, $\overline{Y}_i \Leftrightarrow y_i = 1$ — to transform the switching functions of Fig. 4–1 into logic functions, you obtain the functions in the right-hand column, marked 'Negative logic', Fig. 5–11. Note that the logic formulas $X_i \Leftrightarrow X_i \leftrightarrow \mathbf{1}$, $\overline{X}_i \Leftrightarrow X_i \leftrightarrow \mathbf{0}$ and $Y_i \Leftrightarrow Y_i \leftrightarrow \mathbf{1}$, $\overline{Y}_i \Leftrightarrow Y_i \leftrightarrow \mathbf{0}$, to be used with negative logic remain valid, as it is irrelevant what interpretation the logic variables are given.

You will notice that *all* **0**s and **1**s in the right-hand column are the negation of equally positioned **0**s and **1**s in the left-hand column. The input events (X_1, X_2) in the negative-logic column, read from right to left, are identical with the input events (X_1, X_2) in the positive-logic column, read from left to right. Therefore, to find the name of a given function in the negative-logic column, you read the sequence of **0**s and **1**s of the output variable, and search for the same sequence in reverse order in a row of the positive-logic column — the function thus found is identical with the given negative-logic function, giving this function its name, and index number, and allowing you to use the symbolic notation (i.e., logic formula). For instance, in the third row of the negative-logic column, the output sequence is **1 1 0 1**. The reverse sequence, **1 0 1 1**, is found in the fifth row from the bottom in the positive-logic column, where the function name is *transposed* IMPLICATION and its index number is 11. Thus this is also the name and the index number of the function in the third row of the negative-logic column.

The elementary logic-functions written in one and the same row of Fig. 5–11 are said to be *dual*. Following this lead, we can define duality generally, i.e., for logic functions of n input variables

(5.15) Duality of logic functions. Two logic functions $F : \{\mathbf{0},\mathbf{1}\}^n \mapsto \{\mathbf{0},\mathbf{1}\}$ and $\widetilde{F} : \{\mathbf{0},\mathbf{1}\}^n \mapsto \{\mathbf{0},\mathbf{1}\}$ are called **dual** functions if both are logic descriptions of the same switching function $f : \{0,1\}^n \mapsto \{0,1\}$, one describing the switching function f in *positive* logic, the other in *negative* logic.

As AND and OR are dual logic functions, the above definition allows you to logically describe two relays in *series* by using the OR function instead of the AND function. The more conventional way to explain duality is to incorporate the concept of negative logic into the explanation, thus avoiding the reference to a common switching function.

(5.16) Duality of logic functions. Two logic functions $F : \{\mathbf{0},\mathbf{1}\}^n \mapsto \{\mathbf{0},\mathbf{1}\}$ and $\widetilde{F} : \{\mathbf{0},\mathbf{1}\}^n \mapsto \{\mathbf{0},\mathbf{1}\}$ are **dual** if each output $\widetilde{F}(X_1,\ldots,X_n)$ of $\widetilde{F}$, initiated by $(X_1,\ldots,X_n)$, is the *complement* of the output $F(\overline{X}_1,\ldots,\overline{X}_n)$ of F, initiated by the input event $(\overline{X}_1,\ldots,\overline{X}_n)$ of negated or inverted inputs:

$$\widetilde{F}(X_1,\ldots,X_n) \Leftrightarrow \overline{F(\overline{X}_1,\ldots,\overline{X}_n)}.$$

Positive logic		**Negative logic**	
Logic	$X_1 \leftrightarrow 0\ 0\ 1\ 1$	$X_1 \leftrightarrow 1\ 1\ 0\ 0$	Logic
formula	$X_2 \leftrightarrow 0\ 1\ 0\ 1$	$X_2 \leftrightarrow 1\ 0\ 1\ 0$	formula
FALSITY		TRUTH	
$Y_0 \Leftrightarrow 0$	$Y_0 \leftrightarrow 0\ 0\ 0\ 0$	$Y_{15} \leftrightarrow 1\ 1\ 1\ 1$	$Y_{15} \Leftrightarrow 1$
AND		OR	
$Y_1 \Leftrightarrow X_1 \wedge X_2$	$Y_1 \leftrightarrow 0\ 0\ 0\ 1$	$Y_7 \leftrightarrow 1\ 1\ 1\ 0$	$Y_7 \Leftrightarrow X_1 \vee X_2$
INHIBITION		*transposed* IMPLICATION	
$Y_2 \Leftrightarrow X_1 > X_2$	$Y_2 \leftrightarrow 0\ 0\ 1\ 0$	$Y_{11} \leftrightarrow 1\ 1\ 0\ 1$	$Y_{11} \Leftrightarrow X_1 \leftarrow X_2$
IDENTITY		IDENTITY	
$Y_3 \Leftrightarrow X_1$	$Y_3 \leftrightarrow 0\ 0\ 1\ 1$	$Y_3 \leftrightarrow 1\ 1\ 0\ 0$	$Y_3 \Leftrightarrow X_1$
transposed INHIBITION		IMPLICATION	
$Y_4 \Leftrightarrow X_1 < Y_2$	$Y_4 \leftrightarrow 0\ 1\ 0\ 0$	$Y_{13} \leftrightarrow 1\ 0\ 1\ 1$	$Y_{13} \Leftrightarrow X_1 \rightarrow X_2$
IDENTITY		IDENTITY	
$Y_5 \Leftrightarrow X_2$	$Y_5 \leftrightarrow 0\ 1\ 0\ 1$	$Y_5 \leftrightarrow 1\ 0\ 1\ 0$	$Y_5 \Leftrightarrow X_2$
XOR		EQUIVALENCE	
$Y_6 \Leftrightarrow X_1 \oplus X_2$	$Y_6 \leftrightarrow 0\ 1\ 1\ 0$	$Y_9 \leftrightarrow 1\ 0\ 0\ 1$	$Y_9 \Leftrightarrow X_1 \leftrightarrow X_2$
OR		AND	
$Y_7 \Leftrightarrow X_1 \vee X_2$	$Y_7 \leftrightarrow 0\ 1\ 1\ 1$	$Y_1 \leftrightarrow 1\ 0\ 0\ 0$	$Y_1 \Leftrightarrow X_1 \wedge X_2$
NOR		NAND	
$Y_8 \Leftrightarrow X_1 \tilde{\vee} X_2$	$Y_8 \leftrightarrow 1\ 0\ 0\ 0$	$Y_{14} \leftrightarrow 0\ 1\ 1\ 1$	$Y_{14} \Leftrightarrow X_1 \tilde{\wedge} X_2$
EQUIVALENCE		XOR	
$Y_9 \Leftrightarrow X_1 \leftrightarrow X_2$	$Y_9 \leftrightarrow 1\ 0\ 0\ 1$	$Y_6 \leftrightarrow 0\ 1\ 1\ 0$	$Y_6 \Leftrightarrow X_1 \oplus X_2$
NEGATION		NEGATION	
$Y_{10} \Leftrightarrow \overline{X}_2$	$Y_{10} \leftrightarrow 1\ 0\ 1\ 0$	$Y_{10} \leftrightarrow 0\ 1\ 0\ 1$	$Y_{10} \Leftrightarrow \overline{X}_2$
transposed IMPLICATION		INHIBITION	
$Y_{11} \Leftrightarrow X_1 \leftarrow X_2$	$Y_{11} \leftrightarrow 1\ 0\ 1\ 1$	$Y_2 \leftrightarrow 0\ 1\ 0\ 0$	$Y_2 \Leftrightarrow X_1 > X_2$
NEGATION		NEGATION	
$Y_{12} \Leftrightarrow \overline{X}_1$	$Y_{12} \leftrightarrow 1\ 1\ 0\ 0$	$Y_{12} \leftrightarrow 0\ 0\ 1\ 1$	$Y_{12} \Leftrightarrow \overline{X}_1$
IMPLICATION		*transposed* INHIBITION	
$Y_{13} \Leftrightarrow X_1 \rightarrow X_2$	$Y_{13} \leftrightarrow 1\ 1\ 0\ 1$	$Y_4 \leftrightarrow 0\ 0\ 1\ 0$	$Y_4 \Leftrightarrow X_1 < X_2$
NAND		NOR	
$Y_{14} \Leftrightarrow X_1 \tilde{\wedge} X_2$	$Y_{14} \leftrightarrow 1\ 1\ 1\ 0$	$Y_8 \leftrightarrow 0\ 0\ 0\ 1$	$Y_8 \Leftrightarrow X_1 \tilde{\vee} X_2$
TRUTH		FALSITY	
$Y_{15} \Leftrightarrow 1$	$Y_{15} \leftrightarrow 1\ 1\ 1\ 1$	$Y_0 \leftrightarrow 0\ 0\ 0\ 0$	$Y_0 \Leftrightarrow 0$

Fig. **5–11** Mutually dual elementary logic-functions

The duality of functions is also expressed by the duality of their respective logic formulas Y and $\widetilde{Y}$. The duality of two logic formulas Y and $\widetilde{Y}$ follows directly from the duality of the elementary logic functions of Fig. 5–11.

(5.17) **Duality of logic formulas.** Two logic formulas Y and $\widetilde{Y}$ are **dual** if, by replacing all **0**s by **1**s, and **1**s by **0**s, and all dyadic connectives by their duals according to Fig. 5–11 in one of the formulas, it becomes identical with the other.

Note that all variables and negations remain unchanged as IDENTITY and NEGATION are the only self-dual functions. Also, the sequence in which the connectives are acted upon (as defined by parentheses) remains unaltered. For all practical purposes, duality is interesting due to what is called the

(5.18) Duality theorem. If two logic formulas $P(\mathbf{x})$ and $Q(\mathbf{x})$ can be proven to be equivalent then the respective dual formulas, $\widetilde{P}(\mathbf{x})$ and $\widetilde{Q}(\mathbf{x})$, are also equivalent:

$$\textit{if} \quad P(\mathbf{x}) \Leftrightarrow Q(\mathbf{x}) \quad \textit{then} \quad \widetilde{P}(\mathbf{x}) \Leftrightarrow \widetilde{Q}(\mathbf{x}).$$

To see the theorem's validity, interpret it this way: If the logic formulas $P(\mathbf{x})$ and $Q(\mathbf{x})$ are equivalent, they can both be taken as logic descriptions of some switching function $f : \{0,1\}^n \mapsto \{0,1\}$. Then the dual formulas $\widetilde{P}(\mathbf{x})$ and $\widetilde{Q}(\mathbf{x})$ both describe the same switching function f — one in positive logic, the other in negative logic — and are thus equivalent.

The practical importance of the duality theorem lies in the fact that it halves the burden of proof. For example, if you are able to prove the equivalence

$$X_1 \vee \overline{X}_1 X_2 \Leftrightarrow X_1 \vee X_2,$$

then by the duality theorem it is proven that

$$X_1(\overline{X}_1 \vee X_2) \Leftrightarrow X_1 X_2.$$

Let me add one example for writing the dual of a given formula according to (5.17). Looking back at the logic circuit of Fig. 2–11a, you can interpret it's gates as in Chapter 4, obtaining the formula

$$Y \Leftrightarrow \left((\overline{X_1 \vee_1 X_2}) <_3 (X_2 >_2 X_1)\right) \vee_6 \Big(X_1 \wedge_5 \big(X_2 \vee_4 (X_2 >_2 X_1)\big)\Big).$$

The indices assigned to the logical connectives are the numbers of the gates in Fig. 2–11a and are intended to help you see the correspondence between the circuit and the formula. The dual formula according to (5.17) is:

$$\widetilde{Y} \Leftrightarrow \left((\overline{X_1 \wedge X_2}) \rightarrow (X_2 \leftarrow X_1)\right) \wedge \Big(X_1 \vee \big(X_2 \wedge (X_2 \leftarrow X_1)\big)\Big).$$

Chapter 6

Karnaugh Maps

Writing the logical formula of a given table of asserted events as a canonical normal form, as discussed in the previous chapter, is more or less a standard procedure. The formulas obtained are often excessively long, so that a lot of energy was invested in finding methods to *minimise* these long formulas. Contrary to most minimisation methods, the map method, first presented by **Veitch** [1952], but put into a more practical form by **Karnaugh** [1953], bypassed writing a canonical normal form, enabling us instead to directly write a minimised or, at least, nearly minimised logic formula.

When working manually, i.e., without a computer, the **Karnaugh maps** (K-maps), as they have come to be known, belong to the most versatile tools we have. Karnaugh maps are **Venn diagrams** that depict very special sets, sets that in this text are called **maximum Karnaugh sets**. Let us start this chapter by briefly reviewing what little we need to know of sets.

6.1 Speaking of Sets...

Cantor (1845 – 1918), a brilliant and unorthodox mathematical thinker, conceived the idea of sets and their cardinal numbers to cope with the concept of infinity. A set, according to Cantor, could be a collection of any identifiable real or abstract objects — the so-called elements of the set. But, tragically, this approach, that imposes no limit to *what* and *how much* we collect, allows contradictions (or paradoxes, as they have come to be called) to be formulated.

Mainly to avoid paradoxes, set theory is nowadays introduced axiomatically, not saying *what* a set is but, rather, saying *how* to operate with sets. We shall introduce set theory in its so-called *naive* form, not axiomatizing it, because switching algebra employs only finite sets so there is no danger of running into a paradox. Undefined concepts of a theory are called its **primitives**. Set theory has two primitives — *set* and *membership*. With Cantor, yet quite informally, take a **set** $\mathcal{S}$ to be a collection of specified elements. If x is one such element we say 'x is a **member of** $\mathcal{S}$' or 'x is an **element of** $\mathcal{S}$' and write this as $x \in \mathcal{S}$.

A set $\mathcal{S}$ is always a collection of elements x that belong to some predefined **universal set** $\mathcal{U}$ and usually share a certain property P ($P(x)$ being read as 'x has the property P'):

$$x \in \mathcal{S} \Leftrightarrow x \in \mathcal{U} \ \wedge \ P(x). \tag{6.1}$$

This states that x, to be an element of $\mathcal{S}$, must be an element of $\mathcal{U}$ and must have the property P. The above formula does not directly specify (or 'define') the set $\mathcal{S}$, but it does allow us to decide for *each* element of the universe $\mathcal{U}$ whether it belongs to (is an element of) $\mathcal{S}$ or not, thus defining $\mathcal{S}$ *implicitly*. In the desire to specify the set $\mathcal{S}$ *explicitly*, the vague formulation

$$\mathcal{S} = \{x | x \in \mathcal{U} \ \wedge \ P(x)\} \tag{6.2}$$

is common. It is read as '$\mathcal{S}$ is the set consisting of all x for which $x \in \mathcal{U} \wedge P(x)$ is true', the vagueness lying in the expression 'consisting of'. The only way to understand (6.2) is to take it as an alternative way of writing (6.1).

There are two relations between sets, called **subset** $\mathcal{S} \subset \mathcal{T}$ ($\mathcal{S}$ is said to be the *subset* of $\mathcal{T}$ whereas I refer to $\mathcal{T}$ as a *relative universe* of $\mathcal{S}$), and **identity** $\mathcal{S} = \mathcal{T}$:

$$\mathcal{S} \subset \mathcal{T} \stackrel{df}{\Leftrightarrow} \text{For all } x: \ (x \in \mathcal{S} \ \rightarrow \ x \in \mathcal{T}), \tag{6.3}$$

$$\mathcal{S} = \mathcal{T} \stackrel{df}{\Leftrightarrow} \text{For all } x: \ (x \in \mathcal{S} \ \leftrightarrow \ x \in \mathcal{T}), \tag{6.4}$$

By (6.1), each set considered is *a priori* a subset of a previously agreed upon universe $\mathcal{U}$. For instance, the **complement** $\overline{\mathcal{S}}$ *of* $\mathcal{S}$ *consists of all elements not in* $\mathcal{S}$ *but still in* $\mathcal{U}$. Apart from the just mentioned *complement* $\overline{\mathcal{S}}$, the algebra of sets uses two further operations, the *intersection* and *union*. The **intersection** $\mathcal{S} \cap \mathcal{T}$ *is the set of all elements in* $\mathcal{S}$ AND ($\wedge$) *in* $\mathcal{T}$, while the **union** $\mathcal{S} \cup \mathcal{T}$ *is the set of all elements in* $\mathcal{S}$ OR ($\vee$) *in* $\mathcal{T}$. For all elements in $\mathcal{U}$ we can define these operations as

$$x \in \overline{\mathcal{S}} \stackrel{df}{\Leftrightarrow} (x \notin \mathcal{S}) \ \wedge \ (x \in \mathcal{U}) \tag{6.5}$$

$$x \in (\mathcal{S} \cap \mathcal{T}) \stackrel{df}{\Leftrightarrow} (x \in \mathcal{S}) \ \wedge \ (x \in \mathcal{T}) \tag{6.6}$$

$$x \in (\mathcal{S} \cup \mathcal{T}) \stackrel{df}{\Leftrightarrow} (x \in \mathcal{S}) \ \vee \ (x \in \mathcal{T}) \tag{6.7}$$

The right side of (6.5) uses the definition $x \notin \mathcal{S} \ \stackrel{df}{\Leftrightarrow} \ \neg(x \in \mathcal{S})$. The above operations are often pictured in *Venn diagrams*, as in Fig. 6–1.

Each set is represented by the points of a certain area in the **Venn diagram**. As each area has an infinite number of points the Venn diagram can represent infinite sets. To work with finite sets, you actually draw, or at least think of, distinct points in the plain. As a matter of principle, you always start by drawing the area of the universe $\mathcal{U}$. The *complement* $\overline{\mathcal{S}}$ of

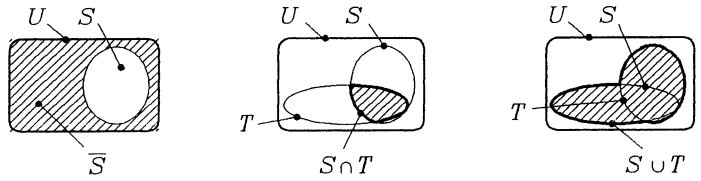


Fig. 6–1 The Venn diagram

some set $\mathcal{S}$ is then the area of $\mathcal{U}$ that is outside $\mathcal{S}$. The *intersection* $\mathcal{S} \cap \mathcal{T}$ of two sets $\mathcal{S}$ and $\mathcal{T}$ is the area where $\mathcal{S}$ and $\mathcal{T}$ overlap; the *union* $\mathcal{S} \cup \mathcal{T}$ is the area covered by $\mathcal{S}$ or by $\mathcal{T}$.

An important concept is that of the **null set** $\emptyset$, a virtual set containing no elements, and thus said to be empty. The null set is explained as a set the defining property of which is always false (**0**), e.g., $\{x|\ (x \text{ is odd}) \wedge ((x-2)(x-4) = 0)\}$, or $\{x|x \neq x\}$. By the definition of the equality of sets there is only one null set. Two sets, say $\mathcal{S}$ and $\mathcal{T}$, that have no elements in common, this being described by $\mathcal{S} \cap \mathcal{T} = \emptyset$, are said to be **disjoint**. Furthermore, the null set is a subset of every set.

A concept important for sequential circuits is the **power set** denoted here as $\wp(\mathcal{S})$, or sometimes as $\wp\mathcal{S}$, and explained as *the set of all subsets of a given set* $\mathcal{S}$, e.g., $\wp\{0,1\} = \{\emptyset, \{0\}, \{1\}, \{0,1\}\}$. The number of subsets of a given set $\mathcal{S}$ is

$$\#(\wp\mathcal{S}) = 2^{\#\mathcal{S}}. \tag{6.8}$$

6.2 Introducing the Karnaugh Map *

Despite the fact that the Karnaugh map is the most widely used graphical tool in switching algebra, it has, to date, successfully defied being explained in any formal way. So, hopefully, you can accept the following

(6.9) Definition. A **Karnaugh map** for n input variables $x_1, \cdots, x_n$ is a Venn diagram whose universe consists of all 2^n input events $(x_1, \cdots, x_n)$. The sets defined and pictured on the universe are n non-disjoint sets, here named $\mathcal{X}_1, \cdots, \mathcal{X}_n$ and referred to as **maximum Karnaugh sets**. Any such set $\mathcal{X}_i$ contains those input events $(x_1, \cdots, x_i, \cdots, x_n)$ whose i-th input variable x_i is 1:

$$(x_1, \cdots, x_i, \cdots, x_n) \in \mathcal{X}_i \Leftrightarrow x_i = 1 \quad \text{or} \quad \mathbf{x} \in \mathcal{X}_i \Leftrightarrow X_i \tag{6.9a}$$

when using $\mathbf{x} \stackrel{df}{=} (x_1, x_2, \cdots, x_n)$ and $X_i \stackrel{df}{\Leftrightarrow} x_i = 1$.

To illustrate, let us, in this section, consider the practically simplest case of a Karnaugh map, one for 2 input variables $x_1, x_2 \in \{0,1\}$, its universe thus being the set $\{0,1\}^2 = \{(0,0),(0,1),(1,0),(1,1)\}$ of all input events (x_1, x_2). The two sets to be depicted in such a Karnaugh map are the maximum Karnaugh sets $\mathcal{X}_1$ and $\mathcal{X}_2$ defined by

$$(x_1, x_2) \in \mathcal{X}_1 \Leftrightarrow x_1 = 1 \qquad \text{and} \qquad (x_1, x_2) \in \mathcal{X}_2 \Leftrightarrow x_2 = 1.$$

Let us first make sure we understand how these formulas determine the elements of $\mathcal{X}_1$ and $\mathcal{X}_2$. For these logical formulas to define the sets $\mathcal{X}_1$ and $\mathcal{X}_2$, the propositional forms $x_1 = 1$ and $x_2 = 1$ must be made into propositions that are true. This obviously means substituting 1 for x_1 in the left formula and for x_2 in the right formula, giving us

$$(1, x_2) \in \mathcal{X}_1 \Leftrightarrow \mathbf{1}, \qquad (x_1, 1) \in \mathcal{X}_2 \Leftrightarrow \mathbf{1}.$$

$(1, x_2) \in \mathcal{X}_1$ being true, it must be true for every instance, 0 and 1, of x_2, i.e., for all ordered pairs $(1,0)$, $(1,1)$, while $(x_1, 1) \in \mathcal{X}_2$ is required to be true for all values of x_1, i.e., for $(0,1)$, $(1,1)$. Thus, we can write the sets $\mathcal{X}_1$ and $\mathcal{X}_2$ as

$$\mathcal{X}_1 = \{(1,0),(1,1)\}, \qquad \mathcal{X}_2 = \{(0,1),(1,1)\}.$$

The above results can also be read from the table of input events of Fig. 6–2a. The Venn diagram of these maximum Karnaugh sets is depicted in Fig. 6–2b. *The Karnaugh map itself, Fig. 6–2c, is drawn in terms of the logic variables X_1 and X_2 that are used to define the maximum Karnaugh sets $\mathcal{X}_1$ and $\mathcal{X}_2$.* The bar defining the extent of X_1 is to be understood to replace the loop named X_1. Likewise, the bar for X_2 describes all cells to the right of that bar. The (optional) small number in the top left corner of each cell is the decimal equivalents e of the (binary) input event (e_1, e_2). The space in each cell of the Karnaugh map is intended for the output values caused by the input event of that cell and usually written into the output row of the table of asserted events.

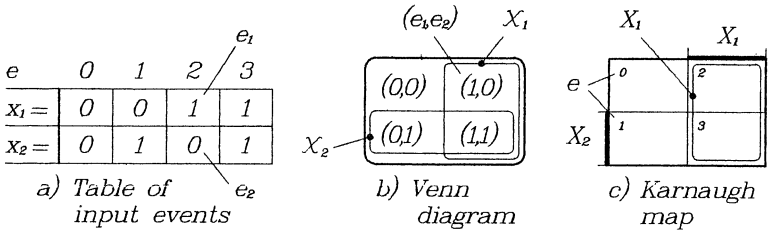


Fig. 6–2 Table of asserted events, Venn Diagram and Karnaugh Map

The **advantage of using maximum Karnaugh sets** $\mathcal{X}_i$ lies in the fact that their defining properties are the *logic variables* X_i. Using these instead of the sets $\mathcal{X}_i$, allows us to employ the conjunction (the AND) $X_1 \wedge X_2$ in place of the intersection $\mathcal{X}_1 \cap \mathcal{X}_2$, and the disjunction (the OR) $X_1 \vee X_2$ instead of the union $\mathcal{X}_1 \cup \mathcal{X}_2$, as inferred from the following formulas.

$$\mathbf{x} \in (\mathcal{X}_1 \cap \mathcal{X}_2) \overset{(6.6)}{\Longleftrightarrow} (\mathbf{x} \in \mathcal{X}_1) \wedge (\mathbf{x} \in \mathcal{X}_2) \overset{(6.9)}{\Longleftrightarrow} X_1 \wedge X_2 \qquad \textbf{(6.10)}$$

$$\mathbf{x} \in (\mathcal{X}_1 \cup \mathcal{X}_2) \overset{(6.7)}{\Longleftrightarrow} (\mathbf{x} \in \mathcal{X}_1) \vee (\mathbf{x} \in \mathcal{X}_2) \overset{(6.9)}{\Longleftrightarrow} X_1 \vee X_2 \qquad \textbf{(6.11)}$$

As shown in Fig. 6–3 for two inputs, you intersect and unite the loops X_1 and X_2 of the Karnaugh map — creating the expressions $X_1 \wedge X_2$ and $X_1 \vee X_2$ — in the same way you intersect and unite the sets $\mathcal{X}_1$ and $\mathcal{X}_2$ to obtain the sets $\mathcal{X}_1 \cap \mathcal{X}_2$ and $\mathcal{X}_1 \cup \mathcal{X}_2$ in the Venn diagram. It is also worth mentioning that *the full area of the Karnaugh map represents* **1**, i.e. TRUTH, as it is true that all the input events **e** are contained in the full area of the Venn diagram.

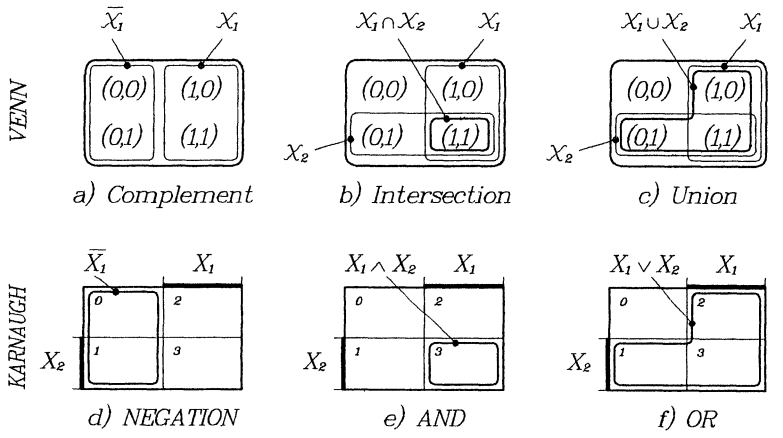


Fig. 6–3 Basic Operations in a Karnaugh Map

Let us take a more detailed look at the Karnaugh maps of Fig. 6–3. Whenever you explicitly need a NEGATION, say $\overline{X}_1$, you must refer to the area of the Karnaugh map that does *not* belong to X_1. This example is shown as a loop in Fig. 6–3d. In Section 5.1 we discussed minterms $C_e(x_1, x_2)$, and maxterms $D_e(x_1, x_2)$. A cell e of a Karnaugh map represents the minterm $C_e(x_1, x_2)$. The area outside the cell e represents the maxterm $D_e(x_1, x_2)$. For instance, the cell number 3, marked by a loop in Fig. 6–3e, depicts $C_3(x_1, x_2) \Leftrightarrow X_1 \wedge X_2$. The area marked by a loop in Fig. 6–3f represents the maxterm $D_0(x_1, x_2) \Leftrightarrow X_1 \vee X_2$. If you are new to this field, you should

take your time to get acquainted with the representation of minterms and maxterms in Karnaugh maps.

6.3 Karnaugh Maps for multiple Inputs

The look of a Karnaugh map, see Fig. 6–4, depends primarily on the number of input variables to be represented and secondarily on the arrangement of the input variables around the perimeter of the map. The number of input variables is referred to as the **dimension** of the K-map. In this text, the following arrangement of the variables has been chosen: *The first half of the input variables (or one more than the first half, in the case of n being odd) mark the columns, the rest mark the rows.*

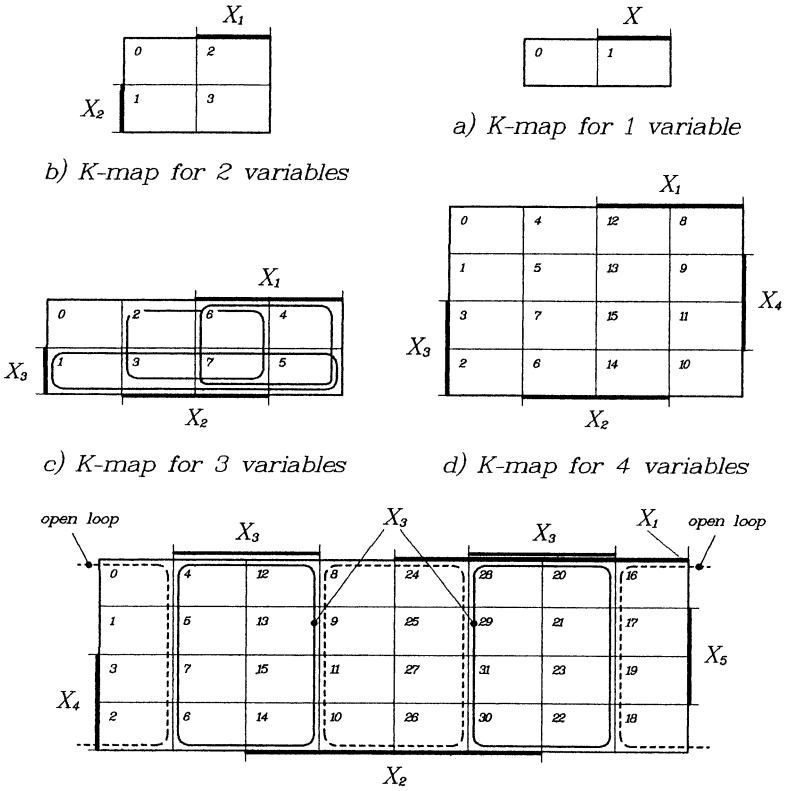


Fig. 6–4 Karnaugh maps for 1, 2, 3, 4 and 5 variables

As long as $n \leq 4$, you will notice that the circumference of each maximum Karnaugh set $\mathcal{X}_i$ is a (single) closed loop, as demonstrated in Fig. 6–4c for

$n = 3$. It is this property that makes working with Karnaugh maps up to four input variables so comfortable. This advantage is lost for five (or more) input variables, as shown in Fig. 6–4e for the area defined by the property X_3, and for the complementary area, defined by $\overline{X}_3$. The area defined by X_3 consists of *two separate areas*. The area defined by $\overline{X}_3$ also consists of two separate areas, the 'open loop' being thought of as a single closed line. You can quite easily imagine the open loop as a single closed line if you think of the Karnaugh map for $n = 5$ as being wrapped around a standing cylinder.

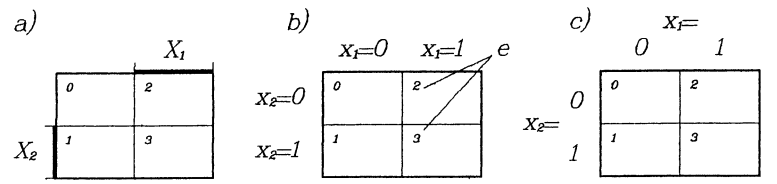


Fig. 6–5 Using numeric variables x_i with a Karnaugh map

Karnaugh's achievement was the practical way in which the logic variables were arranged around the perimeter of the map. We first switch from using logical variables to using numerical variables, obtaining a Karnaugh map as shown in Fig. 6–5c. The numerical values of the variables assigned to the columns, as well as those assigned to the rows, are ordered according to a Gray code. The **Gray codes** for one, two and three variables are shown in Fig. 6–6, this figure also explaining how the Gray code is developed. The characterising property of this code is that, as you progress from one column (i.e., code value) to the next, the new code differs from the old one in only a single position. Looping back from the last column to the first, you will notice that, here too, the codes differ in only one position. We shall follow the implications of this characterising property in the next section.

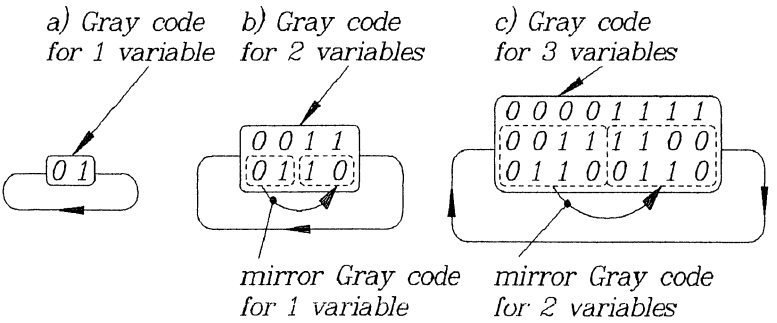


Fig. 6–6 Gray codes

To see how the Gray code is used to organise the input variables, compare Figs. 6–4e and 6–6. Interpret the rows of the Gray code in Fig. 6–6c, from top to bottom, as values of x_1, x_2 and x_3 respectively, and you will notice that this coincides with the ordering of x_1, x_2 and x_3 in Fig. 6–4e. Analogously, the rows in Fig. 6–6b, when interpreted as x_4 and x_5, explain the rows of Fig. 6–4e.

6.4 Karnaugh Sets *

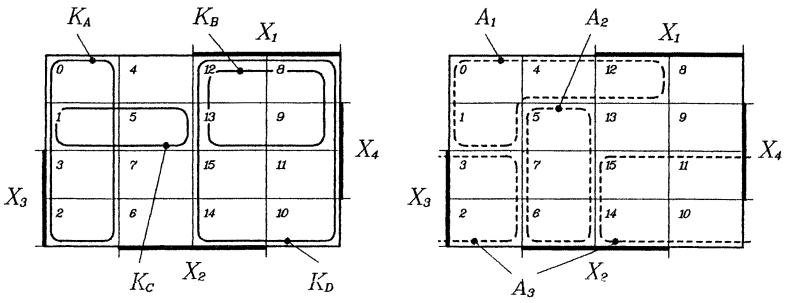
All the work you do with Karnaugh maps centres around what I choose to call *Karnaugh sets*. A maximum Karnaugh set $\mathcal{X}_i$ consists of those input events which share the property that their i-th input variable x_i is 1. A general **Karnaugh set** $\mathcal{K}$ is characterised by the fact that specified variables (say, $x_i, x_j, \ldots, x_k$) are assigned certain values, either 0 or 1. We can state this more formally as

$$(x_1, x_2, \ldots, x_n) \in \mathcal{K} \overset{df}{\Leftrightarrow} x_i = e_i \wedge x_j = e_j \wedge \ldots \wedge x_k = e_k \qquad \textbf{(6.12)}$$

in which $e_i, e_j, \ldots, e_k \in \{0,1\}$. By this definition, you can see that the following sets are Karnaugh sets and that, using (6.6), every Karnaugh set can be represented by the intersection of maximum Karnaugh sets or their complements.

$$\mathcal{K}_A = \{(x_1,x_2,x_3,x_4) | \overline{X}_1 \wedge \overline{X}_2\} \overset{(6.6)}{=} \overline{\mathcal{X}}_1 \cap \overline{\mathcal{X}}_2$$
$$\mathcal{K}_B = \{(x_1,x_2,x_3,x_4) | X_1 \wedge \overline{X}_3\} = \mathcal{X}_1 \cap \overline{\mathcal{X}}_3$$
$$\mathcal{K}_C = \{(x_1,x_2,x_3,x_4) | \overline{X}_1 \wedge \overline{X}_3 \wedge X_4\} = \overline{\mathcal{X}}_1 \cap \overline{\mathcal{X}}_3 \cap \mathcal{X}_4$$
$$\mathcal{K}_D = \{(x_1,x_2,x_3,x_4) | X_1\} = \mathcal{X}_1$$

Their representation in the Karnaugh map of Fig. 6–7a is due simply to the graphical interpretation of the intersection. Of all Karnaugh sets the maximum Karnaugh sets have the largest number of elements — thus the name *maximum*.



a) Examples of Karnaugh sets b) These are NOT Karnaugh sets

Fig. 6–7 Karnaugh and other sets

Informally, a Karnaugh set can be characterised by the following two properties. *For up to 4 input variables, the cells of a Karnaugh set can be enclosed by a single closed loop that has* four *corners. The number of cells in a Karnaugh set is always a power of 2 (i.e., 1, 2, 4, 8, 16,...).* By these criteria, all sets in Fig. 6–7a qualify as Karnaugh sets, while none of those in Fig. 6–7b do.

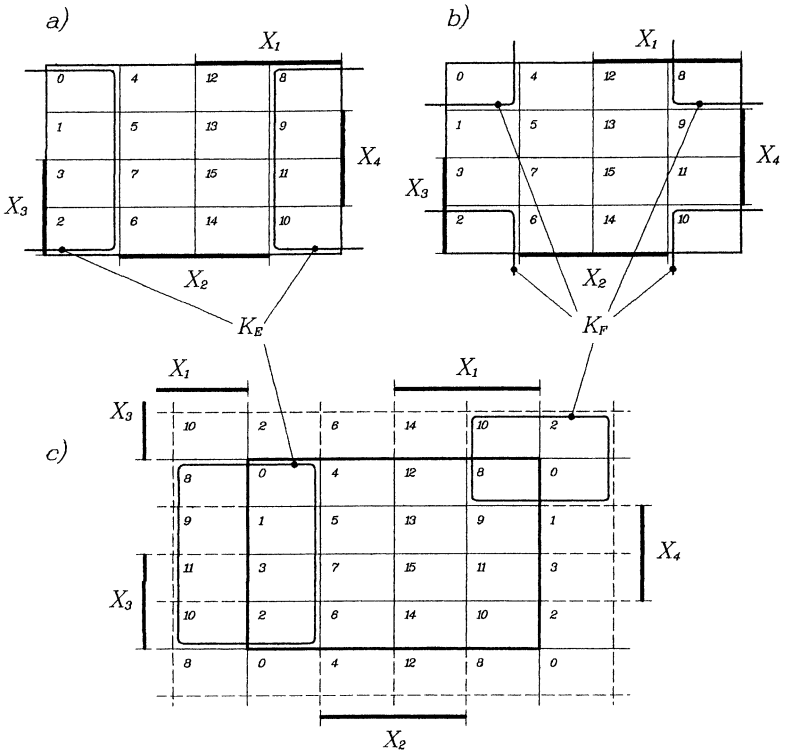


Fig. 6–8 'Tiling' Karnaugh maps

The Karnaugh maps of Figs. 6–8a and 6–8b contain standard examples of Karnaugh sets — $\mathcal{K}_E$ and $\mathcal{K}_F$ — whose loops seem not to be closed. But, imagine extending the original Karnaugh map as if laying tiles of identical maps in rows and columns (**'tiling'**), as indicated in Fig. 6–8c. By drawing $\mathcal{K}_E$ and $\mathcal{K}_F$ in this extended environment, so that they exceed the boundaries of the original Karnaugh maps, one can (for up to four input variables) always obtain contiguous areas and closed loops.

Chapter 7

Utilising Karnaugh Maps

There are two different ways in which Karnaugh maps (K-maps) are usually used. In the first three sections we discuss the K-map's conventional use, i.e., how to specify and evaluate (combinational) circuits. The next two sections are devoted to the question of developing and proving logical equations, a topic oddly lacking in the literature.

7.1 Specifying Switching Circuits in K-maps *

The cells of a Karnaugh map are intended to hold the output values $y(e)$ caused by the input events e. Writing the output values into appropriate cells is called *specifying or defining a switching circuit in a K-map.* It is simply done by entering the assertion $y = y(e)$ into the cell of the input event e in the same way as you fill in the output row of the *explicit* form of a table of asserted events.

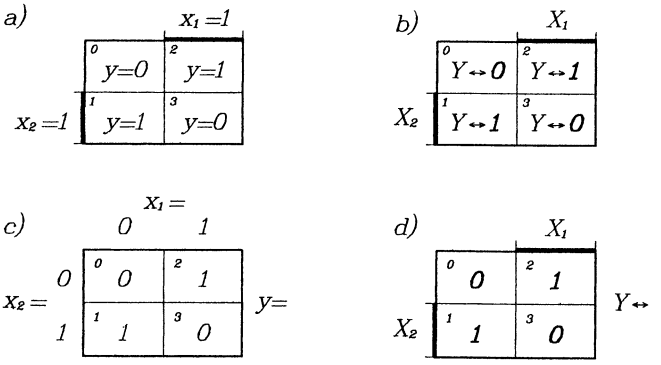


Fig. 7–1 Specifying a switching circuit in a K-map

Defining the running example of Chapter 1 (Fig. 1–2a) in a K-map leads to that of Fig. 7–1a. Extracting input and output variables leads to the frequently used **numeric specification** shown in Fig. 7–1c. On the other

hand, the **logical specification** of Fig. 7–1d is obtained from Fig. 7–1a by employing the correspondence between $(y_i, =, 0, 1)$ and $(Y_i, \leftrightarrow, \mathbf{0}, \mathbf{1})$, developed in Section 4–3, leading to Fig. 7–1b and then extracting the logic output variable Y.

7.2 Obtaining Disjunctive Formulas

The most common usage of a Karnaugh map is to employ it in designing logic switching circuits. There are three steps we have to take. You **first** *specify the circuit* which, in the context of this chapter, means filling-in a K-map with **0** and **1**. The **second** step is to *read the logical formula from the K-map*, and then, this being the **third** and last step, to *draw the circuit* by connecting the appropriate logic gates. The most obvious way to read a K-map leads to disjunctive formulas, and this is what we shall be covering here.

The basic idea is to assemble all the **1***s into* **as few and as large** *K-sets as possible, describe these by conjunctions and, finally, write the output as the disjunction of the conjunctions obtaining, what is called, a disjunctive form.* Take, as a first example of a K-map for which we wish to design a circuit, that of Fig. 7–2a .

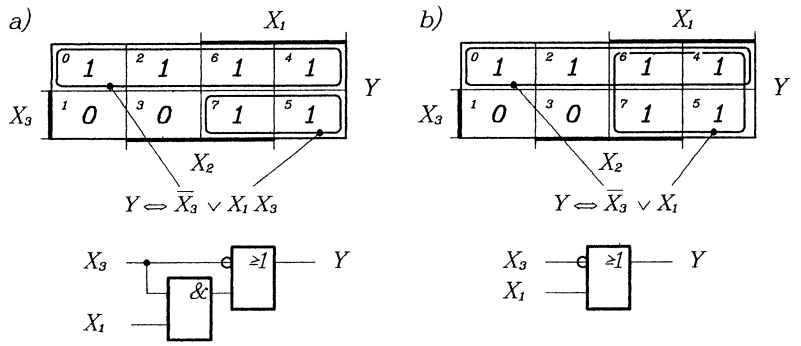


Fig. 7–2 Developing a disjunctive form

In a first attempt at assembling **1**s into K-sets, you might put the top row of **1**s into one K-set, while putting the bottom two **1**s into a second K-set. The first K-set is described by $\overline{X}_3$, the second by X_1X_3. The disjunction of these conjunctions leads to the disjunctive form $Y \Leftrightarrow \overline{X}_3 \vee X_1X_3$ for the output of the circuit. The logic circuit is shown in Fig. 7–2a. As Fig. 7–2b demonstrates, we can design larger K-sets if we allow them to *overlap*. The reason for K-sets being allowed to overlap is that when uniting them, each element is counted only once. In our example, the bottom two **1**s can be

taken as part of X_1 which leads to $Y \Leftrightarrow \overline{X}_3 \vee X_1$ and the circuit shown in Fig. 7–2b.

Choosing the largest possible K-sets *and* letting them overlap are important design principles. As we already know, the logical description of a K-set becomes simpler, the larger it is. As important as this aspect is, circuits designed via large and overlapping K-sets are far less prone to producing faulty output values, as is discussed in some detail in a later chapter. In fact, one will often incorporate large but *redundant* K-sets, the only reason being to avoid faulty outputs as much as possible. Such a redundant K-set is $\mathcal{K}_E$ in Fig. 7–3, the logical formula for the circuit being

$$Y \Leftrightarrow \underbrace{X_1\overline{X}_3X_4}_{Y_A} \vee \underbrace{\overline{X}_1X_2\overline{X}_3}_{Y_B} \vee \underbrace{\overline{X}_1X_3X_4}_{Y_C} \vee \underbrace{X_1X_2X_3}_{Y_D} \vee \underbrace{X_2X_4}_{Y_E}$$

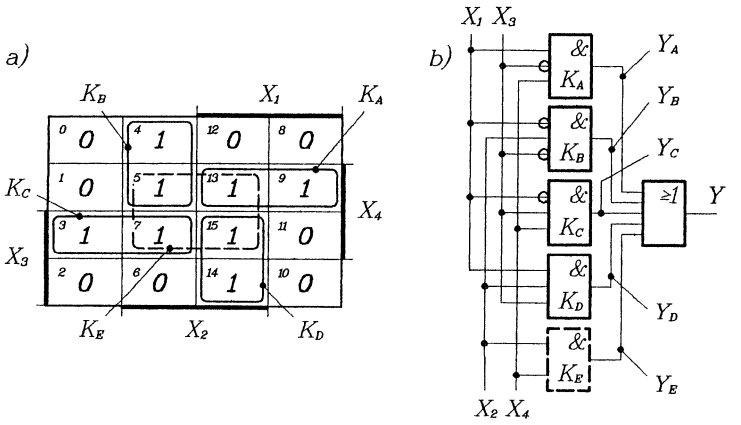


Fig. 7–3 Using redundant K-sets

This configuration of **1**s, often called a windmill, is completely covered by the two-element K-sets $\mathcal{K}_A$, $\mathcal{K}_B$, $\mathcal{K}_C$, $\mathcal{K}_D$, these being the fewest possible K-sets in which to assemble all **1**s, as is apparent. The four-element K-set $\mathcal{K}_E$ is **redundant**, meaning, it contributes nothing not already contributed by the two-element K-sets, and could therefore be omitted. Yet, it is often incorporated in a hardware design to reduce the risk of faulty output values.

7.3 Obtaining Conjunctive Formulas *

Reading a *conjunctive formula* from a K-map is a bit arduous as compared to reading a disjunctive formula from the K-map. The procedure itself is a

straight forward generalisation of that shown in the subsection 'using max-terms as partial outputs' of Section 5.2.

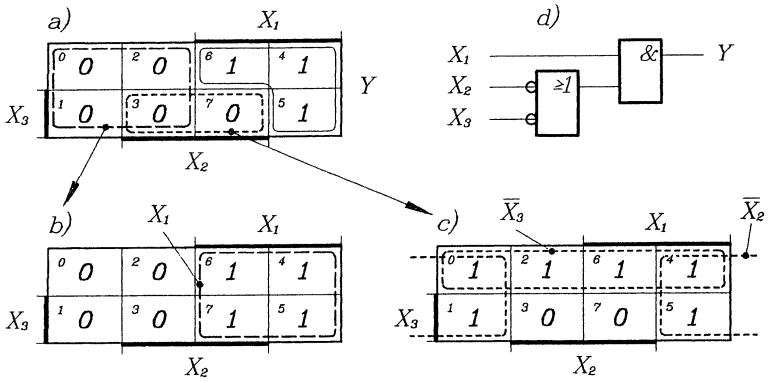


Fig. 7–4 Developing a conjunctive formula

Let us follow the example in Fig. 7–4 which illustrates the steps to take. **Step 1** requires us to assemble all **0**s (not the **1**s) in as few and as large K-sets as possible (Fig.7–4a). **Step 2**: For each K-set of assembled **0**s draw a new K-map in which the K-set of **0**s is repeated, filling the remaining cells of the K-map with **1**s (Figs. 7–4b and 7–4c). **Step 3**: Each of the newly drawn K-maps allow us to read their **1**s as *disjunctive forms*, e.g., X_1 for Fig. 7–4b and $\overline{X}_2 \vee \overline{X}_3$ for Fig. 7–4c. **Step 4**: The output is written as the *conjunction* of the disjunctive forms, $Y \Leftrightarrow X_1(\overline{X}_2 \vee \overline{X}_3)$, and is called a *conjunctive formula*. The **last step** is to draw the circuit according to the formula.

In step 2 you are requested to draw a new K-map for each K-set chosen in step 1. As this sometimes requires quite an amount of drawing, it is common to do step 2 as a mental exercise, putting none of it to paper. With a bit of practice, this is not difficult to do. An example of medium complexity is shown in Fig. 7–5.

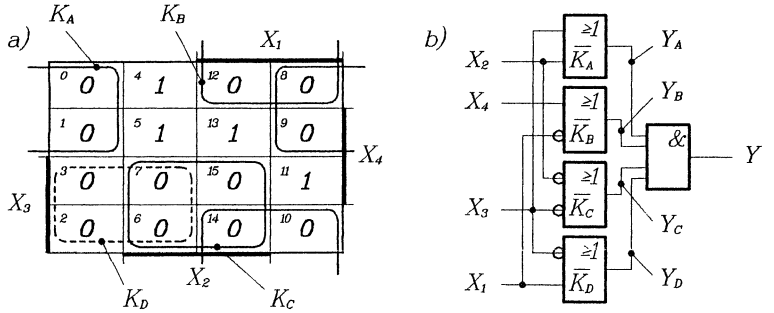


Fig. 7–5 Omitting step 2 to developing a conjunctive formula

Assume you have assembled all the **0**s into the K-sets $\mathcal{K}_A$ to $\mathcal{K}_D$ of Fig. 7–5a. Now let us attempt to do step 2 as a *gedanken* experiment. We first concentrate on the **0**s of $\mathcal{K}_A$, imagining that the rest of the cells contain **1**s. In this case the two centre columns and the two bottom rows would contain **1**s, and these columns and rows could be described by the disjunction $X_2 \vee X_3$. Reasoning in a like manner for the other K-sets allows us to write the output Y as the conjunctive formula

$$Y \Leftrightarrow \underbrace{(X_2 \vee X_3)}_{Y_A}\underbrace{(\overline{X}_1 \vee X_4)}_{Y_B}\underbrace{(\overline{X}_2 \vee \overline{X}_3)}_{Y_C}\underbrace{(X_1 \vee \overline{X}_3)}_{Y_D}$$

A final aspect to note when choosing the K-sets for this example is that not all K-sets are of equal importance. Whereas $\mathcal{K}_A$, $\mathcal{K}_B$ and $\mathcal{K}_C$ are necessary, $\mathcal{K}_D$ may be dropped in favour of a K-set $\mathcal{K}_E = \{0, 1, 3, 2\}$, as a brief examination of the K-map of Fig. 7–5a should reveal.

7.4 Logically Equivalent Expressions *

The Karnaugh map is an excellent tool for developing logically equivalent expressions. The basic idea to the method is this: *All logical descriptions of a certain loop or area of a Karnaugh map are logically equivalent. We accept formulas, thus obtained, as proven.* Let us work through a few examples.

We start by choosing some area in a K-map, for instance, the area of loop X_1 in Fig. 7–6a. Trivially, one logical description of this area is X_1 itself. A second way to describe this area, also shown in Fig. 7–6a, is to argue that the area X_1 can also be obtained by *uniting* (which means using the $\vee$-operation) the loops $X_1\overline{X}_2$ and X_1X_2. So, we can also describe the area of X_1 as $X_1\overline{X}_2 \vee X_1X_2$. These being descriptions of one and the same area, they are logically equivalent:

$$X_1 \Leftrightarrow X_1\overline{X}_2 \vee X_1X_2. \tag{7.1}$$

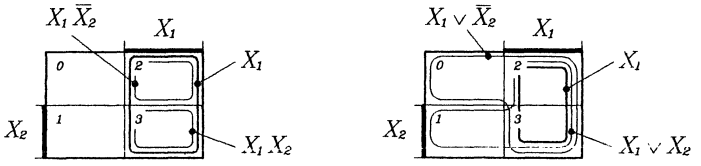


Fig. 7–6 Developing dual minimisation formulas

The logical formula dual to (7.1) is obtained, you will remember from Section 5.3, by interchanging the truth values **0** and **1**, and substituting the dual connective (Fig. 5–11) for every given connective. Following this procedure, (7.1) is transformed into

$$X_1 \Leftrightarrow (X_1 \vee \overline{X}_2)(X_1 \vee X_2). \tag{7.2}$$

Due to the duality theorem (5.18), the above formula (7.2) — being dual to a proven formula — need not itself be proven. Yet, to do so is quite informative, and is demonstrated in Fig. 7–6b. The right side disjunctions $(X_1 \vee \overline{X}_2)$ and $(X_1 \vee X_2)$ of (7.2) are shown as loops in Fig. 7–6b. The intersection of these loops is X_1, thus proving (7.2).

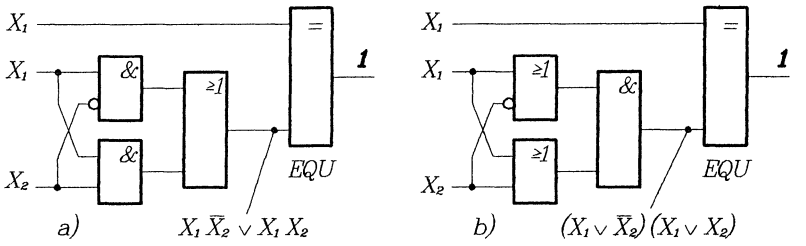


Fig. 7–7 Circuits verifying logical equivalence

What is the difference between **logical equivalence** and **material equivalence**? Material equivalence (for two input variables) is just one of the logic functions defined in the table of Fig. 4–3. Its output value is true, i.e. **1**, when both input variables have identical truth values, if their truth values differ, the output value is **0**. **Logical equivalence**, on the other hand, *refers to a material equivalence that is always true, i.e., it is* **1** *for every input event.* You can test this, and I strongly invite you to do so, on the two circuits of Fig. 7–7 that depict (7.1) and (7.2), respectively. Logical equivalence is so important because it allows us to **replace** an expression by one that is logically equivalent.

The method used in this section to develop and/or prove logical formulas can of course be used on formulas of more than two variables. Let us show this for, e.g., one of the laws of distributivity:

$$X_1(X_2 \vee X_3) \Leftrightarrow X_1X_2 \vee X_1X_3. \tag{7.3}$$

The intersection of the loops X_1 and $X_2 \vee X_3$, see Fig. 7–8a, is the loop described as $X_1(X_2 \vee X_3)$. The same area is obtained, see Fig. 7–8b, when uniting the loops X_1X_2 and X_1X_3, thus proving (7.3). As before, you can verify (7.3) by showing that the material equivalence of Fig. 7–8c is **1** for *all* input events (x_1, x_2, x_3).

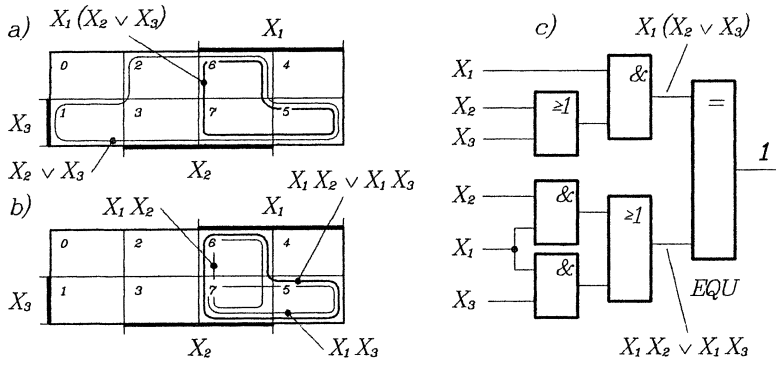


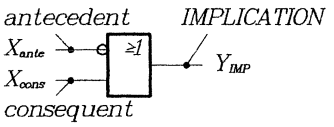
Fig. 7–8 Proving one of the laws of distributivity

7.5 Logical Implications *

Just as a logical equivalence is a material equivalence that is true for every input event, so is a **logical implication** *a material implication that is true for all input events.* As we will see in the next chapter when discussing *modus ponens*, the logical implication lays the foundation for developing *new* logical formulas. Thus, its status is of the same importance as that of logical equivalence.

$X_{ante} \leftrightarrow$	0	0	1	1
$X_{cons} \leftrightarrow$	0	1	0	1
$Y_{IMP} \leftrightarrow$	1	1	0	1

a) Truth Table



b) Gate representation

Fig. 7–9 IMPLICATION $Y_{IMP} \Leftrightarrow X_{ante} \rightarrow X_{cons} \Leftrightarrow \overline{X}_{ante} \vee X_{cons}$

The input variables of an implication are given names, as shown in Fig. 7–9, and the implication $X_{ante} \rightarrow X_{cons}$, if true, is read as 'if X_{ante} then X_{cons}'. This expresses the fact that the *consequent* can be deduced from the *antecedent* if the implication is true for all input events, i.e., if it is a logical implication. The question now is, how can we find logical implications? The answer is astonishingly simple:

> *If the set defined by the* antecedent *is a subset of the set defined by the* consequent *then the implication is a logical implication.*

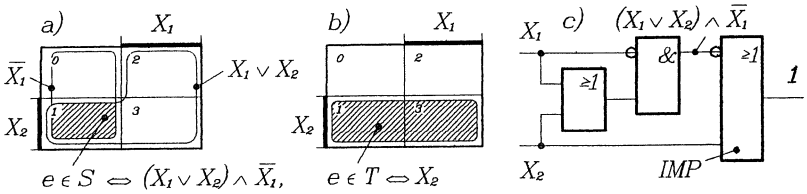


Fig. 7–10 Developing a logical implication

Let us work through a few examples, the first one being visualised in Fig. 7–10. $\mathcal{S} = \{1\}$ of Fig. 7–10a is obviously a subset of $\mathcal{T} = \{1, 3\}$ of Fig. 7–10b. A possible logic description of $\mathcal{S}$ (but by no means the only one) is $(X_1 \vee X_2) \wedge \overline{X}_1$, while $\mathcal{T}$ is best described by X_2. As the sets $\mathcal{S}$ and $\mathcal{T}$ stand in the relationship $\mathcal{S} \subset \mathcal{T}$, the description $(X_1 \vee X_2) \wedge \overline{X}_1$ of $\mathcal{S}$ is the antecedent, and the description X_2 of $\mathcal{T}$ is the consequent of a logical implication:

$$(X_1 \vee X_2) \wedge \overline{X}_1 \Rightarrow X_2. \tag{7.4}$$

You can test this, as shown in Fig. 7–11, by verifying that the output of the logical circuit is **1** for every input event.

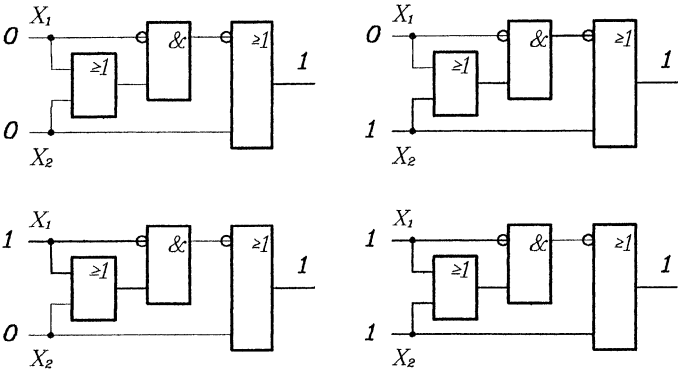


Fig. 7–11 Verifying (7.4)

In the example of Fig. 7–12a again $\mathcal{S} = \{0, 1, 3, 7\}$ is a subset of $\mathcal{T} = \{0, 1, 2, 3, 5, 7\}$. The description $(X_1 \rightarrow X_2) \wedge (X_2 \rightarrow X_3)$ of $\mathcal{S}$ is thus the antecedent, while the description $(X_1 \rightarrow X_3)$ of $\mathcal{T}$ is the consequent:

$$(X_1 \rightarrow X_2) \wedge (X_2 \rightarrow X_3) \Rightarrow (X_1 \rightarrow X_3). \tag{7.5}$$

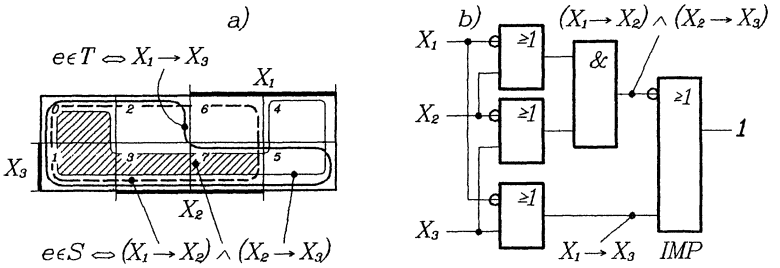


Fig. 7–12 Developing a logical implication

7.6 K-maps of Dual Functions *

Once in a while, you may find it advantageous to work *not* with a function F originally given, but rather with its dual function $\widetilde{F}$. You will then want to develop the table of asserted events (TAE) of $\widetilde{F}$ from that of F, or the K-map of $\widetilde{F}$ from that of F. The definition of duality, as stated in

$$\widetilde{F}(X_1, \ldots, X_n) \Leftrightarrow \overline{F(\overline{X}_1, \ldots, \overline{X}_n)}, \tag{5.16}$$

allows you to draw the TAE of $\widetilde{F}$ by inverting *all* **0**s and **1**s in the TAE of F — as shown in the example of Fig. 7–13.

$e \ldots$	0	1	2	3	4	5	6	7
$X_1 \leftrightarrow$	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1**
$X_2 \leftrightarrow$	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**1**
$X_3 \leftrightarrow$	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**
$Y \leftrightarrow$	**0**	**1**	**0**	**1**	**1**	**0**	**1**	**1**

(a) A given function F

$\overline{e} \ldots$	7	6	5	4	3	2	1	0
$X_1 \leftrightarrow$	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**
$X_2 \leftrightarrow$	**1**	**1**	**0**	**0**	**1**	**1**	**0**	**0**
$X_3 \leftrightarrow$	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**
$\widetilde{Y} \leftrightarrow$	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**0**

(b) The dual function $\widetilde{F}$

Fig. 7–13 The TAEs of dual functions

It is more convenient (and quicker) to append the dual output $\widetilde{F}$ to the TAE of the given function F as shown in Fig. 7–14: You first write the output $\overline{Y}$ of the inverted function $\overline{F}$, and then reverse the sequence of $\overline{Y}$ to obtain the output $\widetilde{Y}$ of the dual function $\widetilde{F}$. Of course, you can omit writing the inverted output $\overline{Y}$.

The variable $\overline{e}$ used in the top row of Fig. 7–13b still needs soom explanation. It is best introduced by its binary representation. The **complement** $\overline{e}_i$ of the binary variable $e_i \in \{0, 1\}$ is defined as

$$\overline{e}_i \stackrel{df}{=} 1 - e_i, \qquad \text{with} \qquad e_i \in \{0, 1\}. \tag{7.7}$$

$e\ldots$	0	1	2	3	4	5	6	7
$X_1 \leftrightarrow$	0	0	0	0	1	1	1	1
$X_2 \leftrightarrow$	0	0	1	1	0	0	1	1
$X_3 \leftrightarrow$	0	1	0	1	0	1	0	1
$Y \leftrightarrow$	0	1	0	1	1	0	1	1
$\overline{Y} \leftrightarrow$	1	0	1	0	0	1	0	0
$\widetilde{Y} \leftrightarrow$	0	0	1	0	0	1	0	1

Fig. 7–14 Defining $\overline{F}$ and $\widetilde{F}$ of F

Given the binary representation $(e_1, \ldots, e_n)$ of a decimal number $(e)_{10}$,

$$e = (e_1, \ldots, e_n) = \sum_{i=1}^{n} e_i \cdot 2^{n-i} \qquad \text{with} \qquad e_i \in \{0, 1\}, \tag{7.8}$$

we define the **complement** $\overline{e}$ of e as

$$\overline{e} \stackrel{df}{=} (\overline{e}_1, \ldots, \overline{e}_n) = \sum_{i=1}^{n} \overline{e}_i \cdot 2^{n-i} \qquad \text{with} \qquad \overline{e}_i \in \{0, 1\}, \tag{7.9}$$

meaning, $\overline{e}$ is the decimal number representing a complemented input event $(\overline{e}_1, \ldots, \overline{e}_n)$. The relationship between e and $\overline{e}$ is simply

$$\overline{e} = (2^n - 1) - e \tag{7.10}$$

as is easy to see:

$$\overline{e} = \sum_{i=1}^{n} \overline{e}_i \cdot 2^{n-i} = \sum_{i=1}^{n} (1 - e_i) \cdot 2^{n-i} =$$

$$\sum_{i=1}^{n} (2^{n-i} - e_i \cdot 2^{n-i}) = \underbrace{\sum_{i=1}^{n} 2^{n-i}}_{2^n - 1} - \underbrace{\sum_{i=1}^{n} e_i \cdot 2^{n-i}}_{e}.$$

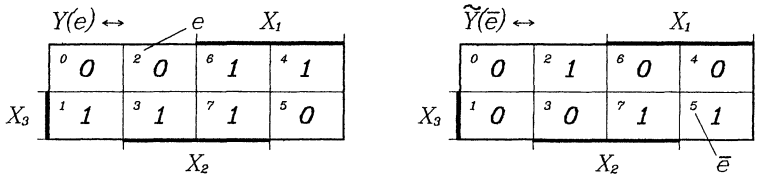
Using e and $\overline{e}$ to introduce the abbreviations

$$\widetilde{Y}(e) \stackrel{df}{\Leftrightarrow} \widetilde{F}(X_1, \ldots, X_n) \qquad \text{and} \qquad Y(\overline{e}) \stackrel{df}{\Leftrightarrow} F(\overline{X}_1, \ldots, \overline{X}_n), \tag{7.11}$$

we can express $\widetilde{F}(X_1, \ldots, X_n) \Leftrightarrow \overline{F(\overline{X}_1, \ldots, \overline{X}_n)}$, (5.16), as

$$\widetilde{Y}(e) \Leftrightarrow \overline{Y\big((2^n - 1) - e\big)} \qquad \text{or} \qquad \widetilde{Y}\big((2^n - 1) - e\big) \Leftrightarrow \overline{Y(e)}. \tag{7.12}$$

The latter formula, $\widetilde{Y}\big((2^n-1)-e\big) \Leftrightarrow \overline{Y(e)}$, can be used in the following way to develop the K-map of the dual $\widetilde{F}$ from the K-map of a given function F. As an example, consider a function F specified by the K-map of Fig. 7–15a.



a) A given function F *b) The function $\widetilde{F}$ dual to F*

Fig. 7–15 K-maps of dual functions

The output value $\widetilde{Y}\big((2^n-1)-e\big)$ is the negation of the output value $Y(e)$, and is written into the cell of $\widetilde{F}$ whose cell number $\overline{e} = (2^n-1)-e$ is the complement of e. It is appropriate to first transform those cells of F which contain the truth value that occurs least frequently (in our case **0** in the cells 0, 2, 5 of F): Write the negated truth value (**1**) into the complemented cells of $\widetilde{F}$ (i.e., into 7, 5, 2). The remaining cells of the K-map of the dual function $\widetilde{F}$ (the cells (0, 1, 3, 4, 6) then take the negated truth value of the cells first filled.

Division Two

Logic

8 Tautologies 79 (10 pages)
9 Propositional Logic 89 (12 pages)
10 Summary of Theorems 101 (9 pages)
11 Algebraic Proofs 110 (11 pages)
12 On Predicate Logic 121 (8 pages)
13 Predicate Logic 129 (10 pages)

It was the flair of **Shannon**'s [1938] paper that triggered the use of **propositional logic** in designing switching circuits, and thus initiating what is now known as switching algebra. Although it has since become in vogue to develop switching theory by using propositional logic—mostly presented as a *Boolean algebra*—the meagre theoretical results have let the original drive slacken. In this second Division, I adapt **predicate logic** to switching algebra. Predicate logic, being a more powerful tool than propositional logic, opens new possibilities, as the study of combinational circuits in the third Division of this book shows.

Chapter 8 introduces the basic tools needed in working with logic, and the all-important concept of **tautology**. We start our discussion of logic with propositional logic (Chapter 9), as everything valid in this realm remains so in predicate logic (Chapter 13). The axiom system presented was chosen because it defines *each* of the main connectives ($\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$). It was introduced by **Hilbert, Bernays** [1968], and minimally modified by **Asser** [1982]. In the context of this book, the axiom system employed is of less importance than it is to demonstrate how to work with a given system. The axiom system could be replaced by any of a multitude of other well established systems.

Let me stress the importance of the summary of theorems in Chapter 10 by pointing out that *all commutative, associative, and distributive laws* are listed in the tables of the chapter—no further laws of the kind mentioned exist.

The examples of Chapter 11 are intended to motivate the introduction of predicate logic, especially the example considered in Section 11.2. The basic concepts of predicate logic are then introduced informally in Chapter 12, the more formal approach being left for Chapter 13. The two main points of this final chapter are the summary of the theorems on quantification (not readily available for non-specialists) and the important theorem on identical functions (hardly known in applied mathematics).

Chapter 8

Tautologies

A *tautology* is a logical expression, such as $X_1 \Leftrightarrow X_1\overline{X}_2 \vee X_1X_2$ or $(X_1 \vee X_2) \wedge \overline{X}_1 \Rightarrow X_2$, that is true for *all* input events. This likens the tautology to what is sometimes called an identity in conventional algebra, e.g. $(a+b)^2 \equiv a^2 + 2ab + b^2$. In the last two sections of the previous chapter, we already saw how to use K-maps to obtain tautologies. In this chapter we introduce *truth tables* to prove tautologies.

Most importantly, we discuss in-depth how tautologies are used, that is, we discuss *replacement*, *substitution* and *modus ponens*.

8.1 Logic Expressions

Not every sequence of variables, connectives, and parentheses is what we are willing to accept as a *well formed logic expression*, as you might guess when looking at the following examples.

$$\neg(X_1 \wedge) X_2$$

$$X_1 \vee \rightarrow X_2 \neg X_3)$$

To be able to refer more generally to single variables and logic expressions, we use the variables A, B, C, and D (with or without indices) to represent any logical input or output variable X, X_1, X_2, ..., Y, Y_1, Y_2, ..., and the symbols P, Q, R, S, T (also with or without indices) to represent logic expressions.

To prohibit getting meaningless expressions, like the ones above, it is common to use so-called **formation rules**. Their application ensures that only *well formed logic expressions* can be constructed. But, instead of always referring to *well formed* logic expressions, we shall be content to simply call them *logic expressions*, arguing that only well formed logic expressions are **logic expressions**.

The **first formation rule** states that *writing a constant,* **0** *or* **1**, *or a single logical variable, say A, provides us with a logic expression.* The **second formation rule** states how to write the monadic function of NEGATION:

If a string of characters, say P, is a logic expression then you obtain a new logic expression by writing $(\neg P)$. Do note that the parentheses are an intrinsic part of the result. The **third formation rule** tells us how to correctly insert dyadic connectives: *If the character strings P and Q are logic expressions then so are* $(P \wedge Q)$, $(P \vee Q)$, $(P \rightarrow Q)$, *and* $(P \leftrightarrow Q)$. Here too, the parentheses are an intrinsic part of the result. The **fourth formation rule** is optional but nevertheless quite practical in indicating that the building process has terminated: *The outermost parentheses of the final logic expressions are dropped.*

The resulting string, when using the fourth formation rule, has one of the forms $\neg P$, $P \wedge Q$, $P \vee Q$, $P \rightarrow Q$, or $P \leftrightarrow Q$ and is referred to as a **complete logic expression**. The monadic or dyadic connective of a complete logic expression is called the **main connective** of the logic expression, this connective giving the logic expression its name. It is always the last connective to be acted upon when evaluating the truth value of the logic expression.

A direct consequence of the formation rules is that there is no priority between the connectives, the sequence in which they are acted upon is determined solely by parentheses. Furthermore, the formation rules cause the number of parentheses to grow quite rapidly. To counteract this, one uses a *convention of adhesion* that introduces an artificial priority between logical connectives.

> **Convention of adhesion**: A given connective of the quintuple $(\neg, \wedge, \vee, \rightarrow, \leftrightarrow)$ has a stronger adhesion to its arguments than any connective listed to its right has.

This convention allows us to rewrite the following top formula in the indicated ways.

$$\begin{gathered}((A \wedge B) \vee ((\neg A) \wedge B)) \leftrightarrow B \Leftrightarrow \\ (A \wedge B) \vee (\neg A \wedge B) \leftrightarrow B \Leftrightarrow \\ A \wedge B \vee \neg A \wedge B \leftrightarrow B\end{gathered}$$

When $\wedge$- and $\vee$-connectives occur alternatively, one is prone to confuse them. It is thus common to adhere to the

> **Convention of omitting the $\wedge$-connective**: One may write PQ instead of $P \wedge Q$ when it seems appropriate.

The last formula may then be written $AB \vee \neg AB \leftrightarrow B$ which is quite easy to read, as you will hopefully agree with. In contrast to the above procedure, it is sometimes advantageous not to **bind** arguments to (dyadic) connectives but to **separate** the arguments from the connectives. This is done in the

> **Convention of separation**: *Dots* attached to dyadic connectives (as in $.\wedge.$, $:\leftrightarrow:$, $.:\rightarrow:.$) separate these from their arguments. The more dots, the stronger the separation. But, for two different kinds of connectives, both with the same number of dots, the above *convention of adhesion* prevails.

The following example of the application of the *convention of separation*

demonstrates that the dots need not occur symmetrically.

$$(A \to (B \to C)) \to (A \wedge B \to C) \Leftrightarrow$$
$$A \to (B \to C) \;.\!\to. A \wedge B \to C \Leftrightarrow$$
$$A \;.\!\to. B \to C \;:\!\to. A \wedge B \to C$$

We now turn to the question of how to methodically evaluate logic expressions.

8.2 Truth Tables

A **truth table***) for a given logic expression is a table that — for each input event — allows us to methodically calculate and write every interim result leading to the truth value of the main connective. The process as such is often called **perfect induction** (not to be confused with *mathematical induction).*

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \to B$	$A \leftrightarrow B$
0	0	1	0	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	0	0
1	1	0	1	1	1	1

Fig. 8–1 Defining the connectives $\neg$, $\wedge$, $\vee$, $\to$, $\leftrightarrow$ in truth tables

The simplest truth tables are those that define the elementary connectives (Fig. 8–1) used in logic expressions. Fig. 8–1 actually contains five truth tables, one for each of the logic expressions stated in the headers of the five rightmost columns. To economise, the input events are written only once in a column to the left. You may have already noticed that Fig. 8–1 is an excerpt of Fig. 4–3.

Let us employ an example to see how to use a truth table. Say we want to determine the truth value of the main connective for each input event of the logic expression $(A \vee B) \wedge ((A \vee (\neg B)) \wedge ((\neg A) \vee B))$. The truth table consists of two columns, the left one containing as header the input variables A and B, and then — row for row — all the input events.

The header of the right column is the logic expression to be evaluated. Its notation should comply with the second formation rule (that is, you should write $\neg P$, and not $\overline{P}$, ensuring that there is a negation symbol under which its truth values may be written), and — for a similar reason — no $\wedge$-connective should be omitted. As there is often no *unique* sequence by which the logic

*) Truth tables were introduced by **Ludwig Wittgenstein** (1889-1951)

		1	7	5	2	6	3	4
A	B	$(A \vee B)$	$\wedge$	$((A \vee$	$(\neg B))$	$\wedge$	$((\neg A)$	$\vee B))$
0	0	0	0	1	1	1	1	1
0	1	1	0	0	0	0	1	1
1	0	1	0	1	1	0	0	0
1	1	1	1	1	0	1	0	1

Fig. 8–2 Successive evaluation of logic connectives

connectives must be evaluated, it is good practice to number the logic connectives according to the sequence used. The logic connective with the highest number is the main connective.

The sequence of evaluation was chosen in the following way. The logic expression is read from left to right, evaluating any connective possible, a process we call a *pass* of the logic expression. This lets us evaluate $\overset{1}{\vee}$, but not $\overset{7}{\wedge}$, as its right argument (the truth value to be found under $\overset{6}{\wedge}$) is not yet evaluated. Likewise, $\overset{5}{\vee}$ can not yet be evaluated, its right argument (found under $\overset{2}{\neg}$) still being unknown. Although we can calculate $\overset{2}{\neg}$, we do not go back to determine $\overset{5}{\vee}$. Rather, we continue reading the logic expression from left to right, next encountering $\overset{6}{\wedge}$. This connective can not be evaluated as its left argument (the value under $\overset{5}{\vee}$) is still unknown. But, the next connective, $\overset{3}{\neg}$, having a given value as its argument, can be evaluated, and with it $\overset{4}{\vee}$.

This *pass* of the logic expression enables us to calculate the truth value under the connectives $\overset{1}{\vee}$,$\overset{2}{\neg}$, $\overset{3}{\neg}$,$\overset{4}{\vee}$. A second *pass* allows us to determine $\overset{5}{\vee}$ and $\overset{6}{\wedge}$ as the next connectives to be evaluated, while a final *pass* (a mere formality) lets us determine $\overset{7}{\wedge}$ as the main connective.

Having found a sequence in which the connectives are or may be evaluated, one column of truth values after another is calculated. This proves to be faster than calculating row after row.

8.3 Speaking of Tautologies ...

A **tautology** *is a logical expression, say* $R(\mathbf{x})$*, that is true for every event* $\mathbf{x}$ *of its variables.* Saying that $R(\mathbf{x})$ is *true for every event* $\mathbf{x}$ is symbolically expressed as:

$$R(\mathbf{x}) \Leftrightarrow \mathbf{1}. \tag{8.1}$$

Although a tautology is assigned a truth value it is not itself a proposition. Rather, containing variables, it stands for a whole set of propositions, each a substitution instance of the tautology and each true.

Logical equivalence & logical implication

There are two kinds of tautologies, called *logical equivalencies* and *logical implications*. All other forms of a tautology can be expressed by the just mentioned two.

A **logical equivalence** is a material EQUIVALENCE, i.e., a complete logical expression of the form $P(\mathbf{x}) \leftrightarrow Q(\mathbf{x})$ that *is true for all events* $\mathbf{x}$ of the expression's variables, this being written as $P(\mathbf{x}) \leftrightarrow Q(\mathbf{x}) \Leftrightarrow \mathbf{1}$, or as $P(\mathbf{x}) \Leftrightarrow Q(\mathbf{x})$, for short:

$$P(\mathbf{x}) \Leftrightarrow Q(\mathbf{x}) \stackrel{df}{.\Leftrightarrow}. P(\mathbf{x}) \leftrightarrow Q(\mathbf{x}) \Leftrightarrow \mathbf{1}. \tag{8.2}$$

A **logical implication** is a material IMPLICATION, i.e., a complete logical expression of the form $P(\mathbf{x}) \rightarrow Q(\mathbf{x})$ that *is true for all events* $\mathbf{x}$ of the expression's variables, this being written as $P(\mathbf{x}) \rightarrow Q(\mathbf{x}) \Leftrightarrow \mathbf{1}$, or as $P(\mathbf{x}) \Rightarrow Q(\mathbf{x})$, for short:

$$P(\mathbf{x}) \Rightarrow Q(\mathbf{x}) \stackrel{df}{.\Leftrightarrow}. P(\mathbf{x}) \rightarrow Q(\mathbf{x}) \Leftrightarrow \mathbf{1}. \tag{8.3}$$

There is one prototype of a tautology that is often expressed using an $\vee$-connective

$$(P \vee \neg P) \Leftrightarrow \mathbf{1}. \tag{8.4}$$

But, as the truth tables of Fig. 8–3 illustrate, you can always express $(P \vee \neg P)$ as either $(P \leftrightarrow P)$ or as $(P \rightarrow P)$, allowing us to write

$$(P \leftrightarrow P) \Leftrightarrow (P \rightarrow P) \Leftrightarrow (P \vee \neg P) \Leftrightarrow \mathbf{1}. \tag{8.5}$$

P	$P \leftrightarrow P$	$P \rightarrow P$	$P \vee (\neg P)$
0	**1**	**1**	**1 1**
1	**1**	**1**	**1 0**

Fig. 8–3 Proving $(P \leftrightarrow P) \Leftrightarrow (P \rightarrow P) \Leftrightarrow (P \vee \neg P) \Leftrightarrow \mathbf{1}$

Proving tautologies by truth table

Tautologies can be proved efficiently and quickly by using truth tables to test if the main connectives — a material equivalence, or a material implication — are true for all events (a method prematurely used in Fig. 8–3). To illustrate, let us check whether the logic expressions

$$(P \wedge (\neg Q)) \vee Q \leftrightarrow P \vee Q, \tag{8.6}$$

$$(P \vee Q) \wedge (\neg P) \rightarrow Q. \tag{8.7}$$

are tautologies. This is accomplished in Fig. 8–4. Note that the main connectives are **1** for all input events as is required by definitions (8.2) and (8.3).

		3	1	4	5	2	1	3	2	4
P	Q	$(P\wedge(\neg Q))$		$\vee$	Q	$\leftrightarrow$	$P\vee Q$	$(P\vee Q)$	$\wedge$	$(\neg P)$	$\rightarrow$	Q
0	0	0	1	0	1	0	0	0	1	1		
0	1	0	0	1	1	1	1	1	1	1		
1	0	1	1	1	1	1	1	0	0	1		
1	1	0	0	1	1	1	1	0	0	1		
		logical equivalence					logical implication					

Fig. 8–4 Proving tautologies

Having proved that the logic expressions are true for *all* events, you can replace their *material* main connectives $\rightarrow$ and $\leftrightarrow$ by their corresponding *logical* connectives $\Rightarrow$ and $\Leftrightarrow$, thereby demonstrating that the logic expressions are proved to be tautologies:

$$(P\wedge(\neg Q))\vee Q \;\Leftrightarrow P\vee Q, \tag{8.8}$$

$$(P\vee Q)\wedge(\neg P) \;\Rightarrow Q. \tag{8.9}$$

Tautologies & K-maps *

In Sections 7.4 and 7.5 we saw how to develop logical equivalencies and logical implications in K-maps. But, lacking the definitions (8.2) and (8.3) we could not then prove the procedure. In this subsection we shall show why subsets correspond to logical implications (leaving the proof for the procedure of Section 7.4 to you).

Consider a logical implication

$$P(\mathbf{x})\Rightarrow Q(\mathbf{x}) \tag{8.10}$$

the antecedent $P(\mathbf{x})$ of which defines a set $\mathcal{S}$, while the consequent $Q(\mathbf{x})$ defines a set $\mathcal{T}$

$$\mathbf{x}\in\mathcal{S}\Leftrightarrow P(\mathbf{x}),\qquad \mathbf{x}\in\mathcal{T}\Leftrightarrow Q(\mathbf{x}). \tag{8.11}$$

Inserting the left sides of (8.11) into (8.10) gives us

$$\mathbf{x}\in\mathcal{S}\Rightarrow\mathbf{x}\in\mathcal{T} \tag{8.12}$$

which is true for all events $\mathbf{x}$, and thus allows us to state the concept of subset (6.3) as

$$\mathcal{S}\subset\mathcal{T}\;.\Leftrightarrow.\;\mathbf{x}\in\mathcal{S}\Rightarrow\mathbf{x}\in\mathcal{T}. \tag{8.13}$$

Returning to the logic expressions $P(\mathbf{x})$ and $Q(\mathbf{x})$, according to (8.11), leads to

$$\mathcal{S}\subset\mathcal{T}\;.\Leftrightarrow.\;P(\mathbf{x})\Rightarrow Q(\mathbf{x}). \tag{8.14}$$

By this formula a subset relationship defines a logical implication, and, *vice versa*, a logical implication defines a subset relationship. Thus, we can use the K-map to *develop* and to *prove* logical implications. The same reasoning holds for the logical equivalence.

8.4 Replacement versus Substitution

The importance of the logical equivalence lies in its use in *replacement* and *substitution*, two concepts we take a brief look at in this section.

Assume a complete logic expression, say P, to contain a sub-expression, say S, one or more times. Further, assume that we know S to be logically equivalent to an expression we refer to as R. Then the **rule of replacement** states: *Replacing some or all of the sub-expressions S of P with the replacement expression R, logically equivalent to S, transforms P into a new, yet logically equivalent expression, say Q.*

This rule is employed in the following example, using (8.8).

$$\overbrace{\underbrace{A\overline{B} \vee B}_{S} \vee C \vee D\underbrace{(A\overline{B} \vee B)}_{S}}^{P} \Leftrightarrow \overbrace{\underbrace{A\overline{B} \vee B}_{S} \vee C \vee D\underbrace{(A \vee B)}_{R}}^{Q_1},$$

$$\overbrace{\underbrace{A\overline{B} \vee B}_{S} \vee C \vee D\underbrace{(A\overline{B} \vee B)}_{S}}^{P} \Leftrightarrow \overbrace{\underbrace{A \vee B}_{R} \vee C \vee D\underbrace{(A \vee B)}_{R}}^{Q_2}.$$

As you can see, you are not obliged to replace all occurrences of a sub-expression by some other logically equivalent expression. In fact, you don't have to use the same logical expression for each replacement, you only need to use logically equivalent expressions. Do also note that there is nothing special about the expression P, notably, it need not be a tautology.

X_1	X_2	$(X_1 \wedge X_2) \vee (\neg(X_1 \wedge X_2))$				
0	**0**	**0**	**1**	**1**	**0**	
0	**1**	**0**	**1**	**1**	**0**	
1	**0**	**0**	**1**	**1**	**0**	
1	**1**	**1**	**1**	**0**	**1**	
		1	4	3	2	

Fig. 8–5 A substitution instance of $A \vee (\neg A)$

On the other hand, the more basic **rule of substitution** is applicable only to a tautology. It states: *substituting an arbitrary logic expression for a variable in a tautology provides us with a new logical expression that is itself a tautology.*

Three points should be stressed. Firstly, substitution refers to replacing *variables*, and not *whole expressions*, as would be the case with replacement.

Secondly, note that *each* occurrence of the variable of your choice is replaced. Thirdly, in each substitution instance (i.e., for each variable being replaced) you write *identical (albeit arbitrary) logic expressions.*

To illustrate, let us look at the following example. Given the tautology $A \vee (\neg A)$, let us substitute the arbitrarily chosen expression $(X_1 \wedge X_2)$ for A, obtaining $(X_1 \wedge X_2) \vee (\neg(X_1 \wedge X_2))$. By the rule of substitution, this expression should be a tautology, as verified in Fig. 8–5.

8.5 Logical Reasoning

Logic started in ancient Greece as the art of drawing a **conclusion** from given **premises**. A premise is a declarative sentence, a proposition, as is the conclusion. The premises together with their conclusion pose an **argument**. The conclusion is drawn from *all* premises together, i.e., from the *conjunction of premises.* An argument may have only a single premise; the *'conjunction of premises'* then refers to that premise. When a conclusion can validly be drawn from its premises, the argument is said to be a *good* or **valid argument**. The premises of a valid argument are **sufficient** for drawing or implying the conclusion. The conclusion of a valid argument **follows of necessity** from the premises.

Logical reasoning tells you how to formulate a valid argument, and lets you decide whether a given argument is valid. But what, precisely, is a valid argument? Let us postulate

> **(8.15)** A **valid argument** is a substitution instance of a **logic implication** $P \Rightarrow Q$ in which the antecedent P stands for the conjunction of the premises, and consequent Q stands for the conclusion.

When formulating an argument, the antecedent of a logic implication is substituted by the conjunction of premises (each premise can be, and usually is, a compound proposition), while the consequent (now called the conclusion of the argument) is determined by the logic implication itself as a (compound) statement of one or more of the premises. Note that (8.15) allows you to use *any* logic implication as the basis for a valid argument. Let us look at the following examples, each being a substitution instance of

(8.9)
$$\underbrace{\underbrace{(R \vee S)}_{P_1} \wedge \underbrace{(\neg R)}_{P_2}}_{P} \Rightarrow \underbrace{S}_{Q}.$$

Argument 1: Given the declarative sentences *'A swan is a bird'* and *'A cat is a mammal'*, we refer to these propositions by the abbreviations A_1 and B_1:

$$A_1 \Leftrightarrow \textit{'A swan is a bird'}, \qquad B_1 \Leftrightarrow \textit{'A cat is a mammal'}.$$

Substituting these for R and S ($R \hookleftarrow A_1$, $S \hookleftarrow B_1$) in (8.9) we write the valid argument as

$$\underbrace{(A_1 \vee B_1)}_{P_1} \wedge \underbrace{(\neg A_1)}_{P_2} \Rightarrow B_1,$$

or colloquially (which you will find much easier to read) as:

	'A swan is a bird OR *a cat is a mammal'*	$\ldots P_1$
	AND	
	'A swan is NOT *a bird'*	$\ldots P_2$
Therefore:	*'A cat is a mammal'*	$\ldots Q$

Let me point out that both premises are compound propositions, $P_1 \Leftrightarrow A_1 \vee B_1$ and $P_2 \Leftrightarrow \overline{A}_1$, and that the conjunction of premises is false, $(A_1 \vee B_1)\,\overline{A}_1 \Leftrightarrow (\mathbf{1} \vee \mathbf{1})\,\mathbf{0} \Leftrightarrow \mathbf{0}$, whereas the drawn conclusion is true (*'A cat is a mammal'* $\Leftrightarrow \mathbf{1}$).

Argument 2: Given $A_2 \Leftrightarrow$ *'Earth is a disk'*, and $B_2 \Leftrightarrow$ *'Earth is a sun'*, (8.9) allows us to argue validly:

	'Earth is a disk OR *it is a sun'*
	AND
	'Earth is NOT *a disk'*
Therefore:	*'Earth is a sun'*

In this example, the conjunction of premises is again false, $(A_2 \vee B_2)\overline{A}_2 \Leftrightarrow (\mathbf{0} \vee \mathbf{0})\,\mathbf{1} \Leftrightarrow \mathbf{0}$, whereas this time the drawn conclusion is false (*'Earth is a sun'* $\Leftrightarrow \mathbf{0}$).

Argument 3: Substituting the propositions $A_3 \Leftrightarrow$ *'Humans grow feathers'* for R, and $B_3 \Leftrightarrow$ *'Humans grow hair'* for S in (8.9), we obtain the valid argument:

	'Humans grow feathers OR *they grow hair'*
	AND
	'Humans do NOT *grow feathers'*
Therefore:	*'Humans grow hair'*

For the above argument we note that both the conjunction of the premises $\big((A_3 \vee B_3)\overline{A}_3 \Leftrightarrow (\mathbf{0} \vee \mathbf{1})\mathbf{1} \Leftrightarrow \mathbf{1}\big)$ and the drawn conclusion are true (*'Humans grow hair'* $\Leftrightarrow \mathbf{1}$).

The arguments 1 and 2 are examples of arguments called *unsound.* The conjunction of premises of an **unsound argument** is false allowing you to validly[1)] draw a conclusion that can be either true or false. This is often formulated as: *'You can draw any conclusion (true or false) from false premises'.*

[1)] The phrase *'to validly draw a conclusion'* should remind you that an argument is always a substitution instance of a *logic implication.*

A **sound argument**, on the other hand, is a valid argument *all* premises of which are true — the conclusion of a sound argument is *always true*. If an argument whose premises are all true provides you with a *false conclusion*, it is said to be an **invalid argument**, the (material) implication on which it was based being non-tautological.

No law of astronomy, physics, or any natural science in general is responsible for the truth or falsity of an argument's conclusion as a novice is sometimes apt to believe, possibly due to the word *'therefore'* for the logic implication ($\Rightarrow$). Whether a conclusion is true or false is decided solely on the formal basis of validity. For instance, consider argument 3: The conclusion that *'humans grow hair'* is a formal deduction — it is not a necessary biological antethesis to humans growing feathers.

Now, let us consider how to employ the concept of logical reasoning to formal logic. We shall develop two forms of formal logic, first, propositional logic in Chapter 9, and then predicate logic in Chapter 13. In both cases we start with a set of **axioms** from which we develop **theorems**. Axioms are logic expressions we take to be tautological *per se*. Theorems are logic expressions developed from the axioms in such a way that they themselves (the theorems, that is) are tautological. So, generally speaking, theorems are developed from theorems — if we extend the meaning of theorem to include that of axiom.

The question of course is, what *rule* lets us decide unequivocally when a logic expression developed from theorems is itself a theorem? Such a rule is sometimes called a *metatheorem* as it itself is not part of the logic system. Rather, it says something *about* the system, in our case, how to use it. The answer to the above question is to use logical reasoning which we adapt in the following way. *As axioms are* per se *true, and theorems are provably so, we can use both as true premises P in a valid argument of the form $P \Rightarrow Q$, where Q is then the new theorem.* This rule is usually formulated more formally, and called

(8.16) modus ponens (meaning, method of affirming): If P and $P \Rightarrow Q$ are axioms or theorems of the logic system, then Q is a theorem of the logic system. Or, briefly

$$\text{If } P \Leftrightarrow \mathbf{1} \text{ and } P \Rightarrow Q, \text{ then } Q \Leftrightarrow \mathbf{1}.$$

You will find many examples of the employment of *modus ponens* in the next chapter in which propositional logic is introduced axiomatically.

Chapter 9

Propositional Logic

This chapter introduces you to a systematic way of developing and proving logical formulas and lays down the law on what you may and may not do in manipulating them. It thus lays the basis for all algebraic manipulation, especially for the predicate logic approach introduced in Chapters 12 and 13. It is assumed you have worked through Chapter 8. If the subject matter of this chapter is not to your liking, you might want to restrict your reading to the first two sections.

9.1 Axiomatic Approach to Propositional Logic

The title of this section possibly sounds impressive, if not formidable. But the basic ideas, as so often, are quite simple, and I shall restrict myself to these. Nevertheless, if you want a really masterly and entertaining introduction to *formal systems* (of which propositional logic is but one), try **Hofstadter** [1979].

In principle, **formal systems** work like this: You are given **(a)** a set of **symbols** (for constants, variables, operators, etc.), **(b)** a set of rules stating how to create **well formed strings** using the given symbols, strings that are in our case called **logic expressions**, **(c)** an initial collection of logic expressions or formulas usually called **axioms**, and **(d)** a number of **rules** by which to develop new formulas from the axioms. And this is precisely the object — namely, *to develop new formulas — so-called* **theorems** *— from given ones by applying the stated rules.*

a) We start with an enumeration of the symbols used, **0** for **falsity** and **1** for **truth**, the logical operators, and the left and right parentheses '(' and ')' as technical symbols used to express priority: $\{\mathbf{0}, \mathbf{1}, (,), \neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$. Furthermore, we shall use the variables X and Y (usually with indices) to represent the constants **0** and **1**, the variables A, B, C, D (with or without indices) to stand for any variable X_i, Y_i, or constants, and finally the variables P, Q, R, S, T (also with or without indices) to represent logic expressions.

b) The rules for creating logic expressions, the so-called **formation rules**, were already formulated in Section 8.1, and here, we accept everything written in that section.

c) We can now state the initial laws, the **axioms** (9.1) to (9.17), of propositional logic.

$$A \Rightarrow B \rightarrow A \tag{9.1}$$
$$(A \rightarrow B) \rightarrow A \Rightarrow A \tag{9.2}$$
$$A \rightarrow B \Rightarrow (B \rightarrow C) \rightarrow (A \rightarrow C) \tag{9.3}$$

$$A \wedge B \Rightarrow A \tag{9.4}$$
$$A \wedge B \Rightarrow B \tag{9.5}$$
$$A \rightarrow B \Rightarrow (A \rightarrow C) \rightarrow (A \rightarrow B \wedge C) \tag{9.6}$$

$$A \Rightarrow A \vee B \tag{9.7}$$
$$B \Rightarrow A \vee B \tag{9.8}$$
$$A \rightarrow C \Rightarrow (B \rightarrow C) \rightarrow (A \vee B \rightarrow C) \tag{9.9}$$

$$A \leftrightarrow B \Rightarrow A \rightarrow B \tag{9.10}$$
$$A \leftrightarrow B \Rightarrow B \rightarrow A \tag{9.11}$$
$$A \rightarrow B \Rightarrow (B \rightarrow A) \rightarrow (A \leftrightarrow B) \tag{9.12}$$

$$A \rightarrow B \Rightarrow \neg B \rightarrow \neg A \tag{9.13}$$
$$A \Rightarrow \neg\neg A \tag{9.14}$$
$$\neg\neg A \Rightarrow A \tag{9.15}$$

$$\mathbf{0} \Rightarrow \neg\mathbf{1} \tag{9.16}$$
$$\mathbf{1} \Rightarrow \neg\mathbf{0} \tag{9.17}$$

d) Finally, we have to state the **rules of deduction** which enable us to develop new tautologies — the theorems — from the axioms. There are two such rules, and we encountered both in Sections 8.4 and 8.5.

> **Rule of substitution:** Substituting an arbitrary logic expression for a variable in a tautology provides us with a new logical expression that is itself a tautology.
>
> **Modus ponens:** If you can postulate, or by some form of reasoning prove that a certain IMPLICATION $P \rightarrow Q$ and its antecedent P are tautologies then the consequent Q is also a tautology. Or, briefly

(8.16) if $P \Rightarrow Q$ and $P \Leftrightarrow \mathbf{1}$ then $Q \Leftrightarrow \mathbf{1}$.

We take the liberty of adding the *rule of replacement* of Section 8.4 to this list although, in actual fact, it can be deduced:

rule of replacement: Replacing some or all of the subexpressions S of P by the replacement expression R, logically equivalent to S, transforms P into a new, yet logically equivalent expression, say Q.

Let us now take a closer look at the axioms. First, note that the axioms (9.1) to (9.15) contain no constants (**0** and **1**) and are grouped into three axioms each which, from top to bottom, refer to $\rightarrow$, $\wedge$, $\vee$, $\leftrightarrow$, $\neg$. More importantly, all axioms contain a symbol, the logical implication $\Rightarrow$, that is not a symbol of our formal system. It is called a *metasymbol*, and is used to express something *about* the system, namely, that a certain implication is taken to be true for all input events. It is explained, together with the logical equivalence $\Leftrightarrow$, in Section 8.3. The important thing to note is that *by using the logical implication $\Rightarrow$ in formulating the axioms, we are postulating that each axiom is a tautology.* You will have noticed that the axioms, being of the form $P \Rightarrow Q$, idealy fit our rule for logical reasoning or deduction, namely *modus ponens*.

A general remark seems appropriate. When axiomatising, we don't say *what* we are talking about, rather, we state *how* to talk about it. For instance, in the formal system stated, we do not say what 'truth' or 'falsity' is. They are simply elements of the domains and co-domain of functions we refer to as $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$. Contrary to our previous approach, the functions $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ are not defined by truth tables, as in Fig. 8–1. Rather, they are explained by the axioms (9.1) to (9.17). If we want to possess the truth tables of these functions we shall have to develop them from the axioms. But it is safe to say that we have chosen the function symbols $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ to coincide with their previous interpretation shown in Fig. 8–1.

The axiom system of (9.1) to (9.17) is very weak, meaning, it contains a minimum of information on each of the functions $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$. Therefore proofs of theorems tend to be quite long. Nevertheless, each proof is stated so comprehensively that you should be able to follow the reasoning without the parallel use of paper and pencil, thus being able to save a considerable amount of time.

9.2 Complementation

We start our excursion into complementation by developing two logical implications that are very similar to the axioms (9.16) $\mathbf{0} \Rightarrow \neg\mathbf{1}$ and (9.17) $\mathbf{1} \Rightarrow \neg\mathbf{0}$. We first prove

$$\neg\mathbf{0} \Rightarrow \mathbf{1}. \tag{9.18}$$

To prove this formula, we use (9.1) $A \Rightarrow B \rightarrow A$ in which A **is substituted by 1** (for which we introduce the shorthand $A \hookleftarrow \mathbf{1}$), and B is substituted by $\neg\mathbf{0}$ ($B \hookleftarrow \neg\mathbf{0}$), allowing us to write (9.1) as $\mathbf{1} \Rightarrow \neg\mathbf{0} \rightarrow \mathbf{1}$. As we shall encounter this three-step process — using a stated axiom (or

theorem), substituting for its variables, writing the result of the substitution — again and again, we shall reduce verbosity and simply write the following three successive lines, shown for the example of the previous sentence.

(9.1) $A \Rightarrow B \to A$
$\{A \hookleftarrow \mathbf{1}, B \hookleftarrow \neg\mathbf{0}\}$
$\mathbf{1} \Rightarrow \neg\mathbf{0} \to \mathbf{1}$

As the formula $\mathbf{1} \Rightarrow \neg\mathbf{0} \to \mathbf{1}$ is of the form $P \Rightarrow Q$ with a true antecedent P, we may apply *modus ponens* allowing us to deduce the truth of the consequent Q, writing this as $\neg\mathbf{0} \to \mathbf{1} \Leftrightarrow \mathbf{1}$, and thus proving (9.18).

The second of the logical implications we wish to prove is

$$\neg\mathbf{1} \Rightarrow \mathbf{0}. \tag{9.19}$$

Looking ahead to the next section, we make use of theorem

(9.24) $\neg A \to B \Rightarrow \neg B \to A$
$\{A \hookleftarrow \mathbf{0}, B \hookleftarrow \mathbf{1}\}$
$\underbrace{\neg\mathbf{0} \to \mathbf{1}}_{(9.18)} \Rightarrow \neg\mathbf{1} \to \mathbf{0}$ (due to (9.18) the antecedent is true)
$\mathbf{1} \Rightarrow \neg\mathbf{1} \to \mathbf{0}$ (apply *modus ponens*)
$\neg\mathbf{1} \to \mathbf{0} \Leftrightarrow \mathbf{1}$ (9.19)

We can now show that the function $\neg$ (NEGATION) always maps $\mathbf{0}$ to $\mathbf{1}$ and $\mathbf{1}$ to $\mathbf{0}$ which we express as *inversion* or *complementation*:

$$\neg\mathbf{0} \Leftrightarrow \mathbf{1} \tag{9.20}$$
$$\neg\mathbf{1} \Leftrightarrow \mathbf{0} \tag{9.21}$$

Here is the proof of (9.20), but I leave the proof of (9.21) to you:

(9.12) $A \to B \Rightarrow (B \to A) \to (A \leftrightarrow B)$
$\{A \hookleftarrow \neg\mathbf{0}, B \hookleftarrow \mathbf{1}\}$
$\underbrace{\neg\mathbf{0} \to \mathbf{1}}_{(9.18)} \Rightarrow \underbrace{(\mathbf{1} \to \neg\mathbf{0}) \to (\neg\mathbf{0} \leftrightarrow \mathbf{1})}_{Q}$ (apply *modus ponens*)
Q: $\underbrace{\mathbf{1} \to \neg\mathbf{0}}_{(9.17)} \Rightarrow \neg\mathbf{0} \leftrightarrow \mathbf{1}$ (apply *modus ponens*)
$\neg\mathbf{0} \leftrightarrow \mathbf{1} \Leftrightarrow \mathbf{1}$ (9.20)

The concept of negation, underlying (9.20) and (9.21), cannot be expressed by a logical formula using a variable. The closest we can get is to establish the principle that *double negation leaves an expression unchanged*:

$$A \Leftrightarrow \neg\neg A. \tag{9.22}$$

But to speak of double *negation* is only possible after the concept of negation has been formulated as (9.20) and (9.21). Let us now take (9.22) as an example by which to take a closer look at the proof process. The logic

expression we want to prove must be part of the consequent Q of a tautology $P \Rightarrow Q$. In our case we want to prove $A \leftrightarrow \neg\neg A$ (note that we can only write $A \Leftrightarrow \neg\neg A$ *after* $A \leftrightarrow \neg\neg A$ has been shown to be true). You will notice that the only axiom containing an equivalence '$\leftrightarrow$' in its consequent Q is (9.12). We therefore start with this axiom.

$$\begin{aligned} & A \rightarrow B \Rightarrow (B \rightarrow A) \rightarrow (A \leftrightarrow B) \\ & \{B \hookleftarrow \neg\neg A\} \\ & \underbrace{A \rightarrow \neg\neg A}_{(9.14)} \Rightarrow (\neg\neg A \rightarrow A) \rightarrow (A \leftrightarrow \neg\neg A). \end{aligned} \tag{9.22a}$$

According to the *rule of substitution*, the above logic expression (9.22a) is a tautology as it is derived from a tautology — namely the axiom (9.12) — by substitution. In order to apply *modus ponens* to the logical implication (9.22a), we first note that its antecedent $A \rightarrow \neg\neg A$ is true, as it is axiom (9.14) written as $A \rightarrow \neg\neg A \Leftrightarrow \mathbf{1}$. We can now apply *modus ponens* to (9.22a) allowing us to deduce that its consequent is a tautology:

$$\underbrace{\neg\neg A \rightarrow A}_{(9.15)} \Rightarrow A \leftrightarrow \neg\neg A.$$

By (9.15) the antecedent of this logical implication is also true allowing us, again, to employ *modus ponens* to state the truth of the consequente — $A \leftrightarrow \neg\neg A \Leftrightarrow \mathbf{1}$ — thus proving (9.22).

9.3 IMPLICATION and NEGATION

IMPLICATION together with NEGATION is governed by axiom (9.13) $A \rightarrow B \Rightarrow \neg B \rightarrow \neg A$. We first show that its more stringent form (9.23) also holds

$$A \rightarrow B \Leftrightarrow \neg B \rightarrow \neg A. \tag{9.23}$$

(9.12) $\quad A \rightarrow B \Rightarrow (B \rightarrow A) \rightarrow (A \leftrightarrow B)$

$\{A \hookleftarrow A \rightarrow B,\ B \hookleftarrow \neg B \rightarrow \neg A\}$

$$\underbrace{A \rightarrow B \ .\rightarrow.\ \neg B \rightarrow \neg A}_{(9.13)} \Rightarrow \underbrace{(\neg B \rightarrow \neg A) \rightarrow (A \rightarrow B) \ .\rightarrow.\ (A \rightarrow B) \leftrightarrow (\neg B \rightarrow \neg A)}_{Q_1}$$

The antecedent is a tautology, thus by *modus ponens*:

Q_1: $\quad \underbrace{(\neg B \rightarrow \neg A) \rightarrow (A \rightarrow B)}_{P_2} \Rightarrow \underbrace{(A \rightarrow B) \leftrightarrow (\neg B \rightarrow \neg A)}_{Q_2}$

We now show that P_2 is a logical implication:

(9.13)
$$A \to B \Rightarrow \neg B \to \neg A$$
$$\{A \hookleftarrow \neg B,\ B \hookleftarrow \neg A\}$$
$$\neg B \to \neg A \Rightarrow \underbrace{\neg\neg A} \to \underbrace{\neg\neg B}$$
$$\{\text{replacement} : \neg\neg A \overset{(9.22)}{\Longleftrightarrow} A,\ \neg\neg B \overset{(9.22)}{\Longleftrightarrow} B\}$$
$$\neg B \to \neg A \Rightarrow A \to B$$

allowing us to apply *modus ponens* to Q_2:

Q_2: $(A \to B) \leftrightarrow (\neg B \to \neg A) \Leftrightarrow \mathbf{1}$ ... proving (9.23).

The next two formulas follow effortlessly from (9.23). But if this subject is new to you, try to write the (extremely simple) proofs in the spirit of the ones shown above.

$$\neg A \to B \Leftrightarrow \neg B \to A, \tag{9.24}$$
$$A \to \neg B \Leftrightarrow B \to \neg A. \tag{9.25}$$

9.4 DeMorgan's Theorems

The two DeMorgan theorems

$$\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B, \tag{9.26}$$
$$\neg(A \vee B) \Leftrightarrow \neg A \wedge \neg B \tag{9.27}$$

allow us, with the help of NEGATION, to switch from AND to OR and *vice versa*. Let us work through the lengthy but quite straight forward proof of (9.26).

(9.12)
$$A \to B \Rightarrow (B \to A) \to (A \leftrightarrow B)$$
$$\{A \hookleftarrow \neg(A \wedge B),\ B \hookleftarrow \neg A \vee \neg B\}$$
$$\underbrace{\neg(A \wedge B) \to (\neg A \vee \neg B)}_{P_1} \Rightarrow$$
$$\underbrace{(\neg A \vee \neg B) \to \neg(A \wedge B) \ .\!\!\to .\ \overbrace{\neg(A \wedge B) \leftrightarrow (\neg A \vee \neg B)}^{\text{(9.26) to be proved}}}_{Q_1}$$

We prove the antecedent P_1 in the equivalent form P_1^*:

(9.24)
$$\neg A \to B \Leftrightarrow \neg B \to A$$
$$\{A \hookleftarrow A \wedge B,\ B \hookleftarrow \neg A \vee \neg B\}$$
$$\underbrace{\neg(A \wedge B) \to (\neg A \vee \neg B)}_{P_1} \Leftrightarrow \underbrace{\neg(\neg A \vee \neg B) \to (A \wedge B)}_{P_1^*}$$

and now start the proof of P_1^*:

(9.6) $A \to B \Rightarrow (A \to C) \to (A \to B \wedge C)$
$\{A \hookleftarrow \neg(\neg A \vee \neg B),\ B \hookleftarrow A,\ C \hookleftarrow B\}$
$\underbrace{\neg(\neg A \vee \neg B) \to A}_{P_2} \Rightarrow$

$$\underbrace{\neg(\neg A \vee \neg B) \to B) \to \overbrace{\neg(\neg A \vee \neg B) \to A \vee B)}^{P_1^*}}_{Q_2}.$$

The antecedent P_2 is proved in its equivalent form P_2^*:

(9.24) $\neg A \to B \Leftrightarrow \neg B \to A$
$\{A \hookleftarrow \neg A \vee \neg B,\ B \hookleftarrow A\}$
$\underbrace{\neg(\neg A \vee \neg B) \to A}_{P_2} \Leftrightarrow \underbrace{\neg A \to \neg A \vee \neg B}_{P_2^*}$

the proof of P_2^* following directly from axiom

(9.7) $A \Rightarrow A \vee B$
$\{A \hookleftarrow \neg A,\ B \hookleftarrow \neg B\}$
$\neg A \Rightarrow \neg A \vee \neg B$ ... This proves P_2^* and thus P_2

allowing us to deduce Q_2 from $P_2 \Rightarrow Q_2$ by *modus ponens*:

Q_2: $\underbrace{\neg(\neg A \vee \neg B) \to B}_{P_3} \Rightarrow \underbrace{\neg(\neg A \vee \neg B) \to A \wedge B}_{P_1^*}$

Here too, P_3 is proved in its equivalent form P_3^*:

(9.24) $\neg A \to B \Leftrightarrow \neg B \to A$
$\{A \hookleftarrow \neg A \vee \neg B\}$
$\underbrace{\neg(\neg A \vee \neg B) \to B}_{P_3} \Leftrightarrow \underbrace{\neg B \to (\neg A \vee \neg B)}_{P_3^*}$

The proof of P_3^* following directly from axiom:

(9.8) $B \Rightarrow A \vee B$
$\{A \hookleftarrow \neg A,\ B \hookleftarrow \neg B\}$
$\neg B \Rightarrow \neg A \vee \neg B$ ... This proves P_3^* and thus P_3

allowing us to deduce P_1 from $P_3 \Rightarrow P_1^*$ by *modus ponens*:

P_1: $\neg(A \wedge B) \Rightarrow (\neg A \vee \neg B)$

and this allows us to deduce Q_1 from $P_1 \Rightarrow Q_1$:

Q_1 : $\underbrace{(\neg A \vee \neg B) \to \neg(A \wedge B)}_{P_4} \Rightarrow \underbrace{\neg(A \wedge B) \leftrightarrow (\neg A \vee \neg B)}_{(9.26)}$

To prove P_4 we use axiom (9.9) as follows:

(9.9) $A \to C \Rightarrow (B \to C) \to (A \vee B \to C)$
$\{A \hookleftarrow \neg A,\ B \hookleftarrow \neg B,\ C \hookleftarrow \neg(A \wedge B)\}$
$\underbrace{\neg A \to \neg(A \wedge B)}_{P_5} \Rightarrow$

$$\underbrace{\neg B \to \neg(A \wedge B) .\to. \overbrace{(\neg A \vee \neg B) \to \neg(A \wedge B)}^{P_4}}_{Q_5}$$

To prove P_5 we express it as the consequent of axiom

(9.13) $A \to B \Rightarrow \neg B \to \neg A$
$\{A \hookleftarrow A \wedge B, B \hookleftarrow A\}$
$\underbrace{A \wedge B \to A}_{(9.4)} \Rightarrow \underbrace{\neg A \to \neg(A \wedge B)}_{P_5}$

As the antecedent $A \wedge B \to A$ is true we may deduce P_5:

P_5: $\neg A \Rightarrow \neg(A \wedge B)$

this allowing us to deduce Q_5 from $P_5 \Rightarrow Q_5$:

Q_5: $\underbrace{\neg B \to \neg(A \wedge B)}_{P_6} \Rightarrow \underbrace{(\neg A \vee \neg B) \to \neg(A \wedge B)}_{P_4}$

We must next prove P_6:

(9.13) $A \to B \Rightarrow \neg B \to \neg A$
$\{A \hookleftarrow A \wedge B\}$
$\underbrace{A \wedge B \to B}_{(9.5)} \Rightarrow \underbrace{\neg B \to \neg(A \wedge B)}_{P_6}$

As the antecedent $A \wedge B \to B$ is true we may deduce P_6:

P_6: $\neg B \Rightarrow \neg(A \wedge B)$

and from Q_5: $P_6 \Rightarrow P_4$ we deduce P_4:

P_4: $\neg A \vee \neg B \Rightarrow \neg(A \wedge B)$.

Finally, from Q_1: $P_4 \Rightarrow$ (9.26) we deduce (9.26):

(9.26) $\neg(A \wedge B) \leftrightarrow (\neg A \vee \neg B) \Leftrightarrow \mathbf{1}$.

9.5 Commutativity of AND and OR

Even such seemingly obvious formulas as commutativity of AND and OR

$$A \wedge B \Leftrightarrow B \wedge A \qquad \textbf{(9.28)}$$
$$A \vee B \Leftrightarrow B \vee A \qquad \textbf{(9.29)}$$

need to be proved. Here, we shall only prove the commutativity of the AND function.

(9.12) $A \to B \Rightarrow (B \to A) \to (A \leftrightarrow B)$
$\{A \hookleftarrow A \wedge B, B \hookleftarrow B \wedge A\}$
$\underbrace{(A \wedge B) \to (B \wedge A)}_{P_1} \Rightarrow$

$$\underbrace{(B \wedge A) \to (A \wedge B) \mathrel{.\!\to\!.} \overbrace{(A \wedge B) \leftrightarrow (B \wedge A)}^{\text{(9.28) to be proved}}}_{Q_1}$$

To prove the antecedent P_1 we start with axiom

(9.6) $A \to B \Rightarrow (A \to C) \to (A \to B \wedge C)$
$\{A \leftarrow A \wedge B,\ C \leftarrow A\}$

$$\underbrace{(A \wedge B) \to B}_{(9.5)} \Rightarrow \underbrace{(A \wedge B) \to A \mathrel{.\!\to\!.} \overbrace{(A \wedge B) \to (B \wedge A)}^{P_1}}_{Q_2}$$

The antecedent being true we may deduce

Q_2: $\underbrace{(A \wedge B) \to A}_{(9.4)} \Rightarrow \underbrace{(A \wedge B) \to (B \wedge A)}_{P_1}.$

Again, the antecedent being true we may now deduce

P_1: $A \wedge B \Rightarrow B \wedge A$

allowing us to deduce Q_1 from $P_1 \Rightarrow Q_1$:

Q_1: $\underbrace{(B \wedge A) \to (A \wedge B)}_{P_2} \Rightarrow \underbrace{(A \wedge B) \leftrightarrow (B \wedge A)}_{(9.28)}.$

To prove P_2 we apply the just proved:

P_1: $A \wedge B \Rightarrow B \wedge A$
$\{A \leftarrow B,\ B \leftarrow A\}$
$B \wedge A \Rightarrow A \wedge B$

this allowing us to deduce (9.28) from $P_2 \Rightarrow$ (9.28):

$(A \wedge B) \leftrightarrow (B \wedge A) \Leftrightarrow \mathbf{1}.$

9.6 Logical Implications of IMPLICATIONS

For use in further proofs it is useful to have a number of logical implications at hand, e.g.

$$A \to (A \to B) \Rightarrow A \to B, \tag{9.30}$$
$$A \Rightarrow A, \tag{9.31}$$
$$A \to (B \to C) \Rightarrow B \to (A \to C), \tag{9.32}$$
$$A \to (B \to C) \Rightarrow (A \to B) \to (A \to C), \tag{9.33}$$
$$(A \to B) \to C \Rightarrow (A \to C) \to C. \tag{9.34}$$

For our purposes, let it suffice to prove the first two. We start by adapting axiom

(9.3) $A \to B \Rightarrow (B \to C) \to (A \to C)$

$$\{A \hookleftarrow A \to (A \to B),\ B \hookleftarrow (A \to B) \to B \mathrel{.\to.} A \to B,$$
$$C \hookleftarrow A \to B\}$$
$$\underbrace{A \to (A \to B) \mathrel{.\to:} (A \to B) \to B \mathrel{.\to.} A \to B}_{P_1} \Rightarrow$$
$$\underbrace{((A \to B) \to B \mathrel{.\to.} A \to B \mathrel{:\to.} A \to B)}_{P_2} \to$$
$$\underbrace{(A \to (A \to B) \mathrel{.\to.} A \to B)}_{\text{(9.30) to be proved}}$$

The antecedent P_1 is a substitution instant of axiom

(9.3) $A \to B \Rightarrow (B \to C) \to (A \to C)$
$\{B \hookleftarrow A \to B,\ C \hookleftarrow B\}$
$$\underbrace{A \to (A \to B) \Rightarrow (A \to B) \to B \mathrel{.\to.} A \to B}_{P_1}$$

Having thus proved P_1 we deduce $P_2 \Rightarrow$ (9.30) from above

$$\underbrace{(A \to B) \to B \mathrel{.\to.} A \to B \mathrel{:\to.} A \to B}_{P_2} \Rightarrow$$
$$\underbrace{A \to (A \to B) \mathrel{.\to.} A \to B}_{\text{(9.30) to be proved}}$$

P_2 is a substitution instant of axiom (9.2) and thus true:

(9.2) $(A \to B) \to A \Rightarrow A$
$\{A \hookleftarrow A \to B\}$
$$\underbrace{(A \to B) \to B \mathrel{.\to.} A \to B \Rightarrow A \to B}_{P_2}$$

allowing us to deduce (9.30) from $P_2 \Rightarrow$ (9.30) above:

(9.30) $A \to (A \to B) \mathrel{.\to.} A \to B \Leftrightarrow \mathbf{1}$.

The proof of the next formula

(9.31) $$A \Rightarrow A$$

follows from

(9.30) $A \to (A \to B) \Rightarrow A \to B$
$\{B \hookleftarrow A\}$
$$\underbrace{A \to (A \to A)}_{P_1} \Rightarrow \underbrace{A \to A}_{(9.31)}$$

P_1 is a substitution instant of axiom (9.1) and thus true:

(9.1) $A \Rightarrow B \to A$
$\{B \hookleftarrow A\}$
$A \Rightarrow A \to A$

allowing us to deduce (9.31) from $P_1 \Rightarrow$ (9.31):

(9.31) $A \to A \Leftrightarrow \mathbf{1}$.

9.7 Formulas with a single Variable

The only formula we have developed so far which uses a single variable is $A \Rightarrow A$, i.e., (9.31). Let us here take a look at two more:

$$A \vee A \Leftrightarrow A, \tag{9.35}$$

$$\neg A \vee A \Leftrightarrow \mathbf{1}. \tag{9.36}$$

The proof of (9.35) is simple:

(9.12) $A \to B \Rightarrow (B \to A) \to (A \leftrightarrow B)$
$\{B \leftarrow\!\shortmid A \vee A\}$
$\underbrace{(A \to A \vee A)}_{(9.7)} \Rightarrow \underbrace{(A \vee A \to A) \to (A \leftrightarrow A \vee A)}_{Q_1}$

The antecedent $A \to A \vee A$ being true we may deduce:

Q_1: $A \vee A \to A \Rightarrow A \leftrightarrow A \vee A$

To prove the antecedent $A \vee A \to A$ we use:

(9.9) $A \to C \Rightarrow (B \to C) \to (A \vee B \to C)$
$\{B \leftarrow\!\shortmid A,\ C \leftarrow\!\shortmid A\}$
$\underbrace{A \to A}_{(9.31)} \Rightarrow \underbrace{(A \to A) \to (A \vee A \to A)}_{Q_2}$

The antecedent $A \to A$ (9.31) being true we deduce:

Q_2: $\underbrace{A \to A}_{(9.31)} \Rightarrow \underbrace{A \vee A \to A}_{Q_3}$

Again, the antecedent $A \to A$ (9.31) being true we deduce:

Q_3: $A \vee A \Rightarrow A$,

allowing us to deduce $A \Leftrightarrow A \vee A$ (9.35) from Q_1.

To prove $\neg A \vee A \Leftrightarrow \mathbf{1}$ (9.36) we need

$$A \to B \Leftrightarrow \neg A \vee B. \tag{9.37}$$

Here, this formula will remain unproved so as not to bore you with a proof all too long. The proof of (9.36) can now be written as:

(9.37) $\neg A \vee B \Leftrightarrow A \to B$
$\{B \leftarrow\!\shortmid A\}$
$\neg A \vee A \Leftrightarrow A \to A \overset{(9.31)}{\Longleftrightarrow} \mathbf{1}$.

One thing the formulas of this section can be used for is to develop formulas equivalent to the truth table for OR:

$$
\begin{array}{rll}
A \vee A \Leftrightarrow A & \text{with } A \hookleftarrow \mathbf{0}: & \mathbf{0} \vee \mathbf{0} \Leftrightarrow \mathbf{0}, \\
\neg A \vee A \Leftrightarrow \mathbf{1} & \text{with } A \hookleftarrow \mathbf{1} \text{ and } \neg\mathbf{1} \Leftrightarrow \mathbf{0}: & \mathbf{0} \vee \mathbf{1} \Leftrightarrow \mathbf{1}, \\
\neg A \vee A \Leftrightarrow \mathbf{1} & \text{with } A \hookleftarrow \mathbf{0} \text{ and } \neg\mathbf{0} \Leftrightarrow \mathbf{1}: & \mathbf{1} \vee \mathbf{0} \Leftrightarrow \mathbf{1}, \\
A \vee A \Leftrightarrow A & \text{with } A \hookleftarrow \mathbf{1}: & \mathbf{1} \vee \mathbf{1} \Leftrightarrow \mathbf{1}.
\end{array}
$$

Chapter 10

Summary of Theorems

All the theorems presented in this chapter are *logical equivalencies* allowing you to use them together with the rule of replacement (Section 8.4). To maintain an overall view of the theorems their proofs have been omitted. These proofs are easy to come by when using the truth table method of Section 8.3 (or when using Karnaugh maps as described in Section 7.4). All the logic symbols used in the theorems were introduced in the table of Fig. 4–3.

To be able to discuss general aspects of the theorems, aspects that could apply to any connective, we introduce two symbols, *circle* $\circ$ and *diamond* $\diamond$, that stand for any of the *dyadic* logic connectives $\wedge$, $>$, $<$, $\oplus$, $\vee$, $\tilde{\vee}$, $\leftrightarrow$, $\leftarrow$, $\rightarrow$, $\tilde{\wedge}$.

The theorems of this chapter are *not* intended to be learnt by heart, they are meant as a source of reference and comparison.

10.1 Commutative and Associative Laws *

Commutativity refers to a formula of the following kind

$$A \circ B \Leftrightarrow B \circ A, \tag{10.1}$$

meaning that a dyadic logic connective $\circ$ is called commutative if you get the same result when interchanging the arguments. **Associativity** is described by the formula

$$A \circ (B \circ C) \Leftrightarrow (A \circ B) \circ C \tag{10.2}$$

meaning that, in whichever sequence the arguments of an associative dyadic connective are acted upon, you always get the same result. For an associative dyadic connective it is common to omit the parenthesis, writing $A \circ B \circ C$ instead of $A \circ (B \circ C)$ or $(A \circ B) \circ C$.

Commutativity and associativity (taken together) are of practical importance. They influence the way we employ gates and switching devices, as touched upon in Section 4.4. Commutativity allows you to interchange the

input signals to a gate as you like without being able to notice a change in the behaviour of the output value. Not all logical connectives are commutative, as shown in the table of Fig. 10–1. In particular, the functions that are *not* commutative are the INHIBITIONs and IMPLICATIONs.

Commutativity	
$A \wedge B \Leftrightarrow B \wedge A$	$A \vee B \Leftrightarrow B \vee A$
$A \oplus B \Leftrightarrow B \oplus A$	$A \leftrightarrow B \Leftrightarrow B \leftrightarrow A$
$A \tilde{\vee} B \Leftrightarrow B \tilde{\vee} A$	$A \tilde{\wedge} B \Leftrightarrow B \tilde{\wedge} A$

Fig. **10–1** All the commutative laws

Associativity permits you to cascade identical gate types thus creating a multiple inputs gate of that type. Associativity only holds for a subset of the commutative logic functions, i.e., for the connectives AND, OR, XOR and EQUIVALENCE, as Fig. 10–2 demonstrates.

Associativity	
$A \wedge (B \wedge C) \Leftrightarrow (A \wedge B) \wedge C$	$A \vee (B \vee C) \Leftrightarrow (A \vee B) \vee C$
$A \oplus (B \oplus C) \Leftrightarrow (A \oplus B) \oplus C$	$A \leftrightarrow (B \leftrightarrow C) \Leftrightarrow (A \leftrightarrow B) \leftrightarrow C$

Fig. **10–2** All the associative laws

Notice in Figs. 10–3a,b the correspondence between the parentheses and the way two-input gates are interconnected. Whenever associativity holds, you can leave away the parentheses, as symbolised in Fig. 10–3c.

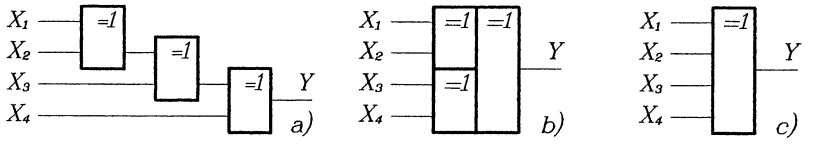


Fig. 10–3 Analogy of formula and circuit

There is a difference in the real world behaviour of the circuits of Figs. 10–3a and 10–3b. Each physical realisation of a gate always causes a time lag between the input and output signals. The propagation time a signal takes from an input to the output of a circuit depends on the number of gates the signal has to pass. The propagation time can vary markedly from input to

input of an asymmetric circuit, whereas it will hardly vary for the various inputs of a symmetric circuit. Symmetrical designs are obviously preferred.

10.2 Single-Variable Formulas

The theoretical importance of single-variable formulas was pointed out in Section 9.7 where we used them to develop formulas equivalent to truth tables. Here we summarise all the single-variable formulas except those for INHIBITION (which you can develop as dual to IMPLICATION). The commutative connectives are considered in dual pairs.

AND	OR
$A \wedge \mathbf{0} \Leftrightarrow \mathbf{0}$	$A \vee \mathbf{1} \Leftrightarrow \mathbf{1}$
$A \wedge \mathbf{1} \Leftrightarrow \mathrm{A}$	$A \vee \mathbf{0} \Leftrightarrow \mathrm{A}$
$A \wedge A \Leftrightarrow \mathrm{A}$	$A \vee A \Leftrightarrow \mathrm{A}$
$A \wedge \overline{A} \Leftrightarrow \mathbf{0}$	$A \vee \overline{A} \Leftrightarrow \mathbf{1}$

Fig. **10–4** The Single-Variable Formulas for AND and OR

The AND and OR functions both have what is called a **neutral element**. To demonstrate what a neutral element is, take the operations of addition and multiplication. Writing $x + 0 = x$ shows that adding zero to any number x leaves the number unchanged — one calls zero the neutral element of addition. Similarly, 1 is the neutral element of multiplication, because multiplying any number x by 1 leaves the number unchanged: $x \times 1 = x$.

Now returning to logic functions, let $\circ$ be some dyadic logic connective. Then, ν is said to be a neutral element if $A \circ \nu = A$. As you can see from the table of Fig. 10–4, the neutral element for the AND function is **1** as follows from $A \wedge \mathbf{1} \Leftrightarrow A$; due to duality, the neutral element of the OR function is **0**.

The second property to point out is *idempotence* of a function. Again, let $\circ$ be some dyadic logic connective. Then the connective $\circ$ is said to be **idempotent** if $A \circ A \Leftrightarrow A$. Both AND and OR are idempotent, meaning $A \wedge A \Leftrightarrow A$ and $A \vee A \Leftrightarrow A$.

EQU	XOR
$A \leftrightarrow \mathbf{0} \Leftrightarrow \overline{A}$	$A \oplus \mathbf{1} \Leftrightarrow \overline{A}$
$A \leftrightarrow \mathbf{1} \Leftrightarrow \mathrm{A}$	$A \oplus \mathbf{0} \Leftrightarrow \mathrm{A}$
$A \leftrightarrow A \Leftrightarrow \mathbf{1}$	$A \oplus A \Leftrightarrow \mathbf{0}$
$A \leftrightarrow \overline{A} \Leftrightarrow \mathbf{0}$	$A \oplus \overline{A} \Leftrightarrow \mathbf{1}$

Fig. **10–5** The Single-Variable Formulas for EQU and XOR

As you can see from Fig. 10–5, EQUIVALENCE and XOR have, respectively, the neutral elements **1** and **0**, i.e., $A \leftrightarrow \mathbf{1} \Leftrightarrow A$ and $A \oplus \mathbf{0} \Leftrightarrow A$. But, note that neither EQUIVALENCE nor XOR are idempotent.

NAND	NOR
$A \mathbin{\tilde{\wedge}} \mathbf{0} \Leftrightarrow \mathbf{1}$	$A \mathbin{\tilde{\vee}} \mathbf{1} \Leftrightarrow \mathbf{0}$
$A \mathbin{\tilde{\wedge}} \mathbf{1} \Leftrightarrow \overline{A}$	$A \mathbin{\tilde{\vee}} \mathbf{0} \Leftrightarrow \overline{A}$
$A \mathbin{\tilde{\wedge}} A \Leftrightarrow \overline{A}$	$A \mathbin{\tilde{\vee}} A \Leftrightarrow \overline{A}$
$A \mathbin{\tilde{\wedge}} \overline{A} \Leftrightarrow \mathbf{1}$	$A \mathbin{\tilde{\vee}} \overline{A} \Leftrightarrow \mathbf{0}$

Fig. **10–6** The Single-Variable Formulas for NAND and NOR

The last pair of commutative logic functions consists of NAND and NOR, their single-variable formulas being found in Fig. 10–6. Neither of these functions has a neutral element, nor are they idempotent. The IMPLICATION (Fig. 10–7), finally, is neither commutative nor idempotent, and has only a left neutral element, this being **1** (i.e., $\mathbf{1} \rightarrow A \Leftrightarrow A$).

IMPLICATION	
$A \rightarrow \mathbf{0} \Leftrightarrow \overline{A}$	$\mathbf{0} \rightarrow A \Leftrightarrow \mathbf{1}$
$A \rightarrow \mathbf{1} \Leftrightarrow \mathbf{1}$	$\mathbf{1} \rightarrow A \Leftrightarrow A$
$A \rightarrow A \Leftrightarrow \mathbf{1}$	$A \rightarrow A \Leftrightarrow \mathbf{1}$
$A \rightarrow \overline{A} \Leftrightarrow \overline{A}$	$\overline{A} \rightarrow A \Leftrightarrow A$

Fig. **10–7** The Single-Variable Formulas for IMPLICATION

10.3 Distributive Laws *

Distributivity refers to a formula that is of one of the following types

$$A \circ (B \diamond C) \Leftrightarrow (A \circ B) \diamond (A \circ C), \tag{10.3}$$

$$(B \diamond C) \circ A \Leftrightarrow (B \circ A) \diamond (C \circ A). \tag{10.4}$$

The terminology when speaking of the first formula (10.3) is '*(the operator)* $\circ$ *distributes over (the operator)* $\diamond$ *from the left*' or '$\circ$ *is left-distributive over* $\diamond$'. If (10.4) holds, one says '$\circ$ *is right-distributive over* $\diamond$'. One refers

to the operator $\circ$ as being *(completely) distributive* over the operator $\diamond$ *iff* $\circ$ distributes over $\diamond$ from the left *and* from the right. You should also note that the two operators, *circle* $\circ$ and *diamond* $\diamond$, may stand for one and the same logic connective.

Complete distributivity	
Left distributivity	Right distributivity
$A \wedge (B \wedge C) \Leftrightarrow (A \wedge B) \wedge (A \wedge C)$	$(B \wedge C) \wedge A \Leftrightarrow (B \wedge A) \wedge (C \wedge A)$
$A \wedge (B > C) \Leftrightarrow (A \wedge B) > (A \wedge C)$	$(B > C) \wedge A \Leftrightarrow (B \wedge A) > (C \wedge A)$
$A \wedge (B < C) \Leftrightarrow (A \wedge B) < (A \wedge C)$	$(B < C) \wedge A \Leftrightarrow (B \wedge A) < (C \wedge A)$
$A \wedge (B \oplus C) \Leftrightarrow (A \wedge B) \oplus (A \wedge C)$	$(B \oplus C) \wedge A \Leftrightarrow (B \wedge A) \oplus (C \wedge A)$
$A \wedge (B \vee C) \Leftrightarrow (A \wedge B) \vee (A \wedge C)$	$(B \vee C) \wedge A \Leftrightarrow (B \wedge A) \vee (C \wedge A)$
$A \vee (B \vee C) \Leftrightarrow (A \vee B) \vee (A \vee C)$	$(B \vee C) \vee A \Leftrightarrow (B \vee A) \vee (C \vee A)$
$A \vee (B \leftarrow C) \Leftrightarrow (A \vee B) \leftarrow (A \vee C)$	$(B \leftarrow C) \vee A \Leftrightarrow (B \vee A) \leftarrow (C \vee A)$
$A \vee (B \rightarrow C) \Leftrightarrow (A \vee B) \rightarrow (A \vee C)$	$(B \rightarrow C) \vee A \Leftrightarrow (B \vee A) \rightarrow (C \vee A)$
$A \vee (B \leftrightarrow C) \Leftrightarrow (A \vee B) \leftrightarrow (A \vee C)$	$(B \leftrightarrow C) \vee A \Leftrightarrow (B \vee A) \leftrightarrow (C \vee A)$
$A \vee (B \wedge C) \Leftrightarrow (A \vee B) \wedge (A \vee C)$	$(B \wedge C) \vee A \Leftrightarrow (B \vee A) \wedge (C \vee A)$

Fig. **10–8** Left *and* right distributivity

The term distributivity is thus ambiguous. You have to state which two connectives are concerned, which connective distributes over the other, and whether the connective that distributes does so from the left or from the right. As shown in the table of Fig. 10–8, only the AND ($\wedge$) and OR ($\vee$) functions distribute completely over other, but not all other, connectives.

AND, apart from being self-distributive, distributes only over the INHIBITIONs, XOR, and OR. It does not distribute over EQUIVALENCE and over the IMPLICATIONs. The OR function, on the other hand, does distribute over the IMPLICATIONs and EQUIVALENCE, but not over the INHIBITIONs and XOR. Do notice that NAND and NOR play no role whatsoever in connection with distributivity, whether complete or one-sided distributivity.

The AND and OR functions are the only ones that distribute completely, but of all the other logic functions only the INHIBITION and IMPLICATION distribute at least one-sidedly. The INHIBITIONs distribute over the same functions as the AND function, while the IMPLICATIONs distribute over the same functions as the OR function.

Although it is probably obvious, let us take a look at what the empty fields in the table of Fig. 10–9 mean. Consider the first formula, $A < (B \wedge C) \Leftrightarrow (A < B) \wedge (A < C)$, in which $<$ is left-distributive over $\wedge$. If $<$ were right-distributive over $\wedge$, which it is not, we would be able to write the

One sided distributivity	
Left distributivity	Right distributivity
$A<(B\wedge C) \Leftrightarrow (A<B)\wedge(A<C)$ $A<(B>C) \Leftrightarrow (A<B)>(A<C)$ $A<(B<C) \Leftrightarrow (A<B)<(A<C)$ $A<(B\oplus C) \Leftrightarrow (A<B)\oplus(A<C)$ $A<(B\vee C) \Leftrightarrow (A<B)\vee(A<C)$	
	$(B\wedge C)>A \Leftrightarrow (B>A)\wedge(C>A)$ $(B>C)>A \Leftrightarrow (B>A)>(C>A)$ $(B<C)>A \Leftrightarrow (B>A)<(C>A)$ $(B\oplus C)>A \Leftrightarrow (B>A)\oplus(C>A)$ $(B\vee C)>A \Leftrightarrow (B>A)\vee(C>A)$
$A\rightarrow(B\vee C) \Leftrightarrow (A\rightarrow B)\vee(A\rightarrow C)$ $A\rightarrow(B\leftarrow C) \Leftrightarrow (A\rightarrow B)\leftarrow(A\rightarrow C)$ $A\rightarrow(B\rightarrow C) \Leftrightarrow (A\rightarrow B)\rightarrow(A\rightarrow C)$ $A\rightarrow(B\leftrightarrow C) \Leftrightarrow (A\rightarrow B)\leftrightarrow(A\rightarrow C)$ $A\rightarrow(B\wedge C) \Leftrightarrow (A\rightarrow B)\wedge(A\rightarrow C)$	
	$(B\vee C)\leftarrow A \Leftrightarrow (B\leftarrow A)\vee(C\leftarrow A)$ $(B\leftarrow C)\leftarrow A \Leftrightarrow (B\leftarrow A)\leftarrow(C\leftarrow A)$ $(B\rightarrow C)\leftarrow A \Leftrightarrow (B\leftarrow A)\rightarrow(C\leftarrow A)$ $(B\leftrightarrow C)\leftarrow A \Leftrightarrow (B\leftarrow A)\leftrightarrow(C\leftarrow A)$ $(B\wedge C)\leftarrow A \Leftrightarrow (B\leftarrow A)\wedge(C\leftarrow A)$

Fig. **10–9** One-sided Distributivity

formula $(B \wedge C) < A \Leftrightarrow (B < A) \wedge (C < A)$ next to the previous one. But, please do note that the formula just written is wrong, does not exist. The empty fields, therefore, remind us that the alternatively distributive formulas do not exist.

10.4 Generalised DeMorgan Theorems

The original theorems of DeMorgan — (9.26) and (9.27), the top formulas of Fig. 10–10 — refer to the negation of AND and OR. Similar formulas, shown in Fig. 10–10, state how to negate *each* dyadic logic function: *You negate*

DeMorgan's Theorems

$\overline{A \wedge B} \Leftrightarrow \overline{A} \vee \overline{B}$	$\overline{A \vee B} \Leftrightarrow \overline{A} \wedge \overline{B}$
$\overline{A > B} \Leftrightarrow \overline{A} \leftarrow \overline{B}$	$\overline{A \leftarrow B} \Leftrightarrow \overline{A} > \overline{B}$
$\overline{A < B} \Leftrightarrow \overline{A} \rightarrow \overline{B}$	$\overline{A \rightarrow B} \Leftrightarrow \overline{A} < \overline{B}$
$\overline{A \oplus B} \Leftrightarrow \overline{A} \leftrightarrow \overline{B}$	$\overline{A \leftrightarrow B} \Leftrightarrow \overline{A} \oplus \overline{B}$
$\overline{A \,\tilde{\vee}\, B} \Leftrightarrow \overline{A} \,\tilde{\wedge}\, \overline{B}$	$\overline{A \,\tilde{\wedge}\, B} \Leftrightarrow \overline{A} \,\tilde{\vee}\, \overline{B}$

Fig. **10–10** The set of DeMorgan's Theorems

each variable and replace the dyadic operator by its dual operator. The dual operators are listed in Fig. 5–11.

For the four associative operators, DeMorgan's theorems can be extended to multiple variables as shown in Fig. 10–11. To complement or negate any given Boolean expression you can quite conveniently employ the following method, called the **rule of negation**: *Replace every variable and constant by its complement and each connective by its dual connective, but take care to maintain all parentheses.* For some logic function F this is often written symbolically in the following way:

$$\neg F(X_1, X_2, \cdots, X_n; \mathbf{0}, \mathbf{1}; (,); \wedge, >, <, \oplus, \vee, \tilde{\vee}, \leftrightarrow, \leftarrow, \rightarrow, \tilde{\wedge}) \Leftrightarrow$$
$$F(\overline{X}_1, \overline{X}_2, \cdots, \overline{X}_n; \mathbf{1}, \mathbf{0}; (,); \vee, \leftarrow, \rightarrow, \leftrightarrow, \wedge, \tilde{\wedge}, \oplus, >, <, \tilde{\vee}) \qquad (10.5)$$

Generalised DeMorgan Theorems

$$\overline{X_1 \wedge X_2 \wedge \cdots \wedge X_n} \Leftrightarrow \overline{X}_1 \vee \overline{X}_2 \vee \cdots \vee \overline{X}_n$$
$$\overline{X_1 \vee X_2 \vee \cdots \vee X_n} \Leftrightarrow \overline{X}_1 \wedge \overline{X}_2 \wedge \cdots \wedge \overline{X}_n$$
$$\overline{X_1 \oplus X_2 \oplus \cdots \oplus X_n} \Leftrightarrow \overline{X}_1 \leftrightarrow \overline{X}_2 \leftrightarrow \cdots \leftrightarrow \overline{X}_n$$
$$\overline{X_1 \leftrightarrow X_2 \leftrightarrow \cdots \leftrightarrow X_n} \Leftrightarrow \overline{X}_1 \oplus \overline{X}_2 \oplus \cdots \oplus \overline{X}_n$$

Fig. **10–11** DeMorgan's theorems for n variables

The following example demonstrates how to use (10.5) to negate a given logic expression and how important it is to maintain the correct sequence of operation. Given

$$\overline{((A \vee \overline{C}D)AB \vee C)\overline{D}} \Leftrightarrow$$

Basic theorems on negation	
$\overline{\mathbf{0}} \Leftrightarrow \mathbf{1}$	$\overline{\mathbf{1}} \Leftrightarrow \mathbf{0}$
$\overline{\overline{A}} \Leftrightarrow A$	
DeMorgan's theorems	
$\overline{A \wedge B} \Leftrightarrow \overline{A} \vee \overline{B}$	$\overline{A \vee B} \Leftrightarrow \overline{A} \wedge \overline{B}$
Basic theorems on a single variable	
$A \wedge \mathbf{0} \Leftrightarrow \mathbf{0}$	$A \vee \mathbf{1} \Leftrightarrow \mathbf{1}$
$A \wedge \mathbf{1} \Leftrightarrow A$	$A \vee \mathbf{0} \Leftrightarrow A$
$A \wedge A \Leftrightarrow A$	$A \vee A \Leftrightarrow A$
$A \wedge \overline{A} \Leftrightarrow \mathbf{0}$	$A \vee \overline{A} \Leftrightarrow \mathbf{1}$
Commutativity	
$A \wedge B \Leftrightarrow B \wedge A$	$A \vee B \Leftrightarrow B \vee A$
Associativity	
$A \wedge (B \wedge C) \Leftrightarrow (A \wedge B) \wedge C$	$A \vee (B \vee C) \Leftrightarrow (A \vee B) \vee C$
Distributivity	
$A \wedge (B \vee C) \Leftrightarrow (A \wedge B) \vee (A \wedge C)$	$A \vee (B \wedge C) \Leftrightarrow (A \vee B) \wedge (A \vee C)$
Some theorems on minimisation	
$(A \vee B) \wedge (A \vee \overline{B}) \Leftrightarrow A$	$(A \wedge B) \vee (A \wedge \overline{B}) \Leftrightarrow A$
$A \wedge (A \vee B) \Leftrightarrow A$	$A \vee (A \wedge B) \Leftrightarrow A$
$A \wedge (\overline{A} \vee B) \Leftrightarrow A \wedge B$	$A \vee (\overline{A} \wedge B) \Leftrightarrow A \vee B$

Fig. **10–12** Basic Theorems on AND, OR and NOT

we first reinstate the $\wedge$-operators, thus making sure we shan't miss any operators when switching to the dual forms:

$$\overline{\big((A \vee \overline{C} \wedge D) \wedge A \wedge B \vee C\big) \wedge \overline{D}} \Leftrightarrow$$

In the next step we explicitly state the sequence of operation by inserting braces, thereby going back on the *convention of adhesion* (Section 8.1).

$$\overline{\big(\{(A \vee \{\overline{C} \wedge D\}) \wedge A \wedge B\} \vee C\big) \wedge \overline{D}} \Leftrightarrow$$

To this logic expression we can now apply the *rule of negation* (10.5):

$$\big(\{(\overline{A} \wedge \{C \vee \overline{D}\}) \vee \overline{A} \vee \overline{B}\} \wedge \overline{C}\big) \vee D \Leftrightarrow$$

Finally, we simplify the notation of this logic expression by reintroducing the *conventions of adhesion* and *of omitting the* $\wedge$*-connective*:

$$\{\overline{A}\{C \vee \overline{D}\} \vee \overline{A} \vee \overline{B}\}\overline{C} \vee D.$$

With a bit of practice, you can deduce the negated formula directly, i.e., without writingany of the intermediate steps.

10.5 Basic Theorems on AND, OR and NOT

When developing circuits, you will most frequently be using AND, OR, and NOT connectives in various combinations. It is therefore convenient to possess a summery of the basic theorems for these connectives as given in Fig. 10–12. All these theorems have been discussed and/or proved except the last two. I strongly recommend that you prove them with the help of truth tables *and* K-maps.

Chapter 11

Algebraic Proofs

Are the algebraic methods developed in Chapters 9 and 10 a necessity, or are they merely a matter of mathematical aesthetics? Is it possible to do without these methods, and to rely instead and exclusively on truth tables and K-maps? These questions seem fair considering that all the theorems in previous chapters can be proved by employing either a truth table or a K-map. But wait, there is a group of formulas that does *not* fit this picture: *Formulas on an undefined number of variables* such as the generalised DeMorgan theorems of Fig. 10–11. The only way to prove such formulas is algebraically.

The examples I wish to cover in this chapter are centred around minterms and maxterms, these being expressions with an arbitrary number of input variables. For space reasons, the headers of the sections on minterms and maxterms usually only refer to one of two possible formulas, the second formula then being the complement of the one mentioned in the header. To demonstrate the technique of *mathematical induction* we shall also prove one of the generalised DeMorgan theorems.

11.1 Min- and Maxterms are Complementary *

The reason for this section is to stress the necessity to prove even a seemingly obvious relationship, such as

$$D_e(x_1, x_2, \cdots, x_n) \Leftrightarrow \overline{C_e(x_1, x_2, \cdots, x_n)}. \qquad \textbf{(11.1)}$$

Although you can find the above formula in almost any book on switching algebra, I dare say you won't find it proved. The reason for this omission is that, conventionally, neither minterms nor maxterms are defined formally, thus making it impossible to *prove* any relationship between them.

On the other hand, in this presentation, we need only refer to the two general formulas on minterms and maxterms in the form

(5.3) $\quad C_e(x_1, x_2, \cdots, x_n) \Leftrightarrow (x_1, x_2, \cdots, x_n) = (e_1, e_2, \cdots, e_n),$

(5.7) $\quad D_e(x_1, x_2, \cdots, x_n) \Leftrightarrow (x_1, x_2, \cdots, x_n) \neq (e_1, e_2, \cdots, e_n),$

to see that $C_e(\mathbf{x})$ and $D_e(\mathbf{x})$ are complementary for any number n of input variables.

11.2 Disjunction of all Minterms *

The prime formulas on min- and maxterms are

$$\bigvee_{e=0}^{2^n-1} C_e(x_1, x_2, \cdots, x_n) \Leftrightarrow \bigwedge_{i=1}^{n} (\overline{X}_i \vee X_i) \Leftrightarrow \mathbf{1}, \tag{11.2}$$

$$\bigwedge_{e=0}^{2^n-1} D_e(x_1, x_2, \cdots, x_n) \Leftrightarrow \bigvee_{i=1}^{n} \overline{X}_i X_i \Leftrightarrow \mathbf{0}. \tag{11.3}$$

Before proving these formulas, let us try to develop a feeling for how such formulas could have originated. In Section 6.2 — in the paragraph following equation (6.11) — you may remember having read the obscure remark that '*the full area of the Karnaugh map represents* **1**'. The intuitive consequence of this is the assumption that the disjunction of all minterms is **1** as the disjunction of all minterms covers the whole area of the K-map. For one and for two variables this is pictured in Fig. 11–1 for which we shall verify this assumption algebraically.

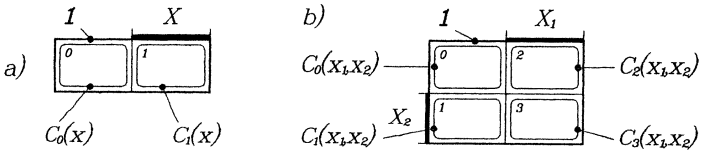


Fig. 11–1 The disjunction of all minterms is **1**

For one input variable x the disjunction of all minterms is:

$$C_0(x) \vee C_1(x) \Leftrightarrow \overline{X} \vee X \overset{(9.36)}{\Longleftrightarrow} \mathbf{1}. \tag{11.4}$$

For two input variables x_1,x_2 you can either read the minterms from the K-map of Fig. 11–1b or refer to Fig. 5–2. Their disjunction is calculated as follows:

$$C_0(x_1, x_2) \vee C_1(x_1, x_2) \vee C_2(x_1, x_2) \vee C_3(x_1, x_2) \Leftrightarrow \tag{11.5a}$$

$$\underbrace{\overline{X}_1\overline{X}_2 \vee \overline{X}_1 X_2} \vee \underbrace{X_1\overline{X}_2 \vee X_1 X_2} \Leftrightarrow \tag{11.5b}$$

Applying distributivity to the underbraced expressions allows us to write

$$\overline{X}_1(\overline{X}_2 \vee X_2) \vee X_1(\overline{X}_2 \vee X_2) \Leftrightarrow \tag{11.5c}$$

$$\underbrace{(\overline{X}_1 \vee X_1)}_{1} \wedge \underbrace{(\overline{X}_2 \vee X_2)}_{1} \Leftrightarrow \mathbf{1}. \tag{11.5d}$$

These two examples pose a strong case for the conjecture that for *any* number of input variables *the disjunction of all minterms is* **1**. Let us try to prove this assumption. To carry out the calculation

$$\bigvee_{e=0}^{2^n-1} C_e(x_1, x_2, \cdots, x_n) \Leftrightarrow \tag{11.6a}$$

we employ the *general minterm formula*

$$C_e(x_1, x_2, \cdots, x_n) \Leftrightarrow \bigwedge_{i=1}^{n} (x_i = e_i) \qquad \text{with} \qquad e = \sum_{i=1}^{n} e_i . 2^{n-i} \tag{5.4}$$

to substitute $C_e(x_1, \cdots, x_n)$ of (11.6a) by $(x_1 = e_1) \wedge \cdots \wedge (x_n = e_n)$, obtaining

$$\bigvee_{e} (x_1 = e_1) \wedge (x_2 = e_2) \wedge \cdots \wedge (x_n = e_n) \Leftrightarrow \tag{11.6b}$$

The multiple disjunction over all integers $e \in \{0, 1, \ldots, 2^n - 1\}$ must be adapted to the notation $(x_1 = e_1) \wedge \cdots \wedge (x_n = e_n)$ in which the integer e is expressed in binary form. Each combination of 0s and 1s in the n-tuple $(e_1, \ldots, e_n)$ represents one of the integers $e \in \{0, 1, \ldots, 2^n - 1\}$, allowing us to replace e in (11.6b) by $(e_1, \ldots, e_n)$:

$$\bigvee_{(e_1, e_2, \ldots, e_n)} (x_1 = e_1) \wedge (x_2 = e_2) \wedge \cdots \wedge (x_n = e_n) \Leftrightarrow \tag{11.6c}$$

It should be noted that the abbreviated multiple disjunction $\bigvee_{e_1, e_2, \ldots, e_n}$ stands for $\bigvee_{e_1} \bigvee_{e_2} \ldots \bigvee_{e_n}$ allowing us to rewrite the above expression as

$$\bigvee_{e_1} \bigvee_{e_2} \ldots \bigvee_{e_n} (x_1 = e_1) \wedge (x_2 = e_2) \wedge \cdots \wedge (x_n = e_n) \Leftrightarrow \tag{11.6d}$$

Distributing the multiple disjunctions provides us with the interim result

$$\Big(\bigvee_{e_1} x_1 = e_1\Big) \wedge \Big(\bigvee_{e_2} x_2 = e_2\Big) \wedge \cdots \wedge \Big(\bigvee_{e_n} x_n = e_n\Big). \tag{11.6e}$$

Let us take a closer look at the meaning of the notation $\bigvee_{e_i} x_i = e_i$ in which $e_i \in \{0, 1\}$:

$$\bigvee_{e_i} x_i = e_i \Leftrightarrow \bigvee_{e_i \in \{0,1\}} x_i = e_i \Leftrightarrow (x_i = 0) \vee (x_i = 1) \Leftrightarrow \overline{X}_i \vee X_i \Leftrightarrow \mathbf{1}. \tag{11.7}$$

Finally, combining (11.6a), (11.6e), and (11.7) we can state how to calculate the disjunction of all minterms, and that the result is **1** for any number n of input variables

$$\bigvee_{e=0}^{2^n-1} C_e(x_1, x_2, \cdots, x_n) \Leftrightarrow (\overline{X}_1 \vee X_1) \wedge (\overline{X}_2 \vee X_2) \wedge \cdots \wedge (\overline{X}_n \vee X_n) \Leftrightarrow \mathbf{1}$$

thus obtaining (11.2). Its negation is (11.3).

11.3 Conjunction of two Minterms *

The two formulas we want to prove here are

$$C_e(\mathbf{x}) \wedge C_\varepsilon(\mathbf{x}) \Leftrightarrow \begin{cases} \mathbf{0} & \ldots e \neq \varepsilon \\ C_e(\mathbf{x}) & \ldots e = \varepsilon \end{cases}, \tag{11.8}$$

$$D_e(\mathbf{x}) \vee D_\varepsilon(\mathbf{x}) \Leftrightarrow \begin{cases} \mathbf{1} & \ldots e \neq \varepsilon \\ D_e(\mathbf{x}) & \ldots e = \varepsilon \end{cases}. \tag{11.9}$$

The two formulas are complementary so that it suffices to prove one of them, say, (11.8). As in the previous section, we start by employing the *general minterm formula* (5.4) to expand the minterms $C_e(\mathbf{x})$ and $C_\varepsilon(\mathbf{x})$:

$$\begin{aligned} &C_e(x_1, x_2, \cdots, x_n) \wedge C_\varepsilon(x_1, x_2, \cdots, x_n) \Leftrightarrow \\ &\quad \big((x_1 = e_1) \wedge (x_2 = e_2) \wedge \cdots \wedge (x_n = e_n)\big) \wedge \\ &\quad \big((x_1 = \varepsilon_1) \wedge (x_2 = \varepsilon_2) \wedge \cdots \wedge (x_n = \varepsilon_n)\big) \Leftrightarrow \end{aligned}$$

We now reorder the above expression to bring equal indices together

$$\begin{aligned} &\big((x_1 = e_1) \wedge (x_1 = \varepsilon_1)\big) \wedge \big((x_2 = e_2) \wedge (x_2 = \varepsilon_2)\big) \wedge \cdots \wedge \\ &\quad \big((x_n = e_n) \wedge (x_n = \varepsilon_n)\big) \Leftrightarrow \end{aligned}$$

(a) If $e = \varepsilon$ then all $e_i = \varepsilon_i$ so that $(x_i = e_i) \wedge (x_i = \varepsilon_i) \Leftrightarrow (x_i = e_i)$ for all i thus proving

$$C_e(\mathbf{x}) \wedge C_e(\mathbf{x}) \Leftrightarrow C_e(\mathbf{x}).$$

(b) If $e \neq \varepsilon$ then there exists at least one conjunction, say $(x_i = e_i) \wedge (x_i = \varepsilon_i)$, for which $e_i \neq \varepsilon_i$. Then, referring to $x_i = e_i$ as A_i, the term $x_i = \varepsilon_i$ must be the negation of A_i, i.e. $\overline{A}_i$. It therefore follows that $(x_i = e_i) \wedge (x_i = \varepsilon_i) \Leftrightarrow A_i \wedge \overline{A}_i \Leftrightarrow \mathbf{0}$. Furthermore, it suffices for a single conjunction $(x_i = e_i) \wedge (x_i = \varepsilon_i)$ to be **0** to make the whole expression

$$\begin{aligned} &\big((x_1 = e_1) \wedge (x_1 = \varepsilon_1)\big) \wedge \big((x_2 = e_2) \wedge (x_2 = \varepsilon_2)\big) \wedge \cdots \wedge \\ &\quad \underbrace{\big((x_i = e_i) \wedge (x_i = \varepsilon_i)\big)}_{\mathbf{0}} \wedge \cdots \wedge \big((x_n = e_n) \wedge (x_n = \varepsilon_n)\big) \end{aligned}$$

false, thus proving $C_e(\mathbf{x}) \wedge C_\varepsilon(\mathbf{x}) \Leftrightarrow \mathbf{0}$.

11.4 Maxterm as the Disjunction of Minterms *

The two formulas to be proved in this section are

$$D_e(\mathbf{x}) \Leftrightarrow \bigvee_{\varepsilon \in \mathcal{E} \setminus \{e\}} C_\varepsilon(\mathbf{x}), \tag{11.10}$$

$$C_e(\mathbf{x}) \Leftrightarrow \bigwedge_{\varepsilon \in \mathcal{E} \setminus \{e\}} D_\varepsilon(\mathbf{x}). \tag{11.11}$$

To understand these formulas, we need to understand the range $\mathcal{E} \setminus \{e\}$ over which the multiple OR and multiple AND are explained. The symbol $\mathcal{E}$ — a calligraphic E — is defined as the set $\mathcal{E} \stackrel{df}{=} \{0, 1, 2, \ldots, 2^n - 1\}$ of all 2^n decimal equivalents e of the input events $(e_1, e_2, \ldots, e_n)$, with $e_i \in \{0, 1\}$. Excluding a certain subset $\mathcal{E}_S$ — with $\mathcal{E}_S \subset \mathcal{E}$ — from the set $\mathcal{E}$ is written $\mathcal{E} \setminus \mathcal{E}_S$. To exclude a single element e from $\mathcal{E}$, you exclude the subset $\{e\}$, the only element of which is e, from $\mathcal{E}$, and write this as $\mathcal{E} \setminus \{e\}$. As the above formulas, (11.10) and (11.11), are complementary, it suffices to prove one of them, say (11.10).

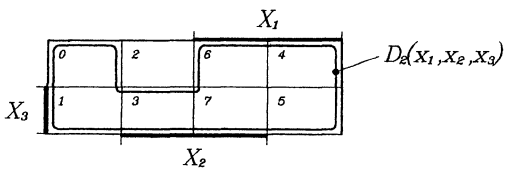


Fig. 11–2 K-map for $D_2 \Leftrightarrow C_0 \vee C_1 \vee C_3 \vee C_4 \vee C_5 \vee C_6 \vee C_7$

The idea that such a relationship could be valid, stems from a K-map such as the one shown in Fig. 11–2. It shows that $D_2(x_1, x_2, x_3)$ is the disjunction of all minterms except $C_2(x_1, x_2, x_3)$

$$\begin{aligned} D_2(x_1, x_2, x_3) \Leftrightarrow & C_0(x_1, x_2, x_3) \vee C_1(x_1, x_2, x_3) \vee C_3(x_1, x_2, x_3) \vee \\ & C_4(x_1, x_2, x_3) \vee C_5(x_1, x_2, x_3) \vee C_6(x_1, x_2, x_3) \vee \\ & C_7(x_1, x_2, x_3) \end{aligned} \tag{11.12}$$

It is of course a simple matter to prove a special case, like the one above, algebraically:

$$
\begin{aligned}
D_2(x_1,x_2,x_3) \Leftrightarrow & X_1 \vee \overline{X}_2 \vee X_3 \Leftrightarrow \\
& X_1(\overline{X}_2 \vee X_2)(\overline{X}_3 \vee X_3) \vee \\
& (\overline{X}_1 \vee X_1)\overline{X}_2(\overline{X}_3 \vee X_3) \vee \\
& (\overline{X}_1 \vee X_1)(\overline{X}_2 \vee X_2)X_3 \Leftrightarrow \\
& \underbrace{X_1\overline{X}_2\overline{X}_3}_{C_4} \vee \underbrace{X_1\overline{X}_2X_3}_{C_5} \vee \underbrace{X_1X_2\overline{X}_3}_{C_6} \vee \underbrace{X_1X_2X_3}_{C_7} \vee \\
& \underbrace{\overline{X}_1\overline{X}_2\overline{X}_3}_{C_0} \vee \underbrace{\overline{X}_1\overline{X}_2X_3}_{C_1} \vee \underbrace{X_1\overline{X}_2\overline{X}_3}_{C_4} \vee \underbrace{X_1\overline{X}_2X_3}_{C_5} \vee \\
& \underbrace{\overline{X}_1\overline{X}_2X_3}_{C_1} \vee \underbrace{\overline{X}_1X_2X_3}_{C_3} \vee \underbrace{X_1\overline{X}_2X_3}_{C_5} \vee \underbrace{X_1X_2X_3}_{C_7}.
\end{aligned}
$$

Dropping duplicates of minterms — i.e., dropping C_4 once, C_5 twice, C_7 once, and C_1 once — (according to $A \vee A \Leftrightarrow A$) leads to (11.12), the special case we wanted to prove.

Now to the proof of the general formula (11.10). Pondering the above example lets it seem likely that generalising the approach taken there, has a rather lean prospect of success. But, a formal proof can take on many guises, e.g., as shown here.

As $D_e(\mathbf{x}) \Leftrightarrow \overline{C}_e(\mathbf{x})$ and $C_e(\mathbf{x}) \Leftrightarrow x = \varepsilon$ we can write (11.10) as

$$
x \neq e \Leftrightarrow \bigvee_{\varepsilon \in \mathcal{E} \setminus \{e\}} x = \varepsilon. \tag{11.13}
$$

While x is an arbitrary element of $\mathcal{E} = \{0, 1, 2, \ldots, e, \ldots, 2^n - 1\}$, e refers to a specific element of $\mathcal{E}$. As x — according to the left side of (11.13) — is required to be unequal to e, x must equal some other element of $\mathcal{E}$, say ε. Removing e from $\mathcal{E}$ gives us the set $\mathcal{E} \setminus \{e\}$, and it is over this set that we let ε vary: $\varepsilon \in \mathcal{E} \setminus \{e\}$. Now, (11.13) is merely the formal notation of the simple truth that if $x \neq e$ there must exist (at least) one $\varepsilon \in \mathcal{E} \setminus \{e\}$ to which x is equal. Accepting this as a proof of (11.13) it only remains to be said that (11.13) can be rewritten as (11.10) which too is hereby proved.

11.5 Minterm AND/OR Maxterm*

The formulas to be proved in this section are (11.14) and (11.15). As one is the negation of the other, it suffices to prove, say, the first one.

$$
C_e(\mathbf{x}) \wedge D_\varepsilon(\mathbf{x}) \Leftrightarrow \begin{cases} C_e(\mathbf{x}) & \ldots e \neq \varepsilon \\ \mathbf{0} & \ldots e = \varepsilon \end{cases}, \tag{11.14}
$$

$$C_e(\mathbf{x}) \vee D_\varepsilon(\mathbf{x}) \Leftrightarrow \begin{cases} D_\varepsilon(\mathbf{x}) & \ldots e \neq \varepsilon \\ \mathbf{1} & \ldots e = \varepsilon \end{cases}. \tag{11.15}$$

Before considering the general case, let us first use the minterm $C_2(x_1, x_2, x_3)$ and maxterm $D_2(x_1, x_2, x_3)$ to illustrate (11.14) for equal indices. Referring to Fig. 11–2, you will notice that $C_2(x_1, x_2, x_3)$ and $D_2(x_1, x_2, x_3)$ represent disjoint areas so that

$$C_2(x_1, x_2, x_3) \wedge D_2(x_1, x_2, x_3) \Leftrightarrow \mathbf{0},$$

just as (11.14) requires. On the other hand, in those cases where the indices differ, the area representing one of the $C_i(x_1, x_2, x_3)$, with i an element of $\{0, 1, 3, 4, 5, 6, 7\}$, is always a subset of the area of $D_2(x_1, x_2, x_3)$. The intersection of these areas is always the area of $C_i(x_1, x_2, x_3)$, which is an illustration of (11.14) for unequal indices.

Let us now turn to the general proof of (11.14).

(a) Assume $e = \varepsilon$. As $D_e(\mathbf{x})$ is defined to be the complement of $C_e(\mathbf{x})$ the proof of (11.14) is all too simple:

$$C_e(\mathbf{x}) \wedge D_e(\mathbf{x}) \Leftrightarrow C_e(\mathbf{x}) \wedge \overline{C}_e(\mathbf{x}) \Leftrightarrow \mathbf{0}.$$

(b) Assume $e \neq \varepsilon$. A convenient way to calculate

$$C_e(\mathbf{x}) \wedge D_\varepsilon(\mathbf{x}) \Leftrightarrow$$

is to employ

$$D_e(\mathbf{x}) \Leftrightarrow \bigvee_{\varepsilon' \in \mathcal{E} \setminus \{e\}} C_{\varepsilon'}(\mathbf{x}), \tag{11.10}$$

obtaining

$$C_e(\mathbf{x}) \wedge \bigvee_{\varepsilon' \in \mathcal{E} \setminus \{\varepsilon\}} C_{\varepsilon'}(\mathbf{x}) \Leftrightarrow$$

$$C_e(\mathbf{x})\Big(C_0(\mathbf{x}) \vee C_1(\mathbf{x}) \vee \cdots \vee C_e(\mathbf{x}) \vee \cdots \vee$$

$$C_{\varepsilon-1}(\mathbf{x}) \vee C_{\varepsilon+1}(\mathbf{x}) \vee \cdots \vee C_{2^n-1}(\mathbf{x})\Big) \Leftrightarrow$$

$$\underbrace{C_e(\mathbf{x})C_0(\mathbf{x})}_{\mathbf{0}} \vee \underbrace{C_e(\mathbf{x})C_1(\mathbf{x})}_{\mathbf{0}} \vee \cdots \vee \underbrace{C_e(\mathbf{x})C_e(\mathbf{x})}_{C_e(\mathbf{x})} \vee \cdots \vee$$

$$\underbrace{C_e(\mathbf{x})C_{\varepsilon-1}(\mathbf{x})}_{\mathbf{0}} \vee \underbrace{C_e(\mathbf{x})C_{\varepsilon+1}(\mathbf{x})}_{\mathbf{0}} \vee \cdots \vee \underbrace{C_e(\mathbf{x})C_{2^n-1}(\mathbf{x})}_{\mathbf{0}} \Leftrightarrow$$

$$\mathbf{0} \vee \cdots \vee \mathbf{0} \vee C_e(\mathbf{x}) \vee \mathbf{0} \vee \cdots \vee \mathbf{0} \Leftrightarrow C_e(\mathbf{x}).$$

The result of (a) and (b), above, prove (11.14).

11.6 Solving a System of Logic Equations *

Consider the definitions of minterms and maxterms for 2 variables:

$$\begin{aligned} \overline{X}_1 \wedge \overline{X}_2 &\Leftrightarrow C_0(x_1, x_2), & X_1 \vee X_2 &\Leftrightarrow D_0(x_1, x_2), \\ \overline{X}_1 \wedge X_2 &\Leftrightarrow C_1(x_1, x_2), & X_1 \vee \overline{X}_2 &\Leftrightarrow D_1(x_1, x_2), \\ X_1 \wedge \overline{X}_2 &\Leftrightarrow C_2(x_1, x_2), & \overline{X}_1 \vee X_2 &\Leftrightarrow D_2(x_1, x_2), \\ X_1 \wedge X_2 &\Leftrightarrow C_3(x_1, x_2), & \overline{X}_1 \vee \overline{X}_2 &\Leftrightarrow D_3(x_1, x_2). \end{aligned}$$

These two groups of logic formulas are sometimes referred to as **systems of logic equations** because of their superficial similarity to a system of linear equations in conventional algebra. We want to 'solve' the above equations by expressing each logic variable X_i as a function of some or all of the minterms (or maxterms). This is simple to do for the above two systems where $X_1 \Leftrightarrow C_2 \vee C_3$ and $X_2 \Leftrightarrow C_1 \vee C_3$, or $X_1 \Leftrightarrow D_0 \wedge D_1$ and $X_2 \Leftrightarrow D_0 \wedge D_2$. But obtaining the general results for an arbitrary number n of variables, i.e.:

$$X_i \Leftrightarrow \bigvee_{e \in \mathcal{X}_i} C_e(\mathbf{x}) \Leftrightarrow \bigwedge_{e \in \overline{\mathcal{X}}_i} D_e(\mathbf{x}), \tag{11.16}$$

$$\overline{X}_i \Leftrightarrow \bigvee_{e \in \overline{\mathcal{X}}_i} C_e(\mathbf{x}) \Leftrightarrow \bigwedge_{e \in \mathcal{X}_i} D_e(\mathbf{x}), \tag{11.17}$$

is a problem of a different magnitude. Remember

$$X_i \Leftrightarrow \mathbf{x} \in \mathcal{X}_i \quad \text{or} \quad \mathcal{X}_i = \{\mathbf{x} | x_i = 1\} \tag{6.9}$$

in which $\mathcal{X}_i$ is the maximum K-set. Next, consider

$$\mathcal{X}_i = \bigcup_{e \in \mathcal{X}_i} \{\mathbf{e}\}. \tag{11.18}$$

Expressing this formula in plain English, you quickly realise that it is valid as it is per definition tautological: The set $\mathcal{X}_i$ is the union ($\bigcup$) of those 1-element sets ($\{\mathbf{e}\}$) the elements of which belong to $\mathcal{X}_i$. Substituting $\mathcal{X}_i$ of (6.9) by $\bigcup_{e \in \mathcal{X}_i} \{\mathbf{e}\}$ leads to

$$X_i \Leftrightarrow \mathbf{x} \in \mathcal{X}_i \Leftrightarrow \mathbf{x} \in \bigcup_{e \in \mathcal{X}_i} \{\mathbf{e}\}. \tag{11.19}$$

To evaluate $\mathbf{x} \in \bigcup_{e \in \mathcal{X}_i} \{\mathbf{e}\}$ we generalise

$$x \in (\mathcal{S} \cup \mathcal{T}) \Leftrightarrow (x \in \mathcal{S}) \ \vee \ (x \in \mathcal{T}) \tag{6.7}$$

so that it applies to an arbitrary number n of sets,

$$x \in (\mathcal{S}_1 \cup \mathcal{S}_2 \cup \cdots \cup \mathcal{S}_n) \Leftrightarrow (x \in \mathcal{S}_1) \ \vee \ (x \in \mathcal{S}_2) \ \vee \cdots \ \vee \ (x \in \mathcal{S}_n),$$

expressing this in compressed form as

$$x \in \bigcup_{i=1}^{n} \mathcal{S}_i \Leftrightarrow \bigvee_{i=1}^{n} (x \in \mathcal{S}_i). \tag{11.20}$$

This formula allows us to rewrite (11.19) as

$$X_i \Leftrightarrow \mathbf{x} \in \mathcal{X}_i \Leftrightarrow \mathbf{x} \in \bigcup_{e \in \mathcal{X}_i} \{\mathbf{e}\} \Leftrightarrow \bigvee_{e \in \mathcal{X}_i} (\mathbf{x} \in \{\mathbf{e}\}).$$

For any 1-element set $\{\mathbf{e}\}$, stating that $\mathbf{x}$ is an element of $\{\mathbf{e}\}$ is the same as saying that $\mathbf{x}$ equals the element $\mathbf{e}$ of the 1-element set $\{\mathbf{e}\}$:

$$\mathbf{x} \in \{\mathbf{e}\} \Leftrightarrow \mathbf{x} = \mathbf{e}. \tag{11.21}$$

We may thus continue:

$$X_i \Leftrightarrow \bigvee_{e \in \mathcal{X}_i} (\mathbf{x} \in \{\mathbf{e}\}) \Leftrightarrow \bigvee_{e \in \mathcal{X}_i} (\mathbf{x} = \mathbf{e}) \Leftrightarrow \bigvee_{e \in \mathcal{X}_i} C_e(\mathbf{x}),$$

this proving the first half of (11.16). The proof of the second half of (11.16) and of (11.17) should now pose no problem.

There is a hidden difficulty in applying (11.16) and (11.17). These formulas assume you know which input events e comprise the maximum K-set $\mathcal{X}_i$ for any given number n of input variables $x_1, \ldots, x_n$. The input events e belonging to $\mathcal{X}_i$ are — for a given number n of input variables — usually calculated by a small computer program. For good reasons this book contains no computer programs. But, I strongly recommend that you write such a program in a computer language convenient to you.

11.7 DeMorgan on DeMorgan

In this section we prove the first of the generalised DeMorgan theorems

$$\overline{X_1 \wedge X_2 \wedge \cdots \wedge X_n} \Leftrightarrow \overline{X}_1 \vee \overline{X}_2 \vee \cdots \vee \overline{X}_n, \tag{11.22}$$

$$\overline{X_1 \vee X_2 \vee \cdots \vee X_n} \Leftrightarrow \overline{X}_1 \wedge \overline{X}_2 \wedge \cdots \wedge \overline{X}_n, \tag{11.23}$$

relying on duality as proof for the second. The method of choice in proving (11.22) is *mathematical induction*, and it is possibly interesting to note that it was DeMorgan himself who, in 1838, formulated this principle (thus the title of this section).

Let us call the logic expression

$$\overline{X_1 \wedge X_2 \wedge \cdots \wedge X_n} \leftrightarrow \overline{X}_1 \vee \overline{X}_2 \vee \cdots \vee \overline{X}_n$$

we wish to prove $P_n(x_1, x_2, \ldots, x_n)$, i.e.,

$$P_n(x_1, x_2, \ldots, x_n) \overset{df}{\Leftrightarrow} \overline{X_1 \wedge X_2 \wedge \cdots \wedge X_n} \leftrightarrow \overline{X}_1 \vee \overline{X}_2 \vee \cdots \vee \overline{X}_n. \quad (11.24)$$

When writing an expression such as (11.24) we are claiming that it holds for any n, thereby inferring that there exists a sequence of expressions such as

$$\begin{aligned} P_2(x_1, x_2) &\overset{df}{\Leftrightarrow} \overline{X_1 \wedge X_2} \leftrightarrow \overline{X}_1 \vee \overline{X}_2 \\ P_3(x_1, x_2, x_3) &\overset{df}{\Leftrightarrow} \overline{X_1 \wedge X_2 \wedge X_3} \leftrightarrow \overline{X}_1 \vee \overline{X}_2 \vee \overline{X}_3 \\ &\vdots \\ P_k(x_1, x_2, \ldots, x_k) &\overset{df}{\Leftrightarrow} \overline{X_1 \wedge X_2 \wedge \cdots \wedge X_k} \leftrightarrow \overline{X}_1 \vee \overline{X}_2 \vee \cdots \vee \overline{X}_k \end{aligned}$$

where each is called an **instance** of (11.24). Now, to prove (11.24), it does not suffice to prove that each of the above instances $P_2(x_1, x_2)$, $P_3(x_1, x_2, x_3)$, $\ldots$, $P_k(x_1, \ldots, x_k)$ is a tautology. For, even if we were to prove each instance, we still would not have shown $P_{k+1}(x_1, \ldots, x_{k+1})$ to be a tautology.

A modification of *modus ponens* — called *mathematical induction* — is designed to cope with this situation. *Modus ponens* (8.16), as you will remember, consists of three steps:

(1) Prove P to be a tautology — $P \Leftrightarrow \mathbf{1}$.
(2) Prove that $P \rightarrow Q$ is a logical implication — $P \Rightarrow Q$.
(3) If steps (1) and (2) are successful then Q is a tautology — $Q \Leftrightarrow \mathbf{1}$.

In formulating **mathematical induction** along the lines of *modus ponens* we liken $P_k(x_1, x_2, \ldots, x_k)$ to P, and $P_{k+1}(x_1, \ldots, x_{k+1})$ to Q, and note that $P_k(x_1, x_2, \ldots, x_k)$ and $P_{k+1}(x_1, \ldots, x_{k+1})$ are instances of the expression $P_n(x_1, x_2, \ldots, x_n)$ we wish to prove.

(0) Prove $P_n(x_1, x_2, \ldots, x_n)$ for some conveniently small n, say, for n is 2, i.e., prove $P_2(x_1, x_2) \Leftrightarrow \mathbf{1}$.
(1) For an arbitrary k, **assume** $P_k(x_1, x_2, \ldots, x_k) \Leftrightarrow \mathbf{1}$.
(2) Prove that $P_k(x_1, x_2, \ldots, x_k) \rightarrow P_{k+1}(x_1, \ldots, x_{k+1}) \Leftrightarrow \mathbf{1}$.
(3) If steps (0) and (2) are successful then $P_n(x_1, x_2, \ldots, x_n)$ is a tautology — i.e., $P_n(x_1, x_2, \ldots, x_n) \Leftrightarrow \mathbf{1}$ for any value of n.

The difference in the practical application between *modus ponens* and mathematical induction is that in *modus ponens* you must *prove* that P is a tautology while in mathematical induction it suffices to *assume* that P_k is true. After all, the purpose of mathematical induction is to prove P_n, so you can't very well be expected to have to first prove P_n for some arbitrary n.

Now let us prove (11.22). (0) For $n = 2$ we first need to prove $\overline{X_1 \wedge X_2} \Leftrightarrow \overline{X}_1 \vee \overline{X}_2$. In this case, employing the truth table would be the quickest proof method. But, you will remember that the above formula was already proved in Section 9.4.

Next, to steps (1) and (2). We start by writing the proven form of DeMorgan's theorem using the variables A and B

$$\overline{A \wedge B} \Leftrightarrow \overline{A} \vee \overline{B}.$$

In this formula we substitute A by $X_1 \wedge X_2 \wedge \cdots \wedge X_k$, and B by X_{k+1}, obtaining

$$\overline{\underbrace{X_1 \wedge X_2 \wedge \cdots \wedge X_k}_{A} \wedge \underbrace{X_{k+1}}_{B}} \Leftrightarrow \overline{\underbrace{X_1 \wedge X_2 \wedge \cdots \wedge X_k}_{A}} \vee \overline{\underbrace{X_{k+1}}_{B}}. \tag{11.25}$$

We now *assume*

$$\overline{X_1 \wedge X_2 \wedge \cdots \wedge X_k} \Leftrightarrow \overline{X}_1 \vee \overline{X}_2 \vee \cdots \vee \overline{X}_k, \tag{11.26}$$

allowing us to *replace* $\overline{X_1 \wedge X_2 \wedge \cdots \wedge X_k}$ on the right of (11.25) by $\overline{X}_1 \vee \overline{X}_2 \vee \cdots \vee \overline{X}_k$ thus developing:

$$\overline{X_1 \wedge X_2 \wedge \cdots \wedge X_k \wedge X_{k+1}} \Leftrightarrow \overline{X}_1 \vee \overline{X}_2 \vee \cdots \vee \overline{X}_k \vee \overline{X}_{k+1}. \tag{11.27}$$

As this formula is an instance of $P_n(x_1, x_2, \ldots, x_n)$, and by its derivation $P_k(x_1, x_2, \ldots, x_k) \rightarrow P_{k+1}(x_1, \ldots, x_{k+1})$ is true, we can, by the principle of mathematical induction, say that $P_n(x_1, x_2, \ldots, x_n)$ has been proved for any value of n.

Chapter 12

On Predicate Logic

Without explicitly saying so, we have been using a characteristic element of predicate logic from the start — the logic variables $X_i \overset{df}{\Leftrightarrow} x_i = 1$ and $Y_i \overset{df}{\Leftrightarrow} y_i = 1$. Then, in the last chapter, we made extensive use of a typical tool of predicate logic — the generalised OR ($\bigvee$) of (11.6b), (11.6c), (11.6d) and (11.7). In the present chapter we are going to take a closer look at predicate logic, a topic we shall use to further develop the theory of combinational switching circuits.

12.1 Inside a Proposition

A **proposition** is a sentence expressing an assertion that is either true (**1**) or false (**0**) — even if we can't always say which (as in the case of the still unproved *Goldbach conjecture: 'Every even number greater than 2 can be represented by the sum of two primes'*). **Propositional logic** takes a proposition to be an *undividable entity* the structure of which is never analysed. **Predicate logic**, on the other hand, does analyse the structure of a proposition, splitting it into constituent parts called *subject*, *predicate*, and *function*.

The simplest form of a proposition asserts that a certain thing or individual (called a **subject**) possesses a specific property or characteristic (called a (monadic) **predicate**), e.g.

$$\underbrace{\text{'}\textit{Churchill}}_{\text{subject}}\ \underbrace{\textit{is a Nobel laureate}}_{\text{(monadic) predicate}}\text{'}. \tag{12.1}$$

Generalising on the above concept of proposition, we allow a proposition to have more than one subject, the predicate then expressing a *relationship* between the subjects, as in

$$\underbrace{\text{'}\textit{Abraham}}_{\text{subject}}\ \underbrace{\textit{is the father of}}_{\text{(dyadic) predicate}}\ \underbrace{\textit{Isaac}}_{\text{subject}}\text{'}. \tag{12.2}$$

The **degree of a predicate** (monadic, dyadic, etc.) states the number of subjects the predicate refers to.

A **function** enables us to restrict or focus our view on certain aspects of a subject (or subjects) thereby creating a new subject:

$$\underbrace{\underbrace{\text{'}\textit{The walls of}}_{\text{function}}\ \underbrace{\textit{Jerico}}_{\text{subject}}}_{\text{new subject}}\ \underbrace{\textit{crumbled}\text{'}}_{\text{predicate}}. \tag{12.3}$$

Often, there are a number of possibilities to partition a given proposition into subjects, predicates and functions, the choice of the partition depending on the aspects one wishes to emphasise, e.g.

$$\underbrace{\underbrace{\text{'}\textit{Churchill}}_{\text{subject}}\ \underbrace{\textit{is author of}}_{\text{dyadic predicate}}}_{\text{monadic predicate}}\ \underbrace{\underbrace{\textit{a book about}}_{\text{monadic function}}\ \underbrace{\textit{world war II}\text{'}}_{\text{subject}}}_{\text{subject}}. \tag{12.4}$$

The above propositions (12.1) to (12.4) are called **atomic propositions** because they cannot be broken down into two 'smaller' propositions connected by any phrase interpretable as one of the logic connectives $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$. In fact, the logic connectives are used to *connect (atomic) propositions*, thus creating **compound propositions**. For instance, the atomic propositions *'The moon consists of water'* and *'My cat has wings'* can be combined to create the compound proposition 'If *the moon consists of water* then *my cat has wings'* — which, being an IMPLICATION, is true (as you can check with Section 8.5).

12.2 Symbolic Notation of Propositions

As a first step towards a symbolic notation we introduce variables for subjects, predicates and functions:

x, y, z	subject variables,
ϱ (rho), σ (sigma), τ (tau)	predicate variables,
f, g, h	function variables,

and allowing for further variables, when needed, by adding indices. We take *all* the subjects we wish to work with to be explicitly listed in a *set of subjects*, a set we call the **universe** (of subjects) $\mathcal{U}_S$.

It is common to express the *general form of an atomic proposition* — thus writing an **atomic expression** — by having a predicate variable being followed by as many subject variables as stated by the degree of the predicate variable. For instance, if ϱ, σ and τ are predicate variables of degree 1, 2, and 3, respectively, then

$$\varrho x_1, \qquad \sigma x_1 x_2, \qquad \tau x_1 x_2 x_3$$

are examples of atomic expressions. The degree of a predicate is sometimes added to the predicate variable as a superscript or, alternatively, parentheses are used:

$$\varrho^1 x_1 \Leftrightarrow \varrho(x_1), \qquad \sigma^2 x_1 x_2 \Leftrightarrow \sigma(x_1, x_2), \qquad \tau^3 x_1 x_2 x_3 \Leftrightarrow \tau(x_1, x_2, x_3).$$

When partitioning a proposition into subjects, predicate and functions, predicate logic uses the concept of *function* in exactly the same way as it is used in the rest of mathematics (and was discussed in Chapter 3). In predicate logic a **function** of degree n (an n-ary function) has as its domain the n-fold Cartesian product of the universe of subjects, $\mathcal{U}_S^n$, and, as its co-domain, the universe of subjects $\mathcal{U}_S$ itself. For any such function, say f, we may write $f : \mathcal{U}_S^n \mapsto \mathcal{U}_S$. The value of a function is written in the same way as an atomic expression: A function variable is followed by as many subject variables as the degree of the function states. Often, the degree of a function is added to the function symbol as a superscript, or you may alternatively use parentheses:

$$h^1 z = h(z), \qquad f^2 x_3 x_1 = f(x_3, x_1),$$

$$g^3 x_1 h^1 z f^2 x_3 x_1 = g\big(x_1, h(z), f(x_3, x_1)\big).$$

The thing to note in the above examples is that $h^1 z$ and $f^2 x_3 x_1$ are expressions for subjects, and it is these subjects that have been used as arguments in g^3. Moreover, subjects are not propositions so that they cannot be logically equivalent ($\Leftrightarrow$), but they can be identical ($=$). Thus, we can write $h^1 z = h(z)$, but not $h^1 z \Leftrightarrow h(z)$.

A **term** is a subject, either directly or indirectly, meaning it is either a subject in its own right, or the *value* of any function (the function mapping to a subject). E.g., t_1, t_2, t_3, t_4 are examples for terms:

$$t_1 \stackrel{df}{=} x_1, \qquad t_2 \stackrel{df}{=} h^1 z, \qquad t_3 \stackrel{df}{=} f^2 x_3 x_1, \qquad t_4 \stackrel{df}{=} g^3 t_1 t_2 t_3.$$

Using terms, we obtain the *general form* of an **atomic expression** by writing a predicate that is followed by the appropriate number of *terms*, e.g.

$$\varrho(t_1), \qquad \sigma(t_1, t_3), \qquad \tau(t_2, t_3, t_4).$$

Identity ($=$) is an outstanding or singular dyadic predicate. Its arguments, *which are always terms*, are commonly written to the left and right of the identity symbol, as in $t_1 = t_2$. The atomic expression $t_1 = t_2$ asserts that t_1 and t_2 refer to the same subject (thing or individual). For instance, in the example *'The pope is the bishop of Rome'* we are referring to a person who has two different functions or titles. When the identity predicate is used one speaks of **predicate logic with identity**.

The simplest form of a **logic expression** is, as you might imagine, an *atomic expression.* Assuming P and Q to be *any* logic expression (simple or not), we obtain further logic expressions as compound logic expressions by writing $\neg P$, $P \wedge Q$, $P \wedge Q$, $P \rightarrow Q$, or $P \leftrightarrow Q$.

12.3 The Switching Algebra Connection

In a rather informal way we have been using concepts of predicate logic from the start (Section 1.3) by introducing the *atomic expressions* $x_i = 1$ and $x_i = 0$ and stating that all *subjects*, used to substitute x_i, are to be taken from the *universe of subjects* $\{0,1\}$. These atomic expressions were taken as special cases of the *term* $x_i = e_i$ (Section 5.1) — with $e_i \in \{0,1\}$ — establishing the *identity predicate* $(=)$ as basic to switching algebra.

Speaking of predicates: *The* **identity predicate** *is the only predicate used when applying predicate logic to combinational circuits. In other words, when developing a switching algebra for combinational circuits, we shall be using predicate logic* with *identity, but, we shall not be using predicate variables, as the set of predicates is empty.* The identity predicate is always considered in its own right, i.e., quite apart from any other predicates. If no other predicates are considered one says that the set of predicates is empty.

The (n-ary) functions of predicate logic map n-tuples of subjects to subjects, or, as formally stated in the previous section, $f^n : \mathcal{U}_S^n \mapsto \mathcal{U}_S$. On the other hand (see Section 3.3), combinational circuits were shown to be the functions $f^n : \{0,1\}^n \mapsto \{0,1\}$. For combinational circuits the universe of subjects $\mathcal{U}_S$ is the set $\{0,1\}$ which means that the functions of predicate logic describe combinational circuits. Thus, those theorems of predicate logic with identity that refer to functions are also theorems of switching algebra for combinational circuits. The predicate logic notation $f^3(x_1, x_2, x_3)$, used in the previous section, is to be interpreted as describing a combinational circuit of three inputs. But, writing $f^3(x_1, t, x_3)$, in which $t = g^3(x_1, x_2, x_3)$, represents a **composite function** in which the output of a combinational circuit g^3 is the input to a combinational circuit f^3. The aspects of this paragraph are discussed in detail in later chapters.

12.4 Quantifiers

Never, when writing $x_i = 1$ (with x_i varying over $\{0,1\}$), have we claimed that x_i really *is* 1. Quite the contrary — because we can substitute either 0 or 1 for x_i whenever writing $x_i = 1$, we are inferring that *'not* **every** $x_i = 1$*'*. Introducing the symbol $\bigwedge$ (a large $\wedge$) for **every** (or *each*, or *all*) allows us to write *'not every* $x_i = 1$*'* as

$$\neg \bigwedge x_i = 1.$$

The symbol $\bigwedge$ itself is called the **universal quantifier** while its application to some logic expression — say, $x_i = 1$ — is referred to as the **generalisation** of that expression. Because a logic expression may contain two or more different variables, it is customary, and usually necessary, to specify the variable to which the generalisation applies, e.g., writing

$$\bigwedge_{x_i} x_i = e_i \quad \text{or} \quad \bigwedge_{x_i} x_i = e_i$$

instead of $\bigwedge x_i = e_i$ and reading the above as *'For every* x_i *:* $x_i = e_i$*'*. **Generalisation** $(\bigwedge)$ of an expression — $x_i = e_i$ — over a specified variable — x_i — is understood as the conjunction $(\wedge)$ of the expressions obtained by substituting *all* values of the set of subjects — $\{0, 1\}$ — for the subject variable x_i over which generalisation is to take place, e.g.

$$\neg \bigwedge_{x_i \in \{0,1\}} x_i = e_i \Leftrightarrow \neg\big((0 = e_i) \wedge (1 = e_i)\big). \tag{12.5}$$

The result of generalisation is a new expression in which the variable over which generalisation took place has been eliminated. If you generalise an expression that has only a single variable, the result is a proposition

$$\neg \bigwedge_{x_i \in \{0,1\}} x_i = 1 \Leftrightarrow \neg\big((0 = 1) \wedge (1 = 1)\big) \Leftrightarrow \neg \mathbf{0} \Leftrightarrow \mathbf{1}, \tag{12.6}$$

this example rendering the introductory remark *'not every* $x_i = 1$*'* true.

Possibly, you might like to convey the meaning of *'not every* $x_i = 1$*'* in a positive way, i.e., without referring to negation (*not*). You can do so by using the logically equivalent phrase *'***some** $x_i = 0$*'*:

$$\textit{'not } \textbf{every } x_i = 1\textit{'} \Leftrightarrow \textit{'}\textbf{some } x_i = 0\textit{'}.$$

For **some** we introduce the symbol $\bigvee$ (a large $\vee$) called the **existential quantifier**. Its application to logic expressions, as in

$$\bigvee_{x_i} x_i = e_i \quad \text{or} \quad \bigvee_{x_i} x_i = e_i,$$

is called the **instantiation** of the expression. The above example is read as *'For some* x_i *:* $x_i = e_i$*'* or as *'There exists an* x_i *such that:* $x_i = e_i$*'* or, more precisely, *'There exists* **at least one** x_i *such that:* $x_i = e_i$*'*. **Instantiation** $(\bigvee)$ of an expression — $x_i = e_i$ — over a specified variable — x_i — is understood as the disjunction $(\vee)$ of the expressions obtained by substituting *all* values of the set of subjects — $\{0, 1\}$ — for the subject variable x_i over which instantiation is to take place, as in the examples

$$\bigvee_{x_i \in \{0,1\}} x_i = e_i \Leftrightarrow \big((0 = e_i) \vee (1 = e_i)\big), \tag{12.7}$$

$$\bigvee_{x_i \in \{0,1\}} x_i = 0 \Leftrightarrow \big((0 = 0) \vee (1 = 0)\big) \Leftrightarrow \mathbf{1}. \tag{12.8}$$

12.5 Quantification and Replication *

Quantification is a summary expression for *generalisation* and *instantiation.* But not everything that looks like quantification is quantification. In the following two formulas, note how the marked similarity of the expressions on the left of $\Leftrightarrow$ contrasts with their meaning written on the right. The first formula is an example of *quantification*, the second of what I in this text call **replication**, simply to be able to refer to it in some way.

$$\text{Quantification:} \quad \bigwedge_{x_i \in \{0,1\}} x_i = e_i \Leftrightarrow \big((0 = e_i) \wedge (1 = e_i)\big)$$

$$\text{Replication:} \quad \bigwedge_{i \in \{1,2\}} x_i = e_i \Leftrightarrow \big((x_1 = e_1) \wedge (x_2 = e_2)\big)$$

To distinguish between *quantification* and *replication*, it might help to think of **quantification** as *expansion by value* or, in more detail, as *expansion over the values of variables* while taking **replication** to be *expansion by name* or, more exhaustively, *expansion over the names of variables and/or functions.* You can find a demonstration of the simultaneous use of quantification and replication if you look back at Section 11.2. The rules employed there in transforming the quantified expressions (11.6b), (11.6c), (11.6d) and (11.7) are discussed later (Section 13.3). Such rules pose an important result of predicate logic.

Quantification and replication are concepts not restricted to generalisation ($\bigwedge$) and instantiation ($\bigvee$), in fact, both concepts are standard to the operators $\sum$ and $\prod$, as demonstrated in the examples for $\sum$.

$$\text{Quantification:} \quad \sum_{i=1}^{n} i = 1 + 2 + \ldots + n$$

$$\text{Replication:} \quad \sum_{i=1}^{n} x_i = x_1 + x_2 + \ldots + x_n$$

In conventional algebra, the concepts of quantification and replication are not always strictly held apart, as when writing $e = \sum_{i=1}^{n} e_i \cdot 2^{n-i}$, where e_i is replicated, and 2^{n-i} is quantified.

12.6 Free and Bound Variables

The subject variables (x, y, z, ...) of any propositional form (e.g., ϱxy) or any compound logic expression (such as $\varrho x \rightarrow \sigma yz \leftrightarrow \tau z$) *not* containing a quantifier are said to be **free.** We refer to such subject variables as being *free* because we are free to substitute any subject of appropriately chosen sets.

Quantifying a propositional form, as in

$$\bigwedge_x \varrho xy, \quad \text{or} \quad \bigvee_y \varrho xy, \quad \text{or} \quad \bigwedge_x \bigvee_y \varrho xy,$$

is said to **bind** the variable or variables. In $\bigwedge_x \varrho xy$ the subject variable x is said to be **bound** by $\bigwedge_x$, in $\bigvee_y \varrho xy$ the variable y is *bound* by $\bigvee_y$, and in $\bigwedge_x \bigvee_y \varrho xy$ both x and y are *bound*. Conversely, y is *free* in $\bigwedge_x \varrho xy$ and x is *free* in $\bigvee_y \varrho xy$, and of course, z is *free* in $\bigwedge_x \varrho z$ because quantification is not taking place over z.

Looking at how quantifiers, incorporated in logic expressions, bind subject variables, makes it necessary to discuss the *scope* of a quantifier. A quantifier *acts only on* or is *applied only to* an expression to the right of the quantifier. In

$$\varrho x \rightarrow \bigvee_y \sigma yz \leftrightarrow \bigwedge_z \tau z$$

the existential quantifier $\bigvee_y$ acts on the whole expression $\sigma yz \leftrightarrow \bigwedge_z \tau z$ although the variable y is not contained in $\bigwedge_z \tau z$; the universal quantifier $\bigwedge_z$ acts only on τz (and *not* on any subexpression to its left although z occurs to the left of $\bigwedge_z \tau z$).

The expression, or subexpression, to which a quantifier applies is called the **scope** of the quantifier. You can limit the scope of a quantifier by using parentheses, e.g., $\varrho x \rightarrow (\bigvee_y \sigma yz) \leftrightarrow \bigwedge_z \tau z$ in which the scope of $\bigvee_y$ is limited to σyz. Without parentheses to limit the scope of a quantifier, the scope extends to the end of the logic expression or till a logical implication ($\Rightarrow$) or logical equivalence ($\Leftrightarrow$) is encountered so that

$$\varrho x \rightarrow \bigvee_y \sigma yz \leftrightarrow \bigwedge_z \tau z \Leftrightarrow \varrho x \rightarrow \Big(\bigvee_y \sigma yz \leftrightarrow \big(\bigwedge_z \tau z\big)\Big).$$

Let $P(x_1, x_2, \ldots, x_n)$ be some arbitrary logic expression in which *no* quantifiers occur. Then the symbols $\bigwedge_{x_1,x_2,\ldots,x_n}$ and $\bigvee_{x_1,x_2,\ldots,x_n}$ are defined as multiple quantification over the same scope $P(x_1, x_2, \ldots, x_n)$

$$\bigwedge_{x_1,x_2,\ldots,x_n} P(x_1, x_2, \ldots, x_n) \overset{df}{\Leftrightarrow} \bigwedge_{x_1} \bigwedge_{x_2} \cdots \bigwedge_{x_n} P(x_1, x_2, \ldots, x_n), \tag{12.9}$$

$$\bigvee_{x_1,x_2,\ldots,x_n} P(x_1, x_2, \ldots, x_n) \overset{df}{\Leftrightarrow} \bigvee_{x_1} \bigvee_{x_2} \cdots \bigvee_{x_n} P(x_1, x_2, \ldots, x_n), \tag{12.10}$$

the quantifiers being acted upon from right to left.

The name (symbol) used for a bound subject variable is of minor importance, meaning, you can change the name of any bound subject variable to

any other name not in the scope of the quantifier, as in

$$\bigvee_x \varrho xy \Leftrightarrow \bigvee_z \varrho zy,$$

$$\varrho x \to \Big(\bigvee_y \sigma yz \leftrightarrow (\bigwedge_z \tau z)\Big) \Leftrightarrow \varrho x \to \Big(\bigvee_y \sigma yz \leftrightarrow (\bigwedge_x \tau x)\Big).$$

This is quite the same as in conventional algebra as the example $\sum_{i=1}^{n} i = \sum_{k=1}^{n} k$ shows.

Chapter 13

Predicate Logic

In a rough outline, one might say propositional logic developed ponderously from the days of **Aristotle** in the fourth century BCE to the latter half of the nineteenth century when, after **DeMorgan** and especially **Boole**, many felt it had culminated. But, a quantum leap occurred in 1879 with the publication of **Frege**'s *Begriffsschrift*. In this booklet Frege broke propositions into predicates and subjects (called, respectively, *'functions'* and *'arguments'* by him) and introduced quantification of subjects. Thus was born what we now call *predicate logic*, and it has proved the greatest single development in logic since Aristotle's time.

This chapter presents the axioms of predicate logic, summarises some of its simpler theorems and proves one or the other to give you an idea how the system works.

13.1 Axioms and Rules of Switching Algebra*

Predicate logic does *not* render propositional logic obsolete, it augments it. Thus, in principle, the *axiomatic approach to propositional logic* of Section 9.1 remains valid, needing only minor adaptations.

a) To the **set of symbols** we add variables for subjects and functions

x, y, z	subject variables,
f, g, h	function variables,

allowing for further variables, when needed, by adding indices. The subjects we work with in switching algebra, 0 and 1, comprise the **universe** (of subjects) $\mathcal{U}_S = \{0, 1\}$. As stated in Section 12.3, no predicates (except the dyadic predicate of *identity*) are employed in applying predicate logic to combinational circuits, which is why we need no predicate variables. But we do, of course, need a symbol for *identity*, i.e., $=$.

b) The **formation rules**, the rules for writing valid *terms*, *atomic expressions* and *logic expressions* are stated at the end of Section 12.2.

c) The **axioms of switching algebra for combinational circuits** must (of course) incorporate those of propositional logic, albeit with a slight

difference. The propositional variables A, B, C must be replaced by variables for logic expressions, say, $P(x)$, $Q(x)$, $R(x)$. Furthermore, we need to add **axioms for the identity predicate**.

$$P(x) \Rightarrow Q(x) \rightarrow P(x) \tag{13.1}$$
$$\big(P(x) \rightarrow Q(x)\big) \rightarrow P(x) \Rightarrow P(x) \tag{13.2}$$
$$P(x) \rightarrow Q(x) \Rightarrow \big(Q(x) \rightarrow R(x)\big) \rightarrow \big(P(x) \rightarrow R(x)\big) \tag{13.3}$$

$$P(x) \wedge Q(x) \Rightarrow P(x) \tag{13.4}$$
$$P(x) \wedge Q(x) \Rightarrow Q(x) \tag{13.5}$$
$$P(x) \rightarrow Q(x) \Rightarrow \big(P(x) \rightarrow R(x)\big) \rightarrow \big(P(x) \rightarrow Q(x) \wedge R(x)\big) \tag{6}$$

$$P(x) \Rightarrow P(x) \vee Q(x) \tag{13.7}$$
$$Q(x) \Rightarrow P(x) \vee Q(x) \tag{13.8}$$
$$P(x) \rightarrow R(x) \Rightarrow \big(Q(x) \rightarrow R(x)\big) \rightarrow \big(P(x) \vee Q(x) \rightarrow R(x)\big) \tag{9}$$

$$P(x) \leftrightarrow Q(x) \Rightarrow P(x) \rightarrow Q(x) \tag{13.10}$$
$$P(x) \leftrightarrow Q(x) \Rightarrow Q(x) \rightarrow P(x) \tag{13.11}$$
$$P(x) \rightarrow Q(x) \Rightarrow \big(Q(x) \rightarrow P(x)\big) \rightarrow \big(P(x) \leftrightarrow Q(x)\big) \tag{13.12}$$

$$P(x) \rightarrow Q(x) \Rightarrow \neg Q(x) \rightarrow \neg P(x) \tag{13.13}$$
$$P(x) \Rightarrow \neg\neg P(x) \tag{13.14}$$
$$\neg\neg P(x) \Rightarrow P(x) \tag{13.15}$$

$$\mathbf{0} \Rightarrow \neg\mathbf{1} \tag{13.16}$$
$$\mathbf{1} \Rightarrow \neg\mathbf{0} \tag{13.17}$$

$$x = x \Rightarrow \mathbf{1} \tag{13.18}$$
$$x_1 = x_2 \Rightarrow x_1 = x_3 \rightarrow x_2 = x_3 \tag{13.19}$$
$$x_i = x \Rightarrow f^n(\ldots, x_i, \ldots) = f^n(\ldots, x, \ldots) \tag{13.20}$$

When writing $P(x)$ we want to indicate that x is *free* in the logic expression P. But, this is not intended to imply that x is the *only* free subject. The last axiom, allowing you to substitute a subject by an *equal* one in any function, will prove to be of utmost theoretical and practical interest.

d) The **rules of deduction** for predicate logic go well beyond those for propositional logic because we need to say how to use quantification. Do note that no attempt is made here to *'explain'* quantification, to say *'what'* it is (the best I could do in this respect, I did in Sections 12.4 and 12.5). The only thing we do here, is to say *how to use* quantifiers — and this is, after all, what axiomatisation is about.

Modus ponens: If an IMPLICATION $P \rightarrow Q$ and its antecedent P are tautologies then the consequent Q is also a tautology. Or, briefly

$$\text{If} \quad P \Rightarrow Q \quad \text{and} \quad P \Leftrightarrow \mathbf{1} \quad \text{then} \quad Q \Leftrightarrow \mathbf{1}. \tag{13.21}$$

Generalisation of the antecedent: If a certain IMPLICATION $P(x) \rightarrow Q$, that has an antecedent P in which x is free, is a tautology $(P(x) \rightarrow Q \Rightarrow \mathbf{1})$ then the IMPLICATION obtained by generalising the antecedent $P(x)$ over x is also a tautology — $(\bigwedge_x P(x)) \rightarrow Q \Leftrightarrow \mathbf{1}$. Or, briefly

$$\text{If} \quad P(x) \Rightarrow Q \quad \text{then} \quad \bigwedge_x P(x) \Rightarrow Q. \tag{13.22}$$

Note that nothing is said about the logic expression Q. The consequent Q need not contain x at all, but it may. If x does occur in Q it can do so as a *free* or *bound* variable.

Generalisation of the consequent: If a certain IMPLICATION $P \rightarrow Q(x)$, that has an antecedent P in which the variable x *does not* occur, is a tautology $(P \rightarrow Q(x) \Rightarrow \mathbf{1})$ then the IMPLICATION obtained by generalising the consequent $Q(x)$ over x is also a tautology — $P \rightarrow \bigwedge_x Q(x) \Leftrightarrow \mathbf{1}$. Or, briefly

$$\text{If} \quad P \Rightarrow Q(x) \quad \text{then} \quad P \Rightarrow \bigwedge_x Q(x). \tag{13.23}$$

Instantiation of the antecedent: If a certain IMPLICATION $P(x) \rightarrow Q$, that has a consequent Q in which x *does not* occur, is a tautology $(P(x) \rightarrow Q \Rightarrow \mathbf{1})$ then the IMPLICATION obtained by instantiating the antecedent $P(x)$ over x is also a tautology — $(\bigvee_x P(x)) \rightarrow Q \Leftrightarrow \mathbf{1}$. Or, briefly

$$\text{If} \quad P(x) \Rightarrow Q \quad \text{then} \quad \bigvee_x P(x) \Rightarrow Q. \tag{13.24}$$

Instantiation of the consequent: If a certain IMPLICATION $P \rightarrow Q(x)$, that has a consequent Q in which x is free, is a tautology $(P \rightarrow Q(x) \Rightarrow \mathbf{1})$ then the IMPLICATION obtained by instantiating the consequent $Q(x)$ over x is also a tautology — $P \rightarrow \bigvee_x Q(x) \Leftrightarrow \mathbf{1}$. Or, briefly

$$\text{If} \quad P \Rightarrow Q(x) \quad \text{then} \quad P \Rightarrow \bigvee_x Q(x). \tag{13.25}$$

Here, the antecedent P need not contain x at all, but it may. If x does occur in P it can do so as a *free* or *bound* variable.

Substituting a free subject variable: If $P(x)$ is a tautology in which x is a *free* subject variable then substituting x by an arbitrary term t (symbolically, $x \leftarrowtail t$) leads to the new tautology $P(t)$. Symbolically:

$$\text{If} \quad P(x) \Leftrightarrow \mathbf{1} \quad \text{and} \quad x \leftarrowtail t \quad \text{then} \quad P(t) \Leftrightarrow \mathbf{1}. \tag{13.26}$$

To help in further work, let us consider the special case of (13.23) in which P is a tautology. For $P \Leftrightarrow \mathbf{1}$ the expression $P \Rightarrow Q(x)$ — due to $\mathbf{1} \rightarrow A \Leftrightarrow A$ — reduces to $Q(x) \Leftrightarrow \mathbf{1}$, while $P \Rightarrow \bigwedge_x Q(x)$ reduces to $\bigwedge_x Q(x) \Leftrightarrow \mathbf{1}$. Having thus eliminated P from (13.23) we obtain the rule for the

Generalisation of a tautology: If $Q(x)$ is a tautology then its generalisation $\bigwedge_x Q(x)$ is also a tautology. Or, briefly

$$\text{If} \quad Q(x) \Leftrightarrow \mathbf{1} \quad \text{then} \quad \bigwedge_x Q(x) \Leftrightarrow \mathbf{1}. \tag{13.27}$$

Furthermore, it is worthwhile to note that **you may formulate any logic implication as a rule of deduction** using *modus ponens*. Let us see how this works for the axiom

$$P \rightarrow Q \Rightarrow (Q \rightarrow R) \rightarrow (P \rightarrow R). \tag{13.3}$$

By *modus ponens*, it follows from the truth of (13.3) that *if* $P \rightarrow Q$ is true then $(Q \rightarrow R) \rightarrow (P \rightarrow R)$ is true. Because $(Q \rightarrow R) \rightarrow (P \rightarrow R) \Rightarrow \mathbf{1}$, it again follows from *modus ponens* that *if* $Q \rightarrow R$ is true then $P \rightarrow R$ is true. The combination of these two successive applications of *modus ponens* can be stated as the **rule of transitivity**:

$$\text{If} \quad P \Rightarrow Q \quad \text{and} \quad Q \Rightarrow R \quad \text{then} \quad P \Rightarrow R. \tag{13.28}$$

This *rule* states, given the two logic implications $P \Rightarrow Q$ and $Q \Rightarrow R$, you may deduce the new implication $P \rightarrow R$ knowing it to be a tautology — $P \rightarrow R \Rightarrow \mathbf{1}$.

In the next example, we start from the theorem

$$P \rightarrow (Q \rightarrow R) \Rightarrow (P \rightarrow Q) \rightarrow (P \rightarrow R). \tag{9.33}$$

By reasoning analogous to that which led to (13.28), the above law leads to the following rule:

$$\text{If} \quad P \Rightarrow Q \rightarrow R \quad \text{and} \quad P \Rightarrow Q \quad \text{then} \quad P \Rightarrow R. \tag{13.29}$$

This is a brief form for stating that, given the two logic implications $P \Rightarrow Q \rightarrow R$ and $P \Rightarrow R$, you may deduce the new implication $P \rightarrow R$ knowing it to be a tautology — $P \rightarrow R \Rightarrow \mathbf{1}$.

13.2 Theorems on Identity

The first two theorems to be pointed out are those referring to *symmetry* and *transitivity* respectively:

$$x_1 = x_2 \Rightarrow x_2 = x_1, \tag{13.30}$$

$$x_1 = x_2 \Rightarrow (x_2 = x_3 \rightarrow x_1 = x_3). \tag{13.31}$$

To prove *symmetry* (13.30) we start with axiom (13.19) $x_1 = x_2 \rightarrow (x_1 = x_3 \rightarrow x_2 = x_3) \Rightarrow \mathbf{1}$ interchanging its antecedents $x_1 = x_2$ and $x_1 = x_3$ as allowed by (9.32),

$$x_1 = x_3 \rightarrow (x_1 = x_2 \rightarrow x_2 = x_3) \Rightarrow \mathbf{1},$$

and substituting x_3 by x_1 ($x_3 \hookleftarrow x_1$) as allowed by (13.26),

$$x_1 = x_1 \Rightarrow (x_1 = x_2 \rightarrow x_2 = x_1).$$

Axiom (13.18) states that $x_1 = x_1$ is true so that we may apply *modus ponens* to the above formula to deduce and thereby prove (13.30).

To prove *transitivity* (13.31) we write axiom (13.19) as

$$\underbrace{x_2 = x_1}_{Q} \Rightarrow \underbrace{(x_2 = x_3 \rightarrow x_1 = x_3)}_{R} \tag{13.32}$$

Now, by employing the *rule of transitivity* (13.28) 'If $P \Rightarrow Q$ and $Q \Rightarrow R$ then $P \Rightarrow R$' to (13.30) $\underbrace{x_1 = x_2}_{P} \Rightarrow \underbrace{x_2 = x_1}_{Q}$ and to (13.32) allows us to deduce and thus prove (13.31).

Next, we consider two theorems that extend the validity of axiom (13.20). This axiom

(13.20) $$x_i = e_i \Rightarrow f^n(\ldots, x_i, \ldots) = f^n(\ldots, e_i, \ldots),$$

which allows us to substitute a single subject (or argument) by an *equal* one in any function, may, intuitively, be extended to allow us to substitute *all* arguments, which we could express as

$$\bigwedge_{i=1}^{n} x_i = e_i \Rightarrow f^n(x_1, x_2, \ldots, x_n) = f^n(e_1, e_2, \ldots, e_n). \tag{13.33}$$

A further generalisation can be imagined by substituting the subjects by *terms*, obtaining

$$\bigwedge_{i=1}^{n} t_i = t_i^* \Rightarrow f^n(t_1, t_2, \ldots, t_n) = f^n(t_1^*, t_2^*, \ldots, t_n^*). \tag{13.34}$$

We shall merely prove the first of these two important **theorems on identical functions**. By axiom (13.4) $P \wedge Q \Rightarrow P$ we know the following expression to be a tautology

$$\underbrace{\overbrace{(x_1 = e_1)}^{P} \wedge \overbrace{(x_2 = e_2) \wedge \ldots \wedge (x_n = e_n)}^{Q}}_{P^*} \Rightarrow \overbrace{(x_1 = e_1)}^{P}. \tag{13.35}$$

Writing axiom (13.20) as

$$\underbrace{x_1 = e_1}_{P} \Rightarrow \underbrace{f^n(x_1, x_2, x_3, \ldots, x_n) = f^n(e_1, x_2, x_3, \ldots, x_n)}_{Q^*}, \tag{13.36}$$

and applying the *rule of transitivity* (13.28) 'If $P^* \Rightarrow P$ and $P \Rightarrow Q^*$ then $P^* \Rightarrow Q^*$' to (13.35) and (13.36) leads to the logic implication

$$\underbrace{\bigwedge_{i=1}^{n} x_i = e_i}_{P_1} \Rightarrow \underbrace{f^n(x_1, x_2, x_3, \ldots, x_n) = f^n(e_1, x_2, x_3, \ldots, x_n)}_{P_2}. \tag{13.37}$$

For the second variable, x_2, be can — in analogy to (13.35) and (13.36) — write

$$\bigwedge_{i=1}^{n} x_i = e_i \Rightarrow x_2 = e_2 \tag{13.38}$$

$$x_2 = e_2 \Rightarrow f^n(e_1, x_2, x_3, \ldots, x_n) = f^n(e_1, e_2, x_3, \ldots, x_n), \tag{13.39}$$

combining these by the *rule of transitivity* (13.28) to obtain

$$\underbrace{\bigwedge_{i=1}^{n} x_i = e_i}_{P} \Rightarrow \underbrace{f^n(e_1, x_2, x_3, \ldots, x_n) = f^n(e_1, e_2, x_3, \ldots, x_n)}_{Q}. \tag{13.40}$$

This second step in our derivation has not yet yielded what we want it to yield, for in the left side of the consequent of the above expression — i.e., in $f^n(e_1, x_2, x_3, \ldots, x_n)$ — there is the unwanted argument e_1, where we would like to see x_1. To achieve this we start by using

(13.31) $$x_1 = x_2 \Rightarrow (x_2 = x_3 \rightarrow x_1 = x_3)$$

with the following substitutions, allowed by (13.26),

$$x_1 \hookleftarrow f^n(x_1, x_2, x_3, \ldots, x_n),$$
$$x_2 \hookleftarrow f^n(e_1, x_2, x_3, \ldots, x_n),$$
$$x_3 \hookleftarrow f^n(e_1, e_2, x_3 \ldots, x_n),$$

so that (13.31) can be expressed as

$$\underbrace{f^n(x_1,x_2,x_3,\ldots,x_n)=f^n(e_1,x_2,x_3,\ldots,x_n)}_{P_2}\Rightarrow$$
$$f^n(e_1,x_2,x_3,\ldots,x_n)=f^n(e_1,e_2,x_3,\ldots,x_n)$$
$$\underbrace{\rightarrow f^n(x_1,x_2,x_3,\ldots,x_n)=f^n(e_1,e_2,x_3,\ldots,x_n)}_{P_3}. \qquad (13.41)$$

Again applying the *rule of transitivity* (13.28) 'If $P_1 \Rightarrow P_2$ and $P_2 \Rightarrow P_3$ then $P_1 \Rightarrow P_3$' to (13.37) and (13.41) leads to the logic implication

$$\underbrace{\bigwedge_{i=1}^{n} x_i = e_i}_{P} \Rightarrow \underbrace{f^n(e_1,x_2,x_3,\ldots,x_n)=f^n(e_1,e_2,x_3,\ldots,x_n)}_{Q}$$
$$\rightarrow \underbrace{f^n(x_1,x_2,x_3,\ldots,x_n)=f^n(e_1,e_2,x_3,\ldots,x_n)}_{R}. \qquad (13.42)$$

We now apply (13.29) 'If $P \Rightarrow (Q \rightarrow R)$ and $P \Rightarrow Q$ then $P \Rightarrow R$' to (13.42) and (13.40) obtaining

$$\bigwedge_{i=1}^{n} x_i = e_i \Rightarrow f^n(x_1,x_2,x_3,\ldots,x_n)=f^n(e_1,e_2,x_3,\ldots,x_n). \qquad (13.43)$$

Continuing the proof in a like manner for the rest of the variables $x_3,\ldots,x_n$ leads to theorem (13.33).

13.3 Theorems on Quantification

This summary of theorems on quantification is by no means complete and only a few of the theorems are proved, the proofs merely intended as a demonstration of a technique. $P(x)$ and $Q(x)$ denote logic expressions in which x occurs freely, whereas P and Q are understood to be logic expressions in which x does not occur freely. To save space, the variable over which quantification takes place is omitted, but if not stated otherwise, quantification is assumed to be over x. To emphasise the scope of a quantifier, sometimes more parentheses are used than is necessary.

Quantification of a single logic expression:

$$\bigwedge P(x) \Rightarrow \bigvee P(x) \qquad \textbf{(13.44)}$$

$$\bigwedge P(x) \Rightarrow P(x) \qquad \textbf{(13.45)}$$

$$P(x) \Rightarrow \bigvee P(x) \tag{13.46}$$

$$\bigwedge P \Leftrightarrow P \tag{13.47}$$

$$P \Leftrightarrow \bigvee P \tag{13.48}$$

Distributing Quantifiers:

$$\bigwedge(P(x) \wedge Q(x)) \Leftrightarrow (\bigwedge P(x)) \wedge (\bigwedge Q(x)) \tag{13.49}$$

$$(\bigwedge P(x)) \vee (\bigwedge Q(x)) \Rightarrow \bigwedge(P(x) \vee Q(x)) \tag{13.50}$$

$$\bigwedge(P(x) \rightarrow Q(x)) \Rightarrow (\bigwedge P(x)) \rightarrow (\bigwedge Q(x)) \tag{13.51}$$

$$\bigwedge(P(x) \leftrightarrow Q(x)) \Rightarrow (\bigwedge P(x)) \leftrightarrow (\bigwedge Q(x)) \tag{13.52}$$

$$\bigvee(P(x) \wedge Q(x)) \Rightarrow (\bigvee P(x)) \wedge (\bigvee Q(x)) \tag{13.53}$$

$$\bigvee(P(x) \vee Q(x)) \Leftrightarrow (\bigvee P(x)) \vee (\bigvee Q(x)) \tag{13.54}$$

$$(\bigvee P(x)) \rightarrow (\bigvee Q(x)) \Rightarrow \bigvee(P(x) \rightarrow Q(x)) \tag{13.55}$$

Repositioning Quantifiers:

$$\bigwedge(P \wedge Q(x)) \Leftrightarrow P \wedge \bigwedge Q(x) \tag{13.56}$$

$$\bigwedge(P \vee Q(x)) \Leftrightarrow P \vee \bigwedge Q(x) \tag{13.57}$$

$$\bigwedge(P \rightarrow Q(x)) \Leftrightarrow P \rightarrow \bigwedge Q(x) \tag{13.58}$$

$$\bigvee(P \wedge Q(x)) \Leftrightarrow P \wedge \bigvee Q(x) \tag{13.59}$$

$$\bigvee(P \vee Q(x)) \Leftrightarrow P \vee \bigvee Q(x) \tag{13.60}$$

$$\bigvee(P \rightarrow Q(x)) \Leftrightarrow P \rightarrow \bigvee Q(x) \tag{13.61}$$

Converting Quantifiers:

$$\bigwedge(P(x) \rightarrow Q) \Leftrightarrow (\bigvee P(x)) \rightarrow Q \tag{13.62}$$

$$\bigvee(P(x) \rightarrow Q) \Leftrightarrow (\bigwedge P(x)) \rightarrow Q \tag{13.63}$$

Interchanging Quantifiers:

$$\bigwedge_x \bigwedge_y P(x,y) \Leftrightarrow \bigwedge_y \bigwedge_x P(x,y) \tag{13.64}$$

$$\bigvee_x \bigvee_y P(x,y) \Leftrightarrow \bigvee_y \bigvee_x P(x,y) \tag{13.65}$$

$$\bigvee_x \bigwedge_y P(x,y) \Rightarrow \bigwedge_y \bigvee_x P(x,y) \tag{13.66}$$

DeMorgan:

$$\neg \bigwedge P(x) \Leftrightarrow \bigvee \neg P(x) \tag{13.67}$$

$$\neg \bigvee P(x) \Leftrightarrow \bigwedge \neg P(x) \tag{13.68}$$

The few proofs that follow will show you that the methods demonstrated in Chapter 9 remain valid.

Proving (13.45) $\bigwedge P(x) \Rightarrow P(x)$ and (13.46) $P(x) \Rightarrow \bigvee P(x)$: Generalising the antecedent of (9.31) $P(x) \Rightarrow P(x)$ — according to (13.22) — leads to (13.45) while generalising the consequent — according to (13.23) — leads to (13.46).

Proving (13.44) $\bigwedge P(x) \Rightarrow \bigvee P(x)$: Instantiating the consequent of (13.45) $\bigwedge P(x) \Rightarrow P(x)$ — according to (13.25) — or generalising the antecedent of (13.46) $P(x) \Rightarrow \bigvee P(x)$ — according to (13.22) — leads to (13.44).

Proving (13.49) $\bigwedge\big(P(x) \wedge Q(x)\big) \Leftrightarrow \big(\bigwedge P(x)\big) \wedge \big(\bigwedge Q(x)\big)$: Interpreting (13.12), we need to prove the logic implication ($\Leftrightarrow$) in both directions. That is, we need to prove

$$\bigwedge\big(P(x) \wedge Q(x)\big) \Rightarrow \big(\bigwedge P(x)\big) \wedge \big(\bigwedge Q(x)\big) \tag{13.49a}$$

$$\big(\bigwedge P(x)\big) \wedge \big(\bigwedge Q(x)\big) \Rightarrow \bigwedge\big(P(x) \wedge Q(x)\big) \tag{13.49b}$$

Proving (13.49a) $\bigwedge\big(P(x) \wedge Q(x)\big) \Rightarrow \big(\bigwedge P(x)\big) \wedge \big(\bigwedge Q(x)\big)$: Let us start by writing axiom

(9.6) $A \to B \Rightarrow (A \to C) \to (A \to B \wedge C)$

$$\{A \leftarrow \bigwedge\big(P(x) \wedge Q(x)\big),\quad B \leftarrow \bigwedge P(x),\quad C \leftarrow \bigwedge Q(x)\}$$

$$\underbrace{\big(\bigwedge P(x) \wedge Q(x)\big) \to \bigwedge P(x)}_{P_1} \Rightarrow$$

$$\underbrace{\big(\bigwedge P(x) \wedge Q(x)\big) \to \bigwedge Q(x)}_{P_2} .\to.$$

$$\underbrace{\big(\bigwedge P(x) \wedge Q(x)\big) \to \bigwedge P(x) \wedge \bigwedge Q(x)}_{Q_2}$$

To show that P_1 is a tautology we start with (13.4) $P(x) \wedge Q(x) \Rightarrow P(x)$, generalising the antecedent, (13.22), and the consequent, (13.23). Similarly, P_2 is shown to be a tautology when starting with (13.5) $P(x) \wedge Q(x) \Rightarrow Q(x)$. P_1 being a tautology, we may apply *modus ponens* to the above $P_1 \Rightarrow P_2 \to Q_2$, deducing $P_2 \Rightarrow Q_2$. And, as P_2 was shown to be a tautology we may deduce Q_2, i.e. (13.49a).

The next step, proving

(13.49b) $\big(\bigwedge P(x)\big) \wedge \big(\bigwedge Q(x)\big) \Rightarrow \bigwedge\big(P(x) \wedge Q(x)\big)$

is achieved by first showing that

$$(*) \qquad (\bigwedge P(x)) \wedge (\bigwedge Q(x)).\Rightarrow.P(x) \wedge Q(x)$$

is a tautology. For then, by generalising the consequent according to (13.23) we would obtain and thus prove (13.49b). To prove (*) we again start with axiom

(9.6) $A \rightarrow B \Rightarrow (A \rightarrow C) \rightarrow (A \rightarrow B \wedge C)$

$\{A \leftarrow \bigwedge(P(x)) \wedge (Q(x)), \quad B \leftarrow \bigwedge P(x), \quad C \leftarrow \bigwedge Q(x)\}$

$$\underbrace{(\bigwedge P(x)) \wedge (Q(x)) \rightarrow \bigwedge P(x)}_{P_1} \Rightarrow$$

$$\underbrace{(\bigwedge P(x)) \wedge (Q(x)) \rightarrow \bigwedge Q(x)}_{P_2}.\rightarrow.$$

$$\underbrace{(\bigwedge P(x)) \wedge (Q(x)) \rightarrow \bigwedge P(x) \wedge \bigwedge Q(x)}_{Q_2}$$

We first need to show that P_1 is a tautology:

(9.4) $A \wedge B \Rightarrow A$

$\{A \leftarrow \bigwedge P(x), \quad B \leftarrow \bigwedge Q(x)\}$

$$\underbrace{(\bigwedge P(x)) \wedge (\bigwedge Q(x))}_{P^*} \Rightarrow \underbrace{\bigwedge P(x).}_{Q^*}$$

(13.45) $$\underbrace{\bigwedge P(x)}_{Q^*} \Rightarrow \underbrace{P(x).}_{R^*}$$

Applying the *rule of transitivity*, (13.28) 'If $P^* \Rightarrow Q^*$ and $Q^* \Rightarrow R^*$ then $P^* \Rightarrow R^*$' allows us to deduce:

$$P_1 : \qquad (\bigwedge P(x)) \wedge (\bigwedge Q(x)) \Rightarrow P(x),$$

and in a like manner also:

$$P_2 : \qquad (\bigwedge P(x)) \wedge (\bigwedge Q(x)) \Rightarrow Q(x).$$

P_1 being a tautology, we may apply *modus ponens* to the above $P_1 \Rightarrow P_2 \rightarrow Q_2$, deducing $P_2 \Rightarrow Q_2$. And, as P_2 was shown to be a tautology we may deduce Q_2, i.e. (13.49b).

Division Three

Combinational Circuits

14 Canonical Normal Forms 141 (10 pages)
15 Shegalkin Normal Forms 151 (11 pages)
16 Synthesis Examples 162 (9 pages)
17 Concepts Old and New 171 (11 pages)
18 Minimisation Preliminaries 182 (11 pages)
19 Minimisation—an Excerpt 193 (14 pages)
20 Reduced Karnaugh Maps 207 (12 pages)
21 NOT-AND and NOT-OR 219 (12 pages)
22 Composition of Circuits 231 (11 pages)
23 Hazards 242 (11 pages)

Just as combinational circuits are used in building memory circuits (i.e., latches and sequential circuits) with feedback, the *theory* of combinational circuits is the basis on which the theory of all other types of circuits is founded. Thus the theory is presented quite extensively.

The backbone to the theory of combinational circuits is the theory of **normal forms** which to a good degree is put forth in Chapters 14 and 15. For the first time, Chapter 14 presents a formal derivation of the **canonical normal forms**—a derivation only possible due to the application of predicate logic (as opposed to propositional logic).

The prime chapter in this third Division of the book is Chapter 17 on **Concepts Old and New**, culminating in so-called **evaluation formulas**. These formulas, based on **covers** enable a new approach to minimisation. By this approach, minimisation refers to finding a minimal cover, while applying an evaluation formula to the cover found provides the minimised disjunctive or conjunctive logic form.

The evaluation formulas allow us to state how to evaluate so-called **reduced Karnaugh maps** (Chapter 20). Their importance goes well beyond their use with combinational circuits: They are central in evaluating memory and sequential circuits. The power of the method—even when applied to combinational circuits—is demonstrated in the example of Section 20.4.

The common theme of Chapters 21 and 22 is the design of **multilevel circuits**. While Chapter 21—on NAND and NOR circuits—is conventional, Chapter 22 is an original approach to decomposition. The presented solution is quite general and yet astonishingly simple in its basic idea.

Hazards (Chapter 23) are not all that important for combinational circuits. But, employing hazard-free combinational circuits in the design of a memory circuit turns out to be essential in obtaining faultless and fast circuits. Thus, hazards are considered in a more than customary thoroughness.

Chapter 14

Canonical Normal Forms

Normal forms lie at the heart of the theory of combinational circuits. A **normal form** is a general logic formula that allows us to develop a *unique* logic expression of a combinational circuit described by, say, a table of asserted events (TAE). There are a number of different normal forms, each based on one of the associative output connectives AND, OR, XOR, or EQU. In this chapter, we look at the more common normal forms whose output connectives are AND and OR, called **conjunctive** and **disjunctive normal forms**.

14.1 An Overview

We start this chapter with a summary. The first formula, (14.1), is called the **disjunctive normal form** (DNF) — or is sometimes referred to by the catchy phrase 'AND-to-OR formula'— the second formula, (14.2), is the **conjunctive normal form** (CNF) — or an OR-to-AND formula.

$$Y(x_1, x_2, \ldots, x_n) \Leftrightarrow \bigvee_{e=0}^{2^n-1} C_e(x_1, x_2, \ldots, x_n) \wedge Y(e), \qquad \textbf{(14.1)}$$

$$Y(x_1, x_2, \ldots, x_n) \Leftrightarrow \bigwedge_{e=0}^{2^n-1} D_e(x_1, x_2, \ldots, x_n) \vee Y(e). \qquad \textbf{(14.2)}$$

A formula is called a **normal form** if it produces an unique result when applied to a given TAE. **Disjunctive** refers to the *instantiation* ($\bigvee$) in (14.1) which is understood to be a multiple disjunction ($\vee$) while **conjunctive** refers to the *generalisation* ($\bigwedge$) in (14.2) which is understood to be a multiple conjunction ($\wedge$). Furthermore, the above normal forms are **canonical**, an expression referring to the *complete set of* — in this case — minterms $C_e(x_1, x_2, \cdots, x_n)$ and maxterms $D_e(x_1, x_2, \cdots, x_n)$.

The $Y(e) \in \{\mathbf{0}, \mathbf{1}\}$ of the above normal forms may, at this point, be viewed as logic *constants* which we call **specified** or **specifying output values** as a given combinational circuit is specified by them, or as they

	0	1	⋯	e	⋯	2^n-2	2^n-1
$x_1=$	0	0	⋯	e_1	⋯	1	1
$x_2=$	0	0	⋯	e_2	⋯	1	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
$x_{n-1}=$	0	0	⋯	e_{n-1}	⋯	1	1
$x_n=$	0	1	⋯	e_n	⋯	0	1
$y(x)=$	$y(0)$	$y(1)$	⋯	$y(e)$	⋯	$y(2^n-2)$	$y(2^n-1)$

Fig. 14–1 General table of asserted events

specify a circuit, as shown in the general TAE of Fig. 14–1 for which $Y(e)$ is taken to be $y(e) = 1$.

Each normal form can be expressed as a logic circuit. In the case of (14.1) this leads to the AND-to-OR circuit of Fig. 14–2. Each of the 2^n multiple conjunctions of the top row represents a minterm $C_e(x_1, x_2, \cdots, x_n)$. In the middle row, for each input event e, the minterm $C_e(\mathbf{x})$ is ANDed with the specifying output value $Y(e)$ which the input event is said to initiate. The disjunction of the two-input ANDs provides us with the output $Y(x_1, x_2, \ldots, x_n)$. This circuit is usually called a **multiplexer** (MUX). The characteristic of a multiplexer is that an input event e ensures that the *datum* $Y(e)$ is fed through to the output. Interpreting the circuit of Fig. 14–2 as a multiplexer allows us to view each $Y(e)$ as the output of some other circuit, and thus not as a constant but, rather, as a variable. This aspect will later prove important when discussing *Reduced Karnaugh Maps* in Chapter 20.

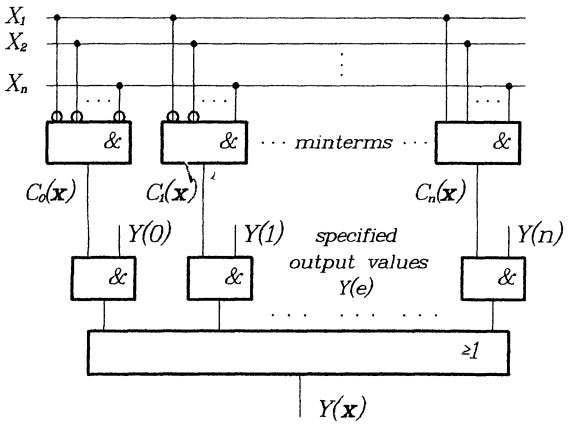


Fig. 14–2 Circuit of the disjunctive normal form

In a like manner as above, (14.2) can be represented by an OR-to-AND circuit. To understand how such a circuit works, remember that, for a given input event e, all maxterms are **1** except $D_e(\mathbf{e})$ which is **0**.

	0	1	2	3	4	5	6	7
$x_1 =$	0	0	0	0	1	1	1	1
$x_2 =$	0	0	1	1	0	0	1	1
$x_3 =$	0	1	0	1	0	1	0	1
$y =$	0	1	1	0	0	0	1	1

a)

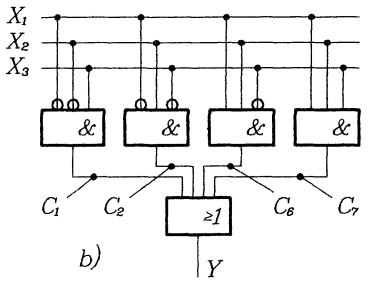


Fig. 14–3 An example AND-to-OR circuit

The circuits designed according to (14.1) or (14.2) are called **two level circuits**. What this means is best explained with the help of an example. Applying (14.1) to the TAE of Fig. 14–3a leads to

$$
\begin{aligned}
&Y(x_1, x_2, x_3) \Leftrightarrow \\
&\quad \big(C_0(\mathbf{x}) \wedge \mathbf{0}\big) \vee \big(C_1(\mathbf{x}) \wedge \mathbf{1}\big) \vee \big(C_2(\mathbf{x}) \wedge \mathbf{1}\big) \vee \big(C_3(\mathbf{x}) \wedge \mathbf{0}\big) \\
&\quad \vee \big(C_4(\mathbf{x}) \wedge \mathbf{0}\big) \vee \big(C_5(\mathbf{x}) \wedge \mathbf{0}\big) \vee \big(C_6(\mathbf{x}) \wedge \mathbf{1}\big) \vee \big(C_7(\mathbf{x}) \wedge \mathbf{1}\big) \Leftrightarrow \\
&\quad \mathbf{0} \vee C_1(\mathbf{x}) \vee C_2(\mathbf{x}) \vee \mathbf{0} \vee \mathbf{0} \vee \mathbf{0} \vee C_6(\mathbf{x}) \vee C_7(\mathbf{x}) \Leftrightarrow \\
&\quad C_1(\mathbf{x}) \vee C_2(\mathbf{x}) \vee C_6(\mathbf{x}) \vee C_7(\mathbf{x}) \Leftrightarrow \\
&\quad \overline{X}_1 \overline{X}_2 X_3 \vee \overline{X}_1 X_2 \overline{X}_3 \vee X_1 X_2 \overline{X}_3 \vee X_1 X_2 X_3 .
\end{aligned}
$$

The circuit that corresponds to this formula is shown on Fig. 14–3b. The batch of conjunctions, the gates the input signals first encounter, are said to belong to the first *level* while the output disjunction is said to comprise the second *level* of gates. These circuits are potentially fast as the input signals, on their individual paths to the output, traverse a minimal number of gates.

When looking at the final formula of the above example, you might like to remember the **procedure to obtain a DNF** of Section 5.3: *The logical output variable Y of a DNF is written as the disjunction of those minterms C_e for which the switching function is 1.*

The last aspect I wish to mention here is the necessity to simplify, or **minimise**, the logic expressions obtained directly by the above normal forms. For instance, the logic expression for the above example may be reduced to

$$Y(x_1, x_2, x_3) \Leftrightarrow X_1 X_2 \vee X_2 \overline{X}_3 \vee \overline{X}_1 \overline{X}_2 X_3 \Leftrightarrow X_2 (X_1 \vee \overline{X}_3) \vee \overline{X}_1 \overline{X}_2 X_3 .$$

It also turns out that the simpler expressions are less prone to a certain kind of error called a **static hazard**, a topic we shall find to be of some importance.

14.2 Direct Derivation of Normal Forms *

The challenge when developing a normal form is to do so by the exclusive use of axioms and theorems. This can not be done within the framework of propositional logic. As has proved true for mathematics in general, we need to employ predicate logic, and we shall use it in the form developed in Chapters 12 and 13. We start with the *theorem on identical functions*

$$(13.33) \qquad (x_1 = e_1) \wedge (x_2 = e_2) \wedge \ldots \wedge (x_n = e_n) \rightarrow f(x_1, x_2, \ldots x_n) = f(e_1, e_2, \ldots e_n) \Leftrightarrow \mathbf{1}.$$

But, to facilitate further development, note that we may switch relatively freely — i.e., with due caution — between a vector notation (3.12) $\mathbf{e} \stackrel{df}{=} (e_1, e_2, \ldots, e_n)$, a decimal representation (3.13) $e \stackrel{df}{=} \sum_{i=1}^{n} e_i.2^{n-i}$, and a representation as n-tuple $(e_1, e_2, \ldots, e_n)$. Analogous representations hold for the variables x_i.

The antecedent of the first formula of this section, (13.33), is — by (5.4) $C_e(x) \Leftrightarrow \bigwedge_{i=1}^{n}(x_i = e_i)$ — easily recognised as the minterm $C_e(x)$. Furthermore, using the abbreviations

$$y(x) \stackrel{df}{=} f(x_1, x_2, \ldots x_n), \qquad y(e) \stackrel{df}{=} f(e_1, e_2, \ldots e_n) \tag{14.3}$$

(13.33) may be expressed as

$$C_e(x) \rightarrow y(x) = y(e) \Leftrightarrow \mathbf{1}. \tag{14.4}$$

By (13.27) 'if $Q(x) \Leftrightarrow \mathbf{1}$ then $\bigwedge_x Q(x) \Leftrightarrow \mathbf{1}$' the generalisation of a tautology is itself a tautology:

$$\bigwedge_e C_e(x) \rightarrow y(x) = y(e) \Leftrightarrow \mathbf{1}. \tag{14.5}$$

This is the exact logical description of the TAE of Fig. 14–4.

0	1	$\cdots$	e	$\cdots$	$2^n - 1$
$x_1 = 0$	$x_1 = 0$	$\cdots$	$x_1 = e_1$	$\cdots$	$x_1 = 1$
$x_2 = 0$	$x_2 = 0$	$\cdots$	$x_2 = e_2$	$\cdots$	$x_2 = 1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$x_{n-1} = 0$	$x_{n-1} = 0$	$\cdots$	$x_{n-1} = e_{n-1}$	$\cdots$	$x_{n-1} = 1$
$x_n = 0$	$x_n = 1$	$\cdots$	$x_n = e_n$	$\cdots$	$x_n = 1$
$y(x) = y(0)$	$y(x) = y(1)$	$\cdots$	$y(x) = y(e)$	$\cdots$	$y(x) = y(2^n - 1)$

Fig. 14–4 General table of asserted events

In the above formula we change from using the numeric variables $y(x)$ and $y(e)$ to using logic variables $Y(x) \Leftrightarrow y(x) = 1$ and $Y(e) \Leftrightarrow y(e) = 1$. This transition is achieved with the help of the logical equivalence

$$y(x) = y(e) \Leftrightarrow Y(x) \leftrightarrow Y(e) \tag{14.6}$$

which is proved by the truth table of Fig. 14–5.

$y(x)$	$y(e)$	1 $y(x) = y(e)$	5 $\cdot\leftrightarrow\cdot$	2 $y(x) = 1$	4 $\leftrightarrow$	3 $y(e) = 1$
0	0	**1**	**1**	**0**	**1**	**0**
0	1	**0**	**1**	**0**	**0**	**1**
1	0	**0**	**1**	**1**	**0**	**0**
1	1	**1**	**1**	**1**	**1**	**1**

Fig. 14–5 Proving (14.6)

By *replacing* $y(x) = y(e)$ of (14.5) by $Y(x) \leftrightarrow Y(e)$ we obtain

$$\bigwedge_e \underbrace{C_e(x)}_{A} \rightarrow \big(\underbrace{Y(x)}_{B} \leftrightarrow \underbrace{Y(e)}_{C}\big) \Leftrightarrow \mathbf{1}. \tag{14.7}$$

As $\rightarrow$ is left distributive over $\leftrightarrow$, i.e. $A \rightarrow (B \leftrightarrow C) \Leftrightarrow (A \rightarrow B) \leftrightarrow (A \rightarrow C)$ — see Fig. 10–9 — we may write (14.7) as

$$\bigwedge_e \underbrace{C_e(x) \rightarrow Y(x)}_{P(e)} \leftrightarrow \underbrace{C_e(x) \rightarrow Y(e)}_{Q(e)} \Leftrightarrow \mathbf{1} \tag{14.8}$$

by the *convention of adhesion* (section 8.1) which states that $\rightarrow$ has a greater adhesion to its arguments than $\leftrightarrow$ has. Looking back at (13.52), $\bigwedge(P(e) \leftrightarrow Q(e)) \Rightarrow (\bigwedge P(e)) \leftrightarrow (\bigwedge Q(e))$, we may distribute the universal quantifier $(\bigwedge)$ of (14.8), obtaining the logic implication

$$\bigwedge_e (C_e(x) \rightarrow Y(x) \leftrightarrow C_e(x) \rightarrow Y(e)) \Rightarrow$$
$$(\bigwedge_e C_e(x) \rightarrow Y(x)) \leftrightarrow (\bigwedge_e C_e(x) \rightarrow Y(e)). \tag{14.9}$$

The theorem (13.52), employed above, ensures that (14.9) is a tautology, while (14.8) ensures that the antecedent of (14.9) is also a tautology. Thus, *modus ponens* (13.21) allows us to deduce that the consequent of (14.9) is also a tautology:

$$\bigwedge_e \big(\underbrace{C_e(x)}_{P(e)} \rightarrow \underbrace{Y(x)}_{Q}\big) \Leftrightarrow \bigwedge_e (C_e(x) \rightarrow Y(e)). \tag{14.10}$$

Please note that $Y(x)$, on the left side of the above expression, does *not* contain e as a free variable so that we may apply (13.62) $\bigwedge(P(e) \rightarrow Q) \Leftrightarrow (\bigvee P(e)) \rightarrow Q$, rewriting (14.10) as

$$(\bigvee_e C_e(x)) \rightarrow Y(x) \Leftrightarrow \bigwedge_e (C_e(x) \rightarrow Y(e)). \tag{14.11}$$

By (11.2) $\bigvee_e C_e(x) \Leftrightarrow \mathbf{1}$ the disjunction of all minterms is a tautology, and as $\mathbf{1} \rightarrow Y(x) \Leftrightarrow Y(x)$ we can write (14.11) as

$$Y(x) \Leftrightarrow \bigwedge_e C_e(x) \rightarrow Y(e). \tag{**14.12**}$$

Now, transforming the IMPLICATION to an OR with the help of $A \rightarrow B \Leftrightarrow \neg A \vee B$, and avoiding the incurred negation by using (11.1) $D_e(x) \Leftrightarrow \neg C_e(x)$, we rewrite (14.12) as

$$Y(x) \Leftrightarrow \bigwedge_e D_e(x) \vee Y(e) \tag{14.13}$$

which is (14.2), the first of the formulas we set out to derive. Had we chosen to use $\overline{Y}(x) \Leftrightarrow y(x) = 0$ and $\overline{Y}(e) \Leftrightarrow y(e) = 0$, and thus

$$y(x) = y(e) \Leftrightarrow \overline{Y}(x) \leftrightarrow \overline{Y}(e) \tag{14.14}$$

instead of (14.6), we would have obtained the **conjunctive normal form of the inverted function**

$$\overline{Y}(x) \Leftrightarrow \bigwedge_e D_e(x) \vee \overline{Y}(e). \tag{**14.15**}$$

We use the above normal forms, (14.13) and (14.15) to develop the *disjunctive normal forms.* We first negate (14.15)

$$\overline{\overline{Y}}(x) \Leftrightarrow \neg \bigwedge_e D_e(x) \vee \overline{Y}(e).$$

Due to (13.67) $\neg \bigwedge P(x) \Leftrightarrow \bigvee \neg P(x)$ the above formula may be rewritten as

$$\overline{\overline{Y}}(x) \Leftrightarrow \bigvee_e \neg(D_e(x) \vee \overline{Y}(e)).$$

Now employing DeMorgan (9.27) $\neg(A \vee B) \Leftrightarrow \neg A \wedge \neg B$

$$\overline{\overline{Y}}(x) \Leftrightarrow \bigvee_e \overline{D}_e(x) \wedge \overline{\overline{Y}}(e)$$

and then (9.22) $A \Leftrightarrow \neg\neg A$ and (11.1) $D_e(x) \Leftrightarrow \neg C_e(x)$ leads to the *disjunctive normal form* (DNF)

$$Y(x) \Leftrightarrow \bigvee_e C_e(x) \wedge Y(e). \tag{14.1}$$

Had we started by negating (14.13), we would have wound up with the **disjunctive normal form of the inverted function**:

$$\overline{Y}(x) \Leftrightarrow \bigvee_e C_e(x) \wedge \overline{Y}(e). \tag{14.16}$$

14.3 Shannon's Expansion Theorems *

Shannon [1938] asserted that the (output) value $F(X_1, X_2, \ldots, X_n)$ of any logic function $F : \{\mathbf{0}, \mathbf{1}\}^n \mapsto \{\mathbf{0}, \mathbf{1}\}$ could be expressed by one of the following *dual* formulas

$$\begin{aligned} &F(X_1, \ldots, X_{i-1}, X_i, X_{i+1}, \ldots, X_n) \Leftrightarrow \\ &\qquad (X_i \wedge F(X_1, \ldots, X_{i-1}, \mathbf{1}, X_{i+1}, \ldots, X_n)) \\ &\qquad \vee (\overline{X}_i \wedge F(X_1, \ldots, X_{i-1}, \mathbf{0}, X_{i+1}, \ldots, X_n)), \end{aligned} \tag{14.17}$$

$$\begin{aligned} &F(X_1, \ldots, X_{i-1}, X_i, X_{i+1}, \ldots, X_n) \Leftrightarrow \\ &\qquad (X_i \vee F(X_1, \ldots, X_{i-1}, \mathbf{0}, X_{i+1}, \ldots, X_n)) \\ &\qquad \wedge (\overline{X}_i \vee F(X_1, \ldots, X_{i-1}, \mathbf{1}, X_{i+1}, \ldots, X_n)). \end{aligned} \tag{14.18}$$

Clearly, these **expansion** formulas are correct as you can see by first substituting **1** and then **0** for X_i. But, as these formulas can only be intuitively guessed at within the framework of propositional logic, it is interesting to see how they can systematically be developed from the axioms of predicate logic. To reduce writing space, we set the value of the index i to 1 which, by the way, we can do without the loss of generality. We start by writing axiom

$$x_1 = e_1 \Rightarrow f(x_1, x_2, \ldots, x_n) = f(e_1, x_2, \ldots, x_n) \tag{13.20}$$

separately for each of the cases $e \hookleftarrow 0$ and $e \hookleftarrow 1$

$$\begin{aligned} x_1 = 0 &\Rightarrow f(x_1, x_2, \ldots, x_n) = f(0, x_2, \ldots, x_n), \\ x_1 = 1 &\Rightarrow f(x_1, x_2, \ldots, x_n) = f(1, x_2, \ldots, x_n). \end{aligned}$$

Writing these formulas in a logic form

$$\begin{aligned} \overline{X}_1 &\rightarrow (F(X_1, X_2, \ldots, X_n) \leftrightarrow F(\mathbf{0}, X_2, \ldots, X_n)) \Leftrightarrow \mathbf{1}, \\ X_1 &\rightarrow (F(X_1, X_2, \ldots, X_n) \leftrightarrow F(\mathbf{1}, X_2, \ldots, X_n)) \Leftrightarrow \mathbf{1}, \end{aligned}$$

then using distributivity

$$(\overline{X}_1 \to F(X_1, X_2, \ldots, X_n)) \leftrightarrow (\overline{X}_1 \to F(\mathbf{0}, X_2, \ldots, X_n)) \Leftrightarrow \mathbf{1},$$
$$(X_1 \to F(X_1, X_2, \ldots, X_n)) \leftrightarrow (X_1 \to F(\mathbf{1}, X_2, \ldots, X_n)) \Leftrightarrow \mathbf{1},$$

and finally employing $P \to Q \Leftrightarrow \overline{P} \vee Q$ leads to

$$\underbrace{(X_1 \vee F(X_1, X_2, \ldots, X_n))}_{P_1} \Leftrightarrow \underbrace{(X_1 \vee F(\mathbf{0}, X_2, \ldots, X_n))}_{Q_1},$$
$$\underbrace{(\overline{X}_1 \vee F(X_1, X_2, \ldots, X_n))}_{P_2} \Leftrightarrow \underbrace{(\overline{X}_1 \vee F(\mathbf{1}, X_2, \ldots, X_n))}_{Q_2}.$$

It is worth noting that the expressions P_1 and P_2, on the left of the above logical equivalencies ($P_1 \Leftrightarrow Q_1$, and $P_2 \Leftrightarrow Q_2$) and those on the right, Q_1 and Q_2, may be ANDed separately according to

$$(P_1 \leftrightarrow Q_1) \wedge (P_2 \leftrightarrow Q_2) \Rightarrow (P_1 \wedge P_2) \leftrightarrow (Q_1 \wedge Q_2). \tag{14.19}$$

				1	5	2	7	3	6	4
P_1	Q_1	P_2	Q_2	$(P_1 \leftrightarrow Q_1)$	$\wedge$	$(P_2 \leftrightarrow Q_2)$	$.\to.$	$(P_1 \wedge P_2)$	$\leftrightarrow$	$(Q_1 \wedge Q_2)$
0	0	0	0	1	1	1	1	0	1	0
0	0	0	1	1	0	0	1	0	1	0
0	0	1	0	1	0	0	1	0	1	0
0	0	1	1	1	1	1	1	0	1	0
0	1	0	0	0	0	1	1	0	1	0
0	1	0	1	0	0	0	1	0	0	1
0	1	1	0	0	0	0	1	0	1	0
0	1	1	1	0	0	1	1	0	0	1
1	0	0	0	0	0	1	1	0	1	0
1	0	0	1	0	0	0	1	0	1	0
1	0	1	0	0	0	0	1	1	0	0
1	0	1	1	0	0	1	1	1	0	0
1	1	0	0	1	1	1	1	0	1	0
1	1	0	1	1	0	0	1	0	0	1
1	1	1	0	1	0	0	1	1	0	0
1	1	1	1	1	1	1	1	1	1	1

Fig. 14–6 Proving (14.19)

This logical implication is proved in the truth table of Fig. 14–6. In our case, the equivalencies $P_1 \leftrightarrow Q_1$ and $P_2 \leftrightarrow Q_2$ are tautologies so that by *modus*

ponens we deduce from (14.19) that $P_1 \wedge P_2 \Leftrightarrow Q_1 \wedge Q_2$, i.e.

$$\underbrace{(X_1 \vee F(X_1, X_2, \ldots, X_n))}_{P_1} \wedge \underbrace{(\overline{X}_1 \vee F(X_1, X_2, \ldots, X_n))}_{P_2} \Leftrightarrow$$
$$\underbrace{(X_1 \vee F(\mathbf{0}, X_2, \ldots, X_n))}_{Q_1} \wedge \underbrace{(\overline{X}_1 \vee F(\mathbf{1}, X_2, \ldots, X_n))}_{Q_2}.$$

Applying distributivity $(X \vee F) \wedge (\overline{X} \vee F) \Leftrightarrow (X \wedge \overline{X}) \vee F$ to the left side of the above formula leads to (14.18). (14.17) is valid due to

$$(A \vee B) \wedge (\overline{A} \vee C) \Leftrightarrow (A \wedge C) \vee (\overline{A} \wedge B), \tag{14.20}$$

the proof of which I lay trustingly in your hands.

14.4 Shannon's Expansion to Normal Forms

Both forms of Shannon's expansion theorem (14.17) and (14.18) are used to completely expand a logic function $F : \{\mathbf{0},\mathbf{1}\}^n \mapsto \{\mathbf{0},\mathbf{1}\}$ to either its *disjunction of minterms* or its *conjunction of maxterms* representation. This expansion process is demonstrated by successively applying (14.17) to one variable after another, starting with the expansion of $F(X_1, X_2, \ldots, X_n)$ by X_1

$$\begin{aligned} F(X_1, X_2, \ldots, X_n) \Leftrightarrow & X_1 F(\mathbf{1}, X_2, \ldots, X_n) \\ & \vee \overline{X}_1 F(\mathbf{0}, X_2, \ldots, X_n). \end{aligned}$$

Each of the expressions $F(\mathbf{1}, X_2, \ldots, X_n)$ and $F(\mathbf{0}, X_2, \ldots, X_n)$ is then expanded by X_2

$$\begin{aligned} & F(X_1, X_2, \ldots, X_n) \Leftrightarrow \\ & \quad X_1 \big(X_2 F(\mathbf{1}, \mathbf{1}, X_3 \ldots, X_n) \vee \overline{X}_2 F(\mathbf{1}, \mathbf{0}, X_3 \ldots, X_n)\big) \\ & \quad \vee \overline{X}_1 \big(X_2 F(\mathbf{0}, \mathbf{1}, X_3 \ldots, X_n) \vee \overline{X}_2 F(\mathbf{0}, \mathbf{0}, X_3 \ldots, X_n)\big) \Leftrightarrow \\ & \quad X_1 X_2 F(\mathbf{1}, \mathbf{1}, X_3 \ldots, X_n) \vee X_1 \overline{X}_2 F(\mathbf{1}, \mathbf{0}, X_3 \ldots, X_n) \\ & \quad \vee \overline{X}_1 X_2 F(\mathbf{0}, \mathbf{1}, X_3 \ldots, X_n) \vee \overline{X}_1 \overline{X}_2 F(\mathbf{0}, \mathbf{0}, X_3 \ldots, X_n). \end{aligned}$$

After expanding by *all* variables, we obtain

$$\begin{aligned} F(X_1, X_2, \ldots, X_n) \Leftrightarrow & X_1 X_2 \ldots X_n F(\mathbf{1}, \mathbf{1}, \ldots, \mathbf{1}) \\ & \vee X_1 X_2 \ldots \overline{X}_n F(\mathbf{1}, \mathbf{1}, \ldots, \mathbf{0}) \\ & \vdots \\ & \vee \overline{X}_1 \overline{X}_2 \ldots X_n F(\mathbf{0}, \mathbf{0}, \ldots, \mathbf{1}) \\ & \vee \overline{X}_1 \overline{X}_2 \ldots \overline{X}_n F(\mathbf{0}, \mathbf{0}, \ldots, \mathbf{0}). \end{aligned}$$

You will, no doubt, recognise the conjunction of input variables of each row, from top to bottom, as the minterms $C_{2^n-1}(\mathbf{x})$, $C_{2^n-2}(\mathbf{x})$, ..., $C_1(\mathbf{x})$, $C_0(\mathbf{x})$ while the output values $F(\mathbf{1},\mathbf{1},\ldots,\mathbf{1})$, $F(\mathbf{1},\mathbf{1},\ldots,\mathbf{0})$, $F(\mathbf{0},\mathbf{0},\ldots,\mathbf{1})$, and $F(\mathbf{0},\mathbf{0},\ldots,\mathbf{0})$ were previously referred to as $Y(2^n-1)$, $Y(2^n-2)$, $Y(1)$ and $Y(0)$, respectively. Thus, the above formula is (14.1).

Chapter 15

Shegalkin Normal Forms

The normal forms discussed in this chapter are based on XOR and EQU as output connectives. The XOR-normal form is obscurely ascribed to **Reed** and **Muller**. Yet, as it seems to have been first considered and discussed systematically by **Shegalkin** [1927] I feel it a matter of fairness to call it and its dual a **Shegalkin normal form**. For further reading see **Almaini** [1989].

15.1 An Overview

The **Shegalkin normal form** consists of conjunctions of *noninverted* logic input variables X_i connected via the XOR operation. Which *combinations* of logic input variables occur in the final polynomial is determined by those **Shegalkin coefficients** $\Phi \in \{\mathbf{0}, \mathbf{1}\}$ that are **1**. The Shegalkin polynomial has the following general form:

$$
\begin{aligned}
F(X_1, X_2, \ldots, X_n) \Leftrightarrow \\
&\underbrace{\Phi_{0\ldots0}}_{\binom{n}{0}} \oplus \\
&\underbrace{X_1\Phi_{x_1 0\ldots0} \oplus X_2\Phi_{0x_2 0\ldots0} \oplus \cdots \oplus X_n\Phi_{0\ldots0x_n}}_{\binom{n}{1}\ \text{single-variable expressions}} \oplus \\
&\underbrace{X_1X_2\Phi_{x_1x_2 0\ldots0} \oplus X_1X_3\Phi_{x_1 0x_3 0\ldots0} \oplus \cdots \oplus\ X_{n-1}X_n\Phi_{0\ldots0x_{n-1}x_n}}_{\binom{n}{2}\ \text{dual-variable expressions}} \oplus \\
&\ \vdots \\
&\underbrace{X_1X_2\ldots X_n\Phi_{x_1x_2\ldots x_n}}_{\binom{n}{n}} .
\end{aligned}
\qquad (15.1)
$$

Each AND-term of (15.1) contains a *combination* of input variables. One way of finding all possible combinations is illustrated in the column of

the input variables x_1, x_2, x_3 of Fig. 15–1. Let $e_1, e_2, e_3 \in \{0,1\}$ refer to the binary digits of a specified row of the column of input variables. Then, multiplying the binary digits (e_1, e_2, e_3) by (x_1, x_2, x_3) we obtain $(x_1 \cdot e_1,\ x_2 \cdot e_2,\ x_3 \cdot e_3)$. We refer to this triple as $(\varphi_1, \varphi_2, \varphi_3)$ and use it as the index triple of the coefficient $\Phi_{\varphi_1\varphi_2\varphi_3}$. The combinations are obtained by dropping (i.e., not writing) those φ_i that are zero, e.g., writing x_3 instead of $00x_3$.

$x_1\cdot$	$x_2\cdot$	$x_3\cdot$	$\Phi_{\varphi_1\varphi_2\varphi_3}$		$Y(0)$	$Y(1)$	$Y(2)$	$Y(3)$	$Y(4)$	$Y(5)$	$Y(6)$	$Y(7)$
0	0	0	Φ_{000}	$\Leftrightarrow$	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**1**
0	0	1	Φ_{00x_3}	$\Leftrightarrow$	**1**	**0**	**1**	**0**	**0**	**1**	**0**	
0	1	0	Φ_{0x_20}	$\Leftrightarrow$	**1**	**1**	**1**	**0**	**1**	**1**		
0	1	1	$\Phi_{0x_2x_3}$	$\Leftrightarrow$	**0**	**0**	**1**	**1**	**0**			
1	0	0	Φ_{x_100}	$\Leftrightarrow$	**0**	**1**	**0**	**1**				
1	0	1	$\Phi_{x_10x_3}$	$\Leftrightarrow$	**1**	**1**	**1**			XOR		
1	1	0	$\Phi_{x_1x_20}$	$\Leftrightarrow$	**0**	**0**						
1	1	1	$\Phi_{x_1x_2x_3}$	$\Leftrightarrow$	**0**							

Fig. 15-1 The XOR logic triangle

To determine the Shegalkin coefficients $\Phi_{\varphi_1\varphi_2\varphi_3}$ we make use of the **XOR logic triangle** of truth values in Fig. 15–1. Each truth value of the *top row* of the logic triangle is understood to be a specified logic output value $Y(e)$ of a given TAE, in our case, of the TAE of Fig. 15–2a. All other truth values of the logic triangle are obtained as shown in Fig. 15–2b, i.e., $A_{i+1,j} \Leftrightarrow A_{i,j} \oplus A_{i,j+1}$ where any $A_{i+1,j}$ is a truth value of the logic triangle.

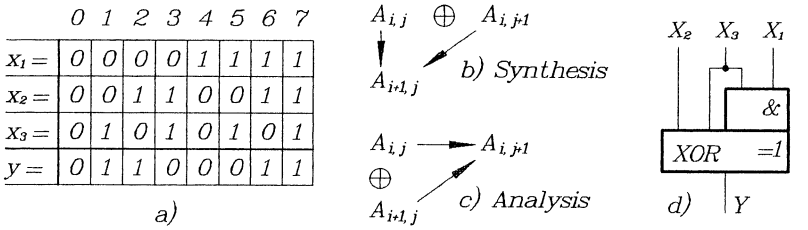


Fig. 15–2 Synthesis and analysis of a Shegalkin circuit

The Shegalkin coefficients are the truth values of the *first column* of the XOR logic triangle (of Fig. 15–1). This allows us to write the Shegalkin normal form as

$$Y(x_1, x_2, x_3) \Leftrightarrow$$

$$\mathbf{0} \oplus (X_3 \wedge \mathbf{1}) \oplus (X_2 \wedge \mathbf{1}) \oplus (X_2X_3 \wedge \mathbf{0}) \oplus (X_1 \wedge \mathbf{0})$$
$$\oplus (X_1X_3 \wedge \mathbf{1}) \oplus (X_1X_2 \wedge \mathbf{0}) \oplus (X_1X_2X_3 \wedge \mathbf{0}) \Leftrightarrow$$

$$\mathbf{0} \oplus X_3 \oplus X_2 \oplus \mathbf{0} \oplus \mathbf{0} \oplus X_1X_3 \oplus \mathbf{0} \oplus \mathbf{0} \Leftrightarrow$$

$$X_3 \oplus X_2 \oplus X_1X_3.$$

The circuit corresponding to the above formula is that of Fig. 15–2d. Developing a circuit from a specification (as is the TAE) is called the **synthesis** of the circuit. The reverse operation, finding, say, the TAE associated with the circuit, is called **analysis**. The Shegalkin coefficients may simply be read off from the formula of the circuit and used as the values of the left column of a XOR logic triangle. All the other values of the logic triangle are obtained as shown in Fig. 15–2c, i.e., $A_{i,j+1} \Leftrightarrow A_{i,j} \oplus A_{i+1,j}$. The logic values of the top row of the logic triangle thus obtained are the specifying output values of the circuit, and thus the result of the analysis.

15.2 Developing the Shegalkin Polynomial *

It is quite straight forward to assume that Shannon's expansion theorem

$$\begin{aligned} F(X_1,\ldots,X_{i-1},X_i,X_{i+1},\ldots,X_n) \Leftrightarrow \\ &\big(X_i \wedge F(X_1,\ldots,X_{i-1},\mathbf{1},X_{i+1},\ldots,X_n)\big) \\ &\vee \big(\overline{X}_i \wedge F(X_1,\ldots,X_{i-1},\mathbf{0},X_{i+1},\ldots,X_n)\big) \end{aligned} \tag{14.17}$$

remains valid if the OR operation is substituted by XOR:

$$\begin{aligned} F(X_1,\ldots,X_{i-1},X_i,X_{i+1},\ldots,X_n) \Leftrightarrow \\ &\big(X_i \wedge F(X_1,\ldots,X_{i-1},\mathbf{1},X_{i+1},\ldots,X_n)\big) \\ &\oplus \big(\overline{X}_i \wedge F(X_1,\ldots,X_{i-1},\mathbf{0},X_{i+1},\ldots,X_n)\big). \end{aligned} \tag{15.2}$$

To prove this assumption, we first substitute X_i by $\mathbf{0}$, and then by $\mathbf{1}$:

$$\begin{aligned} F(X_1,\ldots,X_{i-1},\mathbf{0},X_{i+1},\ldots,X_n) \Leftrightarrow \\ &\big(\mathbf{0} \wedge F(X_1,\ldots,X_{i-1},\mathbf{1},X_{i+1},\ldots,X_n)\big) \\ &\oplus \big(\mathbf{1} \wedge F(X_1,\ldots,X_{i-1},\mathbf{0},X_{i+1},\ldots,X_n)\big) \Leftrightarrow \\ &\mathbf{0} \oplus F(X_1,\ldots,X_{i-1},\mathbf{0},X_{i+1},\ldots,X_n)\big) \Leftrightarrow \\ &F(X_1,\ldots,X_{i-1},\mathbf{0},X_{i+1},\ldots,X_n)\big), \end{aligned}$$

$$\begin{aligned} F(X_1,\ldots,X_{i-1},\mathbf{1},X_{i+1},\ldots,X_n) \Leftrightarrow \\ &\big(\mathbf{1} \wedge F(X_1,\ldots,X_{i-1},\mathbf{1},X_{i+1},\ldots,X_n)\big) \\ &\oplus \big(\mathbf{0} \wedge F(X_1,\ldots,X_{i-1},\mathbf{0},X_{i+1},\ldots,X_n)\big) \Leftrightarrow \\ &F(X_1,\ldots,X_{i-1},\mathbf{1},X_{i+1},\ldots,X_n)\big) \oplus \mathbf{0} \Leftrightarrow \\ &F(X_1,\ldots,X_{i-1},\mathbf{1},X_{i+1},\ldots,X_n)\big), \end{aligned}$$

these results proving (15.2). For our present application, we need to transform (15.2) in the following way:

$$\begin{aligned}
&F(X_1, X_2, X_3, \ldots, X_n) \Leftrightarrow \\
&\quad \big(X_i \wedge F(X_1, \ldots, X_{i-1}, \mathbf{1}, X_{i+1}, \ldots, X_n)\big) \\
&\quad \oplus \big(\overline{X}_i \wedge F(X_1, \ldots, X_{i-1}, \mathbf{0}, X_{i+1}, \ldots, X_n)\big) \Leftrightarrow \\
&\quad \big(X_i \wedge F(X_1, \ldots, X_{i-1}, \mathbf{1}, X_{i+1}, \ldots, X_n)\big) \\
&\quad \oplus \big((\mathbf{1} \oplus X_i) \wedge F(X_1, \ldots, X_{i-1}, \mathbf{0}, X_{i+1}, \ldots, X_n)\big) \Leftrightarrow \\
&\quad \big(X_i \wedge F(X_1, \ldots, X_{i-1}, \mathbf{1}, X_{i+1}, \ldots, X_n)\big) \\
&\quad \oplus \big(\mathbf{1} \wedge F(X_1, \ldots, X_{i-1}, \mathbf{0}, X_{i+1}, \ldots, X_n)\big) \\
&\quad \oplus \big(X_i \wedge F(X_1, \ldots, X_{i-1}, \mathbf{0}, X_{i+1}, \ldots, X_n)\big),
\end{aligned}$$

this being rewritten as

$$\begin{aligned}
&F(X_1, X_2, X_3, \ldots, X_n) \Leftrightarrow \\
&\quad F(X_1, \ldots, X_{i-1}, \mathbf{0}, X_{i+1}, \ldots, X_n) \\
&\quad \oplus X_i \wedge \big(F(X_1, \ldots, X_{i-1}, \mathbf{0}, X_{i+1}, \ldots, X_n) \\
&\qquad\quad \oplus F(X_1, \ldots, X_{i-1}, \mathbf{1}, X_{i+1}, \ldots, X_n)\big) \qquad (\mathbf{15.3})
\end{aligned}$$

The above formula is the **XOR expansion theorem**. Let us now consider the first steps in developing the Shegalkin polynomial. For $i \hookleftarrow 1$ we may write (15.3) as

$$\begin{aligned}
&F(X_1, X_2, X_3, \ldots, X_n) \Leftrightarrow \\
&\quad F(\mathbf{0}, X_2, X_3, \ldots, X_n) \\
&\quad \oplus X_1\big(F(\mathbf{0}, X_2, X_3, \ldots, X_n) \oplus F(\mathbf{1}, X_2, X_3, \ldots, X_n)\big). \qquad (15.4)
\end{aligned}$$

Next, $F(\mathbf{0}, X_2, X_3, \ldots, X_n)$ and $F(\mathbf{1}, X_2, X_3, \ldots, X_n)$ are expanded by X_2 using (15.3):

$$\begin{aligned}
&F(\mathbf{0}, X_2, X_3, \ldots, X_n) \Leftrightarrow F(\mathbf{0}, \mathbf{0}, X_3, \ldots, X_n) \\
&\quad \oplus X_2\big(F(\mathbf{0}, \mathbf{0}, X_3, \ldots, X_n) \oplus F(\mathbf{0}, \mathbf{1}, X_3, \ldots, X_n)\big), \\
&F(\mathbf{1}, X_2, X_3, \ldots, X_n) \Leftrightarrow F(\mathbf{1}, \mathbf{0}, X_3, \ldots, X_n) \\
&\quad \oplus X_2\big(F(\mathbf{1}, \mathbf{0}, X_3, \ldots, X_n) \oplus F(\mathbf{1}, \mathbf{1}, X_3, \ldots, X_n)\big).
\end{aligned}$$

Using these formulas in (15.4) allows us to write

$$\begin{aligned}
&F(X_1, X_2, X_3, \ldots, X_n) \Leftrightarrow \\
&\quad F(\mathbf{0}, \mathbf{0}, X_3, \ldots, X_n) \\
&\quad \oplus X_2 F(\mathbf{0}, \mathbf{0}, X_3, \ldots, X_n) \oplus X_2 F(\mathbf{0}, \mathbf{1}, X_3, \ldots, X_n) \\
&\quad \oplus X_1 F(\mathbf{0}, \mathbf{0}, X_3, \ldots, X_n) \\
&\quad \oplus X_1 X_2 F(\mathbf{0}, \mathbf{0}, X_3, \ldots, X_n) \oplus X_1 X_2 F(\mathbf{0}, \mathbf{1}, X_3, \ldots, X_n) \\
&\quad \oplus X_1 F(\mathbf{1}, \mathbf{0}, X_3, \ldots, X_n) \\
&\quad \oplus X_1 X_2 F(\mathbf{1}, \mathbf{0}, X_3, \ldots, X_n) \oplus X_1 X_2 F(\mathbf{1}, \mathbf{1}, X_3, \ldots, X_n).
\end{aligned}$$

Rearranging the conjunctions and applying distributivity, we may write

$$\begin{aligned}
F(X_1, X_2, X_3, \ldots, X_n) \Leftrightarrow \\
\quad & F(\mathbf{0}, \mathbf{0}, X_3, \ldots, X_n) \\
& \oplus X_1\big(F(\mathbf{0}, \mathbf{0}, X_3, \ldots, X_n) \oplus F(\mathbf{1}, \mathbf{0}, X_3, \ldots, X_n)\big) \\
& \oplus X_2\big(F(\mathbf{0}, \mathbf{0}, X_3, \ldots, X_n) \oplus F(\mathbf{0}, \mathbf{1}, X_3, \ldots, X_n)\big) \\
& \oplus X_1X_2\big(F(\mathbf{0}, \mathbf{0}, X_3, \ldots, X_n) \oplus F(\mathbf{0}, \mathbf{1}, X_3, \ldots, X_n) \\
& \oplus F(\mathbf{1}, \mathbf{0}, X_3, \ldots, X_n) \oplus F(\mathbf{1}, \mathbf{1}, X_3, \ldots, X_n)\big). \qquad (15.5)
\end{aligned}$$

Expanding by the rest of the variables X_3, X_4, ..., X_n leads to Shegalkin's polynomial (15.1) in which the coefficients Φ are multiple XOR of the output values of the function F.

15.3 Shegalkin Coefficients

Continuing the derivation of the previous section, we may write the Shegalkin polynomial for three input variables as

$$\begin{aligned}
F(X_1, X_2, X_3) \Leftrightarrow \\
\quad & F(\mathbf{0}, \mathbf{0}, \mathbf{0}) \\
& \oplus X_1\big(F(\mathbf{0}, \mathbf{0}, \mathbf{0}) \oplus F(\mathbf{1}, \mathbf{0}, \mathbf{0})\big) \\
& \oplus X_2\big(F(\mathbf{0}, \mathbf{0}, \mathbf{0}) \oplus F(\mathbf{0}, \mathbf{1}, \mathbf{0})\big) \\
& \oplus X_3\big(F(\mathbf{0}, \mathbf{0}, \mathbf{0}) \oplus F(\mathbf{0}, \mathbf{0}, \mathbf{1})\big) \\
& \oplus X_1X_2\big(F(\mathbf{0}, \mathbf{0}, \mathbf{0}) \oplus F(\mathbf{0}, \mathbf{1}, \mathbf{0}) \oplus F(\mathbf{1}, \mathbf{0}, \mathbf{0}) \oplus F(\mathbf{1}, \mathbf{1}, \mathbf{0})\big) \\
& \oplus X_1X_3\big(F(\mathbf{0}, \mathbf{0}, \mathbf{0}) \oplus F(\mathbf{0}, \mathbf{0}, \mathbf{1}) \oplus F(\mathbf{1}, \mathbf{0}, \mathbf{0}) \oplus F(\mathbf{1}, \mathbf{0}, \mathbf{1})\big) \\
& \oplus X_2X_3\big(F(\mathbf{0}, \mathbf{0}, \mathbf{0}) \oplus F(\mathbf{0}, \mathbf{0}, \mathbf{1}) \oplus F(\mathbf{0}, \mathbf{1}, \mathbf{0}) \oplus F(\mathbf{0}, \mathbf{1}, \mathbf{1})\big) \\
& \oplus X_1X_2X_3\big(F(\mathbf{0}, \mathbf{0}, \mathbf{0}) \oplus F(\mathbf{0}, \mathbf{0}, \mathbf{1}) \oplus F(\mathbf{0}, \mathbf{1}, \mathbf{0}) \oplus F(\mathbf{0}, \mathbf{1}, \mathbf{1}) \\
& \qquad \oplus F(\mathbf{1}, \mathbf{0}, \mathbf{0}) \oplus F(\mathbf{1}, \mathbf{0}, \mathbf{1}) \oplus F(\mathbf{1}, \mathbf{1}, \mathbf{0}) \oplus F(\mathbf{0}, \mathbf{1}, \mathbf{1})\big). \qquad (15.6)
\end{aligned}$$

Explicitly, the Shegalkin coefficients $\Phi_{\varphi_1,\varphi_2,\varphi_3}$ of the above polynomial are the following expressions:

$$\begin{aligned}
\Phi_{000} &\Leftrightarrow F(\mathbf{0}, \mathbf{0}, \mathbf{0}) \\
\Phi_{x_100} &\Leftrightarrow F(\mathbf{0}, \mathbf{0}, \mathbf{0}) \oplus F(\mathbf{1}, \mathbf{0}, \mathbf{0}) \\
\Phi_{0x_20} &\Leftrightarrow F(\mathbf{0}, \mathbf{0}, \mathbf{0}) \oplus F(\mathbf{0}, \mathbf{1}, \mathbf{0}) \\
\Phi_{00x_3} &\Leftrightarrow F(\mathbf{0}, \mathbf{0}, \mathbf{0}) \oplus F(\mathbf{0}, \mathbf{0}, \mathbf{1}) \\
\Phi_{x_1x_20} &\Leftrightarrow F(\mathbf{0}, \mathbf{0}, \mathbf{0}) \oplus F(\mathbf{0}, \mathbf{1}, \mathbf{0}) \oplus F(\mathbf{1}, \mathbf{0}, \mathbf{0}) \oplus F(\mathbf{1}, \mathbf{1}, \mathbf{0}) \\
\Phi_{x_10x_3} &\Leftrightarrow F(\mathbf{0}, \mathbf{0}, \mathbf{0}) \oplus F(\mathbf{0}, \mathbf{0}, \mathbf{1}) \oplus F(\mathbf{1}, \mathbf{0}, \mathbf{0}) \oplus F(\mathbf{1}, \mathbf{0}, \mathbf{1}) \\
\Phi_{0x_2x_3} &\Leftrightarrow F(\mathbf{0}, \mathbf{0}, \mathbf{0}) \oplus F(\mathbf{0}, \mathbf{0}, \mathbf{1}) \oplus F(\mathbf{0}, \mathbf{1}, \mathbf{0}) \oplus F(\mathbf{0}, \mathbf{1}, \mathbf{1}) \\
\Phi_{x_1x_2x_3} &\Leftrightarrow F(\mathbf{0}, \mathbf{0}, \mathbf{0}) \oplus F(\mathbf{0}, \mathbf{0}, \mathbf{1}) \oplus F(\mathbf{0}, \mathbf{1}, \mathbf{0}) \oplus F(\mathbf{0}, \mathbf{1}, \mathbf{1}) \\
& \qquad \oplus F(\mathbf{1}, \mathbf{0}, \mathbf{0}) \oplus F(\mathbf{1}, \mathbf{0}, \mathbf{1}) \oplus F(\mathbf{1}, \mathbf{1}, \mathbf{0}) \oplus F(\mathbf{0}, \mathbf{1}, \mathbf{1}). \qquad (15.7)
\end{aligned}$$

It is not necessary to memorise any of the above formulas, nor to look them up, as they are easy to develop systematically. Let us use an example to demonstrate the method. We want to find the output values $F(X_1, X_2, X_3)$ which, when connected via XOR, comprise the coefficient $\Phi_{x_1 x_2 0}$. All the input events (X_1, X_2, X_3), with $X_i \in \{\mathbf{0}, \mathbf{1}\}$, which are the arguments of $F(X_1, X_2, X_3)$ are found by writing the K-set

$$\mathcal{K}_{x_1 x_2 0} = \{(x_1, x_2, x_3) | x_3 = 0\} = \{(0,0,0), (0,1,0), (1,0,0), (1,1,0)\}$$

associated with $\Phi_{x_1 x_2 0}$, i.e., defining the associated K-set by employing the constant(s) in the index of $\Phi_{x_1 x_2 0}$ as the specifying property of the set. Writing the input events of $\mathcal{K}_{x_1 x_2 0}$ in *logic* form provides us with the arguments of $F(X_1, X_2, X_3)$, allowing us to write

$$\Phi_{x_1 x_2 0} \Leftrightarrow F(\mathbf{0},\mathbf{0},\mathbf{0}) \oplus F(\mathbf{0},\mathbf{1},\mathbf{0}) \oplus F(\mathbf{1},\mathbf{0},\mathbf{0}) \oplus F(\mathbf{1},\mathbf{1},\mathbf{0}).$$

Of course there is no need to write $\mathcal{K}_{x_1 x_2 0}$ explicitly — the K-set is only a means of explanation. The above method is obviously valid for any number n of input variables.

15.4 On Combinations of Input Variables

Each row of (15.1) contains a *combination* of logic input variables X_1, X_2, ..., X_n. For $n = 3$ the rows would, from top to bottom, start with the following combinations of variables:

$$\underbrace{X_1,\ X_2,\ X_3}_{group\ 1},\ \underbrace{X_1X_2,\ X_1X_3,\ X_2X_3}_{group\ 2},\ \underbrace{X_1X_2X_3}_{group\ 3}.$$

The variables are arranged in three groups, as many as there are input variables. A convenient way to write all combinations is stated in the algorithm below in which, for simplicity, only the indices i of the variables X_i are referred to.

Algorithm for writing all combinations of the given set of indices $\{1, 2, \ldots, n\}$.
Step 1. Each element of $\{1, 2, \ldots, n\}$ — written as (1), (2), ...,(n) — is called a one-element combination, and is placed in a set we call group 1. This set is given a group number g that is 1.
Step 2. In the following step, set the group number g to 2.
Step 3. For each of the $(g-1)$-tuples $(\ldots, k)$ of group $g-1$ for which $k < n$ build the g-tuples $(\ldots, k, k+1)$, $(\ldots, k, k+2)$, ..., $(\ldots, k, n)$ assigning them to a new group g.
Step 4. If $g < n$, set g to $g+1$ and repeat step 3.
Step 5. If $g = n$, terminate the process.

The number of g-tuples or g-element combinations of the n-element set $\{1, 2, \ldots, n\}$ may be calculated as n over g

$$\binom{n}{g} = \frac{n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (n-g+1)}{1 \cdot 2 \cdot 3 \cdot \ldots \cdot g}.$$

Using the notation for factorial, $n! = 1 \cdot 2 \cdot \ldots \cdot n$, one easily finds

$$\binom{n}{g} = \frac{n!}{g!(n-g)!} = \binom{n}{n-g}.$$

As a matter of convention $\binom{n}{0} = 1$. The number of all 1-element, 2-element, to g-element combinations of n elements amounts to

$$\sum_{g=1}^{n} \binom{n}{g} = \binom{n}{1} + \binom{n}{2} + \ldots + \binom{n}{n} = 2^n - 1.$$

15.5 Dual Shegalkin Polynomial

The duality theorem, (5.18), allows us to write the Shegalkin polynomial (15.1) in its dual form knowing it to be valid:

$$\begin{aligned}
&F(X_1, X_2, \ldots, X_n) \Leftrightarrow \\
&\quad \underbrace{\Psi_{0\ldots0}}_{\binom{n}{0}} \leftrightarrow \\
&\quad \underbrace{X_1 \vee \Psi_{x_1 0\ldots0} \leftrightarrow X_2 \vee \Psi_{0 x_2 0\ldots0} \leftrightarrow \ldots \leftrightarrow X_n \vee \Psi_{0\ldots0 x_n}}_{\binom{n}{1}\ \text{single-variable expressions}} \leftrightarrow \\
&\quad \underbrace{X_1 \vee X_2 \vee \Psi_{x_1 x_2 0\ldots0} \leftrightarrow X_1 \vee X_3 \vee \Psi_{x_1 0 x_3 0\ldots0} \leftrightarrow \ldots}_{\binom{n}{2}\ \text{dual-variable expressions}} \leftrightarrow \\
&\quad \vdots \\
&\quad \underbrace{X_1 \vee X_2 \vee \ldots \vee X_n \vee \Psi_{x_1 x_2 \ldots x_n}}_{\binom{n}{n}}. \qquad \textbf{(15.8)}
\end{aligned}$$

The coefficients Ψ are also dual to Φ, and are thus called the *dual Shegalkin coefficients*. For three input variables you get the following coefficients

dual to (15.7)

$$
\begin{aligned}
\Psi_{000} &\Leftrightarrow F(1,1,1)\\
\Psi_{x_1 00} &\Leftrightarrow F(1,1,1) \leftrightarrow F(0,1,1)\\
\Psi_{0x_2 0} &\Leftrightarrow F(1,1,1) \leftrightarrow F(1,0,1)\\
\Psi_{00x_3} &\Leftrightarrow F(1,1,1) \leftrightarrow F(1,1,0)\\
\Psi_{x_1 x_2 0} &\Leftrightarrow F(1,1,1) \leftrightarrow F(1,0,1) \leftrightarrow F(0,1,1) \leftrightarrow F(0,0,1)\\
\Psi_{x_1 0 x_3} &\Leftrightarrow F(1,1,1) \leftrightarrow F(1,1,0) \leftrightarrow F(0,1,1) \leftrightarrow F(0,1,0)\\
\Psi_{0 x_2 x_3} &\Leftrightarrow F(1,1,1) \leftrightarrow F(1,1,0) \leftrightarrow F(1,0,1) \leftrightarrow F(1,0,0)\\
\Psi_{x_1 x_2 x_3} &\Leftrightarrow F(1,1,1) \leftrightarrow F(1,1,0) \leftrightarrow F(1,0,1) \leftrightarrow F(1,0,0)\\
&\quad \leftrightarrow F(0,1,1) \leftrightarrow F(0,1,0) \leftrightarrow F(0,0,1) \leftrightarrow F(1,0,0). \qquad (15.9)
\end{aligned}
$$

Although the result is fully obtained by duality, you may follow an analogue (meaning *dual*) derivation to that shown in Section 15.2. Interestingly, some have more faith in this step by step procedure than in the global approach of duality. You could first develop, or at least prove the **EQU expansion theorem**

$$
\begin{aligned}
&F(X_1, X_2, X_3, \ldots, X_n) \Leftrightarrow\\
&\quad F(X_1, \ldots, X_{i-1}, \mathbf{1}, X_{i+1}, \ldots, X_n)\\
&\quad \leftrightarrow X_i \vee \big(F(X_1, \ldots, X_{i-1}, \mathbf{1}, X_{i+1}, \ldots, X_n)\\
&\qquad \leftrightarrow F(X_1, \ldots, X_{i-1}, \mathbf{0}, X_{i+1}, \ldots, X_n)\big) \qquad \mathbf{(15.10)}
\end{aligned}
$$

Then, applying it as in Section 15.2, you would obtain

$$
\begin{aligned}
&F(X_1, X_2, X_3, \ldots, X_n) \Leftrightarrow\\
&\quad F(\mathbf{1}, X_2, X_3, \ldots, X_n)\\
&\quad \leftrightarrow X_1\big(F(\mathbf{1}, X_2, X_3, \ldots, X_n) \leftrightarrow F(\mathbf{0}, X_2, X_3, \ldots, X_n)\big). \qquad (15.11)
\end{aligned}
$$

and then

$$
\begin{aligned}
&F(X_1, X_2, X_3, \ldots, X_n) \Leftrightarrow\\
&\quad F(\mathbf{1}, \mathbf{1}, X_3, \ldots, X_n)\\
&\quad \leftrightarrow X_1\big(F(\mathbf{1}, \mathbf{1}, X_3, \ldots, X_n) \leftrightarrow F(\mathbf{0}, \mathbf{1}, X_3, \ldots, X_n)\big)\\
&\quad \leftrightarrow X_2\big(F(\mathbf{1}, \mathbf{1}, X_3, \ldots, X_n) \leftrightarrow F(\mathbf{1}, \mathbf{0}, X_3, \ldots, X_n)\big)\\
&\quad \leftrightarrow X_1 X_2\big(F(\mathbf{1}, \mathbf{1}, X_3, \ldots, X_n) \leftrightarrow F(\mathbf{1}, \mathbf{0}, X_3, \ldots, X_n)\\
&\quad \leftrightarrow F(\mathbf{0}, \mathbf{1}, X_3, \ldots, X_n) \leftrightarrow F(\mathbf{0}, \mathbf{0}, X_3, \ldots, X_n)\big). \qquad (15.12)
\end{aligned}
$$

To get a first impression of the results obtained with the dual Shegalkin polynomial we apply it to the TAE of Fig. 15–2a. The easiest way to determine the dual Shegalkin coefficients $\Psi_{\varphi_1 \varphi_2 \varphi_3}$ is to calculate them in the

$Y(0)$	$Y(1)$	$Y(2)$	$Y(3)$	$Y(4)$	$Y(5)$	$Y(6)$	$Y(7)$		$\Psi_{\varphi_1\varphi_2\varphi_3}$	$x_1\cdot$	$x_2\cdot$	$x_3\cdot$
0	**1**	**1**	**0**	**0**	**0**	**1**	**1**	$\Leftrightarrow$	Ψ_{000}	0	0	0
	0	**1**	**0**	**1**	**1**	**0**	**1**	$\Leftrightarrow$	Ψ_{00x_3}	0	0	1
		0	**0**	**0**	**1**	**0**	**0**	$\Leftrightarrow$	Ψ_{0x_20}	0	1	0
			1	**1**	**0**	**0**	**1**	$\Leftrightarrow$	$\Psi_{0x_2x_3}$	0	1	1
				1	**0**	**1**	**0**	$\Leftrightarrow$	Ψ_{x_100}	1	0	0
	EQU				**0**	**0**	**0**	$\Leftrightarrow$	$\Psi_{x_10x_3}$	1	0	1
						1	**1**	$\Leftrightarrow$	$\Psi_{x_1x_20}$	1	1	0
							1	$\Leftrightarrow$	$\Psi_{x_1x_2x_3}$	1	1	1

Fig. 15–3 The EQUIVALENCE logic triangle

logic triangle of Fig. 15–3 the elements of which are evaluated as shown in Fig. 15–4a.

The rightmost column of the logic triangle of Fig. 15–3 contains the coefficients Ψ. With these coefficients the Polynomial may be written as

$$Y \Leftrightarrow \underbrace{X_2}_{A} \leftrightarrow \underbrace{X_1}_{B} \leftrightarrow \underbrace{(X_1 \vee X_3)}_{C}, \tag{15.13}$$

the associated circuit being shown in Fig. 15–4c. But, remembering (4.12b) — which states that, for an odd number of inputs (e.g., A,B, and C), $Y_{XOR} \Leftrightarrow Y_{EQU}$ — the above formula (15.13) may be rewritten as

$$Y \Leftrightarrow \underbrace{X_2}_{A} \oplus \underbrace{X_1}_{B} \oplus \underbrace{(X_1 \vee X_3)}_{C}, \tag{15.14}$$

this leading to the circuit of Fig. 15–4d. As unaccustomed as this may seem when first encountered, the two circuits of Fig. 15–4c,d are equivalent.

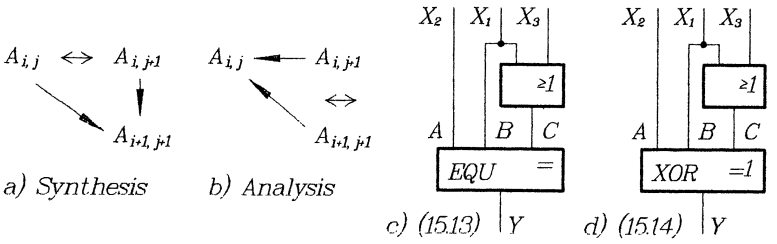


Fig. 15–4 Synthesis and analysis of the dual Shegalkin circuit

15.6 Necessary and Sufficient Connectives

Do we need all the connectives $\neg, \wedge, >, <, \oplus, \vee, \tilde{\vee}, \leftrightarrow, \leftarrow, \rightarrow, \tilde{\wedge}$ to be able to write the logic expression of some arbitrary logic function? Obviously not. For, as the canonical normal forms (14.1) and (14.2) demonstrate, it suffices to employ the connectives of $\{\wedge, \vee, \neg\}$, which is why they are said to comprise a **set of sufficient connectives**. But, it should be noted that not all the connectives of $\{\wedge, \vee, \neg\}$ are *necessary* as you can always express AND by OR and NOT — $A \wedge B \Leftrightarrow \neg(\neg A \vee \neg B)$ — as well as OR by AND and NOT. However, $\{\wedge, \neg\}$ and $\{\vee, \neg\}$ are **sets of necessary and sufficient connectives** meaning each of their connectives are necessary when writing a general logic expression, but no other connectives are needed.

set	$\neg A \Leftrightarrow$	$A \wedge B \Leftrightarrow$	$A \vee B \Leftrightarrow$	dual set
$\{\tilde{\wedge}\}$	$A \mathbin{\tilde{\wedge}} A$	$(A \mathbin{\tilde{\wedge}} B) \mathbin{\tilde{\wedge}} (A \mathbin{\tilde{\wedge}} B)$	$(A \mathbin{\tilde{\wedge}} A) \mathbin{\tilde{\wedge}} (B \mathbin{\tilde{\wedge}} B)$	$\{\tilde{\vee}\}$
$\{\wedge, \neg\}$	$\neg A$	$A \wedge B$	$\neg(\neg A \wedge \neg B)$	$\{\vee, \neg\}$
$\{\rightarrow, \mathbf{0}\}$	$A \rightarrow \mathbf{0}$	$\big(A \rightarrow (B \rightarrow \mathbf{0})\big) \rightarrow \mathbf{0}$	$(A \rightarrow B) \rightarrow B$	$\{>, \mathbf{1}\}$
$\{\rightarrow, \neg\}$	$\neg A$	$\neg(A \rightarrow \neg B)$	$\neg A \rightarrow B$	$\{>, \neg\}$
$\{\oplus, \wedge, \mathbf{1}\}$	$A \oplus \mathbf{1}$	$A \wedge B$	$A \oplus B \oplus (A \wedge B)$	$\{\leftrightarrow, \vee, \mathbf{0}\}$
$\{\oplus, \vee, \mathbf{1}\}$	$A \oplus \mathbf{1}$	$A \oplus B \oplus (A \vee B)$	$A \vee B$	$\{\leftrightarrow, \wedge, \mathbf{0}\}$

Fig. **15–5** Sets of necessary and sufficient functions

The two outer columns of Fig. 15–5 contain dual sets of necessary and sufficient connectives. How do we know that the connectives of such a set are sufficient to write a general logic expression? A straight forward way is to express the functions AND, OR, and NOT by those connectives thought to be sufficient for writing a general function. To claim necessity it suffices to express either NOT and AND, or NOT and OR. In Fig. 15–5 the connectives of the left column are used to express NOT, AND, and OR.

The NAND and NOR functions are especially interesting because they are the only ones that on their own, and without the constants **1** and **0**, are sufficient. Let us see how NAND — which is defined as $A \mathbin{\tilde{\wedge}} B \Leftrightarrow \neg(A \wedge B)$ — can be used to express NOT, AND, and OR.

$$\neg A \Leftrightarrow \neg(A \wedge A) \Leftrightarrow A \mathbin{\tilde{\wedge}} A, \tag{15.15}$$

$$A \wedge B \Leftrightarrow \neg\neg(A \wedge B) \Leftrightarrow \neg(A \mathbin{\tilde{\wedge}} B) \Leftrightarrow \neg\big((A \mathbin{\tilde{\wedge}} B) \wedge (A \mathbin{\tilde{\wedge}} B)\big) \Leftrightarrow (A \mathbin{\tilde{\wedge}} B) \mathbin{\tilde{\wedge}} (A \mathbin{\tilde{\wedge}} B), \tag{15.16}$$

$$A \vee B \Leftrightarrow \neg\neg(A \vee B) \Leftrightarrow \neg(\neg A \wedge \neg B) \Leftrightarrow \neg\big(\neg(A \wedge A) \wedge \neg(B \wedge B)\big) \Leftrightarrow \neg\big((A \mathbin{\tilde{\wedge}} A) \wedge (B \mathbin{\tilde{\wedge}} B)\big) \Leftrightarrow (A \mathbin{\tilde{\wedge}} A) \mathbin{\tilde{\wedge}} (B \mathbin{\tilde{\wedge}} B). \tag{15.17}$$

These formulas are shown in Fig. 15–6. Expressing NOT, AND, and OR as in the last two rows of Fig. 15–5 is achieved by applying the Shegalkin polynomial (15.1) and its dual form (15.8) — and not to forget (4.12).

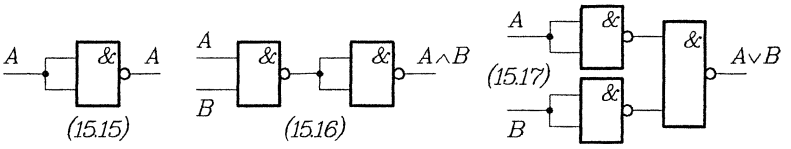


Fig. 15–6 Using NAND to express NOT, AND, OR

Chapter 16

Synthesis Examples

This chapter presents a few design examples of combinational circuits. They were chosen for their simplicity, their design procedures, and their results. You will note that you always need a full understanding of a problem to solve it, and that intuition and the desire not to be restrained by pre-set boundaries are a very helpful prerequisite.

16.1 Multiplexers and Demultiplexers

A common problem in digital information processing is to send information (data) from some **source** to a chosen **destination**. The most costly and inflexible way to do this is to connect each source with each destination. At the receiving end, the possible simultaneous influx of information from diverse sources would pose quite a problem.

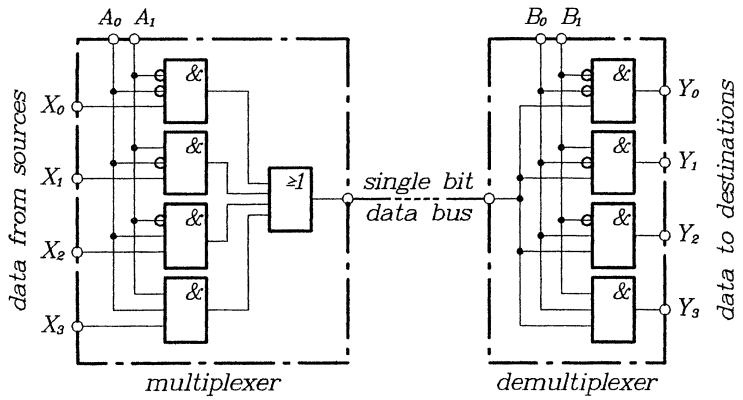


Fig. 16–1 Multiplexer and demultiplexer as end nodes of a bus

The alternative approach is *to connect all sources to a circuit called a* **multiplexer** (MUX) and *to connect all destinations to a circuit called a* **demultiplexer** (DMUX), as exemplified in Fig. 16–1. The lead connecting

the MUX with the DMUX is called a **bus**. An input code (A_0, A_1) to the MUX enables us to determine which source is connected to the bus. The destination to which the source sends its information is chosen via a code (B_0, B_1) to the DMUX. The circuit of the MUX is a direct application of the disjunctive normal form (14.1) as was already mentioned in connection with Fig. 14–2.

16.2 Binary Coded Decimal Digits

While it is most natural for a computer to work with a binary number system, we humans usually find it awkward, instead, preferring the decimal number system. For instance, few people immediately and instinctively associate $(100100111)_2$ with the decimal number 295. A common compromise between the binary needs of a computer and our decimal preferences is to encode *each* digit of a given decimal number as a four digit (four bit) binary number, calling this kind of encoding **binary coded decimal** or BCD. By this method, $(295)_{10}$ corresponds to $(0010\ 1001\ 0101)_{BCD}$ as detailed in Fig. 16–2.

dec weight	100	10	1
$(295)_{10}$	2	9	5
$(100100111)_2$	0 0 1 0	1 0 0 1	0 1 0 1
bin weight	8 4 2 1	8 4 2 1	8 4 2 1

Fig. 16–2 Decimal to BCDs

As our next synthesis example, let us specify and develop a circuit which tests whether a four bit binary number $(b_1 b_2 b_3 b_4)_2$ is a valid BCD code-number or not. If it is a valid code-number the circuit's output y should be 1, otherwise y should be 0. Quite clearly, the TAE of Fig. 16–3a specifies the circuit. Using (14.1), we obtain the disjunctive normal form as the disjunctions of the minterms of those columns for which the outputs are one (of course $B_i \Leftrightarrow b_i = 1$).

$$\begin{aligned} Y \Leftrightarrow & \overline{B}_1\overline{B}_2\overline{B}_3\overline{B}_4 \vee \overline{B}_1\overline{B}_2\overline{B}_3 B_4 \vee \overline{B}_1\overline{B}_2 B_3\overline{B}_4 \vee \overline{B}_1\overline{B}_2 B_3 B_4 \\ & \vee \overline{B}_1 B_2\overline{B}_3\overline{B}_4 \vee \overline{B}_1 B_2\overline{B}_3 B_4 \vee \overline{B}_1 B_2 B_3\overline{B}_4 \vee \overline{B}_1 B_2 B_3 B_4 \\ & \vee B_1\overline{B}_2\overline{B}_3\overline{B}_4 \vee B_1\overline{B}_2\overline{B}_3 B_4 \end{aligned}$$

The conjunctive normal form, on the other hand, is obtained by employing (14.2), i.e., by writing the conjunctions of the maxterms of those columns for which the outputs are zero.

$$\begin{aligned} Y \Leftrightarrow & (\overline{B}_1 \vee B_2 \vee \overline{B}_3 \vee B_4)(\overline{B}_1 \vee B_2 \vee \overline{B}_3 \vee \overline{B}_4)(\overline{B}_1 \vee \overline{B}_2 \vee B_3 \vee B_4) \\ & (\overline{B}_1 \vee \overline{B}_2 \vee B_3 \vee \overline{B}_4)(\overline{B}_1 \vee \overline{B}_2 \vee \overline{B}_3 \vee B_4)(\overline{B}_1 \vee \overline{B}_2 \vee \overline{B}_3 \vee \overline{B}_4) \end{aligned}$$

a)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$b_1=$	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
$b_2=$	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
$b_3=$	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
$b_4=$	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
$y=$	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0

b)

$b_3b_4=$ \ $b_1b_2=$	0,0	0,1	1,1	1,0
0,0	0 1	4 1	12 0	8 1
0,1	1 1	5 1	13 0	9 1
1,1	3 1	7 1	15 0	11 0
1,0	2 1	6 1	14 0	10 0

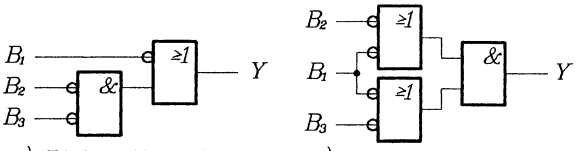
Fig. 16–3 Specifying a circuit that tests for a BCD code

Presumably no detailed explanation is needed to see that the Karnaugh map of Fig. 16–3b specifies the same circuit as the TAE of Fig. 16–3a. The advantage of the K-map is, of course, that we may read off far simpler disjunctive and conjunctive formulas than when employing canonical normal forms (14.1) or (14.2). Respectively, the disjunctive and conjunctive forms (see Sections 7.2 and 7.3) are:

$$DF: \qquad Y \Leftrightarrow \overline{B}_1 \vee \overline{B}_2\overline{B}_3$$

$$CF: \qquad Y \Leftrightarrow (\overline{B}_1 \vee \overline{B}_2)(\overline{B}_1 \vee \overline{B}_3)$$

As these formulas (the circuits of which are shown in Fig. 16–4) are far simpler than the canonical ones, the question of *minimising* canonical expressions is obviously of great importance. We shall start to look into this problem in the next chapter.



a) Disjunctive circuit b) Conjunctive circuit

Fig. 16–4 Circuits for testing the BCD code

Do note that these two formulas (DF and CF) are also equivalent due to distributivity: $\overline{B}_1 \vee \overline{B}_2\overline{B}_3 \Leftrightarrow (\overline{B}_1 \vee \overline{B}_2)(\overline{B}_1 \vee \overline{B}_3)$. In spite of the equivalence of these expressions, and of their circuits of Fig. 16–4, one has the feeling that they must behave differently in some way. And, in fact, there are situations in which they do behave distinctly differently, being prone to different kinds of errors. Studying such errors and how they occur is the subject of a later chapter on *hazards*.

16.3 Priority Decoder

A **priority decoder** may be specified as follows: In an ordered n-tuple $(x_1, x_2, \ldots, x_n)$ of input signals the (decimal) index i of the *first* input signal x_i to be 1 is transformed into a binary number the digits of which are the output signals $z_1, z_2, \ldots, z_k$. Let us use a small decoder of four inputs (x_1, x_2, x_3, x_4) and three outputs z_1, z_2, z_3 to see how to interpret the above specification. Such a decoder is defined in logic terms in Fig. 16–5a.

a) Specification

	X_1	X_2	X_3	X_4	Z_1	Z_2	Z_3
	0	0	0	0	0	0	0
Y_4	0	0	0	1	1	0	0
Y_3	0	0	1	-	0	1	1
Y_2	0	1	-	-	0	1	0
Y_1	1	-	-	-	0	0	1

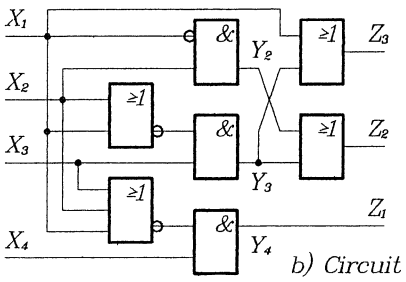


Fig. 16–5 A four input, three output priority decoder

The dashes ('-') stand for either 0 or 1. Thus, (0,0,1,-) is a shorthand for either of the two quadruples (0,0,1,0) or (0,0,1,1). Associating (x_1, x_2, x_3, x_4) with (0,0,1,-), the first variable to be 1 is x_3, its index 3 being expressed in binary form as (0,1,1), these three digits being the output values of (z_1, z_2, z_3). Hopefully, this suffices to illustrate the general specification.

Staying with the example (0,0,1,-) of an input quadruple, we now design a subcircuit the output, call it y_3, of which is 1 if x_3 is the *first* input variable to be 1. This implies that, for y_3 to be 1, x_1 and x_2 must both be 0, and that the value of x_4 is irrelevant. As x_4 has no influence on the output y_3, our subcircuit need only incorporate x_1, x_2, x_3 as input signals. The fact that y_3 is 1 if and only if x_1 is 0, x_2 is 0, and x_3 is 1 may be expressed as $Y_3 \Leftrightarrow X_1 < (X_2 < X_3)$ as you can show by writing a truth table for $X_1 < (X_2 < X_3)$. You might also want to look up what was said on the choice of Iverson's notation $X_1 > X_2$ for the **inhibition** in Section 4.1. In a like manner we can express all variables Y_1, Y_2, Y_3, Y_4 of Fig. 16–5a as:

$$Y_1 \Leftrightarrow X_1,$$

$$Y_2 \Leftrightarrow X_1 < X_2 \overset{(4.14)}{\Longleftrightarrow} \overline{X}_1 \wedge X_2,$$

$$Y_3 \Leftrightarrow X_1 < (X_2 < X_3) \overset{(4.14)}{\Longleftrightarrow} \overline{X}_1 \wedge (\overline{X}_2 \wedge X_3) \overset{10-2}{\Longleftrightarrow} (\overline{X}_1 \wedge \overline{X}_2) \wedge X_3$$
$$\overset{(9.22)}{\Longleftrightarrow} \overline{(\overline{X}_1 \wedge \overline{X}_2)} \wedge X_3 \overset{(9.26)}{\Longleftrightarrow} \overline{(X_1 \vee X_2)} X_3,$$

$$Y_4 \Leftrightarrow X_1 < (X_2 < (X_3 < X_4)) \Leftrightarrow \ldots \Leftrightarrow \overline{(X_1 \vee X_2 \vee X_3)} X_4.$$

Finally, from Fig. 16–5a, the outputs of the priority decoder can be read off as:

$$Z_1 \Leftrightarrow Y_4,$$
$$Z_2 \Leftrightarrow Y_2 \vee Y_3$$
$$Z_3 \Leftrightarrow Y_1 \vee Y_3$$

Combining the formulas for Y_1, Y_2, Y_3, Y_4 and Z_1, Z_2, Z_3 leads to the final circuit of Fig. 16–5b.

16.4 Comparators

This is a rather extensive example, the intention being to stress that real world solutions are seldomly obtained in a straight forward manner. A **comparator** is a circuit that compares two numbers,

$$x = \sum_{i=1}^{n} x_i.2^{n-i} \qquad \text{and} \qquad y = \sum_{i=1}^{n} y_i.2^{n-i},$$

i.e., checks whether $x > y$, $x = y$, or $x < y$. From our study of ordered pairs we already know from (3.4) and (3.5) that two numbers are equal when equally positioned (weighted) digits are equal:

$$x = y \Leftrightarrow (x_1 = y_1) \wedge (x_2 = y_2) \wedge \ldots \wedge (x_n = y_n).$$

This formula can be represented by the circuit of Fig. 16–6a the output of which consists of a n-input AND. A more flexible solution would be to design (identical) modules that could be put together (say, cascaded) to obtain a $(x = y)$-comparator of a desired number of inputs. This can in fact easily be achieved by expanding the above formula with the help of $A \Leftrightarrow A \wedge A$ (Fig. 10–4):

$$\begin{aligned} x = y \Leftrightarrow\ & (x_1 = y_1) \wedge (x_2 = y_2) \wedge \ldots \wedge (x_n = y_n) \Leftrightarrow \\ & (x_1 = y_1) && \ldots R_1 \\ & \wedge (x_1 = y_1) \wedge (x_2 = y_2) && \ldots R_2 \\ & \wedge (x_1 = y_1) \wedge (x_2 = y_2) \wedge (x_3 = y_3) && \ldots R_3 \\ & \vdots \\ & \wedge (x_1 = y_1) \wedge (x_2 = y_2) \wedge \ldots \wedge (x_n = y_n) && \ldots R_n \end{aligned}$$

Referring to the conjunction of any row i, above, as R_i, you can see that, for $i \geq 2$, $R_i \Leftrightarrow R_{i-1} \wedge (x_i = y_i)$, allowing us to design the cascaded circuit shown in Fig. 16–6b.

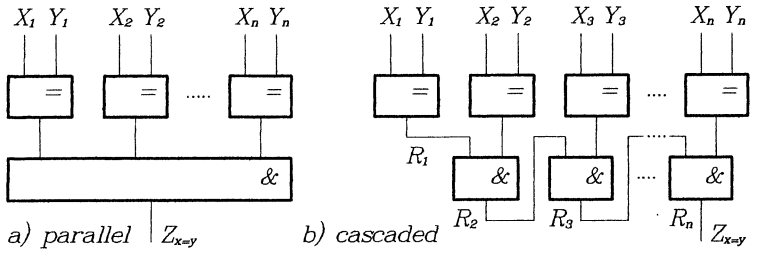


Fig. 16–6 $(x = y)$-comparators

Let us now turn our attention to the question of when two numbers, x and y, are *distinct*. This is decided in the highest position in which two digits are distinct. You thus start by comparing the highest weighted bits, in our notation x_1 and y_1. If they are *equal*, you compare the next lower bits, x_2 and y_2. You carry on in this manner towards lower weighted bits until you come across unequal bits, these deciding which of the numbers is the larger one. This procedure, or algorithm, is expressed in the following formula for the *larger than* relationship $x > y$. Note that, if the truth value of the conjunction of some row i is **1**, the truth value of all foregoing rows are **0**. This mirrors the basic idea of the above algorithm.

$$
\begin{aligned}
x > y \Leftrightarrow\ & (x_1 > y_1) \\
& \vee \big((x_1 = y_1) \wedge (x_2 > y_2)\big) \\
& \vee \big((x_1 = y_1) \wedge (x_2 = y_2) \wedge (x_3 > y_3)\big) \\
& \vee \big((x_1 = y_1) \wedge (x_2 = y_2) \wedge (x_3 = y_3) \wedge (x_4 > y_4)\big) \\
& \vdots \\
& \vee \big((x_1 = y_1) \wedge (x_2 = y_2) \wedge \ldots \wedge (x_{n-1} = y_{n-1}) \wedge (x_n > y_n)\big).
\end{aligned}
$$

An economic way of designing the logic circuit of the above formula is to fall back on the cascaded solution for equals, $x = y$, shown in Fig. 16–6b. Doing so enables us to draw the circuit of Fig. 16–7a. As you can see, the five gates drawn one below the other, and their interconnections, are repeated for the comparison of each pair of input digits. Such repetitive units are usually called **modules**. Modules are often redesigned (see Fig. 16–7b) in such a way as to imply their use — in our case, by cascading them.

Modular circuits have advantages: You need only test a single module, not the whole, often complex circuit. But, if modules are cascaded, the compound circuits tend to be relatively slow as the resulting signals of one module are propagated to the next module. A worthwhile approach to designing potentially fast combinational circuits is to specify the problem directly in a TAE or K-map. Allow me to discuss this approach for our running example

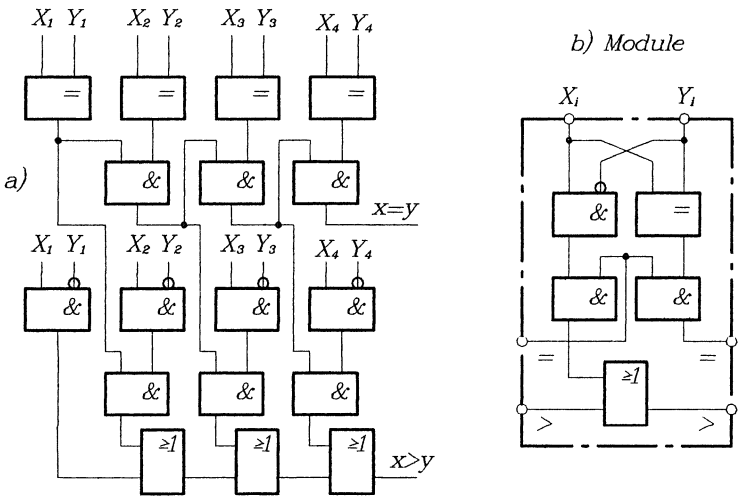


Fig. 16–7 Modular design of a circuit for $x = y$ and $x > y$

$x > y$ using a K-map. Given the 2-digit numbers

$$x = x_1 \cdot 2^1 + x_2 \cdot 2^0, \qquad \text{and} \qquad y = y_1 \cdot 2^1 + y_2 \cdot 2^0,$$

and the logic variable $Z^{(2)}_{x>y} \overset{df}{\Leftrightarrow} x > y$, we may specify its value as shown in Fig. 16–8a. The entry, **0** or **1**, to each cell must be decided individually. This is done according to the aforementioned algorithm by which the distinctness of two numbers is decided in the highest position in which two bits are distinct.

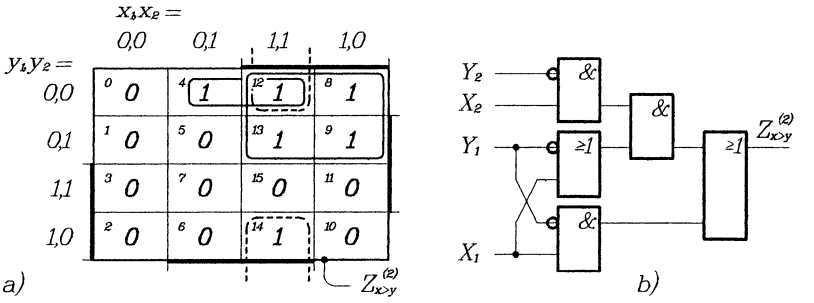


Fig. 16–8 Specification and circuit of $x > y$ for 2-digit numbers

Let us take as examples the entries into the cells 0, 1, and 4. The value **0** in cell 0 expresses the fact that the binary number (0,0), standing for (x_1, x_2), is *not* larger than (0,0), standing for (y_1, y_2). In cell 1 we enter **0** to express

that (0,0) is *not* larger than (0,1) as a comparison of the lowest weighted bits shows. The complementary case is shown in cell 4 containing the value **1** as the comparison of the lowest weighted bits lets us decide that (0,1) *is* larger than (0,0). The loops enable the output $Z^{(2)}_{x>y}$ to be calculated as

$$Z^{(2)}_{x>y} \Leftrightarrow X_1\overline{Y}_1 \vee X_1X_2\overline{Y}_2 \vee X_2\overline{Y}_1\overline{Y}_2 \Leftrightarrow X_1\overline{Y}_1 \vee (X_1 \vee \overline{Y}_1)X_2\overline{Y}_2,$$

this formula leading to the circuit of Fig. 16–8b. This preliminary result is interesting when compared with the first stages (those for (x_1, x_2) and (y_1, y_2)) of the circuit of Fig. 16–7a: No part of the circuit of Fig. 16–8b is related to the question of when (x_1, x_2) and (y_1, y_2) are equal.

To continue this promising approach let us consider a K-map to express when (x_1, x_2, x_3) is larger than (y_1, y_2, y_3), see Fig. 16–9. Its algebraic evaluation leads to

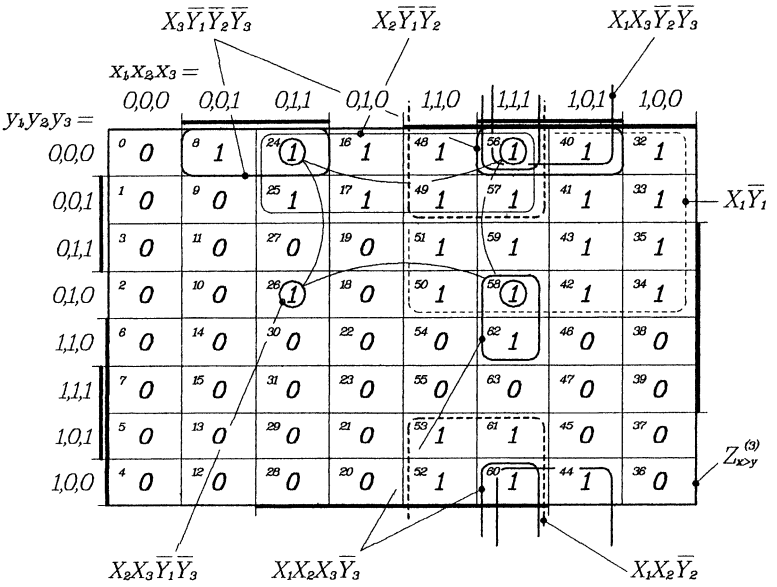


Fig. 16–9 Specification of $x > y$ for 3-digit numbers

$$\begin{aligned} Z^{(3)}_{x>y} \Leftrightarrow\ & X_1\overline{Y}_1 \vee X_1X_2\overline{Y}_2 \vee X_1X_2X_3\overline{Y}_3 \\ & \vee \overline{Y}_1X_2\overline{Y}_2 \vee \overline{Y}_1X_2X_3\overline{Y}_3 \\ & \vee X_1\overline{Y}_2X_3\overline{Y}_3 \\ & \vee \overline{Y}_1\overline{Y}_2X_3\overline{Y}_3 \end{aligned}$$

The conjunctions of this formula have been arranged in such a way that you can easily recognise the possibility of 'extracting' the subexpressions

$X_2\overline{Y}_2$ and $X_3\overline{Y}_3$, thereby obtaining

$$\begin{aligned} Z^{(3)}_{x>y} \Leftrightarrow\ & X_1\overline{Y}_1 \\ & \vee\ (X_1 \vee \overline{Y}_1)X_2\overline{Y}_2 \\ & \vee\ (X_1X_2 \vee \overline{Y}_1X_2 \vee X_1\overline{Y}_2 \vee \overline{Y}_1\overline{Y}_2)X_3\overline{Y}_3. \end{aligned}$$

$$\begin{aligned} Z^{(3)}_{x>y} \Leftrightarrow\ & X_1\overline{Y}_1 \\ & \vee\ (X_1 \vee \overline{Y}_1)X_2\overline{Y}_2 \\ & \vee\ \big((X_1 \vee \overline{Y}_1)X_2 \vee (X_1 \vee \overline{Y}_1)\overline{Y}_2\big)X_3\overline{Y}_3 \\ \Leftrightarrow\ & X_1\overline{Y}_1 \\ & \vee\ (X_1 \vee \overline{Y}_1)X_2\overline{Y}_2 \\ & \vee\ (X_1 \vee \overline{Y}_1)(X_2 \vee \overline{Y}_2)X_3\overline{Y}_3. \end{aligned}$$

Looking at the results for $Z^{(1)}_{x>y} \Leftrightarrow X_1\overline{Y}_1$, $Z^{(2)}_{x>y}$ and $Z^{(3)}_{x>y}$ it does not seem farfetched to conjecture

$$\begin{aligned} Z^{(4)}_{x>y} \Leftrightarrow\ & X_1\overline{Y}_1 \\ & \vee\ (X_1 \vee \overline{Y}_1)X_2\overline{Y}_2 \\ & \vee\ (X_1 \vee \overline{Y}_1)(X_2 \vee \overline{Y}_2)X_3\overline{Y}_3 \\ & \vee\ (X_1 \vee \overline{Y}_1)(X_2 \vee \overline{Y}_2)(X_3 \vee \overline{Y}_3)X_4\overline{Y}_4. \end{aligned}$$

This formula, a proof of which is given in Section 20.4, can be drawn as the circuit of Fig. 16–10. It was designed in a straight forward manner avoiding internal cascading, thus being a potentially fast circuit.

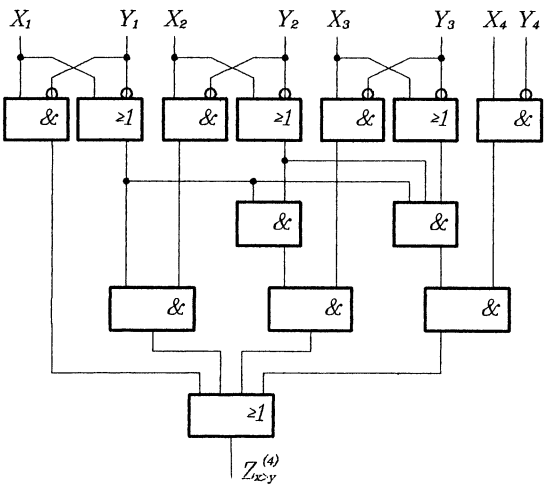


Fig. 16–10 Circuit for $Z^{(4)}_{x>y} \overset{df}{\Leftrightarrow} x > y$ where x and y are 4-bit integers

Chapter 17

Concepts Old and New

Looking back at the canonical normal forms of Chapter 14, the backbone of most algebraic methods for designing switching circuits, we have to admit, the results achieved with the normal forms do not compare well with the simpler ones obtained by the graphic approach of the K-map. The advantage of the K-map lies in the use of the highly adaptive and versatile K-sets. The disadvantage of the canonical normal forms is that they express their results in inflexible and cumbersome min- or maxterms. Clearly, it is desirable to reformulate the canonical normal forms in such a way that they reflect the flexibility of the K-map. To reach this goal, however vague it has been expressed, there are a number of new concepts we need to develop. Each is covered in its own section.

17.1 Multiple Events *

An individual event (or an asserted input-event, as it was called in Section 1.3)

$$(e_1, \ldots, e_n) \qquad \text{with} \qquad e_i \in \{0, 1\} \tag{17.1}$$

is an n-tuple of 0s and 1s, i.e., each e_i, once chosen, remains invariable. For $n \hookleftarrow 4$, each of the quadruples $(0,1,1,1)$, $(1,0,0,1)$ is an example of an individual event.

When trying to find all the individual events $(e_1, \ldots, e_n)$ which comprise a given K-set, we come across a generalisation representing a multitude of individual events. For example, consider the K-set

$$(x_1, x_2, x_3, x_4) \in \mathcal{K}_A \overset{df}{\Leftrightarrow} x_1 = 1 \wedge x_3 = 0.$$

Every input event that is an element of $\mathcal{K}_A$ must make the defining expression $x_1 = 1 \wedge x_3 = 0$ of $\mathcal{K}_A$ a true proposition. As $x_1 = 1 \wedge x_3 = 0$ is true when substituting x_1 by 1 ($x_1 \hookleftarrow 1$) and x_3 by 0 ($x_3 \hookleftarrow 0$), we can write

$$(1, x_2, 0, x_4) \in \mathcal{K}_A \Leftrightarrow 1 = 1 \wedge 0 = 0 \Leftrightarrow \mathbf{1}.$$

Taking the expression $(1, x_2, 0, x_4) \in \mathcal{K}_A$ to be true means it must be true for all combinations $(0,0)$, $(0,1)$, $(1,0)$, $(1,1)$ of (x_2, x_4) so that

$$\mathcal{K}_A = \{(1,0,0,0),(1,0,0,1),(1,1,0,0),(1,1,0,1)\}.$$

In this section, we want to concentrate on n-tuples such as $(1, x_2, 0, x_4)$, noting that they stand for a multitude of input events. Let me refer to the general form of such an n-tuple as a **multiple event** denoting it as

$$(\varphi_1, \ldots, \varphi_n) \qquad \text{with} \qquad \varphi_i \in \{0, 1, x_i\}. \tag{17.2}$$

Characteristic for the multiple event is that any position, say the i-th, stands for one of the three symbols 0, 1, or x_i. We have already made use of these multiple events, albeit tacitly, in classifying Shegalkin coefficients $\Phi_{\varphi_1 \ldots \varphi_n}$ in Sections 15.1 and 15.3.

To get a feeling for the concept of *multiple event* and its specification $(\varphi_1, \ldots, \varphi_n)$, with $\varphi_i \in \{0, 1, x_i\}$, let us look at a few examples taking n to be 4. If each φ_i is a given constant, either 0 or 1, then $(\varphi_1, \varphi_2, \varphi_3, \varphi_4)$ is some individual event, say, $(0,1,1,1)$. Now assume that only 3 variables φ_i are assigned specific values, e.g., $\varphi_1 \hookleftarrow 0$, $\varphi_2 \hookleftarrow 1$, $\varphi_3 \hookleftarrow 1$, while φ_4 remains unspecified ($\varphi_2 \hookleftarrow x_4$), then $(\varphi_1, \varphi_2, \varphi_3, \varphi_4)$ becomes $(0,1,1,x_4)$. As x_4 can be either 0 or 1 ($x_4 \in \{0,1\}$), the multiple event $(0,1,1,x_4)$ represents any of the individual events $(0,1,1,0)$, $(0,1,1,1)$. Similarly, the multiple event $(1, x_2, 0, x_4)$, containing two variable symbols, stands for any of the four individual events $(1,0,0,0)$, $(1,0,0,1)$, $(1,1,0,0)$, $(1,1,0,1)$. Generalising, if the multiple event $(\varphi_1, \ldots, \varphi_n)$ contains k variable symbols $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$, then $(\varphi_1, \ldots, \varphi_n)$ stands for 2^k input events $(e_1, \ldots, e_n)$.

To obtain a **simpler notation**, the indices i of the variables x_i are often omitted, letting the *position* in which the variable is written represent the value of the omitted index. By this convention we may write $(0,1,1,x)$ instead of $(0,1,1,x_4)$, and $(1,x,0,x)$ instead of $(1,x_2,0,x_4)$.

To simplify the notation for multiple events, when desired, let us agree to use φ (without a subscript) as an abbreviation for $(\varphi_1, \ldots, \varphi_n)$:

$$\varphi \overset{df}{\Leftrightarrow} (\varphi_1, \ldots, \varphi_n). \tag{17.3}$$

To distinguish a multitude of, say L, multiple events we employ *superscripts*, writing them in parentheses to avoid interpreting them as realising the power function:

$$\varphi^{(\ell)} \overset{df}{\Leftrightarrow} (\varphi_1^{(\ell)}, \ldots, \varphi_n^{(\ell)}) \quad \text{with} \quad \ell \in \{1, 2, \ldots, L\}. \tag{17.4}$$

17.2 Karnaugh Sets Revisited *

Reconsider the original definition, (6.12), of a Karnaugh set as a set $\mathcal{K}$ of input events $(x_1, x_2, \ldots, x_n)$ whose common and defining property is that

each variable of a given subset $\{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\}$ of the set of input variables $\{x_1, x_2, \ldots, x_n\}$ is assigned a certain value, either 0 or 1. This was stated more formally as

$$(6.12) \qquad (x_1, \ldots, x_n) \in \mathcal{K} \overset{df}{\Leftrightarrow} (x_{i_1} = e_{i_1}) \wedge (x_{i_2} = e_{i_2}) \wedge \ldots \wedge (x_{i_k} = e_{i_k}),$$

with $\{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\} \subseteq \{x_1, x_2, \ldots, x_n\}$ and $e_{i_1}, e_{i_2}, \ldots, e_{i_k} \in \{0, 1\}$.

The above definition gives a good description of what a K-set is, allowing us to use it as in Chapter 7. But to work set-algebraically with K-sets we need to define them more precisely. Here, the notation used for a multiple event proves its value.

(17.5) A Karnaugh set $\mathcal{K}_{\varphi_1 \ldots \varphi_n}$ is a set of the individual events $(x_1, \ldots, x_n)$ that are represented by a given multiple event which, according to (17.2), we denote as $(\varphi_1, \ldots, \varphi_n)$,

$$(x_1, \ldots, x_n) \in \mathcal{K}_{\varphi_1 \ldots \varphi_n} \overset{df}{\Leftrightarrow} \bigwedge_{i=1}^{n} (x_i = \varphi_i), \quad \varphi_i \in \{0, 1, x_i\}.$$

A set $\mathcal{K}_{\varphi_1 \ldots \varphi_n}$, defined by (17.5), is always a Karnaugh set by the weaker definition (6.12) repeated above. Let me make this plausible by considering the examples $(0, 1, 1, x_4)$ and $(1, x_2, 0, x_4)$ of a multiple event $(\varphi_1, \varphi_2, \varphi_3, \varphi_4)$. First, write (17.5) for $(0, 1, 1, x_4)$:

$$\begin{aligned} &(x_1, x_2, x_3, x_4) \in \mathcal{K}_{011x_4} \Leftrightarrow \\ &\qquad \underbrace{(x_1 = 0)}_{\overline{X}_1} \wedge \underbrace{(x_2 = 1)}_{X_2} \wedge \underbrace{(x_3 = 1)}_{X_3} \wedge \underbrace{(x_4 = x_4)}_{1}. \\ &(x_1, x_2, x_3, x_4) \in \mathcal{K}_{011x_4} \Leftrightarrow \overline{X}_1 \wedge X_2 \wedge X_3 \end{aligned}$$

Similarly, we obtain $(x_1, x_2, x_3, x_4) \in \mathcal{K}_{1x_20x_4} \Leftrightarrow X_1 \wedge \overline{X}_3$.

More generally, if φ_i stands for 1 then $x_i = \varphi_i$ becomes $x_i = 1 \Leftrightarrow X_i$; if $\varphi_i \leftarrow\!\!\shortmid\; 0$ then $x_i = \varphi_i \Leftrightarrow x_i = 0 \Leftrightarrow \overline{X}_i$; and if $\varphi_i \leftarrow\!\!\shortmid\; x_i$ we get $x_i = \varphi_i \Leftrightarrow x_i = x_i \Leftrightarrow \mathbf{1}$. Thus the right side of (17.5) reduces to the right side of

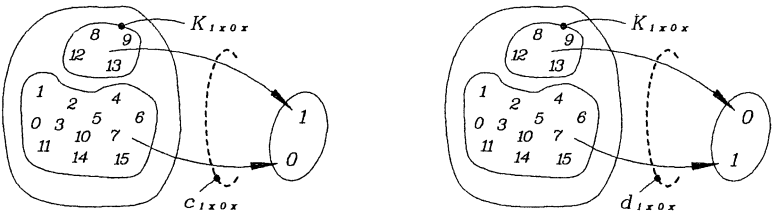
$$(6.12) \qquad (x_1, \ldots, x_n) \in \mathcal{K} \Leftrightarrow (x_{i_1} = e_{i_1}) \wedge (x_{i_2} = e_{i_2}) \wedge \ldots \wedge (x_{i_k} = e_{i_k})$$

in which $e_{i_1}, e_{i_2}, \ldots, e_{i_k} \in \{0, 1\}$, verifying that $\mathcal{K}_{\varphi_1 \ldots \varphi_n}$, as defined in (17.5), is a Karnaugh set in the sense of (6.12). This indexing convention for K-sets was already used, albeit reticently, in Section 15.3 when discussing the connection between a Shegalkin coefficient $\Phi_{\varphi_1 \ldots \varphi_n}$ and the K-set $\mathcal{K}_{\varphi_1 \ldots \varphi_n}$.

17.3 Generalised Minterms *

Minterms were first introduced in Section 5.1 using minterm functions c_e. They were characterised as mapping one and only one event e to 1. The

generalisation to be effected here is to map each event e of a given K-set $\mathcal{K}_{\varphi_1 \ldots \varphi_n}$ to 1, calling the function that does this a generalised minterm function $c_{\varphi_1 \ldots \varphi_n}$. An example of such a function is pictured in Fig. 17–1a.



a) A generalised minterm function *b) A generalised maxterm function*

Fig. 17–1 Examples of generalised min- and maxterm functions

(17.6) The **generalised minterm function** $c_{\varphi_1 \ldots \varphi_n}$ is a switching function $\{0,1\}^n \mapsto \{0,1\}$ that maps the input events $(x_1, \ldots, x_n)$ of the K-set $\mathcal{K}_{\varphi_1 \ldots \varphi_n}$, and only these, to 1:

$$c_{\varphi_1 \ldots \varphi_n}(x_1, \ldots, x_n) = 1 \overset{df}{\Leftrightarrow} \bigwedge_{i=1}^{n} (x_i = \varphi_i), \quad \text{with} \quad \varphi_i \in \{0, 1, x_i\}.$$

The **generalised minterm** $C_{\varphi_1 \ldots \varphi_n}(x_1, \ldots, x_n)$ is simply the abbreviation for the assertion that the output *variable* $c_{\varphi_1 \ldots \varphi_n}(x_1, \ldots, x_n)$ of the generalised minterm *function* $c_{\varphi_1 \ldots \varphi_n} : \{0,1\}^n \mapsto \{0,1\}$ is 1.

$$C_{\varphi_1 \ldots \varphi_n}(x_1, \ldots, x_n) \overset{df}{\Leftrightarrow} c_{\varphi_1 \ldots \varphi_n}(x_1, \ldots, x_n) = 1. \tag{17.7}$$

Using this in (17.6), we get:

$$C_{\varphi_1 \ldots \varphi_n}(x_1, \ldots, x_n) \Leftrightarrow \bigwedge_{i=1}^{n} (x_i = \varphi_i). \tag{**17.8**}$$

As the right sides of this formula and of (17.5) are identical, we may rewrite (17.5):

$$(x_1, \ldots, x_n) \in \mathcal{K}_{\varphi_1 \ldots \varphi_n} \Leftrightarrow C_{\varphi_1 \ldots \varphi_n}(x_1, \ldots, x_n), \text{ each } \varphi_i \in \{0, 1, x_i\} \tag{**17.9**}$$

showing that *the generalised minterm* $C_{\varphi_1 \ldots \varphi_n}$ *is the defining expression of the* K-set $\mathcal{K}_{\varphi_1 \ldots \varphi_n}$. This algebraic result parallels the use of K-sets in K-maps and is, as such, the first step towards algebraic flexibility.

The multiple event $(\varphi_1, \ldots, \varphi_n)$ used to index the generalised minterm $C_{\varphi_1 \ldots \varphi_n}$ classifies it uniquely. The position i of a *constant* value of φ_i is used

as the index i of a logic variable that is inverted if the constant is 0, and not inverted if the constant is 1. Check this for the following examples:

$$C_{001} \Leftrightarrow (x_1 = 0) \wedge (x_2 = 0) \wedge (x_3 = 1) \Leftrightarrow \overline{X}_1\overline{X}_2X_3,$$
$$C_{x_1 10} \Leftrightarrow (x_1 = x_1) \wedge (x_2 = 1) \wedge (x_3 = 0) \Leftrightarrow \mathbf{1} \wedge X_2 \wedge \overline{X}_3 \Leftrightarrow X_2\overline{X}_3,$$
$$C_{11x_3} \Leftrightarrow (x_1 = 1) \wedge (x_2 = 1) \wedge (x_3 = x_3) \Leftrightarrow X_1 \wedge X_2 \wedge \mathbf{1} \Leftrightarrow X_1X_2.$$

17.4 Generalised Maxterms *

As you may remember from Section 5.2, the maxterm function d_e was defined to be the negation of the minterm function c_e, i.e., the maxterm function maps the event e, and only this event, to 0 (not to 1, as c_e does). The generalisation undertaken in this section is to map each event e of a given K-set $\mathcal{K}_{\varphi_1 \ldots \varphi_n}$ to 0 calling the function that does this a generalised maxterm function $d_{\varphi_1 \ldots \varphi_n}$, and making it the negation of the generalised minterm function $c_{\varphi_1 \ldots \varphi_n}$. An example is shown in Fig. 17–1b. It differs from the generalised minterm function $c_{\varphi_1 \ldots \varphi_n}$ of Fig. 17–1a only in that the values of the co-domain have been interchanged.

(17.10) The **generalised maxterm function** $d_{\varphi_1 \ldots \varphi_n}$ is a switching function $\{0,1\}^n \mapsto \{0,1\}$ that maps each of the input events $(x_1, \ldots, x_n)$ of the K-set $\mathcal{K}_{\varphi_1 \ldots \varphi_n}$, but only these, to 0:

$$d_{\varphi_1 \ldots \varphi_n}(x_1, \ldots, x_n) = 0 \overset{df}{\Leftrightarrow} \bigwedge_{i=1}^{n} (x_i = \varphi_i), \quad \text{with} \quad \varphi_i \in \{0, 1, x_i\}.$$

The **generalised maxterm** $D_{\varphi_1 \ldots \varphi_n}(x_1, \ldots, x_n)$ is defined as the assertion that the output variable $d_{\varphi_1 \ldots \varphi_n}(x_1, \ldots, x_n)$ is 1:

$$D_{\varphi_1 \ldots \varphi_n}(x_1, \ldots, x_n) \overset{df}{\Leftrightarrow} d_{\varphi_1 \ldots \varphi_n}(x_1, \ldots, x_n) = 1, \tag{17.11}$$

and is sometimes simply written as $D_{\varphi_1 \ldots \varphi_n}$. In developing a formula by which to evaluate the generalised maxterm $D_{\varphi_1 \ldots \varphi_n}$, we note that it is its inverted form, $\overline{D}_{\varphi_1 \ldots \varphi_n}$, that is stated in (17.10) which may therefore be written as:

$$\overline{D}_{\varphi_1 \ldots \varphi_n}(x_1, \ldots, x_n) \Leftrightarrow \bigwedge_{i=1}^{n} (x_i = \varphi_i), \quad \varphi_i \in \{0, 1, x_i\}. \tag{17.12}$$

Inverting this formula,

$$\overline{\overline{D}}_{\varphi_1 \ldots \varphi_n}(x_1, \ldots, x_n) \Leftrightarrow \neg \bigwedge_{i=1}^{n} \underbrace{(x_i = \varphi_i)}_{P(x)}, \quad \varphi_i \in \{0, 1, x_i\},$$

and employing (9.22) $D_{\varphi_1 \ldots \varphi_n} \Leftrightarrow \overline{\overline{D}}_{\varphi_1 \ldots \varphi_n}$ and (13.67) $\neg \bigwedge P(x) \Leftrightarrow \bigvee \neg P(x)$, leads to the formula by which to evaluate $D_{\varphi_1 \ldots \varphi_n}$:

$$D_{\varphi_1 \ldots \varphi_n}(x_1, \ldots, x_n) \Leftrightarrow \bigvee_{i=1}^{n} (x_i \neq \varphi_i), \quad \varphi_i \in \{0, 1, x_i\}. \tag{17.13}$$

The comparison between (17.12) and (17.8) allows us to infer that $D_{\varphi_1 \ldots \varphi_n}$ is the negation of $C_{\varphi_1 \ldots \varphi_n}$:

$$C_{\varphi_1 \ldots \varphi_n} \Leftrightarrow \overline{D}_{\varphi_1 \ldots \varphi_n}. \tag{17.14}$$

The generalised maxterm $D_{\varphi_1 \ldots \varphi_n}$, on its part, defines a set that is the complement of the K-set $\mathcal{K}_{\varphi_1 \ldots \varphi_n}$

$$\overline{\mathcal{K}}_{\varphi_1 \ldots \varphi_n} = \{(x_1, \ldots, x_n) | D_{\varphi_1 \ldots \varphi_n}\}, \tag{17.15}$$

17.5 Partitions and Equivalents

Partitioning a set refers to splitting it into mutually disjoint subsets. Every function f splits or partitions its domain into mutually disjoint subsets — all the elements of a certain subset being *equivalent* in that they (and only they) map to a specified element of the co-domain. Such a subset is therefore called an **equivalence set** or an **equivalent** for short. The set of all these equivalents is called *the* **partition of the domain** *induced by the function* f.

A switching function $f : \{0,1\}^n \mapsto \{0,1\}$ (whose co-domain consists of 0 and 1) thus partitions its domain (the set $\mathcal{E} = \{0, 1, \ldots, 2^{n-1}\}$ of input events) into a 0-**equivalent** $\mathcal{E}_0$ and a 1-**equivalent** $\mathcal{E}_1$ (the prefix, 0 or 1, being the *output value* to which the elements of the equivalent are mapped):

$$\mathcal{E}_0 \stackrel{df}{=} \{\mathbf{x} | f(\mathbf{x}) = 0\}, \qquad \mathcal{E}_1 \stackrel{df}{=} \{\mathbf{x} | f(\mathbf{x}) = 1\}. \tag{17.16}$$

Remembering that $f(\mathbf{x})$ is a brief form for $f(x_1, \ldots, x_n)$ — see Section 14.2 — and that $y(x) \stackrel{df}{=} f(x_1, \ldots, x_n)$, we use the definitions $Y(x) \Leftrightarrow y(x) = 1$ and $\overline{Y}(x) \Leftrightarrow y(x) = 0$ to rewrite (17.16) as

$$\mathbf{x} \in \mathcal{E}_0 \Leftrightarrow \overline{Y}(x), \qquad \mathbf{x} \in \mathcal{E}_1 \Leftrightarrow Y(x). \tag{17.17}$$

Clearly, $\mathcal{E}_0$ and $\mathcal{E}_1$ are disjoint, meaning their intersection $\mathcal{E}_0 \cap \mathcal{E}_1$ is empty, while on the other hand, together they comprise all input events of the universe $\mathcal{E}$

$$\mathcal{E}_0 \cap \mathcal{E}_1 = \emptyset \qquad \text{and} \qquad \mathcal{E}_0 \cup \mathcal{E}_1 = \mathcal{E}. \tag{17.18}$$

Either $\mathcal{E}_0$ or $\mathcal{E}_1$ can be empty, this being the case when f is either the **one**- or the **zero**-function (see Fig. 4–1), respectively. Now, let the symbol a be a *constant* standing for a specified value — either 0 or 1 — of the output variable y. Then we may write $\mathcal{E}_a$ for either $\mathcal{E}_0$ or $\mathcal{E}_1$, as the case may be:

$$\mathcal{E}_a \stackrel{df}{=} \{\mathbf{x}|f(\mathbf{x}) = a\}, \qquad \text{with} \qquad a \in \{0,1\}. \tag{17.19}$$

It is a customary and approved method to define a switching circuit $f : \{0,1\}^n \mapsto \{0,1\}$ by specifying one of its equivalents, either $\mathcal{E}_0$ or $\mathcal{E}_1$. In fact, we have implicitly been doing this when defining a switching circuit by specifying a K-map. The cells which contain 1s represent the 1-equivalent $\mathcal{E}_1$ whereas the cells in which a 0 is inscribed represent the 0-equivalent $\mathcal{E}_0$.

17.6 Prime Sets

The basic idea in evaluating a K-map is to cover either the 1-equivalent $\mathcal{E}_1$ or the 0-equivalent $\mathcal{E}_0$ with the largest possible K-sets usually using as few of them as possible. A largest possible K-set is called a *prime set.* According to (17.3) we can write $\mathcal{K}_\varphi$ for $\mathcal{K}_{\varphi_1 \ldots \varphi_n}$ to denote K-sets in general. Although prime sets are K-sets, they are sufficiently important to justify the introduction of a characteristic symbol, a calligraphic $\mathcal{P}$ (always with an index), to represent them, e.g., writing $\mathcal{P}_\varphi$ (for $\mathcal{P}_{\varphi_1 \ldots \varphi_n}$), or $\mathcal{P}_\psi$. A formal definition of a prime set could look like this:

(17.20) A **prime set** $\mathcal{P}_\varphi$ of any subset $\mathcal{E}_S$ of $\mathcal{E}$ is a *largest* K-set that is itself a subset of $\mathcal{E}_S$. Let $\mathcal{P}_\varphi$ and $\mathcal{P}_\psi$ be K-sets that are subsets of $\mathcal{E}_S$:

$$\mathcal{P}_\varphi \text{ is a prime set of } \mathcal{E}_S \stackrel{df}{\Leftrightarrow} \neg \bigvee_{\mathcal{P}_\psi} \mathcal{P}_\varphi \subset \mathcal{P}_\psi.$$

The right side of (17.20) states that *not even one K-set* $\mathcal{P}_\psi$ of $\mathcal{E}_S$ *exists* — $(\neg \bigvee_{\mathcal{P}_\psi})$ — *which is a relative universe of* $\mathcal{P}_\varphi$ — $(\mathcal{P}_\varphi \subset \mathcal{P}_\psi)$. In K-maps of four variables or less, prime sets are very easy to recognise. For instance, each loop in Fig. 16–8a represents a prime set of $\mathcal{E}_1 = \{4, 8, 9, 12, 13, 14\}$. But, it takes some practice to find all prime sets on a K-map such as the one of Fig. 16–9. The defining expression $C_\varphi(x)$ of a prime set $\mathcal{P}_\varphi$ of the 1-equivalent $\mathcal{E}_1$ — $x \in \mathcal{P}_\varphi \Leftrightarrow C_\varphi(x)$ — is called a **prime implicant.**

17.7 Covers

The importance of large K-sets, in fact of prime sets, was first demonstrated in Sections 7.2 and 7.3 where the areas of **1**s and then of **0**s in K-maps were *completely covered* by prime sets. This aspect is stressed and generalised by introducing the concept of a *cover.*

(17.21) A **cover** $\mathcal{P}(\mathcal{E}_S)$ of any subset $\mathcal{E}_S \subseteq \mathcal{E}$ — let us refer to it as an $\mathcal{E}_S$-**cover** — is a set of prime sets

$$\mathcal{P}(\mathcal{E}_S) \stackrel{df}{=} \{\mathcal{P}_{\varphi(1)}, \cdots, \mathcal{P}_{\varphi(L)}\},$$

which are said to **cover** $\mathcal{E}_S$, meaning,

$$\mathcal{P}_{\varphi(1)} \cup \cdots \cup \mathcal{P}_{\varphi(L)} = \mathcal{E}_S$$

(a) *The* **full cover** $\mathcal{P}_{max}(\mathcal{E}_S)$ of $\mathcal{E}_S$ is the set of *all* prime sets of $\mathcal{E}_S$.

(b) An **irredundant cover** $\mathcal{P}_{irr}(\mathcal{E}_S)$ of $\mathcal{E}_S$ is a set of those prime sets of $\mathcal{E}_S$ each of which is necessary in comprising a union of $\mathcal{E}_S$ (i.e., removing any prime set from the irredundant cover turns the union of the remaining prime sets into a proper subset of $\mathcal{E}_S$).

(c) A **minimal cover** $\mathcal{P}_{min}(\mathcal{E}_S)$ of $\mathcal{E}_S$ is an irredundant cover with a smallest number of prime sets. (The expression *'minimal'* (instead of *'minimum'*) is used to stress that there can be more than one such cover).

Specialising, we refer to $\mathcal{P}(\mathcal{E}_1)$ as an $\mathcal{E}_1$-**cover**, but more simply as a **1-cover**. Similarly, $\mathcal{P}(\mathcal{E}_0)$ is called an $\mathcal{E}_0$-**cover**, a **0-cover**.

Covers are easily visualised in K-maps where a given subset $\mathcal{E}_a$ of input events is completely covered by loops representing prime sets. In the example of Fig. 17–2a the set $\mathcal{E}_1$ of input events is specified by the **1** entries. Together, all the loops in the figure comprise the *full cover* $\mathcal{P}_{max}(\mathcal{E}_1) = \{\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_C, \mathcal{P}_D, \mathcal{P}_E, \mathcal{P}_F\}$. As you can see, there are two *minimal covers*. One, $\mathcal{P}'_{min}(\mathcal{E}_1)$, represented by the dashed-line loops with $\mathcal{P}'_{min}(\mathcal{E}_1) = \{\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_C\}$, the other, $\mathcal{P}''_{min}(\mathcal{E}_1) = \{\mathcal{P}_D, \mathcal{P}_E, \mathcal{P}_F\}$, pictured as full-line loops.

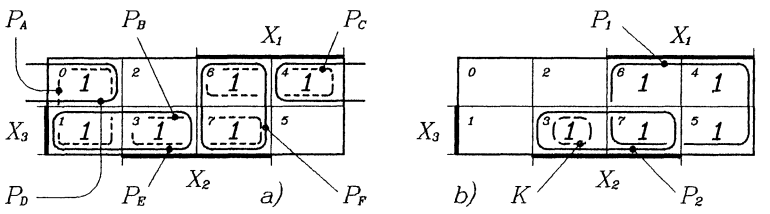


Fig. 17–2 Examples of covers

Of course, in general, prime sets of a minimal cover need not be disjoint, as pictured in Fig. 17–2b where a given subset $\mathcal{E}_1$ is covered by $\mathcal{P}_1$ and $\mathcal{P}_2$. On the other hand, $\{\mathcal{K}, \mathcal{P}_1\}$ is not a cover because $\mathcal{K}$ is not a prime set of $\mathcal{E}_1$.

17.8 Inclusions and Exclusions *

After having found or chosen a cover for a given subset $\mathcal{E}_S \subseteq \mathcal{E}$ of input events, we next need to write the logic formula associated with the cover. To be able to do this we introduce two concepts, *inclusion* and *exclusion*, that are of advantage in developing specialised versions of disjunctive or conjunctive forms, versions which are adapted to specialised applications. For generality, assume you have determined a cover (any cover, not necessarily a minimal cover)

$$\mathcal{P}(\mathcal{E}_S) = \{\mathcal{P}_{\varphi(1)}, \cdots, \mathcal{P}_{\varphi(L)}\}, \qquad \mathcal{E}_S = \mathcal{P}_{\varphi(1)} \cup \cdots \cup \mathcal{P}_{\varphi(L)} \tag{17.22}$$

of a given subset $\mathcal{E}_S$ of the universe $\mathcal{E}$ of input events, the cover consisting of the prime sets $\mathcal{P}_{\varphi(\ell)}$, with $\ell \in \{1, \ldots, L\}$. By $\mathcal{E}_S = \mathcal{P}_{\varphi(1)} \cup \cdots \cup \mathcal{P}_{\varphi(L)}$ and $\mathbf{x} \in \bigcup_{\ell=1}^{L} \mathcal{P}_{\varphi(\ell)} \Leftrightarrow \bigvee_{\ell=1}^{L} (\mathbf{x} \in \mathcal{P}_{\varphi(\ell)})$, (11.20), we can decide when $\mathbf{x}$ is an element of $\mathcal{E}_S$:

$$\mathbf{x} \in \mathcal{E}_S \Leftrightarrow (\mathbf{x} \in \mathcal{P}_{\varphi(1)}) \vee \ldots \vee (\mathbf{x} \in \mathcal{P}_{\varphi(L)}), \tag{17.23}$$

this, not surprisingly, being the case if $\mathbf{x}$ is an element of one or more of the prime sets. Being K-sets, the defining expressions of the prime sets $\mathcal{P}_{\varphi(\ell)}$ are the generalised minterms:

$$C_{\varphi(\ell)}(\mathbf{x}) \Leftrightarrow \mathbf{x} \in \mathcal{P}_{\varphi(\ell)}, \tag{17.9}$$

enabling us to rewrite (17.23) as

$$\mathbf{x} \in \mathcal{E}_S \Leftrightarrow C_{\varphi(1)} \vee \ldots \vee C_{\varphi(L)}, \tag{17.24}$$

or all inclusively as

$$\mathbf{x} \in \mathcal{E}_S \Leftrightarrow \bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_S)} C_\varphi(\mathbf{x}), \tag{17.25}$$

noting that $\varphi: \mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_S)$ is read: *those* φ *for which* $\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_S)$.

Let me refer to an input event $\mathbf{x}$ of a set $\mathcal{P}_\varphi$, which itself is an element of a set $\mathcal{P}(\mathcal{E}_S)$, as being *included* in $\mathcal{P}(\mathcal{E}_S)$. Then, by (17.22) we can say that if $\mathbf{x} \in \mathcal{E}_S$, and thus of one or more of the sets $\mathcal{P}_\varphi$, $\mathbf{x}$ is *included* in $\mathcal{P}(\mathcal{E}_S)$. Now, allow me to introduce a logic variable $U_S(\mathbf{x})$ — for which we shall sometimes simply write U_S — as an abbreviation for the right side of (17.25):

$$U_S(\mathbf{x}) \stackrel{df}{\Leftrightarrow} \bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_S)} C_\varphi(\mathbf{x}). \tag{17.26}$$

As $U_S(\mathbf{x})$ is true *iff* $\mathbf{x}$ is *included* in any of the covers $\mathcal{P}(\mathcal{E}_S)$ of $\mathcal{E}_S$ let us refer to $U_S(\mathbf{x})$ as an **inclusion**. The combination of (17.25) and (17.26),

$$U_S(\mathbf{x}) \Leftrightarrow \mathbf{x} \in \mathcal{E}_S, \tag{17.27}$$

is interpreted as '$U_S(\mathbf{x})$ is true *iff* $\mathbf{x} \in \mathcal{E}_S$'. Next, inverting (17.25),

$$\mathbf{x} \notin \mathcal{E}_S \Leftrightarrow \neg\Big(\bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_S)} C_\varphi(\mathbf{x})\Big) \Leftrightarrow \bigwedge_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_S)} D_\varphi(\mathbf{x}), \tag{17.28}$$

allows us to define the **exclusion** $V_S(\mathbf{x})$,

$$V_S(\mathbf{x}) \overset{df}{\Leftrightarrow} \bigwedge_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_S)} D_\varphi(\mathbf{x}) \Leftrightarrow \mathbf{x} \notin \mathcal{E}_S \Leftrightarrow \overline{U}_S(\mathbf{x}), \tag{**17.29**}$$

as being true *iff* $\mathbf{x}$ is *excluded* from $\mathcal{P}(\mathcal{E}_S)$, i.e., although the $\mathcal{P}_\varphi$ are elements of $\mathcal{P}(\mathcal{E}_S)$, $\mathbf{x}$ is *not included* in $\mathcal{P}(\mathcal{E}_S)$ because $\mathbf{x}$ is not an element of $\mathcal{E}_S$ and thus not of any of the $\mathcal{P}_\varphi$.

17.9 Evaluation Formulas *

The formulas (17.30) and (17.31), developed below, serve the same purpose as the canonical normal forms of Chapter 14 — they allow us to calculate logic formulas for given switching functions. But, in contrast to the *unique solution* provided by canonical normal forms, the *evaluation formulas* (as they are called here) provide multiple solutions — one for each choice of a cover. It is also interesting to note that the derivation of the evaluation formulas is completely independent of the derivation of the canonical normal forms (put forth in Chapter 14). Now, to the derivation: Combining (17.17) and their negations,

$$\begin{aligned} \mathbf{x} \in \mathcal{E}_1 &\Leftrightarrow Y(\mathbf{x}), & \mathbf{x} \in \mathcal{E}_0 &\Leftrightarrow \overline{Y}(\mathbf{x}), \\ \mathbf{x} \notin \mathcal{E}_1 &\Leftrightarrow \overline{Y}(\mathbf{x}), & \mathbf{x} \notin \mathcal{E}_0 &\Leftrightarrow Y(\mathbf{x}), \end{aligned}$$

with

$$\begin{aligned} \mathbf{x} \in \mathcal{E}_1 &\overset{(17.27)}{\Longleftrightarrow} U_1(\mathbf{x}), & \mathbf{x} \in \mathcal{E}_0 &\overset{(17.27)}{\Longleftrightarrow} U_0(\mathbf{x}), \\ \mathbf{x} \notin \mathcal{E}_1 &\overset{(17.29)}{\Longleftrightarrow} V_1(\mathbf{x}), & \mathbf{x} \notin \mathcal{E}_0 &\overset{(17.29)}{\Longleftrightarrow} V_0(\mathbf{x}), \end{aligned}$$

allows us to write so-called **evaluation formulas** as

$$Y(\mathbf{x}) \Leftrightarrow U_1(\mathbf{x}), \qquad \overline{Y}(\mathbf{x}) \Leftrightarrow U_0(\mathbf{x}), \tag{**17.30**}$$

$$\overline{Y}(\mathbf{x}) \Leftrightarrow V_1(\mathbf{x}), \qquad Y(\mathbf{x}) \Leftrightarrow V_0(\mathbf{x}), \tag{**17.31**}$$

or — using (17.26) and (17.29) — in an **explicit form** as

$$Y(\mathbf{x}) \overset{(17.30a)}{\Longleftrightarrow} \bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_1)} C_\varphi(\mathbf{x}), \qquad \overline{Y}(\mathbf{x}) \overset{(17.30b)}{\Longleftrightarrow} \bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_0)} C_\varphi(\mathbf{x}),$$

$$\overline{Y}(\mathbf{x}) \overset{(17.31a)}{\Longleftrightarrow} \bigwedge_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_1)} D_\varphi(\mathbf{x}), \qquad Y(\mathbf{x}) \overset{(17.31b)}{\Longleftrightarrow} \bigwedge_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_0)} D_\varphi(\mathbf{x}).$$

Let me refer to (17.30) as **inclusion evaluation-formulas**, and their usage as *evaluation by inclusion*. The logic formulas obtained by the inclusion evaluation-formulas are called **disjunctive forms**. The equations (17.31) are then referred to as **exclusion evaluation-formulas**, and their usage as *evaluation by exclusion*. The logic formulas obtained by the exclusion evaluation-formulas are called **conjunctive forms**.

The evaluation formulas are in close relationship to the canonical normal forms summarised here

$$Y(\mathbf{x}) \overset{(14.1)}{\Longleftrightarrow} \bigvee_{e=0}^{2^n-1} C_e(\mathbf{x}) \wedge Y(e), \qquad \overline{Y}(\mathbf{x}) \overset{(14.16)}{\Longleftrightarrow} \bigvee_{e=0}^{2^n-1} C_e(\mathbf{x}) \wedge \overline{Y}(e),$$

$$\overline{Y}(\mathbf{x}) \overset{(14.15)}{\Longleftrightarrow} \bigwedge_{e=0}^{2^n-1} D_e(\mathbf{x}) \vee \overline{Y}(e), \qquad Y(\mathbf{x}) \overset{(14.2)}{\Longleftrightarrow} \bigwedge_{e=0}^{2^n-1} D_e(\mathbf{x}) \vee Y(e).$$

To emphasise the close relationship between (14.1) and (17.30) we transform (14.1) using: $\mathcal{E} = \{0, 1, \ldots, 2^n - 1\}$; $\mathcal{E} = \mathcal{E}_0 \cup \mathcal{E}_1$; $\mathcal{E}_0 \cap \mathcal{E}_1 = \emptyset$; $\mathbf{x} \in \mathcal{E}_0 \Leftrightarrow \overline{Y}(\mathbf{x})$; $\mathbf{x} \in \mathcal{E}_1 \Leftrightarrow Y(\mathbf{x})$.

$$\begin{aligned} Y(\mathbf{x}) &\Leftrightarrow \bigvee_{e=0}^{2^n-1} C_e(\mathbf{x}) Y(e) \Leftrightarrow \bigvee_{e \in \mathcal{E}} C_e(\mathbf{x}) Y(e) \\ &\Leftrightarrow \Big(\bigvee_{e \in \mathcal{E}_0} C_e(\mathbf{x}) \underbrace{Y(e)}_{\mathbf{0}} \Big) \vee \Big(\bigvee_{e \in \mathcal{E}_1} C_e(\mathbf{x}) \underbrace{Y(e)}_{\mathbf{1}} \Big) \Leftrightarrow \bigvee_{e \in \mathcal{E}_1} C_e(\mathbf{x}). \end{aligned}$$

A comparison of the above form of (14.1), $Y(\mathbf{x}) \Leftrightarrow \bigvee_{e \in \mathcal{E}_1} C_e(\mathbf{x})$, and the explicit form of the evaluation formula (17.30a) give rise to asserting the following complementary logic equivalencies:

$$\bigvee_{e \in \mathcal{E}_S} C_e(\mathbf{x}) \Leftrightarrow \bigvee_{\varphi: \mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_S)} C_\varphi(\mathbf{x}), \qquad \bigwedge_{e \in \mathcal{E}_S} D_e(\mathbf{x}) \Leftrightarrow \bigwedge_{\varphi: \mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_S)} D_\varphi(\mathbf{x}). \tag{17.32}$$

The minterm version of (17.32) follows from (17.25) and from

$$\mathbf{x} \in \mathcal{E}_S \Leftrightarrow \bigvee_{e \in \mathcal{E}_S} C_e(\mathbf{x}) \Leftrightarrow \bigvee_{e \in \mathcal{E}_S} (\mathbf{x} = \mathbf{e}). \tag{17.33}$$

To prove this formula, assume $\mathcal{E}_S$ to be some arbitrary subset of $\mathcal{E}$, say, $\mathcal{E}_S = \{\mathbf{e}^{(1)}, \mathbf{e}^{(2)}, \ldots, \mathbf{e}^{(L)}\}$. This allows us to write (17.33) as

$$\mathbf{x} \in \{\mathbf{e}^{(1)}, \mathbf{e}^{(2)}, \ldots, \mathbf{e}^{(L)}\} \leftrightarrow (\mathbf{x} = \mathbf{e}^{(1)}) \vee (\mathbf{x} = \mathbf{e}^{(2)}) \vee \ldots \vee (\mathbf{x} = \mathbf{e}^{(L)}),$$

which is clearly true for any $\mathbf{x}$.

Chapter 18

Minimisation Preliminaries

Loosely stated, minimisation usually refers to the simplification of *logic formulas* obtained as canonical normal forms, either disjunctive or conjunctive, from (14.1) and (14.2). In this presentation, emphasis is instead turned to finding *full and minimal covers* while leaving their logic description to the evaluation formulas (17.30) and (17.31). In this approach there is no need to explain the minimisation of disjunctive and conjunctive formulas separately, and the algorithmic (tabular) minimisation procedures are brought closer to the graphic ones.

Switching functions describing real-word problems are often incompletely specified, meaning, we may (in fact, we must) choose output values for those input events whose output values are not specified. This offers a welcome opportunity to complete the specification so as to obtain a fully defined switching function the logic formula of which is of the desired simplicity.

To get a feeling for minimisation, we briefly look at a graphic and at two algebraic methods. Not surprisingly, the graphic method of my choice is the K-map although the n-cube (merely touched on in Section 3.3) could be used equally well (e.g., **Dietmeyer** [1972]). The algebraic methods presented were proposed by **Quine** [1952, 1955], the father of minimisation.

18.1 Aspects of Minimisation

Minimisation covers more aspects than trying to find a simple logic formula for a given switching function. Consider the following points.

Firstly, *reducing cost.* Taking *cost* in a general sense, it can be understood to stand for the number of switching devices, the number of connections, power consumption, space on a printed circuit board, number of its layers, and many aspects more. To enable a simplified formal discussion of minimisation, this text is restricted to finding *two-level circuits consisting of a minimal number of gates with the smallest possible number of connections* — such a circuit being called *minimal.* I leave it to you to take the more device-oriented aspects into account in a separate step, doing so on the basis of a minimal solution found.

Secondly, *promoting theoretical developments.* Due to the rapid development of cheap electronic switching devices, minimisation is sometimes looked upon as an expensive luxury. This is a rather one-sided view, for, minimisation was and is a prime motive for developing new concepts such as in the previous chapter and the next one. Of course, introducing new concepts is not an end unto itself — moreover, they enable us to achieve deeper insight and more general solutions.

Thirdly, *circuit malfunction.* As will be discussed in a later chapter, minimisation has quite a pronounced influence on the way a circuit is prone to erroneous output signals.

18.2 Incomplete Specification

The specification of a technical problem, more often than not, does not require us to state the output value for *every* input event. Such switching functions are called **incompletely specified functions**. Not specifying certain output values usually means we **don't care** what their actual values are. This gives us the possibility to choose appropriate output values by some predefined criterion. But, this is not only a question of freedom gained, for only if we specify or choose output values for *all* input events can we apply the evaluation formulas. Usually one of two reasons prevails if not all output values are specified. Let us consider an example for each.

Non-occurrence of certain Input Events

In a model railway a specific point rail (Fig. 18–1) tends to get stuck between its two connecting rails thereby derailing an incoming train. To remedy this we decide to let the train pass ($y = 1$) only if one of the position switches (x_1 or x_2) is activated. If neither of the switches is activated, the point rail is either moving from one connecting rail to the other, or is stuck. Assuming flawless functioning of the switches, they will never be activated simultaneously, i.e., the input event $(x_1, x_2) = (1, 1)$ will never occur, making the value of the associated output irrelevant.

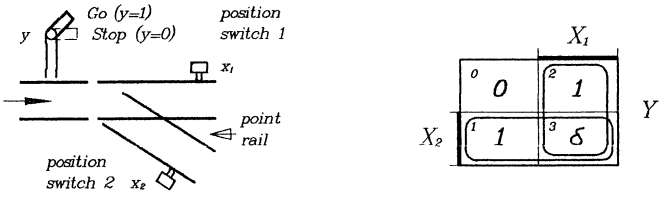


Fig. 18–1 Overseeing the functioning of a point rail.

Unspecified output values are called **don't care output values** or simply **don't cares** and are, for mnemonical reasons, represented in this text by the small greek letter *delta*, δ.

If you imagine the *don't care* symbol δ of Fig. 18–1 substituted by 1, you can cover the 1s by the loops shown, obtaining $Y \Leftrightarrow X_1 \vee X_2$. Had we substituted the *don't care* by 0, the result would have the far more complicated description $Y \Leftrightarrow X_1\overline{X}_2 \vee \overline{X}_1 X_2$. This example shows nicely how output values can be chosen to obtain simpler (sometimes, far simpler) results. But do not be mislead by this possibility. Safety is usually more important than a minimal result. In our example, consider what would happen if a position switch were to jam after having been activated.

Irrelevance of certain Output Values

Computers frequently work with other binary codes than the **BCD code** already mentioned in Section 16.2. The **excess-3 code** (XS3 code), for instance, is a non-weighted binary code with the advantage of being self-complementary. In an n-ary *number system*, with the digits $s_0, s_1, \cdots, s_{n-1}$, the **complement** $\overline{s}_i$ of the digit s_i is defined as $\overline{s}_i \stackrel{df}{=} (n-1) - s_i$. In the decimal number system the pairs (0, 9), (1, 8), (2, 7), etc., are complementary digits, while in the binary number system 0 and 1 are complementary. Self-complementarity of the XS3 code refers to the fact that, given $(d)_{10} = (u_1, u_2, u_3, u_4)_{XS3}$, the complement $(\overline{d})_{10} = \overline{(u_1, u_2, u_3, u_4)}_{XS3}$ is obtained by complementing each XS3-digit u_i individually, expressed as $\overline{(u_1, u_2, u_3, u_4)}_{XS3} = (\overline{u}_1, \overline{u}_2, \overline{u}_3, \overline{u}_4)_{XS3}$. The excess-3 code gets its name from the fact that, in each row of Fig. 18–2, the excess-3 value *exceeds* the BCD value by 3: $(u_1, u_2, u_3, u_4)_{XS3} = (b_1, b_2, b_3, b_4)_{BCD} + (0011)_{BCD}$.

Decimal	BCD				Excess-3				Input
d	b_1	b_2	b_3	b_4	u_1	u_2	u_3	u_4	event
0	0	0	0	0	0	0	1	1	3
1	0	0	0	1	0	1	0	0	4
2	0	0	1	0	0	1	0	1	5
3	0	0	1	1	0	1	1	0	6
4	0	1	0	0	0	1	1	1	7
5	0	1	0	1	1	0	0	0	8
6	0	1	1	0	1	0	0	1	9
7	0	1	1	1	1	0	1	0	10
8	1	0	0	0	1	0	1	1	11
9	1	0	0	1	1	1	0	0	12

Fig. 18–2 The Excess-3 Code

Now let us consider a circuit that will transform any XS3 value into an equivalent BCD value, according to the table of Fig. 18–2. Actually, we shall have to design four circuits — one for each BCD digit b_1, b_2, b_3, b_4, while

each such circuit has as input event the quadruple $(u_1, u_2, u_3, u_4)_{XS3}$. These circuits are specified in the K-maps of Fig. 18–3 for all input events 3 to 12 of the table of Fig. 18–2.

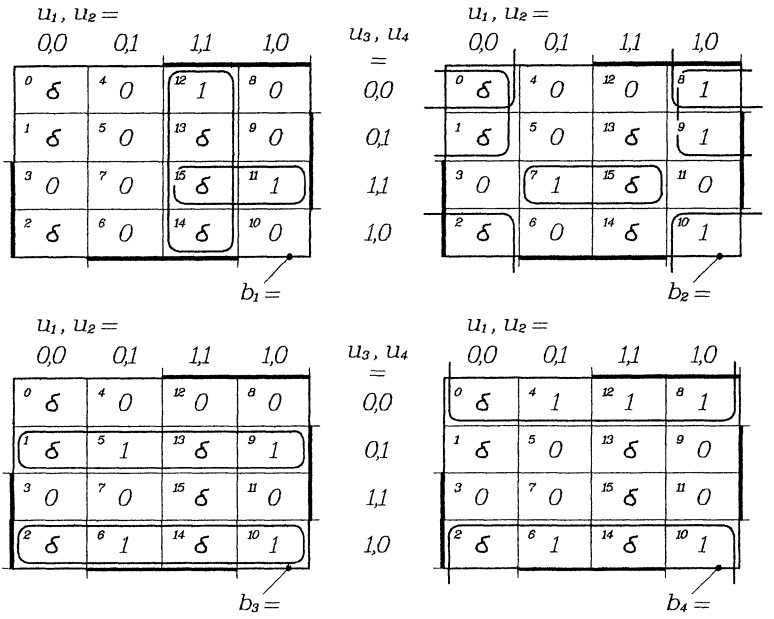


Fig. 18–3 Transforming the Excess-3 code to the BCD code

As we are only concerned with transforming XS3 values to BCD values, let us argue that, for our purposes, the output values (b_1, b_2, b_3, b_4) obtained are irrelevant for input values (u_1, u_2, u_3, u_4) that are not XS3 code values. Such non-XS3 input values are the input events 0, 1, 2, and 13, 14, 15. It is for these that we enter *don't cares* δ into the K-maps.

18.3 Prime Sets that cover Don't Cares

Each cell of a K-map representing an incompletely specified function has one of the symbols of $\{0, 1, \delta\}$ inscribed. The input events that map to δ comprise the **set of unassigned input events** $\mathcal{E}_\delta$. To calculate a logic expression for the output variable, each *don't care* δ must be substituted by either 0 or 1. These substitutions are usually done in accordance with a minimisation, safety, or reliability criterion. Only after we decide which δs to substitute by 0, and which by 1, do we possess a fully specified function, one that partitions its domain solely into a 0-equivalent $\mathcal{E}_0$ and a 1-equivalent $\mathcal{E}_1$.

Those input events of the incompletely specified function whose cells have a 1 inscribed thus do *not* comprise the 1-equivalent $\mathcal{E}_1$, but they are its *nucleus* — so let us call the set of these cells (or input events) the 1-**nucleus** $\mathcal{E}_{1\nu}$. In a like manner, those input events that are mapped to 0 are the 0-**nucleus** $\mathcal{E}_{0\nu}$, i.e., the nucleus of the 0-equivalent $\mathcal{E}_0$. In the incompletely specified function of Fig. 18–4a, for instance, the 1-nucleus $\mathcal{E}_{1\nu}$ is $\{4, 6, 8, 10, 12\}$. If we choose to substitute the *don't cares* δ of the unassigned input events $0, 2, 14$ by the output value 1, then the function obtained would have the 1-equivalent $\mathcal{E}_1 = \mathcal{P}_C = \{0, 2, 4, 6, 8, 10, 12, 14\}$.

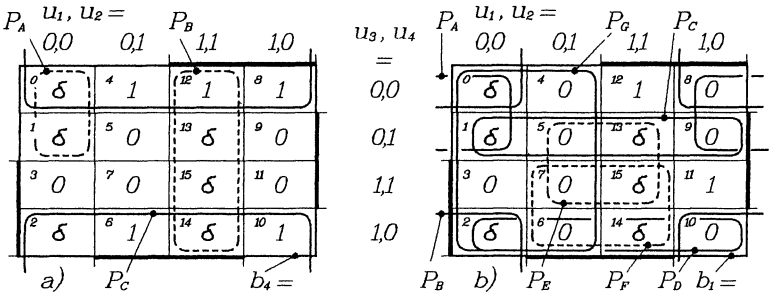


Fig. 18–4 Illustrating the strategy for choosing *don't cares*

When covering an a-nucleus $\mathcal{E}_{a\nu}$ with prime sets, we use any *don't care* input events $\mathbf{x} \in \mathcal{E}_\delta$ which allow us to create larger prime sets than would be possible when restricting ourselves solely to input events $\mathbf{x}$ of the a-nucleus $\mathcal{E}_{a\nu}$. That is, we build prime sets not of $\mathcal{E}_{a\nu}$, but of $\mathcal{E}_{a\nu} \cup \mathcal{E}_{\delta S}$, using any subset $\mathcal{E}_{\delta S} \subseteq \mathcal{E}_\delta$ we deem appropriate.

Unreflectingly using the whole set $\mathcal{E}_\delta$, instead of a subset $\mathcal{E}_{\delta S}$, can lead to some quite unnecessary (*redundant*) prime sets, as the ones drawn as dashed loops in Fig. 18–4. To explain when a prime set is redundant, we must first define the concept of *essential* prime sets, prime sets that under no circumstances can be left out of a solution.

(18.1) A prime set $\mathcal{P}_\varphi \subseteq \mathcal{E}_{a\nu} \cup \mathcal{E}_{\delta S}$ is an **essential prime set** if it contains at least one input event $\mathbf{x}$ belonging to the a-nucleus $\mathcal{E}_{a\nu}$ — $\mathbf{x} \in \mathcal{E}_{a\nu}$ — and which is not contained in any other prime set of $\mathcal{E}_{a\nu} \cup \mathcal{E}_{\delta S}$.

As an example of an essential prime set, consider $\mathcal{P}_G \subseteq \mathcal{E}_{0\nu} \cup \mathcal{E}_\delta$ in Fig. 18–4b. $\mathcal{P}_G$ is an essential prime set because the input-event number 4, contained in $\mathcal{P}_G$, is not contained in any other prime set of $\mathcal{E}_{0\nu} \cup \mathcal{E}_\delta$.

(18.2) A prime set $\mathcal{P}_\varphi \subseteq \mathcal{E}_{a\nu} \cup \mathcal{E}_{\delta S}$ is said to be **redundant**

(a) if each of its input events that is an element of the a-nucleus, $\mathbf{x} \in \mathcal{E}_{a\nu}$, is also an elements of an essential prime set of $\mathcal{E}_{a\nu} \cup \mathcal{E}_{\delta S}$, or

(b) if all of its input events $\mathbf{x}$ are elements of $\mathcal{E}_\delta$, i.e., are *unassigned input events.*

Both prime sets $\mathcal{P}_A$ and $\mathcal{P}_B$ of Fig. 18–4a are redundant. $\mathcal{P}_A$ is redundant because, (18.2b), all its elements are unassigned input events, i.e., belong to $\mathcal{E}_\delta$, while $\mathcal{P}_B$ is redundant because, (18.2a), the input event that evokes the output value 1 — cell number 12 — is an element of an essential prime set. In Fig. 18–4b you will recognise $\mathcal{P}_E$ and $\mathcal{P}_F$ as redundant.

We now turn to the concept of **a cover in the presence of don't cares.** The notation of these covers has to fit cases more complicated than the ones discussed in this chapter — for instance, covers on reduced K-maps (the next chapter), and covers on latches (Chapter 25). We thus choose the expandable notation $\mathcal{P}(\mathcal{E}_L;\mathcal{E}_R)$ in which the set written to the left of the semicolon — $\mathcal{E}_L$ — is the set each element of which we wish to cover, while the set written to the right of the semicolon — $\mathcal{E}_R$ — consists of appropriately chosen elements that do not occur in $\mathcal{E}_L$. The elements of $\mathcal{E}_R$ are chosen so as to facilitate the construction of larger prime sets than would otherwise be possible. In this chapter, we only look into the simple cases in which $\mathcal{E}_L$ stands for either $\mathcal{E}_{1\nu}$ or $\mathcal{E}_{0\nu}$, for which we summarily write $\mathcal{E}_{a\nu}$, with $a \in \{0,1\}$, and in which $\mathcal{E}_R$ is $\mathcal{E}_{\delta S} \subseteq \mathcal{E}_\delta$.

When minimising, the redundant prime sets are omitted from any cover. Let me use the symbolic notation $\mathcal{P}^{\overline{r}}(\mathcal{E}_{a\nu};\mathcal{E}_{\delta S})$ for a cover from which the redundant prime sets have been omitted (mnemonically, you can take the superscript $\overline{r}$ to stand for *'no redundant prime sets'*). We still need to define

(18.3) An $\mathcal{E}_{a\nu}$**-cover** $\mathcal{P}^{\overline{r}}(\mathcal{E}_{a\nu};\mathcal{E}_{\delta S})$ **which takes unassigned input events** $\mathbf{x} \in \mathcal{E}_{\delta S}$ from an arbitrary subset $\mathcal{E}_{\delta S} \subseteq \mathcal{E}_\delta$ **into consideration** is a set of prime sets

$$\mathcal{P}^{\overline{r}}(\mathcal{E}_{a\nu};\mathcal{E}_{\delta S}) = \{\mathcal{P}_{\varphi(1)},\cdots,\mathcal{P}_{\varphi(L)}\},$$

which cover $\mathcal{E}_{a\nu} \cup \mathcal{E}_{\delta S}$,

$$\mathcal{P}_{\varphi(1)} \cup \cdots \cup \mathcal{P}_{\varphi(L)} = \mathcal{E}_{a\nu} \cup \mathcal{E}_{\delta S},$$

none of the prime sets $\mathcal{P}_\varphi^{(\ell)} \subseteq \mathcal{E}_{a\nu} \cup \mathcal{E}_{\delta S}$ being redundant.

(a) The **full cover** $\mathcal{P}^{\overline{r}}_{max}(\mathcal{E}_{a\nu};\mathcal{E}_\delta)$ is the set of *all* prime sets of $\mathcal{E}_{a\nu} \cup \mathcal{E}_{\delta S}$ that are not redundant.

(b) An **irredundant cover** $\mathcal{P}^{\overline{r}}_{ir}(\mathcal{E}_{a\nu};\mathcal{E}_{\delta S})$ of the a-nucleus $\mathcal{E}_{a\nu}$ is a set of non-redundant prime sets that are *necessary* in covering $\mathcal{E}_{a\nu}$ (i.e., removing any prime set from the irredundant cover leaves at least one input event $\mathbf{x} \in \mathcal{E}_{a\nu}$ uncovered, meaning, there is no prime set of which it is an element).

(c) A **minimal cover** $\mathcal{P}^{\overline{r}}_{min}(\mathcal{E}_{a\nu};\mathcal{E}_{\delta S})$ is an irredundant cover with a smallest number of prime sets.

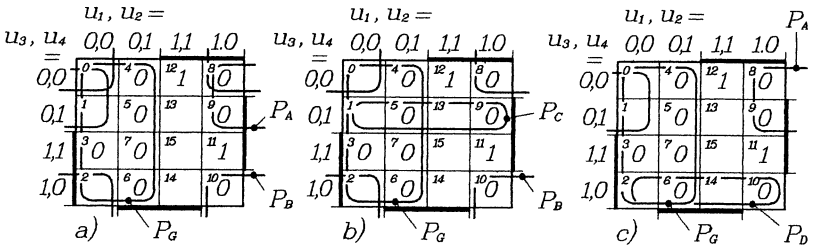


Fig. 18–5 The three minimal covers of b_1 of Fig. 18–4b

To illustrate the concept of cover consider the K-map of Fig. 18–4b. The set of the five prime sets $\mathcal{P}_A$, $\mathcal{P}_B$, $\mathcal{P}_C$, $\mathcal{P}_D$, $\mathcal{P}_G$ comprises the full cover $\mathcal{P}^{\overline{r}}_{max}(\mathcal{E}_{0\nu};\mathcal{E}_\delta)$. As stated, $\mathcal{P}_G$ is essential, and $\mathcal{P}_E$ and $\mathcal{P}_F$ are redundant. Let us now look for *all minimal* covers. If we choose to retain $\mathcal{P}_G$ and $\mathcal{P}_A$, then it suffices to keep either $\mathcal{P}_B$ or $\mathcal{P}_D$ to obtain the minimal covers $\{\mathcal{P}_A,\mathcal{P}_B,\mathcal{P}_G\}$ or $\{\mathcal{P}_A,\mathcal{P}_D,\mathcal{P}_G\}$ of $\mathcal{E}_{0\nu}$, see Figs. 18–5a,c. If, on the other hand, we choose to retain $\mathcal{P}_C$ together with $\mathcal{P}_G$, we need $\mathcal{P}_B$ to obtain the minimal cover $\{\mathcal{P}_B,\mathcal{P}_C,\mathcal{P}_G\}$ of $\mathcal{E}_{0\nu}$, Fig. 18–5b.

18.4 Evaluation Formulas for Don't Cares *

The basic evaluation formulas (17.30) and (17.31) are of course applicable to any set we define to be $\mathcal{E}_1$ (for instance, for $\mathcal{E}_{1\nu} \cup \mathcal{E}_{\delta S}$) or $\mathcal{E}_0$ (such as $\mathcal{E}_{0\nu} \cup \mathcal{E}_{\delta S}$). When taking *don't cares* into account we may write the **inclusion evaluation-formulas** as

$$Y(\mathbf{x}) \Leftrightarrow U_{1;\delta}(\mathbf{x}) \overset{df}{\Leftrightarrow} \bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_{1\nu};\mathcal{E}_{\delta S})} C_\varphi(\mathbf{x}), \tag{18.4}$$

$$\overline{Y}(\mathbf{x}) \Leftrightarrow U_{0;\delta}(\mathbf{x}) \overset{df}{\Leftrightarrow} \bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_{0\nu};\mathcal{E}_{\delta S})} C_\varphi(\mathbf{x}), \tag{18.5}$$

and the **exclusion evaluation-formulas** as

$$Y(\mathbf{x}) \Leftrightarrow V_{0;\delta}(\mathbf{x}) \overset{df}{\Leftrightarrow} \bigwedge_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_{0\nu};\mathcal{E}_{\delta S})} D_\varphi(\mathbf{x}), \tag{18.6}$$

$$\overline{Y}(\mathbf{x}) \Leftrightarrow V_{1;\delta}(\mathbf{x}) \overset{df}{\Leftrightarrow} \bigwedge_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_{1\nu};\mathcal{E}_{\delta S})} D_\varphi(\mathbf{x}). \tag{18.7}$$

Understand $\mathcal{P}(\mathcal{E}_{a\nu};\mathcal{E}_{\delta S})$ to differ from $\mathcal{P}^{\overline{r}}(\mathcal{E}_{a\nu};\mathcal{E}_{\delta S})$, defined in (18.3), only in that the former *may* contain redundant prime sets whereas the latter *may not*.

18.5 Minimisation by K-map

By now, you will probably have become proficient in using K-maps intuitively to obtain disjunctive forms (as in Section 7.2) or conjunctive forms (as in Section 7.3). But, there is a slightly different way of explaining the use of K-maps as follows from the evaluation formulas:

(18.8) **Finding a minimal disjunctive or conjunctive form:**

(a) Given some subset $\mathcal{E}_{a\nu} \cup \mathcal{E}_{\delta S}$, determine a minimal cover such that $\mathcal{P}^{\overline{r}}_{min}(\mathcal{E}_{a\nu}; \mathcal{E}_{\delta S}) = \mathcal{P}_{\varphi(1)} \cup \cdots \cup \mathcal{P}_{\varphi(L)}$ by *'visual inspection'**) of the K-map.

(b) You obtain the disjunctive form by applying (18.4) or (18.5), and the conjunctive forms from (18.6) or (18.7)

As an example, consider the minimal covers $\mathcal{P}^{\overline{r}}_{min}(\mathcal{E}_{0\nu}; \mathcal{E}_{\delta S})$ of Fig. 18–4b obtained by visual inspection (as (18.8a) requires), each minimal cover being pictured in one of the K-maps of Fig. 18–5.

$$\mathcal{P}^{(a)}_{min}(\mathcal{E}_{0\nu}; \mathcal{E}_{\delta}) = \{\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_G\} = \{\mathcal{P}_{x00x}, \mathcal{P}_{x0x0}, \mathcal{P}_{0xxx}\},$$
$$\mathcal{P}^{(b)}_{min}(\mathcal{E}_{0\nu}; \mathcal{E}_{\delta}) = \{\mathcal{P}_B, \mathcal{P}_C, \mathcal{P}_G\} = \{\mathcal{P}_{x0x0}, \mathcal{P}_{xx01}, \mathcal{P}_{0xxx}\},$$
$$\mathcal{P}^{(c)}_{min}(\mathcal{E}_{0\nu}; \mathcal{E}_{\delta}) = \{\mathcal{P}_A, \mathcal{P}_D, \mathcal{P}_G\} = \{\mathcal{P}_{x00x}, \mathcal{P}_{xx10}, \mathcal{P}_{0xxx}\}.$$

Then, let us use the evaluation formula (18.6):

$$Y_1^{(a)} \Leftrightarrow V^{(a)}_{0;\delta}(\mathbf{x}) \Leftrightarrow \bigwedge_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}^{(a)}_{min}(\mathcal{E}_{0\nu};\mathcal{E}_\delta)} D_\varphi$$
$$\Leftrightarrow \mathcal{D}_{x00x}\mathcal{D}_{x0x0}\mathcal{D}_{0xxx} \Leftrightarrow (X_2 \vee X_3)(X_2 \vee X_4)X_1,$$

$$Y_1^{(b)} \Leftrightarrow V^{(b)}_{0;\delta}(\mathbf{x}) \Leftrightarrow \bigwedge_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}^{(b)}_{min}(\mathcal{E}_{0\nu};\mathcal{E}_\delta)} D_\varphi$$
$$\Leftrightarrow \mathcal{D}_{x0x0}\mathcal{D}_{xx01}\mathcal{D}_{0xxx} \Leftrightarrow (X_2 \vee X_4)(X_3 \vee \overline{X}_4)X_1,$$

$$Y_1^{(c)} \Leftrightarrow V^{(c)}_{0;\delta}(\mathbf{x}) \Leftrightarrow \bigwedge_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}^{(c)}_{min}(\mathcal{E}_{0\nu};\mathcal{E}_\delta)} D_\varphi$$
$$\Leftrightarrow \mathcal{D}_{x00x}\mathcal{D}_{xx10}\mathcal{D}_{0xxx} \Leftrightarrow (X_2 \vee X_3)(\overline{X}_3 \vee X_4)X_1.$$

---

*) an euphemism expressing that there is no way to state formally and systematically what to do.

18.6 Algebraic Minimisation

Due to *'visual inspection'* graphical minimisation methods are more an art than a science, but, when mastered give very good results. Algebraic minimisation methods are more systematic than graphic ones, but are onerous to use and only provide half the result. The latter meaning, you calculate the full cover $\mathcal{P}_{max}(\mathcal{E}_a)$ and not a minimal cover $\mathcal{P}_{min}(\mathcal{E}_a)$. Algebraic minimisation methods are systematic enough to act as the basis for computerizable algorithms (the topic of the next chapter).

(18.9) **Quine's first minimisation method**: From the *disjunctive normal form* you obtain the disjunctive form associated with a *full cover* by repeated application of the theorem

$$XP(\mathbf{x}) \vee \overline{X}P(\mathbf{x}) \Leftrightarrow P(\mathbf{x})$$

whenever possible and finally dropping duplicate expressions .

May the following example suffice as illustration. Given:

$$\begin{aligned} Y \Leftrightarrow\ & \underbrace{\overline{X}_1\overline{X}_2\overline{X}_3\overline{X}_4}_{1} \vee \underbrace{\overline{X}_1\overline{X}_2\overline{X}_3X_4}_{2} \vee \underbrace{\overline{X}_1\overline{X}_2X_3\overline{X}_4}_{3} \vee \underbrace{\overline{X}_1X_2\overline{X}_3X_4}_{4} \\ & \vee \underbrace{\overline{X}_1X_2X_3X_4}_{5} \vee \underbrace{X_1\overline{X}_2\overline{X}_3\overline{X}_4}_{6} \vee \underbrace{X_1\overline{X}_2\overline{X}_3X_4}_{7} \vee \underbrace{X_1\overline{X}_2X_3\overline{X}_4}_{8} \\ & \vee \underbrace{X_1X_2\overline{X}_3\overline{X}_4}_{9} \vee \underbrace{X_1X_2\overline{X}_3X_4}_{10} \vee \underbrace{X_1X_2X_3\overline{X}_4}_{11}. \end{aligned} \tag{18.10}$$

Now, applying Quine's first method a first time, we get

$$\begin{aligned} Y \Leftrightarrow\ & \underbrace{\overline{X}_1\overline{X}_2\overline{X}_3}_{1-2} \vee \underbrace{\overline{X}_1\overline{X}_2\overline{X}_4}_{1-3} \vee \underbrace{\overline{X}_2\overline{X}_3\overline{X}_4}_{1-6} \vee \underbrace{\overline{X}_1\overline{X}_3X_4}_{2-4} \vee \underbrace{\overline{X}_2\overline{X}_3X_4}_{2-7} \\ & \vee \underbrace{\overline{X}_2X_3\overline{X}_4}_{3-8} \vee \underbrace{\overline{X}_1X_2X_4}_{4-5} \vee \underbrace{X_2\overline{X}_3X_4}_{4-10} \vee \underbrace{X_1\overline{X}_2\overline{X}_3}_{6-7} \vee \underbrace{X_1\overline{X}_2\overline{X}_4}_{6-8} \\ & \vee \underbrace{X_1\overline{X}_3\overline{X}_4}_{6-9} \vee \underbrace{X_1\overline{X}_3X_4}_{7-10} \vee \underbrace{X_1X_3\overline{X}_4}_{8-11} \vee \underbrace{X_1X_2\overline{X}_3}_{9-10} \vee \underbrace{X_1X_2\overline{X}_4}_{9-11}. \end{aligned}$$

Apply (18.9) yet again,

$$\begin{aligned} Y \Leftrightarrow\ & \overline{X}_2\overline{X}_3 \vee \overline{X}_2\overline{X}_4 \vee \overline{X}_2\overline{X}_3 \vee \overline{X}_2\overline{X}_4 \vee \overline{X}_3X_4 \\ & \vee \overline{X}_3X_4 \vee \overline{X}_1X_2X_4 \vee X_1\overline{X}_3 \vee X_1\overline{X}_4 \vee X_1\overline{X}_4, \end{aligned}$$

and remove duplicates:

$$Y \Leftrightarrow \overline{X}_2\overline{X}_3 \vee \overline{X}_2\overline{X}_4 \vee \overline{X}_3X_4 \vee \overline{X}_1X_2X_4 \vee X_1\overline{X}_3 \vee X_1\overline{X}_4.$$

As $XP(\mathbf{x}) \vee \overline{X}P(\mathbf{x}) \Leftrightarrow P(\mathbf{x})$ cannot be applied again, the procedure terminates, and the result contains all *prime implicants*.

Quine's first minimisation method must be started from a canonical normal form. To illustrate this necessity, consider starting from an arbitrary disjunctive form of (18.10)

$$Y \Leftrightarrow \overline{X}_1\overline{X}_2\overline{X}_3 \vee \overline{X}_1\overline{X}_2X_3\overline{X}_4 \vee \overline{X}_1X_2X_4 \vee X_1\overline{X}_3 \vee X_1X_3\overline{X}_4. \qquad (18.11)$$

In this example, $XP(\mathbf{x}) \vee \overline{X}P(\mathbf{x}) \Leftrightarrow P(\mathbf{x})$ cannot be applied at all. Even if we could apply it, we would not get very far. We first need to transform an example, as the one above, into a normal form which we do by ANDing the conjunctions by $(X_i \vee \overline{X}_i)$ for each missing variable, and then employing distributivity:

$$\begin{aligned} Y \Leftrightarrow\ & \overline{X}_1\overline{X}_2\overline{X}_3(X_4 \vee \overline{X}_4) \vee \overline{X}_1\overline{X}_2X_3\overline{X}_4 \vee \overline{X}_1X_2(X_3 \vee \overline{X}_3)X_4 \\ & \vee X_1(X_2 \vee \overline{X}_2)\overline{X}_3(X_4 \vee \overline{X}_4) \vee X_1(X_2 \vee \overline{X}_2)X_3\overline{X}_4. \end{aligned}$$

$$\begin{aligned} Y \Leftrightarrow\ & \underbrace{\overline{X}_1\overline{X}_2\overline{X}_3X_4}_{2} \vee \underbrace{\overline{X}_1\overline{X}_2\overline{X}_3\overline{X}_4}_{1} \vee \underbrace{\overline{X}_1\overline{X}_2X_3\overline{X}_4}_{3} \vee \underbrace{\overline{X}_1X_2X_3X_4}_{5} \\ & \vee \underbrace{\overline{X}_1X_2\overline{X}_3X_4}_{4} \vee \underbrace{X_1X_2\overline{X}_3X_4}_{10} \vee \underbrace{X_1X_2\overline{X}_3\overline{X}_4}_{9} \vee \underbrace{X_1\overline{X}_2\overline{X}_3X_4}_{7} \\ & \vee \underbrace{X_1\overline{X}_2\overline{X}_3\overline{X}_4}_{6} \vee \underbrace{X_1X_2X_3\overline{X}_4}_{11} \vee \underbrace{X_1\overline{X}_2X_3\overline{X}_4}_{8}. \end{aligned}$$

This **expansion to normal forms** can itself become arduous. In theory, the expansion can be avoided by employing Quine's concept of *consensus*, this being the clause $P(\mathbf{x})Q(\mathbf{x})$ in what is often called the *consensus theorem* used to formulate

(18.12) **Quine's second minimisation method**: Given a logic expression as an arbitrary disjunctive form, you obtain the equivalent disjunctive form associated with the *full cover* by repeated application of the following steps.

(a) Drop superfluous expressions. By the two theorems

$$P(\mathbf{x})Q(\mathbf{x}) \vee P(\mathbf{x}) \Leftrightarrow P(\mathbf{x}) \quad \text{and} \quad X \vee \overline{X}P(\mathbf{x}) \Leftrightarrow X \vee P(\mathbf{x}),$$

you replace, wherever possible, expressions that coincide with the left sides of the theorems by expressions that coincide with the right sides of the theorems.

(b) Appending consensuses. Whenever allowed by the *consensus theorem*

$$XP(\mathbf{x}) \vee \overline{X}Q(\mathbf{x}) \Leftrightarrow XP(\mathbf{x}) \vee \overline{X}Q(\mathbf{x}) \vee P(\mathbf{x})Q(\mathbf{x})$$

disjunctively append the consensus $P(\mathbf{x})Q(\mathbf{x})$ if it is non contradictory.

Consider the following example of a three-variable function:

$$Y \Leftrightarrow X_1\overline{X}_3 \vee \overline{X}_1X_3 \vee X_1\overline{X}_2X_3 \vee \overline{X}_1X_2\overline{X}_3.$$

Neither of the theorems of the first step are applicable so we try the consensus theorem of the second step (successfully, as you can see):

$$Y \Leftrightarrow \underbrace{X_1\overline{X}_3}_{1} \vee \underbrace{\overline{X}_1X_3}_{2} \vee \underbrace{X_1\overline{X}_2X_3}_{3} \vee \underbrace{\overline{X}_1X_2\overline{X}_3}_{4}$$
$$\vee \underbrace{X_1\overline{X}_2}_{5:1-3} \vee \underbrace{X_2\overline{X}_3}_{6:1-4} \vee \underbrace{\overline{X}_2X_3}_{7:2-3} \vee \underbrace{\overline{X}_1X_2}_{8:2-4}.$$

Now, by $P(\mathbf{x})Q(\mathbf{x}) \vee P(\mathbf{x}) \Leftrightarrow P(\mathbf{x})$ of the first step:

$$\underbrace{\underbrace{X_1}_{P}\underbrace{\overline{X}_2}_{Q}}_{3} \underbrace{X_3}_{} \vee \underbrace{\underbrace{X_1}_{P}\underbrace{\overline{X}_2}_{}}_{5} \Leftrightarrow \underbrace{\underbrace{X_1}_{P}\underbrace{\overline{X}_2}_{}}_{P}, \qquad \text{drop} \qquad \underbrace{X_1\overline{X}_2X_3}_{3}.$$

$$\underbrace{\overline{X}_1X_2\overline{X}_3}_{4} \vee \underbrace{X_2\overline{X}_3}_{6} \Leftrightarrow X_2\overline{X}_3, \qquad \text{drop} \qquad \underbrace{\overline{X}_1X_2\overline{X}_3}_{4}.$$

This leaves us with the following formula which is the final result because none of the theorems of Quine's second method can be applied.

$$Y \Leftrightarrow X_1\overline{X}_3 \vee \overline{X}_1X_3 \vee X_1\overline{X}_2 \vee X_2\overline{X}_3 \vee \overline{X}_2X_3 \vee \overline{X}_1X_2.$$

The two algebraic minimisation methods touched on are far too onerous for practical applications — this is especially true for Quine's second method. But they pose a sound basis on which to develop tabular minimisation methods that can be formulated stringently enough to be used as computer algorithms. The best known is probably *Quine-McCluskey's algorithm* (**McCluskey** [1956]) which is based on (18.9). We shall concentrate on the more versatile consensus method, first taking a detailed look at the general concept of consensus.

Chapter 19

Minimisation — an Excerpt

Minimising relatively small switching functions (of no more than, say, six input variables) can be done quite efficiently by K-map. Larger problems should be solved by computer. To be able to write computer programs for minimisation, we need to (a) develop an algorithm by which to find a *full cover*, and (b) a further algorithm by which to select a *minimal cover* from the full cover. There is a large number of algorithms for finding a full cover. The one I choose to put forth in Section 3 is a tabular method based on the concept of consensus, so that you need not start with a canonical normal form. As **Petrick**'s [1956] approach to finding a minimal cover is neither theoretically founded nor is it algorithmic, you might find the solution proposed in Section 4 of some interest. The basis of any *versatile* minimisation procedure is the concept of consensus. This we develop first.

19.1 On Adjacent K-Sets and their Consensus *

Adjacency of two K-sets is a property defined solely to be able to find a new, and third, K-set called a *consensus*. Informally, two K-sets are said to be **adjacent** if they are disjoint (i.e., have no elements in common), and if they have at least partially common edges (or boarders) in a K-map.

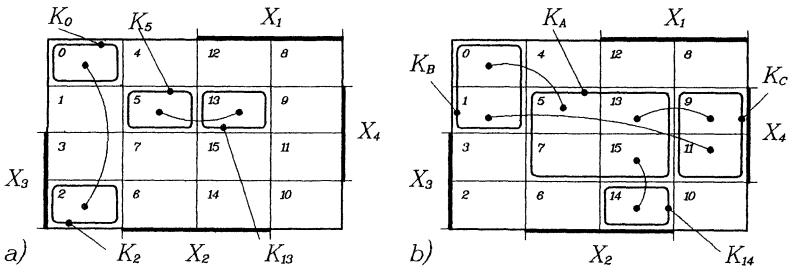


Fig. 19–1 Examples of adjacent K-sets

Adjacency was originally conceived for elementary K-sets $\mathcal{K}_e$, as those pictured in Fig. 19–1a, where $\mathcal{K}_0$ and $\mathcal{K}_2$, as well as $\mathcal{K}_5$ and $\mathcal{K}_{13}$ are adjacent.

This original adjacency concept of two *elementary* K-sets is extended to the adjacency of any two K-sets as stated above.

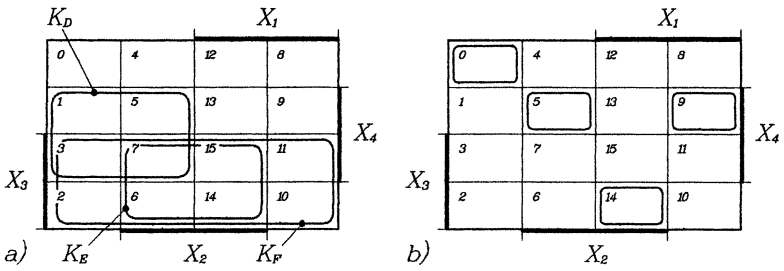


Fig. 19–2 Examples of non-adjacent K-sets

The K-sets $(\mathcal{K}_A, \mathcal{K}_B)$, $(\mathcal{K}_A, \mathcal{K}_C)$, $(\mathcal{K}_A, \mathcal{K}_{14})$, $(\mathcal{K}_B, \mathcal{K}_C)$ of Fig. 19–1b show that adjacent K-sets need not be of the same size and need not be aligned. The criterion of importance is for them to have, at least partially, a boarder in common. Just to round off the picture, no two K-sets of the three shown in Fig. 19–2a are adjacent as they are not pairwise disjoint. Neither are any of the elementary K-sets of Fig. 19–2b adjacent as no two have a boarder in common.

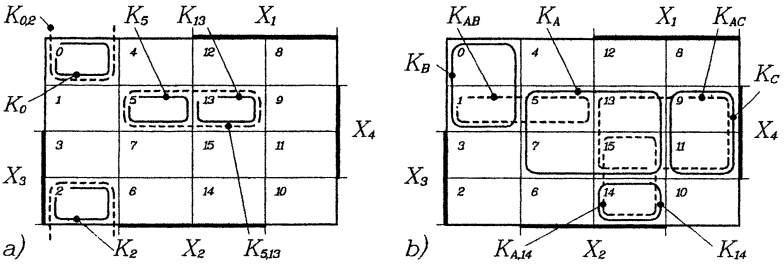


Fig. 19–3 Examples of consensuses

Now, to get a first impression of the consensus, take a look at Fig. 19–3 which shows all but one of the consensuses for the sets of Fig. 19–1. You might want to see if you can find the general rule by which to find or define consensuses — if so interrupt your reading here ...

(19.1) The **consensus** of two given *adjacent* K-sets is the largest K-set that can be constructed exclusively of elements of both the given adjacent K-sets.

By this definition, you will notice that $\mathcal{K}_{0,2}$ is the consensus of $(\mathcal{K}_0, \mathcal{K}_2)$ while $\mathcal{K}_{5,13}$ is the consensus of $(\mathcal{K}_5, \mathcal{K}_{13})$ as shown in Fig. 19–3a. The one K-set missing in Fig. 19–3b, the K-set $\mathcal{K}_{B,C} = \{1, 9\}$ of $(\mathcal{K}_B, \mathcal{K}_C)$, was only left away so as not to overload the picture. In contrast to the consensus

examples of Fig. 19–3, Fig. 19–4 presents counter examples. $\mathcal{K}_U$ cannot be the consensus of $\mathcal{K}_D$ and $\mathcal{K}_F$, for these K-sets — not being disjoint — do not have a consensus. $\mathcal{K}_9$ and $\mathcal{K}_{12}$, on the other hand, *are* disjoint. But, not being adjacent, they also have no consensus, so that $\mathcal{K}_V$ cannot be their consensus. Another argument can also be put forward: $\mathcal{K}_V$ cannot be the consensus of $\mathcal{K}_9$ and $\mathcal{K}_{12}$ as $\mathcal{K}_V$ does not consist *solely* of elements of $\mathcal{K}_9$ and $\mathcal{K}_{12}$.

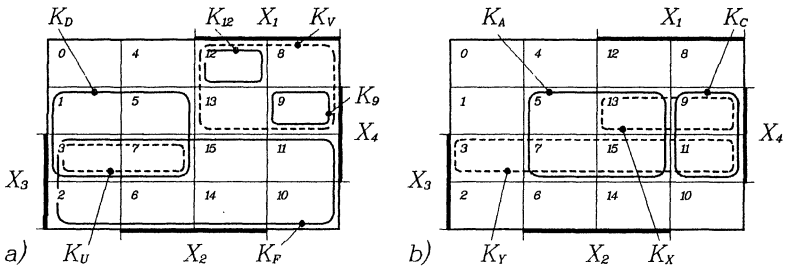


Fig. 19–4 K-sets that are not consensuses

But, why is neither $\mathcal{K}_X$ nor $\mathcal{K}_Y$ of Fig. 19–4b a consensus? $\mathcal{K}_X$ is not the *largest* K-set whose elements are taken exclusively from $\mathcal{K}_A$ and $\mathcal{K}_C$ — whereas $\mathcal{K}_{AC}$ (as shown in Fig. 19–3b) is. We can state this more simply by saying: As $\mathcal{K}_X$ is a subset of $\mathcal{K}_{AC}$, $\mathcal{K}_X$ cannot be the consensus. $\mathcal{K}_Y$, on the other hand, is too big: It contains an element that belongs neither to $\mathcal{K}_A$ nor to $\mathcal{K}_C$.

19.2 Formalising Adjacency and Consensus

All the information on a generalised minterm $C_{\varphi_1 \ldots \varphi_n}$ and on a K-set $\mathcal{K}_{\varphi_1 \ldots \varphi_n}$ is contained in their index n-tuple, the multiple input event $(\varphi_1, \ldots, \varphi_n)$ in which each φ_i stands for one of the symbols 0,1,x_i. Therefore, it suffices to work with the index n-tuples $(\varphi_1, \ldots, \varphi_n)$ of the generalised minterms or the K-sets instead of with the generalised minterms or K-sets themselves. And this we shall subsequently do when formalising the concepts of adjacency and consensus. But first an

> **Auxiliary theorem on disjoint K-sets**: two K-sets $\mathcal{K}_{\varphi_1 \ldots \varphi_n}$ and $\mathcal{K}_{\psi_1 \ldots \psi_n}$ are disjoint if the index n-tuples $(\varphi_1, \ldots, \varphi_n)$ and $(\psi_1, \ldots, \psi_n)$ consist of complementary *constants* in at least one position.

You will remember that two sets are disjoint if their intersection is empty, and that this is the case *iff* the conjunction of the defining properties of

the sets is contradictory. We therefore start the proof of the above auxiliary theorem by writing the intersection of two K-sets.

$$
\begin{aligned}
&(x_1,\ldots,x_n) \in (\mathcal{K}_{\varphi_1\ldots\varphi_n} \cap \mathcal{K}_{\psi_1\ldots\psi_n}) \overset{(6.6)}{\Longleftrightarrow} \\
&\quad \big((x_1,\ldots,x_n) \in \mathcal{K}_{\varphi_1\ldots\varphi_n}\big) \wedge \big((x_1,\ldots,x_n) \in \mathcal{K}_{\psi_1\ldots\psi_n}\big) \overset{(17.9)}{\Longleftrightarrow} \\
&\quad C_{\varphi_1\ldots\varphi_n} \wedge C_{\psi_1\ldots\psi_n} \overset{(17.8)}{\Longleftrightarrow} \\
&\quad \Big(\bigwedge_{i=1}^{n}(x_i = \varphi_i)\Big) \wedge \Big(\bigwedge_{i=1}^{n}(x_i = \psi_i)\Big) \overset{(13.49)}{\Longleftrightarrow} \\
&\quad \bigwedge_{i=1}^{n}(x_i = \varphi_i) \wedge (x_i = \psi_i).
\end{aligned}
$$

Thus, the intersection $\mathcal{K}_{\varphi_1\ldots\varphi_n} \cap \mathcal{K}_{\psi_1\ldots\psi_n}$ of two K-sets $\mathcal{K}_{\varphi_1\ldots\varphi_n}$ and $\mathcal{K}_{\psi_1\ldots\psi_n}$ is empty *iff*, for at least one i, $(x_i = \varphi_i) \wedge (x_i = \psi_i) \Leftrightarrow \mathbf{0}$, i.e., if $(x_i = \varphi_i) \wedge (x_i = \psi_i)$ is a *contradiction*, i.e., if $(x_i = \varphi_i) \wedge (x_i = \psi_i)$ is false for *all* values of x_i. This is the case *iff*, for at least one i

$$
\begin{aligned}
&\varphi_i \hookleftarrow 0 \quad \text{and} \quad \psi_i \hookleftarrow 1, \qquad \text{or} \\
&\varphi_i \hookleftarrow 1 \quad \text{and} \quad \psi_i \hookleftarrow 0.
\end{aligned}
$$

Note that if one of the technical variables φ_i or ψ_i is x_i, i.e., $\varphi_i \hookleftarrow x_i$ or $\psi_i \hookleftarrow x_i$, the expression $(x_i = \varphi_i) \wedge (x_i = \psi_i)$ cannot be a contradiction, this finalising the proof of the above auxiliary theorem on disjoint K-sets.

Having proved the above theorem, no further proof is needed for its converse stated here:

> **Auxiliary theorem on non-disjoint K-sets**: K-sets $\mathcal{K}_{\varphi_1\ldots\varphi_n}$ and $\mathcal{K}_{\psi_1\ldots\psi_n}$ have one or more elements in common if there is no position $i \in \{1,\ldots,n\}$ in which the index n-tuples $(\varphi_1,\ldots,\varphi_n)$ and $(\psi_1,\ldots,\psi_n)$ consist of complementary constants.

Index n-tuples $(\varphi_1,\ldots,\varphi_n)$ are frequently categorised by the number of 1s they contain. As each φ_i stands for one of the symbols of $\{0,1,x_i\}$, we slightly extend the original definition of the Hamming weight h_w of a binary vector, one that consists solely of the symbols 0 and 1:

> (19.2) The **Hamming weight** h_w of $(\varphi_1,\ldots,\varphi_n)$, in which each φ_i stands for one of the symbols of $\{0,1,x_i\}$, is the number of 1s in the index n-tuple $(\varphi_1,\ldots,\varphi_n)$.

For example, the Hamming weight of $(0,1,1)$ is 2, while that of $(x_1,1,0,x_4,0)$ is 1. We next introduce an *adjacency vector* $(\alpha_1,\ldots,\alpha_n)$ as a n-tuple of binary, numeric values $\alpha_i \in \{0,1\}$.

> **(19.3)** For two distinct index n-tuples $(\varphi_1,\ldots,\varphi_n)$, in which each $\varphi_i \in \{0,1,x_i\}$, and $(\psi_1,\ldots,\psi_n)$, in which each $\psi_i \in \{0,1,x_i\}$, the **adjacency vector** $(\alpha_1,\ldots,\alpha_n)$ is obtained as follows:

a) If either φ_i or ψ_i, or both, are x_i then α_i is 0 (i.e., $\alpha_i \hookleftarrow 0$).
b) If φ_i and ψ_i are constants, 0 or 1, then $\alpha_i \hookleftarrow \varphi_i \oplus \psi_i$.

The above auxiliary theorems can now be expressed more constructively when using the concept of the adjacency vector.

(19.4) **Theorem on non-disjoint K-sets:**
Two K-sets $\mathcal{K}_{\varphi_1 \ldots \varphi_n}$ and $\mathcal{K}_{\psi_1 \ldots \psi_n}$ are non-disjoint (i.e., have one or more elements in common) *iff* the adjacency vector $(\alpha_1, \ldots, \alpha_n)$ of their respective index n-tuples $(\varphi_1, \ldots, \varphi_n)$ and $(\psi_1, \ldots, \psi_n)$ contains no 1s (i.e., has a h_w of 0).

As an example for this theorem, consider the tables of Fig. 19–5 showing the index n-tuples of the K-sets $\mathcal{K}_D$, $\mathcal{K}_E$, $\mathcal{K}_F$. These K-sets are depicted in Fig. 19–2a.

$\mathcal{K}_D$: 0 x_2 x_3 1	$\mathcal{K}_D$: 0 x_2 x_3 1	$\mathcal{K}_E$: x_1 1 1 x_4
$\mathcal{K}_E$: x_1 1 1 x_4	$\mathcal{K}_F$: x_1 x_2 1 x_4	$\mathcal{K}_F$: x_1 x_2 1 x_4
(19.3): 0 0 0 0	(19.3): 0 0 0 0	(19.3): 0 0 0 0
a)	b)	c) $\mathcal{K}_E \subset \mathcal{K}_F$

Fig. 19–5 Adjacency vector of non-disjoint K-sets

A special case of non-disjoint K-sets exists when one K-set is a *subset* of another. The following theorem allows you to recognise subsets.

(19.5) A K-set $\mathcal{K}_{\varphi_1 \ldots \varphi_n}$ is a proper **subset** of a K-set $\mathcal{K}_{\psi_1 \ldots \psi_n}$ — $\mathcal{K}_{\varphi_1 \ldots \varphi_n} \subset \mathcal{K}_{\psi_1 \ldots \psi_n}$ — if *all* those φ_i that *differ* from ψ_i are constants, 0 or 1, while the ψ_i are x_i.

By this theorem, you can see that $\mathcal{K}_E \subset \mathcal{K}_F$, Fig. 19–5c: The only position where the index n-tuples of $\mathcal{K}_E$ and $\mathcal{K}_F$ differ is in $i = 2$, and here φ_2 is 1, while ψ_2 is x_2.

(19.6) Theorem on disjoint K-sets: Two K-sets $\mathcal{K}_{\varphi_1 \ldots \varphi_n}$ and $\mathcal{K}_{\psi_1 \ldots \psi_n}$ are disjoint *iff* the adjacency vector $(\alpha_1, \ldots, \alpha_n)$ of the index n-tuples $(\varphi_1, \ldots, \varphi_n)$ and $(\psi_1, \ldots, \psi_n)$ contains one or more 1s (i.e., has a Hamming weight ≥ 1).

Examples of the adjacency vector of disjoint K-sets are shown in the tables of Fig. 19–6, in which the index n-tuples represent K-sets $\mathcal{K}_0$, $\mathcal{K}_2$, $\mathcal{K}_{14}$, $\mathcal{K}_A$, $\mathcal{K}_C$ depicted in Fig. 19–3.

$\mathcal{K}_0$: 0 0 0 0	$\mathcal{K}_{14}$: 1 1 1 0	$\mathcal{K}_A$: x_1 1 x_3 1
$\mathcal{K}_2$: 0 0 1 0	$\mathcal{K}_C$: 1 0 x_3 1	$\mathcal{K}_C$: 1 0 x_3 1
(19.3): 0 0 1 0	(19.3): 0 1 0 1	(19.3): 0 1 0 0
a) adjacent	b) non-adjacent	c) adjacent

Fig. 19–6 Adjacency vectors of disjoint K-sets

Adjacent K-sets are a special case of disjoint K-sets. Their formal definition can be stated as:

(19.7) Two K-sets, $\mathcal{K}_{\varphi_1 \ldots \varphi_n}$ and $\mathcal{K}_{\psi_1 \ldots \psi_n}$, are **adjacent** if the Hamming weight h_w of the adjacency vector $(\alpha_1, \ldots, \alpha_n)$ of their index n-tuples $(\varphi_1, \ldots, \varphi_n)$ and $(\psi_1, \ldots, \psi_n)$ is 1.

You might like to take another look at Fig. 19–6 to note which pairs of K-sets are adjacent, and which are not, and to compare this result with their graphical representation in the K-maps of Fig. 19–3. As already mentioned, for those K-sets that are adjacent, we shall want to determine their consensus. To determine the consensus $\mathcal{K}_{\kappa_1 \ldots \kappa_n}$ of two given *adjacent* K-sets $\mathcal{K}_{\varphi_1 \ldots \varphi_n}$ and $\mathcal{K}_{\psi_1 \ldots \psi_n}$, you employ the constructive definition of consensus:

(19.8) The index n-tuple $(\kappa_1, \kappa_2, \ldots, \kappa_n)$ of the **consensus** $\mathcal{K}_{\kappa_1 \ldots \kappa_n}$ of the *adjacent* K-sets $\mathcal{K}_{\varphi_1 \ldots \varphi_n}$ and $\mathcal{K}_{\psi_1 \ldots \psi_n}$ is obtained from their index n-tuples $(\varphi_1, \ldots, \varphi_n)$ and $(\psi_1, \ldots, \psi_n)$ in the following way:

a) κ_i is assigned the symbol 0 if both φ_i and ψ_i are assigned 0, or one is 0 and the other is the variable symbol x_i.

b) κ_i is assigned the symbol 1 if both φ_i and ψ_i are assigned 1, or one is 1 and the other is the variable symbol x_i.

c) κ_i is assigned the variable symbol x_i if φ_i and ψ_i are symbols for complementary constants, or are both x_i.

$\mathcal{K}_0$: 0 0 0 0	$\mathcal{K}_A$: x_1 1 x_3 1	$\mathcal{K}_A$: x_1 1 x_3 1
$\mathcal{K}_2$: 0 0 1 0	$\mathcal{K}_B$: 0 0 0 x_4	$\mathcal{K}_C$: 1 0 x_3 1
$\mathcal{K}_{0,2}$: 0 0 x_3 0	$\mathcal{K}_{A,B}$: 0 x_2 0 1	$\mathcal{K}_{A,C}$: 1 x_2 x_3 1
a)	b)	c)

Fig. 19–7 Applying (19.8) to determine the consensus

Let me repeat that the above constructive definition of a consensus — (19.8) — is only applicable to two *adjacent* K-sets. If you (wrongly) do apply the above definition to non-adjacent K-sets, as for the examples of Fig. 19–4a, you get the incorrect results $\mathcal{K}_U$ and $\mathcal{K}_V$ shown in that figure. The application of the above definition (19.8) of consensus is demonstrated in the examples of Fig. 19–7 taken from Fig. 19–3.

19.3 Finding the Full Cover*

The first step to take in minimising the logic expression of a given subset $\mathcal{E}_S$ of the universe $\mathcal{E}$ of input events $(x_1, \ldots, x_n)$ is to find the *full cover* of $\mathcal{E}_S$ which we denote as $\mathcal{P}_{max}(\mathcal{E}_S)$.

You can specify the subset $\mathcal{E}_S$ in any way you find appropriate. For the example of this section we take $\mathcal{E}_S$ to be the 1-equivalent $\mathcal{E}_1$ specified in Fig. 19–8. To show that you need not start minimisation with a list of input events, assume that we have pre-collected the input events of $\mathcal{E}_1$ into K-sets

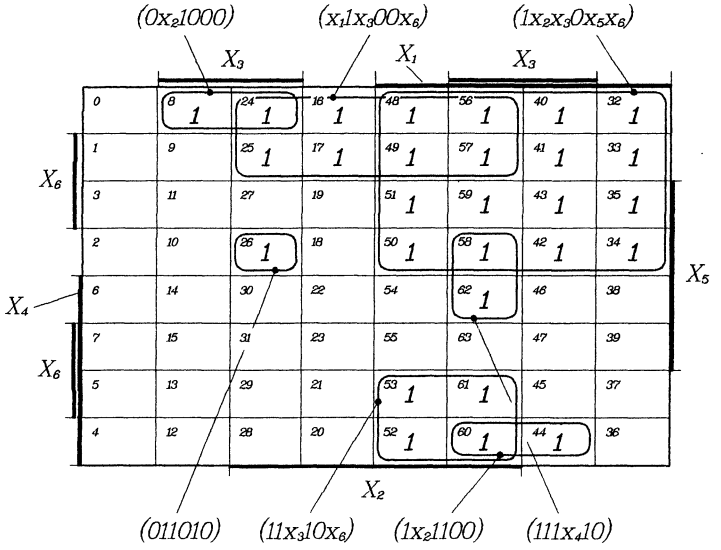


Fig. 19–8 Reworking the example of Fig. 16–9.

$\mathcal{K}_{011010}$, $\mathcal{K}_{0x1000}$, $\mathcal{K}_{1x1100}$, $\mathcal{K}_{111x10}$, $\mathcal{K}_{11x10x}$, $\mathcal{K}_{x1x00x}$, $\mathcal{K}_{1xx0xx}$ as pictured in Fig. 19–8. Let me put forth the following minimisation method.

(19.9) Tabular iterative consensus algorithm for finding the full cover $\mathcal{P}_{max}(\mathcal{E}_S)$ of a given subset $\mathcal{E}_S$.

(a) Start by listing the index n-tuples $(\varphi_1, \cdots, \varphi_n)$ of K-sets that cover $\mathcal{E}_S$ (e.g., list 1 of Fig. 19–9). The duplicates of index n-tuples in this list are dropped, as are all the index n-tuples which, according to (19.5), specify K-sets that are subsets of other K-sets of the list. Call the resulting list a *Basic List* (again, list 1 of Fig. 19–9).

(b) Each of the index n-tuples of the *Basic List* is checked against the others for adjacency according to (19.3) and (19.7). For each pair of index n-tuples that are adjacent you establish their consensus according to (19.8) appending it to the *Basic List* (see list 2 of Fig. 19–9).

(c) The duplicates of index n-tuples in the last list are dropped, as are all the index n-tuples which, according to (19.5), specify K-sets that are subsets of other K-sets of the list. This provides you with a new *Basic List* (e.g., list 3 of Fig. 19–9).

(d) Repeat steps (b) and (c) until two successive *Basic Lists* are identical in which case the algorithm terminates.

List 1 of the table of Fig. 19–10 is the original basic list of our example, obtained by step (a), while lists 2 to 4 are the successive basic lists each ob-

	List 1
1	011010
2	0x1000
3	1x1100
4	111x10
5	11x10x
6	x1x00x
7	1xx0xx

	List 2
1	011010
2	0x1000
3	1x1100
4	111x10
5	11x10x
6	x1x00x
7	1xx0xx
1-2	0110x0
1-4	x11010
1-6	0110x0
1-7	x11010
2-7	xx1000
3-4	1111x0
3-6	111x00
3-7	1x1x00
4-5	1111x0
4-6	1110x0
5-6	11xx0x
5-7	11xx0x

List 3
111x10
x1x00x
1xx0xx
0110x0
x11010
xx1000
1111x0
1x1x00
11xx0x

Fig. 19–9 Illustrating step (a) and the iterative steps (b) and (c) of (19.9)

tained by the iterative steps (b) and (c). This example represents one of those relatively rare cases where the result is a single irredundant cover, meaning that $\mathcal{P}_{max}(\mathcal{E}_1) = \mathcal{P}_{min}(\mathcal{E}_1)$. It has already been visualised in the K-map of Fig. 16–9. However, in general, the resulting list contains an abundance of prime sets allowing us to choose from them a number of different covers. Of these the *minimal covers* are of prime interest with respect to minimisation.

List 1	List 2	List 3	List 4	$C_{\varphi_1 \ldots \varphi_6}$
011010	111x10	x1x00x	x1x00x	$X_2\overline{X}_4\overline{X}_5$
0x1000	x1x00x	1xx0xx	1xx0xx	$X_1\overline{X}_4$
1x1100	1xx0xx	xx1000	xx1000	$X_3\overline{X}_4\overline{X}_5\overline{X}_6$
111x10	0110x0	1x1x00	1x1x00	$X_1X_3\overline{X}_5\overline{X}_6$
11x10x	x11010	11xx0x	11xx0x	$X_1X_2\overline{X}_5$
x1x00x	xx1000	111xx0	111xx0	$X_1X_2X_3\overline{X}_6$
1xx0xx	1111x0	x110x0	x110x0	$X_2X_3\overline{X}_4\overline{X}_6$
	1x1x00			
	11xx0x			

Fig. 19–10 Results of the iteration

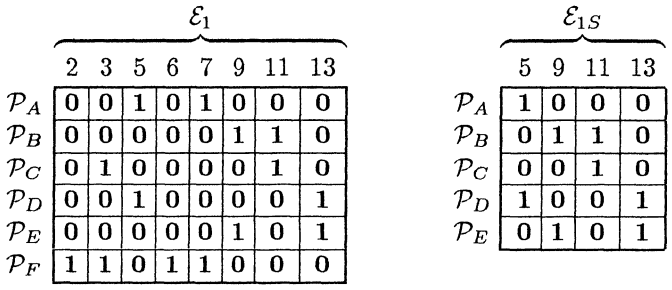
19.4 Finding Minimal Covers *

The procedure presented here allows you to directly find a minimal cover $\mathcal{P}_{min}(\mathcal{E}_S) \subseteq \mathcal{P}_{max}(\mathcal{E}_S)$ among all the prime sets. In fact, the first cover found is a minimal cover, allowing you to terminate the search if you wish. This method assumes you have determined the full cover $\mathcal{P}_{max}(\mathcal{E}_S)$ by some means, possibly by the *tabular iterative consensus algorithm* (19.9) of the previous section.

Let me use an example to show how the method works. Assume a switching function of four input variables is specified by its 1-equivalent $\mathcal{E}_1 = \{2,3,5,6,7,9,11,13\}$, and further assume you have determined all its prime implicants — in this simple case, possibly by drawing the K-map (which you are invited to do):

$$\mathcal{P}_A = \{5,7\} \qquad \mathcal{P}_B = \{9,11\} \qquad \mathcal{P}_C = \{3,11\}$$
$$\mathcal{P}_D = \{5,13\} \qquad \mathcal{P}_E = \{9,13\} \qquad \mathcal{P}_F = \{2,3,6,7\}$$

You start the procedure by writing the **table of prime sets**, as shown in Fig. 19–11a (the equivalent to **McCluskey**'s [1956] *prime implicant table*). Each row represents a prime set $\mathcal{P}_\varphi$, while each column stands for an input event e of the 1-equivalent $\mathcal{E}_1$. The intersection of a column e and a row $\mathcal{P}_\varphi$ contains the truth value (**0** or **1**) of the proposition $e \in \mathcal{P}_\varphi$.



a) Original table b) Reduced table

Fig. 19–11 Tables of prime sets

A table of prime sets can be simplified if it contains one or more *essential prime sets*. An *essential prime set*, according to (18.1), is a prime set that contains at least one input event e — call it a *unique input event* — not contained in any other prime set. The column of a unique input event contains only a single **1**, the row in which it occurs marks the essential prime set. An essential prime set is an element of every irredundant cover. Having found the essential prime sets, we need not search for them any further, and may thus remove them from the table of prime sets. Removing all essential prime sets from the table allows us to also remove the columns of all the input

events that are elements of the essential prime sets. Our table of prime sets contains $\mathcal{P}_F$ as the only essential prime set. Removing it and the columns of all input events it contains leads to Fig. 19–11b.

The last step is to develop the **table of possible covers** from the reduced table of prime sets. The basic idea is to test for each union ($\cup$) of combinations of prime sets, which such union contains all input events e. Fig. 19–12 details how such a test is organised. The combinations in the column 'Possible covers of $\mathcal{E}_{1S}$' are obtained according to the algorithm of Section 15.4. For convenience the group numbers g, referred to in the algorithm, are stated in the first column of Fig. 19–12. The row numbers are only used for easier reference. The logic vectors in the $\mathcal{E}_{1S}$-column are obtained by *scalar disjunction* of those vectors indicated in the combinations. **Scalar disjunction** of logic vectors refers to the disjunction of equally indexed elements. E.g., the logic vector of row 6, $(\mathbf{1},\mathbf{1},\mathbf{1},\mathbf{0})$, is the scalar disjunction of the vectors of rows 1 (for $\mathcal{P}_A$) and 2 (for $\mathcal{P}_B$) — $(\mathbf{1},\mathbf{0},\mathbf{0},\mathbf{0}) \vee (\mathbf{0},\mathbf{1},\mathbf{1},\mathbf{0}) \Leftrightarrow (\mathbf{1},\mathbf{1},\mathbf{1},\mathbf{0})$. Results, once calculated, may be used again. For instance, interpreting $\mathcal{P}_A \cup \mathcal{P}_B \cup \mathcal{P}_E$ of row 18 as $(\mathcal{P}_A \cup \mathcal{P}_B) \cup \mathcal{P}_E$ allows us to calculate its logic vector from rows 6 (for $\mathcal{P}_A \cup \mathcal{P}_B$) and 5 (for $\mathcal{P}_E$) — $(\mathbf{1},\mathbf{1},\mathbf{1},\mathbf{0}) \vee (\mathbf{0},\mathbf{1},\mathbf{0},\mathbf{1}) \Leftrightarrow (\mathbf{1},\mathbf{1},\mathbf{1},\mathbf{1})$.

The table is developed iteratively, i.e., row after row, until the first logic vector containing only **1**s is encountered. The prime sets of the union are a partial minimal cover $\mathcal{P}_{min}(\mathcal{E}_S)$ — see row 11. If there are more minimal covers than one, they are all to be found in the same group. So, if you want to find all minimal covers, you let the algorithm continue till the end of the group. The complete result is obtained by augmenting the covers found here by the previously omitted essential covers — in our example by $\mathcal{P}_F$ — giving us $\mathcal{P}_{min}(\mathcal{E}_1) = \{\mathcal{P}_B, \mathcal{P}_D, \mathcal{P}_F\}$.

If you feel a desire to find *all irredundant covers*, continue with the algorithm as shown in Fig. 19–12, discarding those solutions subsets of which have already proved to be irredundant covers. Thus, row 17 is not accepted as a solution as $\{\mathcal{P}_B, \mathcal{P}_D\}$ of row 11 is a subset of $\{\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_D\}$ of row 17 — $\{\mathcal{P}_B, \mathcal{P}_D\} \subset \{\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_D\}$.

For the example $\mathcal{E}_1 = \{2,3,5,6,7,9,11,13\}$ of this section, we found the minimal cover to be $\mathcal{P}_{min}(\mathcal{E}_1) = \{\mathcal{P}_B, \mathcal{P}_D, \mathcal{P}_F\}$ which is more precisely expressed as $\mathcal{P}_{min}(\mathcal{E}_1) = \{\mathcal{P}_{10x1}, \mathcal{P}_{x101}, \mathcal{P}_{0x1x}\}$. Then, by (17.30) we may write the disjunctive form as

$$\begin{aligned} Y(x) &\Leftrightarrow \bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}_{min}(\mathcal{E}_1)} C_\varphi(x) \Leftrightarrow C_{10x1} \vee C_{x101} \vee C_{0x1x} \\ &\Leftrightarrow X_1\overline{X}_2X_4 \vee X_2\overline{X}_3X_4 \vee \overline{X}_1X_3. \end{aligned} \tag{19.10}$$

For the same example, knowing $\mathcal{E}_1$, we can write its complement $\mathcal{E}_0$ as $\mathcal{E}_0 = \{0,1,4,8,12,14,15\}$. From this, we can calculate the minimal cover $\mathcal{P}_{min}(\mathcal{E}_0) = \{\mathcal{P}_{000x}, \mathcal{P}_{111x}, \mathcal{P}_{xx00}\}$, which I encourage you to do. Then, (17.31)

enables us to write the conjunctive form of the given function:

$$Y(x) \Leftrightarrow \bigwedge_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}_{min}(\mathcal{E}_0)} D_\varphi(x) \Leftrightarrow \mathcal{D}_{000x}\mathcal{D}_{111x}\mathcal{D}_{xxoo}$$
$$\Leftrightarrow (X_1 \vee X_2 \vee X_3)(\overline{X}_1 \vee \overline{X}_2 \vee \overline{X}_3)(X_3 \vee X_4). \qquad (19.11)$$

Group	Row	Possible covers of $\mathcal{E}_{1S}$	$\mathcal{E}_{1S}$ 5	9	11	13	Comments
1	1	$\mathcal{P}_A$	1	0	0	0	
	2	$\mathcal{P}_B$	0	1	1	0	
	3	$\mathcal{P}_C$	0	0	1	0	
	4	$\mathcal{P}_D$	1	0	0	1	
	5	$\mathcal{P}_E$	0	1	0	1	
2	6	$\mathcal{P}_A \cup \mathcal{P}_B$	1	1	1	0	
	7	$\mathcal{P}_A \cup \mathcal{P}_C$	1	0	1	0	
	8	$\mathcal{P}_A \cup \mathcal{P}_D$	1	0	0	1	
	9	$\mathcal{P}_A \cup \mathcal{P}_E$	1	1	0	1	
	10	$\mathcal{P}_B \cup \mathcal{P}_C$	0	1	1	0	
	11	$\mathcal{P}_B \cup \mathcal{P}_D$	1	1	1	1	$\mathcal{P}_{min}(\mathcal{E}_{1S})$
	12	$\mathcal{P}_B \cup \mathcal{P}_E$	0	1	1	1	
	13	$\mathcal{P}_C \cup \mathcal{P}_D$	1	0	1	1	
	14	$\mathcal{P}_C \cup \mathcal{P}_E$	0	1	1	1	
	15	$\mathcal{P}_D \cup \mathcal{P}_E$	1	1	0	1	
3	16	$\mathcal{P}_A \cup \mathcal{P}_B \cup \mathcal{P}_C$	1	1	1	0	
	(17)	$\mathcal{P}_A \cup \mathcal{P}_B \cup \mathcal{P}_D$	1	1	1	1	$11 \subset 17$
	18	$\mathcal{P}_A \cup \mathcal{P}_B \cup \mathcal{P}_E$	1	1	1	1	$\mathcal{P}_{irr}^{(1)}(\mathcal{E}_{1S})$
	19	$\mathcal{P}_A \cup \mathcal{P}_C \cup \mathcal{P}_D$	1	0	1	1	
	20	$\mathcal{P}_A \cup \mathcal{P}_C \cup \mathcal{P}_E$	1	1	1	1	$\mathcal{P}_{irr}^{(2)}(\mathcal{E}_{1S})$
	21	$\mathcal{P}_A \cup \mathcal{P}_D \cup \mathcal{P}_E$	1	1	0	1	
	(22)	$\mathcal{P}_B \cup \mathcal{P}_C \cup \mathcal{P}_D$	1	1	1	1	$11 \subset 22$
	23	$\mathcal{P}_B \cup \mathcal{P}_C \cup \mathcal{P}_E$	0	1	1	1	
	(24)	$\mathcal{P}_B \cup \mathcal{P}_D \cup \mathcal{P}_E$	1	1	1	1	$11 \subset 24$
	25	$\mathcal{P}_C \cup \mathcal{P}_D \cup \mathcal{P}_E$	1	1	1	1	$\mathcal{P}_{irr}^{(3)}(\mathcal{E}_{1S})$
4	(26)	$\mathcal{P}_A \cup \mathcal{P}_B \cup \mathcal{P}_C \cup \mathcal{P}_D$	1	1	1	1	$11 \subset 26$
	(27)	$\mathcal{P}_A \cup \mathcal{P}_B \cup \mathcal{P}_C \cup \mathcal{P}_E$	1	1	1	1	$18, 20 \subset 27$
	(28)	$\mathcal{P}_A \cup \mathcal{P}_B \cup \mathcal{P}_D \cup \mathcal{P}_E$	1	1	1	1	$11 \subset 28$
	(29)	$\mathcal{P}_A \cup \mathcal{P}_C \cup \mathcal{P}_D \cup \mathcal{P}_E$	1	1	1	1	$20, 25 \subset 29$
	(30)	$\mathcal{P}_B \cup \mathcal{P}_C \cup \mathcal{P}_D \cup \mathcal{P}_E$	1	1	1	1	$11 \subset 30$
5	(31)	$\mathcal{P}_A \cup \mathcal{P}_B \cup \mathcal{P}_C \cup \mathcal{P}_D \cup \mathcal{P}_E$	1	1	1	1	$\mathcal{P}_{max}(\mathcal{E}_{1S})$

Fig. 19–12 Table of possible covers

19.5 Minimisation considering Don't Cares *

Let us rework the example documented in Fig. 18–4b, meaning, we want to evaluate the *conjunctive* forms of all minimal covers $\mathcal{P}_{min}(\mathcal{E}_{0\nu};\mathcal{E}_\delta)$ of the 0-nucleus $\mathcal{E}_{0\nu}$. A systematic solution is obtained by firstly employing the *tabular iterative consensus algorithm* (19.9) to find the full cover — as is shown in Fig. 19–13. Do note that this algorithm (19.9) yields a full cover $\mathcal{P}_{max}(\mathcal{E}_{0\nu};\mathcal{E}_\delta)$ that can contain redundant prime sets.

$\mathcal{E}_{0\nu} \cup \mathcal{E}_\delta$	List 1	List 2	List 3	List 4	$\mathcal{P}_{max}(\mathcal{E}_{0\nu};\mathcal{E}_\delta)$
0	0000	000x	00xx	x00x	$\mathcal{P}_{x00x} = \mathcal{P}_A$
1	0001	00x0	0x0x	x0x0	$\mathcal{P}_{x0x0} = \mathcal{P}_B$
3	0011	0x00	x00x	xx01	$\mathcal{P}_{xx01} = \mathcal{P}_C$
2	0010	x000	0xx0	xx10	$\mathcal{P}_{xx10} = \mathcal{P}_D$
4	0100	00x1	x0x0	x1x1	$\mathcal{P}_{x1x1} = \mathcal{P}_E$
5	0101	0x01	0xx1	x11x	$\mathcal{P}_{x11x} = \mathcal{P}_F$
7	0111	x001	xx01	0xxx	$\mathcal{P}_{0xxx} = \mathcal{P}_G$
6	0110	001x	0x1x		
13	1101	0x11	xx10		
15	1111	0x10	01xx		
14	1110	x010	x1x1		
8	1000	010x	x11x		
9	1001	01x0			
10	1010	01x1			
		x101			
		011x			
		x111			
		x110			
		11x1			
		1x01			
		111x			
		1x10			
		100x			
		10x0			

Fig. 19–13 Tabular iterative consensus method

For your convenience the formal results $\mathcal{P}_{\varphi_1 \ldots \varphi_4}$ of the last column of Fig. 19–13 are related to the intuitive ones — $\mathcal{P}_A$, $\mathcal{P}_B$, $\mathcal{P}_C$, $\mathcal{P}_D$, $\mathcal{P}_E$, $\mathcal{P}_F$, $\mathcal{P}_G$ — obtained from the K-map Fig. 18–4b.

Secondly, we use the method developed in the previous section to obtain a *minimal irredundant* cover $\mathcal{P}^{\bar{r}}_{min}(\mathcal{E}_{0\nu};\mathcal{E}_\delta)$, initially writing the *table of prime sets*. For our example, the *complete* table of prime sets is shown in Fig. 19–14. But, if you look back at the definition of an irredundant cover, (18.3b), you will note that the union of its prime sets need only equal — in our case —

$\mathcal{E}_{0\nu}$. The consequence is that we need only develop the table of prime sets for $\mathcal{E}_{0\nu}$, and not for $\mathcal{E}_{0\nu} \cup \mathcal{E}_\delta$. Thus, those columns marked $\mathcal{E}_\delta$ need not have been incorporated in Fig. 19–14.

	$\mathcal{E}_{0\nu}$								$\mathcal{E}_\delta$						
	3	4	5	6	7	8	9	10	0	1	2	13	14	15	
$\mathcal{P}_{x00x}$	0	0	0	0	0	1	1	0	1	1	0	0	0	0	$\mathcal{P}_A$
$\mathcal{P}_{x0x0}$	0	0	0	0	0	1	0	1	1	0	1	0	0	0	$\mathcal{P}_B$
$\mathcal{P}_{xx01}$	0	0	1	0	0	0	1	0	0	1	0	1	0	0	$\mathcal{P}_C$
$\mathcal{P}_{xx10}$	0	0	0	1	0	0	0	1	0	0	1	0	1	0	$\mathcal{P}_D$
$\mathcal{P}_{x1x1}$	0	0	1	0	1	0	0	0	0	0	0	1	0	1	$\mathcal{P}_E$
$\mathcal{P}_{x11x}$	0	0	0	1	1	0	0	0	0	0	0	0	1	1	$\mathcal{P}_F$
$\mathcal{P}_{0xxx}$	1	1	1	1	1	0	0	0	1	1	1	0	0	0	$\mathcal{P}_G$

Fig. 19–14 Complete table of prime sets of $\mathcal{P}_{max}(\mathcal{E}_{0\nu}; \mathcal{E}_\delta)$

Next, in this table of prime sets, Fig. 19–14, remember to look for *essential* prime sets, dropping them and all columns containing their input events. The only essential prime set is $\mathcal{P}_G$, its input events 3 and 4 not being elements of any of the other prime sets. By dropping the row for $\mathcal{P}_G$, and the columns for the input events 3,4,5,6,7, and not considering the columns for $\mathcal{E}_{0\nu}$, we obtain the reduced table of prime sets of Fig. 19–15a. There are two conspicuous rows in Fig. 19–15a — those for $\mathcal{P}_E$ and $\mathcal{P}_F$, both containing only **0**s. These rows represent *redundant* prime sets. Eliminating them leads to the reduced table of Fig. 19–15b.

	$\mathcal{E}_{0\nu S}$			
	8	9	10	
$\mathcal{P}_A$	1	1	0	
$\mathcal{P}_B$	1	0	1	
$\mathcal{P}_C$	0	1	0	
$\mathcal{P}_D$	0	0	1	
$\mathcal{P}_E$	0	0	0	*redundant*
$\mathcal{P}_F$	0	0	0	*redundant*

(a)

	$\mathcal{E}_{0\nu S}$		
	8	9	10
$\mathcal{P}_A$	1	1	0
$\mathcal{P}_B$	1	0	1
$\mathcal{P}_C$	0	1	0
$\mathcal{P}_D$	0	0	1

(b)

Fig. 19–15 Table of reduced prime sets

The table of possible covers, Fig. 19–16, is developed as explained in the previous section. As we are only interested in minimal covers, the first already occurring in *group 2*, we need develop the table for no further group. After augmenting the thus found results by the essential prime set $\mathcal{P}_G$ we, as expected, obtain the same complete result as from the 3 K-maps of Fig. 18–5.

Group	Row	Possible covers of $\mathcal{E}_{0\nu S}$	$\mathcal{E}_{0\nu S}$ 5	9	11	Comments
1	1	$\mathcal{P}_A$	**1**	**1**	**0**	
	2	$\mathcal{P}_B$	**1**	**0**	**1**	
	3	$\mathcal{P}_C$	**0**	**1**	**0**	
	4	$\mathcal{P}_D$	**0**	**0**	**1**	
2	**5**	$\mathcal{P}_A \cup \mathcal{P}_B$	**1**	**1**	**1**	$\mathcal{P}^{(1)}_{min}(\mathcal{E}_{0\nu S})$
	6	$\mathcal{P}_A \cup \mathcal{P}_C$	**1**	**1**	**0**	
	7	$\mathcal{P}_A \cup \mathcal{P}_D$	**1**	**1**	**1**	$\mathcal{P}^{(3)}_{min}(\mathcal{E}_{0\nu S})$
	8	$\mathcal{P}_B \cup \mathcal{P}_C$	**1**	**1**	**1**	$\mathcal{P}^{(2)}_{min}(\mathcal{E}_{0\nu S})$
	9	$\mathcal{P}_B \cup \mathcal{P}_D$	**1**	**0**	**1**	
	10	$\mathcal{P}_C \cup \mathcal{P}_D$	**0**	**1**	**1**	

Fig. 19–16 Table of possible covers

$$\mathcal{P}^{(a)}_{min}(\mathcal{E}_{0\nu};\mathcal{E}_\delta) = \{\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_G\} = \{\mathcal{P}_{x00x}, \mathcal{P}_{x0x0}, \mathcal{P}_{0xxx}\},$$
$$\mathcal{P}^{(b)}_{min}(\mathcal{E}_{0\nu};\mathcal{E}_\delta) = \{\mathcal{P}_B, \mathcal{P}_C, \mathcal{P}_G\} = \{\mathcal{P}_{x0x0}, \mathcal{P}_{xx01}, \mathcal{P}_{0xxx}\},$$
$$\mathcal{P}^{(c)}_{min}(\mathcal{E}_{0\nu};\mathcal{E}_\delta) = \{\mathcal{P}_A, \mathcal{P}_D, \mathcal{P}_G\} = \{\mathcal{P}_{x00x}, \mathcal{P}_{xx10}, \mathcal{P}_{0xxx}\}.$$

And once you have a cover, you can use one of the evaluation formulas (18.4) to (18.7), in our case it is (18.6):

$$Y_1^{(a)} \Leftrightarrow V^{(a)}_{0;\delta}(\mathbf{x}) \Leftrightarrow \bigwedge_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}^{(a)}_{min}(\mathcal{E}_{0\nu};\mathcal{E}_\delta)} D_\varphi \Leftrightarrow (X_2 \vee X_3)(X_2 \vee X_4)X_1,$$

$$Y_1^{(b)} \Leftrightarrow V^{(b)}_{0;\delta}(\mathbf{x}) \Leftrightarrow \bigwedge_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}^{(b)}_{min}(\mathcal{E}_{0\nu};\mathcal{E}_\delta)} D_\varphi \Leftrightarrow (X_2 \vee X_4)(X_3 \vee \overline{X}_4)X_1,$$

$$Y_1^{(c)} \Leftrightarrow V^{(c)}_{0;\delta}(\mathbf{x}) \Leftrightarrow \bigwedge_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}^{(c)}_{min}(\mathcal{E}_{0\nu};\mathcal{E}_\delta)} D_\varphi \Leftrightarrow (X_2 \vee X_3)(\overline{X}_3 \vee X_4)X_1.$$

Chapter 20

Reduced Karnaugh Maps

K-maps have proved very helpful when the number of variables being used does not exceed, say, six. The usefulness of the K-map can be greatly enhanced by transferring one or more of its variables from the perimeter to its cells thus reducing the dimension of the K-map. These **reduced K-maps**, as they have come to be called, were first described systematically by **Clare** [1973]. The idea that leads to reduced K-maps also leads to the concept of **K-maps within K-maps**. Reduced K-maps are also used to evaluate latches and sequential circuits as indicated in Fig. 1–8.

20.1 Reducing the Dimension of a K-map

The dimension of a K-map refers to the number of (input) variables arranged around its perimeter. It is always possible to reduce the dimension of a K-map by transferring one or more input variables from the perimeter of the map to its cells. The variables thus transferred are called **map-entered variables**. For a three-dimensional K-map the reduction process follows from a simple transformation, say, of the disjunctive normal form

$$Y \Leftrightarrow \overline{X}_1\overline{X}_2\overline{X}_3Y(0) \vee \overline{X}_1\overline{X}_2X_3Y(1) \vee \overline{X}_1X_2\overline{X}_3Y(2) \vee \overline{X}_1X_2X_3Y(3)$$
$$\vee X_1\overline{X}_2\overline{X}_3Y(4) \vee X_1\overline{X}_2X_3Y(5) \vee X_1X_2\overline{X}_3Y(6) \vee X_1X_2X_3Y(7).$$

To obtain X_3 as a map-entered variable we *extract* the variables X_1 and X_2 (in inverted or non-inverted form). A similar approach is taken to obtain X_2 and X_1 as map-entered variables. The above disjunctive normal form is thus transformed into the following three logic formulas:

$$Y \Leftrightarrow \overline{X}_1\overline{X}_2\big(\overline{X}_3Y(0) \vee X_3Y(1)\big) \vee \overline{X}_1X_2\big(\overline{X}_3Y(2) \vee X_3Y(3)\big)$$
$$\vee X_1\overline{X}_2\big(\overline{X}_3Y(4) \vee X_3Y(5)\big) \vee X_1X_2\big(\overline{X}_3Y(6) \vee X_3Y(7)\big), \qquad (20.1a)$$
$$Y \Leftrightarrow \overline{X}_1\overline{X}_3\big(\overline{X}_2Y(0) \vee X_2Y(2)\big) \vee \overline{X}_1X_3\big(\overline{X}_2Y(1) \vee X_2Y(3)\big)$$
$$\vee X_1\overline{X}_3\big(\overline{X}_2Y(4) \vee X_2Y(6)\big) \vee X_1X_3\big(\overline{X}_2Y(5) \vee X_2Y(7)\big), \qquad (20.1b)$$
$$Y \Leftrightarrow \overline{X}_2\overline{X}_3\big(\overline{X}_1Y(0) \vee X_1Y(4)\big) \vee \overline{X}_2X_3\big(\overline{X}_1Y(1) \vee X_1Y(5)\big)$$
$$\vee X_2\overline{X}_3\big(\overline{X}_1Y(2) \vee X_1Y(6)\big) \vee X_2X_3\big(\overline{X}_1Y(3) \vee X_1Y(7)\big). \qquad (20.1c)$$

The disjunctive normal form for 3 variables can be depicted in a three-dimensional standard K-map, Fig. 20–1a. The mutually equivalent equations (20.1a) to (20.1c) lead, respectively, to the two-dimensional **reduced K-maps** of Figs. 20–1b to 20–1d.

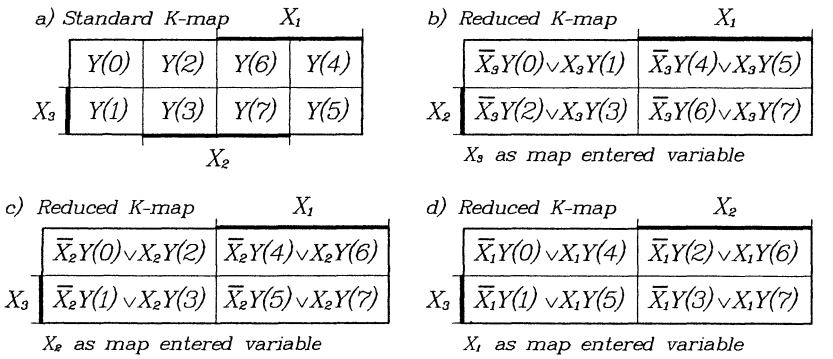


Fig. 20–1 Reducing a 3-dimensional to 2-dimensional K-maps

This decrementation of dimension is detailed in the example of Fig. 20–2 in which X_3 was arbitrarily chosen as map-entered variable.

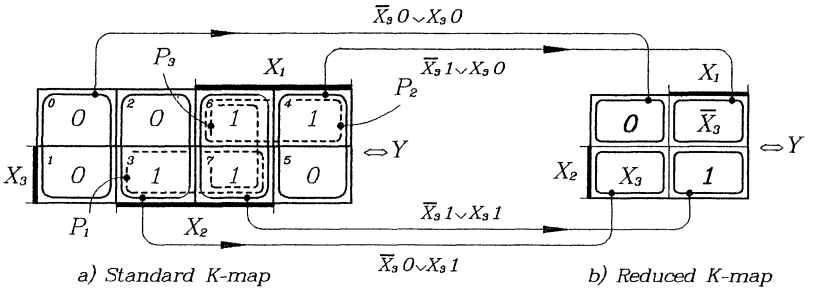


Fig. 20–2 Example showing how to develop a reduced K-map

Having seen how to *obtain* a reduced K-map, we now seek a way to *evaluate* it. Let us start with the standard K-map of Fig. 20–2a whose output variable Y, clearly, may be written as

$$Y \Leftrightarrow X_2X_3 \vee X_1\overline{X}_3 \vee X_1X_2 \Leftrightarrow X_2X_3 \vee X_1\overline{X}_3. \qquad (20.2)$$

Choosing each variable X_1, X_2, X_3 in turn as a map-entered variable leads to the reduced K-maps of Fig. 20–3.

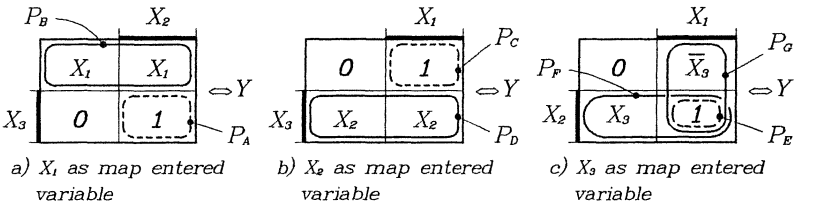


Fig. 20–3 Evaluating reduced K-maps

Stating it quite informally, you can **evaluate** these **reduced K-maps** by (a) describing the cells that have a **1** inscribed, and (b) then describing the cells that have a map-entered variable inscribed, taking any **1**-cells as *don't cares*, and ANDing with the map-entered variable. Let me illustrate this for

Fig. 20–3*a*: $Y \Leftrightarrow X_2X_3 \vee \overline{X}_3X_1$,

Fig. 20–3*b*: $Y \Leftrightarrow X_1\overline{X}_3 \vee X_3X_2$,

Fig. 20–3*c*: $Y \Leftrightarrow X_1X_2 \vee X_2X_3 \vee X_1\overline{X}_3 \Leftrightarrow X_2X_3 \vee X_1\overline{X}_3$.

Clearly, this can only be a first intuitive approach to the evaluation of reduced K-maps. But before looking into the evaluation problem more closely, let us consider further variants of reduced K-maps.

20.2 K-maps within K-maps *

Sometimes, you will find logic expressions consisting of two or more map-entered variables in the cells of reduced K-maps (e.g., Fig. 20–7e). It is convenient to explain the logic expression inscribed into a certain cell of a reduced K-map as the output of a, may I say, **sub-K-map**.

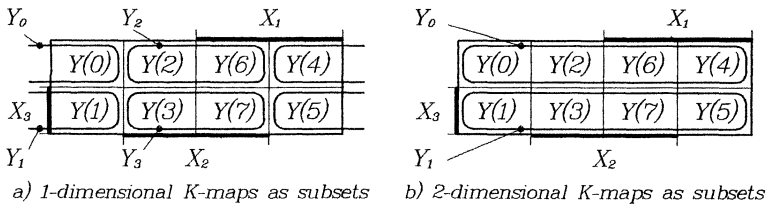


Fig. 20–4 Loops representing K-maps within K-maps

In fact, decrementing the dimensions of a standard K-map by two or more automatically leads to the concept of inscribing sub-K-maps into the cells of reduced K-maps. To illustrate, consider the general three-dimensional standard K-maps of Fig. 20–4. The loops Y_0, Y_1, Y_2, Y_3 in Fig. 20–4a can be

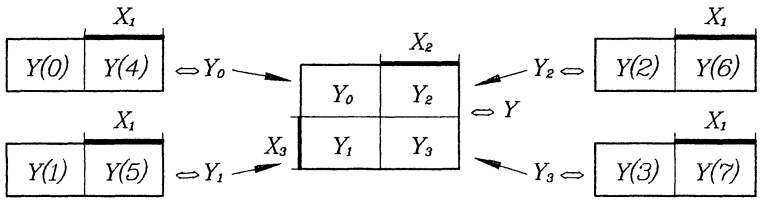


Fig. 20–5 Interpreting the map-entered variables $Y_0, \cdots, Y_3$ of the 2-dimentional K-map as outputs of 1D submaps

taken to represent one-dimensional K-maps as shown in Fig. 20–5. Extracting these one-dimensional K-maps from the three-dimensional map leaves us with the two-dimensional K-map of Fig. 20–5. This is analogous to the algebraic procedure of the previous section where the equation (20.1c) leads to the K-map representation of Fig. 20–1d.

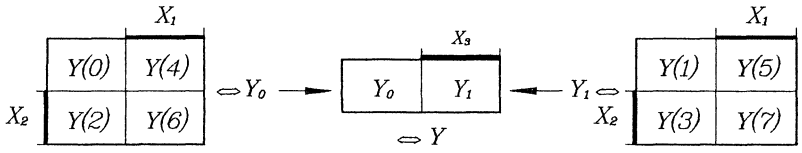


Fig. 20–6 Interpreting the map-entered variables Y_0, Y_1 of the one-dimentional K-map as outputs of 2D submaps

If desired, you can directly extract *two*-dimensional K-maps from a three-dimensional K-map, as the loops in Fig. 20–4b indicate. This leads to the one-dimensional K-map of Fig. 20–6 whose map-entered variables are the outputs Y_0, Y_1 calculated from the two-dimensional K-maps.

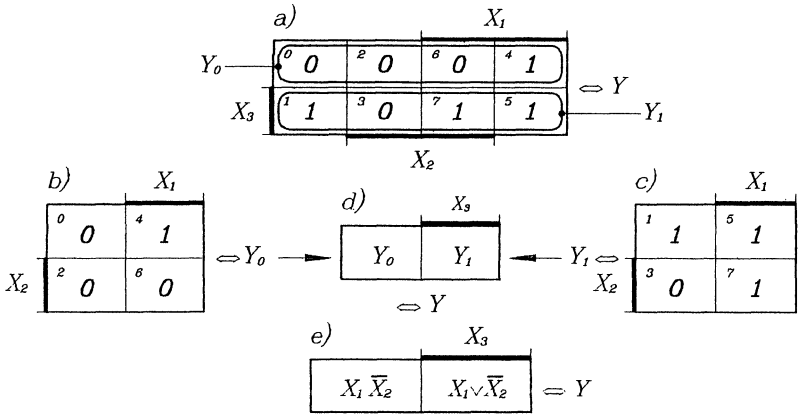


Fig. 20–7 An example showing when map-entered variables can be replaced by logic expressions

To illustrate this latter procedure, take a look at the example pictured in Fig. 20–7. The circuit specified in the three-dimensional K-map of Fig. 20–7a is, as the loops Y_0, Y_1 indicate, broken down into two 2-dimentional K-maps (Figs. 20–7b,c), whose outputs Y_0, Y_1 function as map-entered variables to the 1-dimentional K-map of Fig. 20–7d. A different and quite common way of expressing this, is to draw the 1-dimentional K-map as shown in Fig. 20–7e where the map-entered variables Y_0, Y_1 are replaced by the algebraic expressions obtained from the 2-dimentional K-maps.

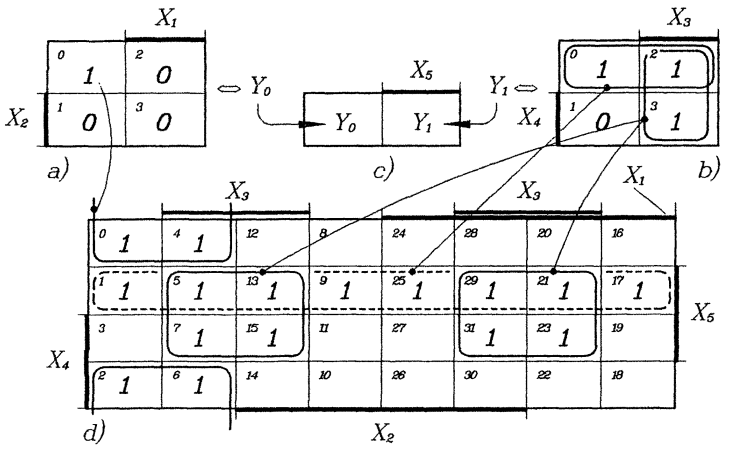


Fig. 20–8 An example of multivariable sub-K-maps

Sub-K-maps (as those of Fig. 20–7b,c) need not all have the same variables around their perimeter. Fig. 20–8 shows such an example, and also indicates how to develop the single 5-variable K-map equivalent to the K-maps of Figs. 20–8a,b,c.

20.3 Evaluating Reduced K-maps *

There is nothing mysterious about evaluating a reduced K-map. You apply one of the canonical normal forms (14.1) or (14.2) in the same way as for a standard K-map. Consider applying the *disjunctive* normal form to the reduced K-map of Fig. 20–9a.

$$Y \Leftrightarrow C_0 Y(0) \vee C_1 Y(1) \vee C_2 Y(2) \vee C_3 Y(3) \Leftrightarrow$$
$$C_0 \vee C_1 Y_1 \vee C_2 Y_2 \Leftrightarrow \overline{X}_1 \overline{X}_2 \vee \overline{X}_1 X_2 Y_1 \vee X_1 \overline{X}_2 Y_2.$$

You can obtain a minimised disjunctive form if you use **1**-cells as *dont't cares*, employing them as if they contained an output variable of your choice. The

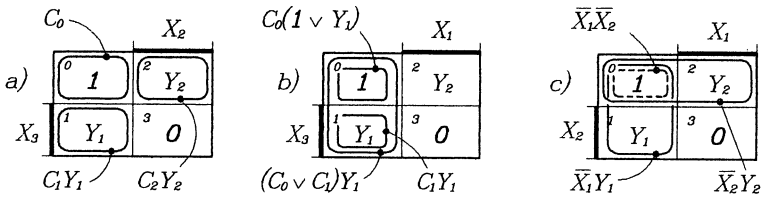


Fig. 20–9 Evaluating a reduced K-map

techneque is explained in Fig. 20–9b. If you want to combine the **1**-cell C_0 with the Y_1-cell C_1, you expand C_0 by $(\mathbf{1} \vee Y_1)$ as in

$$C_0 \Leftrightarrow C_0\mathbf{1} \Leftrightarrow C_0(\mathbf{1} \vee Y_1) \Leftrightarrow C_0 \vee C_0Y_1,$$

allowing you to combine C_0Y_1 with C_1Y_1:

$$C_0Y_1 \vee C_1Y_1 \Leftrightarrow (C_0 \vee C_1)Y_1 \Leftrightarrow \overline{X}_1Y_1.$$

Of course, you can reuse a **1**-cell as often as you like and for differing output variables Y_k. The complete calculation for the above example could look like this:

$$\begin{aligned} Y \Leftrightarrow\ & C_0 \vee C_1Y_1 \vee C_2Y_2 \Leftrightarrow C_0 \vee C_0 \vee C_1Y_1 \vee C_2Y_2 \Leftrightarrow \\ & C_0(\mathbf{1} \vee Y_1) \vee C_0(\mathbf{1} \vee Y_2) \vee C_1Y_1 \vee C_2Y_2 \Leftrightarrow \\ & C_0 \vee C_0Y_1 \vee C_0 \vee C_0Y_2 \vee C_1Y_1 \vee C_2Y_2 \Leftrightarrow \\ & C_0 \vee (C_0 \vee C_1)Y_1 \vee (C_0 \vee C_2)Y_2 \Leftrightarrow \\ & \overline{X}_1\overline{X}_2 \vee \overline{X}_1Y_1 \vee \overline{X}_2Y_2. \end{aligned}$$

The point is, the above minimised result can be read directly from the reduced K-map of Fig. 20–9c without any calculation needed.

The above procedure uses **1**-cells $x \in \mathcal{E}_1$ as *don't cares* in the same way as *unassigned input events* $x \in \mathcal{E}_\delta$ are used. Thus, to avoid any misunderstanding, it is advisable to reformulate the concept of cover stated in (17.21), noting that $\mathcal{E}_{a\nu}$ stands for either $\mathcal{E}_{0\nu}$ or $\mathcal{E}_{1\nu}$:

(20.3) A Y_k**-cover** $\mathcal{P}^{\overline{r}}(\mathcal{E}_{Y_k}; \mathcal{E}_{a\nu S} \cup \mathcal{E}_{\delta S})$, in which $\mathcal{E}_{a\nu S} \subseteq \mathcal{E}_{a\nu}$ and $\mathcal{E}_{\delta S} \subseteq \mathcal{E}_\delta$, is a set of non-redundant prime sets:

$$\begin{aligned} \mathcal{P}(\mathcal{E}_{Y_k}; \mathcal{E}_{a\nu S} \cup \mathcal{E}_{\delta S}) &= \{\mathcal{P}_{\varphi(1)}, \cdots, \mathcal{P}_{\varphi(L)}\}, \\ \mathcal{E}_{Y_k} \cup \mathcal{E}_{a\nu S} \cup \mathcal{E}_{\delta S} &= \mathcal{P}_{\varphi(1)} \cup \cdots \cup \mathcal{P}_{\varphi(L)}. \end{aligned}$$

(a) The **full cover** $\mathcal{P}^{\overline{r}}_{max}(\mathcal{E}_{Y_k}; \mathcal{E}_{a\nu S} \cup \mathcal{E}_{\delta S})$ is the set of *all* prime sets of $\mathcal{E}_{Y_k} \cup \mathcal{E}_{a\nu S} \cup \mathcal{E}_{\delta S}$ that are non-redundant.

(b) An **irredundant cover** $\mathcal{P}_{irr}^{\overline{r}}(\mathcal{E}_{Y_k}; \mathcal{E}_{a\nu S} \cup \mathcal{E}_{\delta S})$ of the Y_k equivalent $\mathcal{E}_{Y_k}$ is a set of non-redundant prime sets that are *necessary* in covering $\mathcal{E}_{Y_k}$ (i.e., removing any prime set from the irredundant cover leaves at least one input event $x \in \mathcal{E}_{Y_k}$ uncovered).

(c) A **minimal cover** $\mathcal{P}_{min}^{\overline{r}}(\mathcal{E}_{Y_k}; \mathcal{E}_{a\nu S} \cup \mathcal{E}_{\delta S})$ of $\mathcal{E}_{Y_k}$, the Y_k equivalent, is an irredundant cover $\mathcal{P}_{irr}^{\overline{r}}(\mathcal{E}_{Y_k}; \mathcal{E}_{a\nu S} \cup \mathcal{E}_{\delta S})$ with a smallest number of prime sets.

In the same vein, we introduce the *inclusions* $U_{Y_k;a\delta}(\mathbf{x})$ and *exclusions* $V_{Y_k;a\delta}(\mathbf{x})$ in accordance with (17.26) and (17.29):

$$U_{Y_k;0\delta}(\mathbf{x}) \Leftrightarrow \bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_{Y_k}; \mathcal{E}_{0\nu S} \cup \mathcal{E}_{\delta S})} C_\varphi(\mathbf{x}), \tag{20.4}$$

$$U_{Y_k;1\delta}(\mathbf{x}) \Leftrightarrow \bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_{Y_k}; \mathcal{E}_{1\nu S} \cup \mathcal{E}_{\delta S})} C_\varphi(\mathbf{x}), \tag{20.5}$$

$$V_{Y_k;0\delta}(\mathbf{x}) \Leftrightarrow \bigwedge_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_{Y_k}; \mathcal{E}_{0\nu S} \cup \mathcal{E}_{\delta S})} D_\varphi(\mathbf{x}), \tag{20.6}$$

$$V_{Y_k;1\delta}(\mathbf{x}) \Leftrightarrow \bigwedge_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_{Y_k}; \mathcal{E}_{1\nu S} \cup \mathcal{E}_{\delta S})} D_\varphi(\mathbf{x}). \tag{20.7}$$

We refer to $U_{Y_k;1\delta}(\mathbf{x})$ and $V_{Y_k;1\delta}(\mathbf{x})$ as a **1-inclusion** and **1-exclusion** respectively. Similarly, $U_{Y_k;0\delta}(\mathbf{x})$ and $V_{Y_k;0\delta}(\mathbf{x})$ are referred to as a **0-inclusion** and **0-exclusion** respectively. Clearly

$$U_{Y_k;1\delta}(\mathbf{x}) \Leftrightarrow \overline{V}_{Y_k;1\delta}(\mathbf{x}), \qquad U_{Y_k;0\delta}(\mathbf{x}) \Leftrightarrow \overline{V}_{Y_k;0\delta}(\mathbf{x}). \tag{20.8}$$

When considering a set $\mathcal{E}_{a\nu S} \cup \mathcal{E}_{\delta S}$, instead of $\mathcal{E}_{Y_k} \cup \mathcal{E}_{a\nu S} \cup \mathcal{E}_{\delta S}$, we write its inclusions as $U_0(\mathbf{x})$ and $U_1(\mathbf{x})$, and its exclusions as $V_0(\mathbf{x})$ and $V_1(\mathbf{x})$.

The *evaluation formulas* of Section 18.4 can now be expanded to take the Y_k-equivalents and the set $\mathcal{E}_\delta$ of unassigned input events into account. The inclusion evaluation-formulas (18.4) and (18.5) thus become the **inclusion evaluation-formulas for reduced K-maps**:

$$Y(\mathbf{x}) \Leftrightarrow U_{1;\delta}(\mathbf{x}) \vee \bigvee_{k=1}^{K} U_{Y_k;1\delta}(\mathbf{x}) Y_k, \tag{20.9}$$

$$\overline{Y}(\mathbf{x}) \Leftrightarrow U_{0;\delta}(\mathbf{x}) \vee \bigvee_{k=1}^{K} U_{Y_k;0\delta}(\mathbf{x}) \overline{Y}_k, \tag{20.10}$$

while the exclusion evaluation-formulas (18.6) and (18.7) become the **exclusion evaluation-formulas for reduced K-maps**:

$$Y(\mathbf{x}) \Leftrightarrow V_{0;\delta}(\mathbf{x}) \wedge \bigwedge_{k=1}^{K} \left(V_{Y_k;0\delta}(\mathbf{x}) \vee Y_k\right), \tag{20.11}$$

$$\overline{Y}(\mathbf{x}) \Leftrightarrow V_{1;\delta}(\mathbf{x}) \wedge \bigwedge_{k=1}^{K} \left(V_{Y_k;1\delta}(\mathbf{x}) \vee \overline{Y}_k\right). \tag{20.12}$$

Let us look at two examples to help get acquainted with the standard evaluation formulas. As a first example, consider calculating a *minimal disjunctive form* by employing (20.9) to the reduced K-map of Fig. 20–10a. As you can see, this K-map is reproduced three times, the only reason being that each K-map is used to picture the loops representing the **1**-inclusions $U_{Y_1;1\delta}$, $U_{Y_2;1\delta}$, $U_{\overline{Y}_2;1\delta}$ of the three map-entered variables Y_1, Y_2, $\overline{Y}_2$. Do note that $\overline{Y}_2$ is a map-entered variable in its own right.

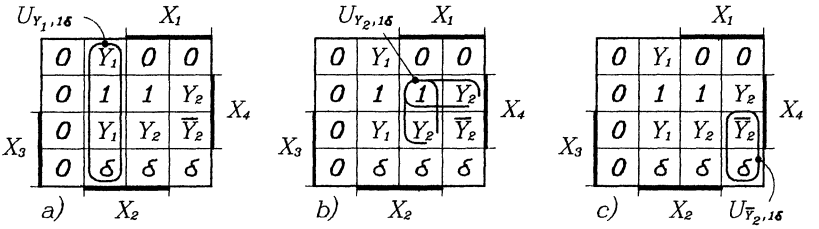


Fig. 20–10 Picturing minimal **1**-covers $U_{Y_1;1\delta}$, $U_{Y_2;1\delta}$, $U_{\overline{Y}_2;1\delta}$

In this example the **1**-inclusion $U_{1;\delta}$ is only represented by the area of **1**s as there are no adjacent *don't cares*. The loop in Fig. 20–10a, representing the **1**-inclusion $U_{Y_1;1\delta}$, must — according to (20.3b) — contain *all* Y_1-cells and can contain any of *don't cares* and **1**-cells. As Fig. 20–10b shows, a **1**-inclusion need not be represented by only one loop, in general it will be represented by the union of two or more loops, i.e., by the union of two or more prime sets. With these comments on Fig. 20–10 we can write the *minimal (quasi) disjunctive form* (20.9) as:

$$Y(\mathbf{x}) \Leftrightarrow \underbrace{X_2\overline{X}_3X_4}_{U_1} \vee \underbrace{\overline{X}_1X_2}_{U_{Y_1;1\delta}}Y_1 \vee (\underbrace{X_1X_2X_4 \vee X_1\overline{X}_3X_4}_{U_{Y_2;1\delta}})Y_2 \vee \underbrace{X_1\overline{X}_2X_3}_{U_{\overline{Y}_2;1\delta}}\overline{Y}_2.$$

To obtain a pure disjunctive form for our example, you need to distribute Y_2:

$$Y(\mathbf{x}) \Leftrightarrow X_2\overline{X}_3X_4 \vee \overline{X}_1X_2Y_1 \vee X_1X_2X_4Y_2 \vee X_1\overline{X}_3X_4Y_2 \vee X_1\overline{X}_2X_3\overline{Y}_2.$$

As our second example let us calculate a *minimal conjunctive form* by employing (20.11) to the previous reduced K-map(s) repeated in Fig. 20–11. The loops of these K-maps picture the **0**-exclusions $V_{Y_1;0\delta}$, $V_{Y_2;0\delta}$, $V_{\overline{Y}_2;0\delta}$ of the three map-entered variables Y_1, Y_2, $\overline{Y}_2$ ($V_{0;\delta}$ is not pictured).
Using these **0**-exclusions on (20.11) allows us to write:

$$Y(\mathbf{x}) \Leftrightarrow \underbrace{(X_1 \vee X_2)(\overline{X}_1 \vee X_4)}_{V_0}\underbrace{(X_4(X_1 \vee \overline{X}_3)}_{V_{Y_1;0\delta}} \vee Y_1)$$
$$\wedge \underbrace{((X_2 \vee X_3)(\overline{X}_1 \vee \overline{X}_2 \vee \overline{X}_3)}_{V_{Y_2;0\delta}} \vee Y_2)\underbrace{((X_2 \vee \overline{X}_3)}_{V_{\overline{Y}_2;0\delta}} \vee \overline{Y}_2).$$

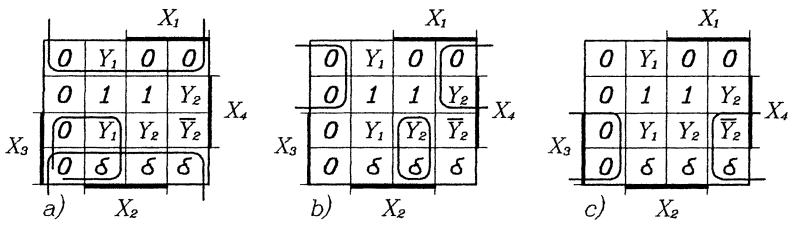


Fig. 20–11 Picturing minimal **0**-covers $V_{Y_1;0\delta}$, $V_{Y_2;0\delta}$, $V_{\overline{Y}_2;0\delta}$

Of course, if you set store in obtaining a pure conjunctive form, you must distribute Y_1 and Y_2:

$$Y(\mathbf{x}) \Leftrightarrow (X_1 \vee X_2)(\overline{X}_1 \vee X_4)(X_4 \vee Y_1)(X_1 \vee \overline{X}_3 \vee Y_1) \\ \wedge (X_2 \vee X_3 \vee Y_2)(\overline{X}_1 \vee \overline{X}_2 \vee \overline{X}_3 \vee Y_2)(X_2 \vee \overline{X}_3 \vee \overline{Y}_2).$$

20.4 Specification by Map-Entered Variables *

In this section, we follow-up the comparator example of Section 16.4 where the formula for $Z^{(4)}_{x>y}$ was expressed as a conjecture. We shan't attempt to develop an eight-dimensional K-map in which to compare the 4-digit binary numbers $x = x_1 \cdot 2^3 + x_2 \cdot 2^2 + x_3 \cdot 2^1 + x_4 \cdot 2^0$ and $y = y_1 \cdot 2^3 + y_2 \cdot 2^2 + y_3 \cdot 2^1 + y_4 \cdot 2^0$. Instead, you will see how to specify the problem in reduced K-maps.

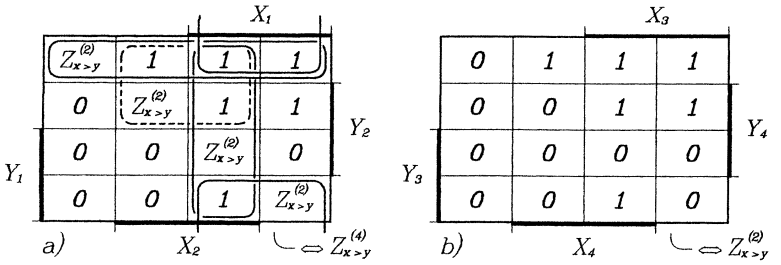


Fig. 20–12 Specifying a comparator for 4-digit binary numbers using map-entered variables

Remember that the **distinctness** of two numbers is decided in the position of highest weight in which the binary digits are unequal. Let us therefore first compare only the two highest digits x_1, x_2 and y_1, y_2. This comparison is carried out in the K-map of Fig. 20–12a. The **1**s state that $(x_1, x_2) > (y_1, y_2)$,

the **0**s refer to (x_1, x_2) being smaller than (y_1, y_2). But, why does the main diagonal of the K-map (Fig. 20–12a) have the variable $Z^{(2)}_{x>y}$ inscribed? You might want to find the answer yourself. If so, interrupt your reading here...

In the main diagonal of Fig. 20–12a you state how to decide which of the numbers (x_1, x_2, x_3, x_4) or (y_1, y_2, y_3, y_4) is greater, given that their two highest digits (x_1, x_2) and (y_1, y_2) are equal. This decision is taken on the bases of the result $Z^{(2)}_{x>y}$, which states which of the lower digits (x_3, x_4) and (y_3, y_4) is greater. Thus the value, **0** or **1**, entered into the main diagonal of Fig. 20–12a must be the value given $Z^{(2)}_{x>y}$, meaning we enter this variable into the main diagonal. $Z^{(2)}_{x>y}$ is specified in Fig. 20–12b (as was demonstrated in the explanation of Fig. 16–8). Evaluating the K-maps of Fig. 20–12 leads to a logic formula for $Z^{(4)}_{x>y}$.

$$Z^{(2)}_{x>y} \Leftrightarrow X_3\overline{Y}_3 \vee (X_3 \vee \overline{Y}_3)X_4\overline{Y}_4, \tag{20.13}$$

$$\begin{aligned} Z^{(4)}_{x>y} \Leftrightarrow & X_1\overline{Y}_1 \vee (X_1 \vee \overline{Y}_1)X_2\overline{Y}_2 \\ & \vee (X_1X_2 \vee \overline{Y}_1X_2 \vee X_1\overline{Y}_2 \vee \overline{Y}_1\overline{Y}_2)Z^{(2)}_{x>y}, \\ Z^{(4)}_{x>y} \Leftrightarrow & X_1\overline{Y}_1 \vee (X_1 \vee \overline{Y}_1)X_2\overline{Y}_2 \\ & \vee (X_1 \vee \overline{Y}_1)(X_2 \vee \overline{Y}_2)Z^{(2)}_{x>y}. \end{aligned} \tag{20.14}$$

Combining the logic formulas (20.14) and (20.13) leads to the circuit of Fig. 20–13. Note that the output $Z^{(2)}_{x>y}$ of the half-comparator is fed into the full-comparator. Quite clearly, this procedure can be extended by connecting the output $Z^{(4)}_{x>y}$ of the full comparator to a further full comparator.

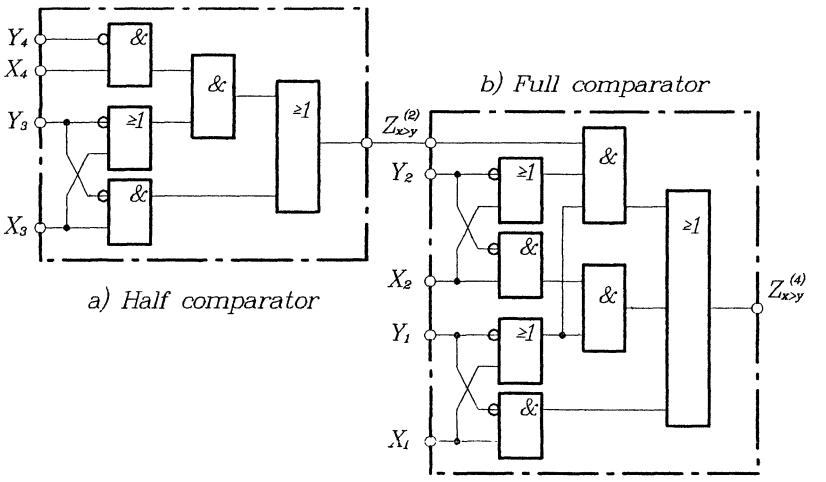


Fig. 20–13 The composite function derived from (20.13) and (20.14)

A different solution than the composite circuit of Fig. 20–13 is obtained when substituting $Z^{(2)}_{x>y}$ of (20.14) by (20.13) — we obtain the formula for $Z^{(4)}_{x>y}$ expressed as a conjecture at the end of Chapter 16 and pictured as a logic circuit in Fig. 16–10.

$$Z^{(4)}_{x>y} \Leftrightarrow X_1\overline{Y}_1 \vee (X_1 \vee \overline{Y}_1)X_2\overline{Y}_2 \vee (X_1 \vee \overline{Y}_1)(X_2 \vee \overline{Y}_2)(X_3\overline{Y}_3 \vee (X_3 \vee \overline{Y}_3)X_4\overline{Y}_4),$$

$$Z^{(4)}_{x>y} \Leftrightarrow X_1\overline{Y}_1 \vee (X_1 \vee \overline{Y}_1)X_2\overline{Y}_2 \vee (X_1 \vee \overline{Y}_1)(X_2 \vee \overline{Y}_2)X_3\overline{Y}_3 \vee (X_1 \vee \overline{Y}_1)(X_2 \vee \overline{Y}_2)(X_3 \vee \overline{Y}_3)X_4\overline{Y}_4.$$

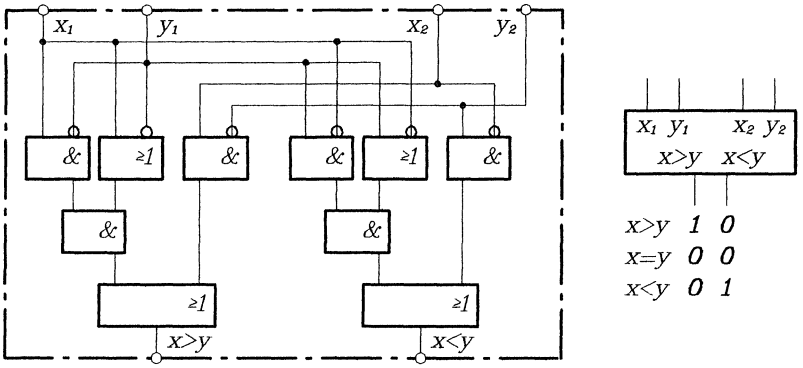


Fig. 20–14 Module for comparisons $x > y$ and $x < y$ of 2-bit integers

This design approach using the map-entered variables of Fig. 20–12 helps us understand how to cascade comparators. As the basic building block, consider the circuit of Fig. 20–14. Note that not both outputs can be **1** at the same time.

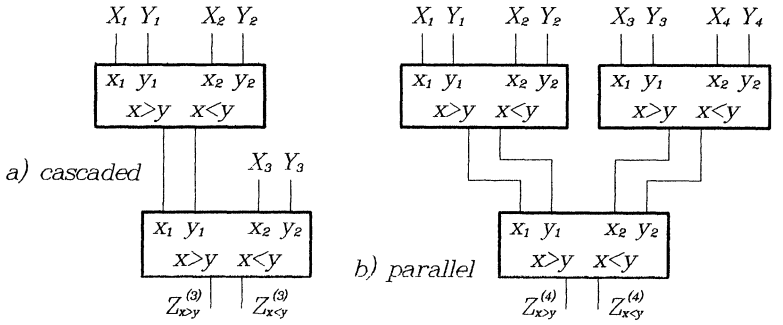


Fig. 20–15 Parallel and serial interconnection of comparators

The same reasoning that allows us to employ map-entered variables in the main diagonal of Fig. 20–12 validates the cascading of these circuits as shown on Fig. 20–15. The details of the explanation I leave to you.

Chapter 21

NOT-AND and NOT-OR

Transistor implementations of switching devices most commonly realise negations of multiple conjunctions, $\overline{X_1 \wedge \cdots \wedge X_n}$, or negations of multiple disjunctions, $\overline{X_1 \vee \cdots \vee X_n}$, reason enough to consider these functions in more detail. For more than two inputs ($n \geq 3$) I refer to them as NOT-AND and NOT-OR functions, respectively. For two inputs, and only for two inputs, are they called NAND and NOR.

But, why this distinction? The NAND and the NOR functions are defined for *two* variables only, allowing us to write $A \mathbin{\tilde{\wedge}} B \Leftrightarrow \overline{A \wedge B}$ and $A \mathbin{\tilde{\vee}} B \Leftrightarrow \overline{A \vee B}$. But, contrary to the dyadic expressions $A \mathbin{\tilde{\wedge}} B$ and $A \mathbin{\tilde{\vee}} B$, neither of the expressions $A \mathbin{\tilde{\wedge}} B \mathbin{\tilde{\wedge}} C$ nor $A \mathbin{\tilde{\vee}} B \mathbin{\tilde{\vee}} C$ are meaningful — let alone similar expressions with more than three variables — as neither NAND ($\tilde{\wedge}$) nor NOR ($\tilde{\vee}$) are *associative*. Furthermore, it is folly to use $A \mathbin{\tilde{\wedge}} B \mathbin{\tilde{\wedge}} C$ as an abbreviation for $\overline{A \wedge B \wedge C}$, and $A \mathbin{\tilde{\vee}} B \mathbin{\tilde{\vee}} C$ as an abbreviation for $\overline{A \vee B \vee C}$, as you will find time and again in otherwise serious texts. For $n \geq 3$ we shall thus abide by the notation for NOT-AND, $\overline{X_1 \wedge \cdots \wedge X_n}$, and NOT-OR, $\overline{X_1 \vee \cdots \vee X_n}$.

21.1 Miscellaneous Theorems

If you want to be able to manipulate logic expressions based on NOT-AND and NOT-OR algebraically, you will want a collection of relevant theorems. The most basic of these are the NAND and NOR theorems *on commutativity, on a single variable*, and *DeMorgan's theorem.*

$$A \mathbin{\tilde{\wedge}} B \Leftrightarrow B \mathbin{\tilde{\wedge}} A, \qquad A \mathbin{\tilde{\vee}} B \Leftrightarrow B \mathbin{\tilde{\vee}} A. \tag{21.1}$$

$$A \mathbin{\tilde{\wedge}} \mathbf{0} \Leftrightarrow \mathbf{1}, \qquad A \mathbin{\tilde{\vee}} \mathbf{1} \Leftrightarrow \mathbf{0}. \tag{21.2}$$

$$A \mathbin{\tilde{\wedge}} \mathbf{1} \Leftrightarrow \overline{A}, \qquad A \mathbin{\tilde{\vee}} \mathbf{0} \Leftrightarrow \overline{A}. \tag{21.3}$$

$$A \mathbin{\tilde{\wedge}} A \Leftrightarrow \overline{A}, \qquad A \mathbin{\tilde{\vee}} A \Leftrightarrow \overline{A}. \tag{21.4}$$

$$A \mathbin{\tilde{\wedge}} \overline{A} \Leftrightarrow \mathbf{1}, \qquad A \mathbin{\tilde{\vee}} \overline{A} \Leftrightarrow \mathbf{0}. \tag{21.5}$$

$$\overline{A \mathbin{\tilde{\wedge}} B \Leftrightarrow \overline{A} \mathbin{\tilde{\vee}} \overline{B}}, \qquad \overline{A \mathbin{\tilde{\vee}} B \Leftrightarrow \overline{A} \mathbin{\tilde{\wedge}} \overline{B}}. \tag{21.6}$$

To recapitulate results of Chapter 10: *Neither* NAND *nor* NOR *are associative, and neither of these functions plays any role in distributivity.* The relationships between (multiple) AND and NAND are governed by the following formulas.

$$A \wedge B \Leftrightarrow \overline{A \mathbin{\tilde{\wedge}} B} \tag{21.7}$$

$$A \wedge B \wedge C \Leftrightarrow \overline{\overline{A \mathbin{\tilde{\wedge}} B} \mathbin{\tilde{\wedge}} C} \tag{21.8}$$

$$A \wedge B \wedge C \wedge D \Leftrightarrow \overline{\overline{A \mathbin{\tilde{\wedge}} B} \mathbin{\tilde{\wedge}} \overline{C \mathbin{\tilde{\wedge}} D}} \tag{21.9a}$$

$$\Leftrightarrow \overline{\overline{A \mathbin{\tilde{\wedge}} B} \mathbin{\tilde{\wedge}} \overline{C \mathbin{\tilde{\wedge}} D}} \tag{21.9b}$$

A pondering glance at (21.8) and (21.9a) will convince you that the right sides of these formulas can be extrapolated straightforwardly for five and more variables. Note that these formulas hold for any permutation of their variables. As (21.9b) indicates, you can rearrange the NEGATIONs in multiple ways for four and more variables. Of practical interest are those formulas that distribute the NEGATIONs symmetrically, or near-symmetrically — as in (21.9b). To see how to derive these formulas look at the following example:

$$A \wedge B \wedge C \Leftrightarrow \overline{\overline{A \wedge B} \wedge C} \Leftrightarrow \overline{A \mathbin{\tilde{\wedge}} B \wedge C} \Leftrightarrow \overline{\overline{A \mathbin{\tilde{\wedge}} B \wedge C}} \Leftrightarrow \overline{\overline{A \mathbin{\tilde{\wedge}} B} \mathbin{\tilde{\wedge}} C}.$$

Applying *duality* (see Section 5.4) to the above formulas provides us with a class of formulas that relate multiple OR to NOR:

$$A \vee B \Leftrightarrow \overline{A \mathbin{\tilde{\vee}} B} \tag{21.10}$$

$$A \vee B \vee C \Leftrightarrow \overline{\overline{A \mathbin{\tilde{\vee}} B} \mathbin{\tilde{\vee}} C} \tag{21.11}$$

$$A \vee B \vee C \vee D \Leftrightarrow \overline{\overline{A \mathbin{\tilde{\vee}} B} \mathbin{\tilde{\vee}} \overline{C \mathbin{\tilde{\vee}} D}} \tag{21.12a}$$

$$\Leftrightarrow \overline{\overline{A \mathbin{\tilde{\vee}} B} \mathbin{\tilde{\vee}} \overline{C \mathbin{\tilde{\vee}} D}} \tag{21.12b}$$

Trivially, the negation of the above multiple AND and multiple OR formulas relate the NOT-AND and the NOT-OR to the NAND and NOR functions. For instance:

$$\overline{A \wedge B \wedge C} \Leftrightarrow \overline{A \mathbin{\tilde{\wedge}} B} \mathbin{\tilde{\wedge}} C, \qquad \overline{A \vee B \vee C} \Leftrightarrow \overline{A \mathbin{\tilde{\vee}} B} \mathbin{\tilde{\vee}} C. \tag{21.13}$$

21.2 Two-Level Circuits

Assume you have — by some means convenient to you — developed an algebraic expression that is a *minimal disjunctive form* of a given switching function. Such a *minimal* solution is expressed by the *evaluation formula*

$$Y(x) \Leftrightarrow \bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}_{min}(\mathcal{E}_1)} C_\varphi(x). \tag{17.30a}$$

Inverting the right side twice,

$$Y(x) \Leftrightarrow \neg\neg \bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}_{min}(\mathcal{E}_1)} C_\varphi(x),$$

and then applying $\neg \bigvee P(x) \Leftrightarrow \bigwedge \neg P(x)$, DeMorgan (13.68), leads to the NOT-AND **evaluation formula**

$$Y(x) \Leftrightarrow \neg \underbrace{\bigwedge_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}_{min}(\mathcal{E}_1)}}_{\text{NOT-AND}} \underbrace{\Big(\neg C_\varphi(x)\Big)}_{\text{NOT-AND}}. \tag{21.14}$$

The corresponding NOT-OR **evaluation formula**, derived from equation (17.31b), is

$$Y(x) \Leftrightarrow \neg \underbrace{\bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}_{min}(\mathcal{E}_0)}}_{\text{NOT-OR}} \underbrace{\Big(\neg D_\varphi(x)\Big)}_{\text{NOT-OR}}. \tag{21.15}$$

As both formulas use minimal covers, $\mathcal{P}_{min}(\mathcal{E}_1)$ and $\mathcal{P}_{min}(\mathcal{E}_0)$, they provide us with (two-level) circuits containing *the smallest number of gates.* To illustrate the use of (21.14), consider the example of a function defined by $\mathcal{E}_1 = \{2,3,5,6,7,9,11,13\}$. This function was minimised in Section 19.2 where we obtained the single minimal cover $\mathcal{P}_{min}(\mathcal{E}_1) = \{\mathcal{P}_B, \mathcal{P}_D, \mathcal{P}_F\} = \{\mathcal{P}_{10x1}, \mathcal{P}_{x101}, \mathcal{P}_{0x1x}\}$. This minimisation result allows us to write (21.14) as

$$\begin{aligned} Y(x) \Leftrightarrow \neg \bigwedge_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}_{min}(\mathcal{E}_1)} \Big(\neg C_\varphi(x)\Big) \Leftrightarrow \\ \neg\Big((\neg \mathcal{C}_{10x1}) \wedge (\neg \mathcal{C}_{x101}) \wedge (\neg \mathcal{C}_{0x1x})\Big) \Leftrightarrow \\ \overline{\overline{X_1 \overline{X}_2 X_4} \wedge \overline{X_2 \overline{X}_3 X_4} \wedge \overline{\overline{X}_1 X_3}}. \end{aligned} \tag{21.16}$$

To obtain the NOT-OR form for the same function, you use (21.15). This formula requires that you work with the 0-equivalent $\mathcal{E}_0$ — as opposed to the 1-equivalent $\mathcal{E}_1$ when determining the NOT-AND circuit by using

(21.14). The 0-equivalent $\mathcal{E}_0$ is the complement of the 1-equivalent $\mathcal{E}_1 = \{2,3,5,6,7,9,11,13\}$ with respect to the universe

$$\mathcal{E} = \{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15\}$$

(if you are considering 4 input variables): $\mathcal{E}_0 = \mathcal{E} \setminus \mathcal{E}_1 = \{0,1,4,8,12,14,15\}$. As mentioned at the end of Section 19.4, this function has a single minimal cover $\mathcal{P}_{min}(\mathcal{E}_0) = \{\mathcal{P}_{000x}, \mathcal{P}_{111x}, \mathcal{P}_{xx00}\}$ allowing us to write (21.15) as

$$Y(x) \Leftrightarrow \neg \bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}_{min}(\mathcal{E}_0)} \Big(\neg D_\varphi(x)\Big) \Leftrightarrow$$

$$\neg\Big(\big(\neg\mathcal{D}_{000x}\big) \vee \big(\neg\mathcal{D}_{111x}\big) \vee \big(\neg\mathcal{D}_{xx00}\big)\Big) \Leftrightarrow$$

$$\overline{\overline{(X_1 \vee X_2 \vee X_3)} \vee \overline{(\overline{X}_1 \vee \overline{X}_2 \vee \overline{X}_3)} \vee \overline{(X_3 \vee X_4)}}. \qquad (21.17)$$

You might like to compare (21.16) and (21.17) to the canonical solutions (19.10) and (19.11) of Section 19.4. The circuits of (21.16) and (21.17) are shown in Fig. 21–1.

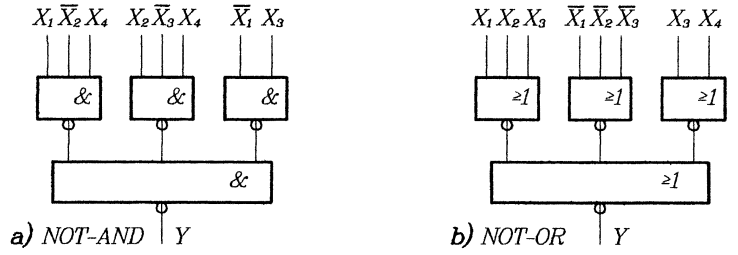


Fig. 21–1 NOT-AND and NOT-OR circuits of (21.16) and (21.17)

21.3 Multilevel NOT-AND Circuits

The two-level circuits discussed in the previous section are the fastest, but sometimes need more gates than slower multilevel designs to be looked at in this section. The reason for a multilevel circuit being slower than a two-level circuit is that at least one input signal must pass through more than two gates to traverse the circuit. An extraordinarily extensive presentation of NOT-AND and NOT-OR design methods is presented by **Muroga** [1979] from which the **map-factoring method** discussed here is taken. This method is only meant for relatively small problems that can conveniently be expressed in K-maps. There are some noteworthy aspects to the method.

(a) You can obtain an exceptionally large number and variety of solutions. But the only way to know whether you have found a 'good' solution or not is to compare it with other solutions found.
(b) You have no direct influence on either the number of NAND gates or the number of inputs to the individual gates.
(c) You must rely completely on your experience and intuition when trying to find a 'good' solution. There is no *'forward looking'* algorithm to help you along.

The above points are not as typical for the map-factored method as they are for the complexity of the problem to be solved. To facilitate the explanation of the map-factored method we need to introduce the concept of an *anchored K-set.*

(21.18) An **anchored K-set** $\mathcal{K}_{\varphi_1 \ldots \varphi_n}$ is a K-set of which the n-tuple $(1, 1, \ldots, 1)$ — consisting exclusively of 1s — is an element.

For a two-input K-map, Fig. 21–2 shows all anchored K-sets — all are *anchored around* the cell $(1, 1)$, marked by an asterisk (*). In defining an anchored K-set, no mention was made in (21.18) of the content of any cells. In a K-map, loops specifying an anchored K-set can contain 0s, 1s, *don't cares*, map-entered variables, or any combinations of these. Muroga calls anchored K-sets *'permissible loops'*, whereas he reserves no name for what in this text is called a *gate-defining set* $\mathcal{G}$ — introduced in step 4 of (21.19).

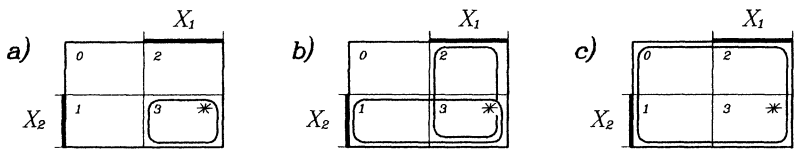


Fig. 21–2 Anchored K-sets of a two-input K-map

(21.19) **The map-factoring method for finding multilevel NOT-AND circuits.**

Step 1. By visual inspection of the K-map choose an **anchored K-set** $\mathcal{K}_\varphi^{(1)}$ and, to start the iterative process of this algorithm, a **gate-defining set** $\mathcal{G}^{(1)}$ equal to $\mathcal{K}_\varphi^{(1)}$.

Step 2. Draw a NOT-AND gate with the inputs of the generalised minterm $C_\varphi \Leftrightarrow \mathbf{x} \in \mathcal{K}_\varphi^{(1)}$ giving the NOT-AND gate a gate number of 1.

Step 3. Start the iteration process of this algorithm by setting the gate number g of the following step to 2 ($g \leftarrow 2$).

Step 4. By visual inspection of the K-map choose an anchored K-set $\mathcal{K}_\varphi^{(g)}$. Choose an associated **gate-defining set**

$$\mathcal{G}^{(g)} \stackrel{df}{=} \mathcal{K}_\varphi^{(g)} \setminus (\mathcal{G}^{(g')} \cup \mathcal{G}^{(g'')} \cup \cdots) \qquad (21.19a)$$

in which $\mathcal{G}^{(g')}$, $\mathcal{G}^{(g'')}$, etc., are any previously specified gate-defining sets whose intersections with $\mathcal{K}_\varphi^{(g)}$ are *not* empty. Note that you need not choose any of the previously specified gate-defining sets, i.e., $\mathcal{G}^{(g')} \cup \mathcal{G}^{(g'')} \cup \cdots$ may be empty.

Step 5. Draw a NOT-AND gate (associated with the gate-defining set $\mathcal{G}^{(g)}$) whose input variables are determined as follows. **Firstly**, the variables that define $\mathcal{C}_\varphi^{(g)}$ (with $\mathcal{C}_\varphi^{(g)} \Leftrightarrow \mathbf{x} \in \mathcal{K}_\varphi^{(g)}$) are inputs to the NOT-AND gate. **Secondly**, the output of every gate assigned to a gate-defining set $\mathcal{G}^{(g')}$, $\mathcal{G}^{(g'')}$, etc., used in $\mathcal{G}^{(g)} = \mathcal{K}_\varphi^{(g)} \setminus (\mathcal{G}^{(g')} \cup \mathcal{G}^{(g'')} \cup \cdots)$ is taken as input to the NOT-AND gate.

Step 6. If the loop specifying the gate-defining set $\mathcal{G}^{(g)} \subseteq \mathcal{K}_\varphi^{(g)}$ chosen in step 4 has at least one symbol inscribed that is not 0, or does not contain all the zeroes of the K-map, then increment g by 1 ($g \hookleftarrow g + 1$) and go to step 4. If the loop contains *all the zeroes of the K-map and only zeroes*, terminate the algorithm.

Let us work through the simple **example** shown in Fig. 21–3 to see how to employ the map-factored method (21.19).

According to **step 1** we choose the first anchored K-set as $\mathcal{K}_{1x_2x_3}^{(1)}$ and the first gate-defining set $\mathcal{G}^{(1)} = \mathcal{K}_{1x_2x_3}^{(1)}$ pictured in the K-map of Fig. 21–3a.

Step 2 requires us to draw a NOT-AND gate (associated with $\mathcal{G}^{(1)}$) the inputs of which are defined as $\mathbf{x} \in \mathcal{K}_{1x_2x_3}^{(1)} \Leftrightarrow C_{1x_2x_3}(\mathbf{x}) \Leftrightarrow X_1$, i.e., our NOT-AND gate has only one input, this being X_1. The gate is given the number 1.

We now start the iterative process with **step 4** by choosing another anchored K-set, the K-set $\mathcal{K}_{x_1x_21}^{(2)}$ pictured in Fig. 21–3b. Next, choosing to not use the previously specified gate-defining set $\mathcal{G}^{(1)}$ — so that the output of gate 1 is not employed as an input to gate 2 — we determine $\mathcal{G}^{(2)}$ as $\mathcal{G}^{(2)} = \mathcal{K}_{x_1x_21}^{(2)} \setminus \emptyset = \mathcal{K}_{x_1x_21}^{(2)}$.

By **step 5** we draw the NOT-AND gate number 2 of Fig. 21–3b, its only input, X_3, being defined by $\mathbf{x} \in \mathcal{K}_{x_1x_21}^{(2)} \Leftrightarrow \mathcal{C}_{x_1x_21}^{(2)}(\mathbf{x}) \Leftrightarrow X_3$.

As $\mathcal{G}^{(2)}$ does not contain only zeroes (in fact, it contains no zeroes) **step 6** requires us to increment g to 3 and to go to ...

Step 4. As shown in Fig. 21–3c, we choose our third anchored K-set to be $\mathcal{K}_{x_11x_3}^{(3)}$ and its associated gate-defining set as $\mathcal{G}^{(3)} = \mathcal{K}_{x_11x_3}^{(3)} \setminus (\mathcal{G}^{(1)} \cup \mathcal{G}^{(2)})$.

Step 5. The inputs to the NOT-AND gate of $\mathcal{G}^{(3)}$ are X_2 — due to $\mathbf{x} \in \mathcal{K}_{x_11x_3}^{(3)} \Leftrightarrow \mathcal{C}_{x_11x_3}^{(3)}(\mathbf{x}) \Leftrightarrow X_2$ — and the outputs of the gates 1 and 2, specified by $\mathcal{G}^{(1)}$ and $\mathcal{G}^{(2)}$.

Step 6. Here, the algorithm terminates because $\mathcal{G}^{(3)}$ contains all the zeroes and only zeroes .

To understand how the connection of the gates is governed by (21.19a),

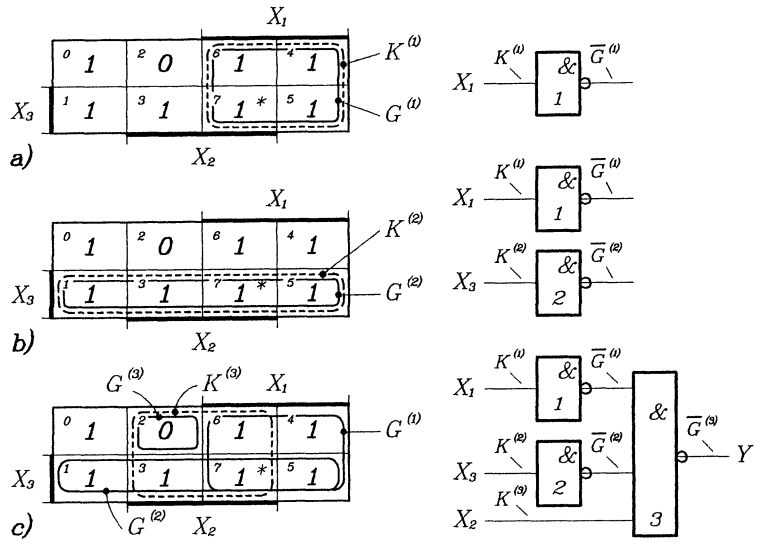


Fig. 21–3 An example illustrating algorithm (21.19)

note that

$$\mathcal{K}_{\varphi}^{(g)} \setminus (\mathcal{G}^{(g')} \cup \mathcal{G}^{(g'')} \cup \ldots) \stackrel{df}{=} \mathcal{K}_{\varphi}^{(g)} \cap \overline{\mathcal{G}^{(g')} \cup \mathcal{G}^{(g'')} \cup \cdots},$$

which allows us to rewrite (21.19a) as

$$\mathcal{G}^{(g)} = \mathcal{K}_{\varphi}^{(g)} \cap \overline{\mathcal{G}}^{(g')} \cap \overline{\mathcal{G}}^{(g'')} \cap \cdots. \tag{21.19b}$$

This formula, written for gate number 3 of Fig. 21–3c,

$$\mathcal{G}^{(3)} = \mathcal{K}_{x_1 1 x_3}^{(3)} \setminus (\mathcal{G}^{(1)} \cup \mathcal{G}^{(2)}) = \mathcal{K}_{x_1 1 x_3}^{(3)} \cap \overline{\mathcal{G}}^{(1)} \cap \overline{\mathcal{G}}^{(2)},$$

is a replica of the connections to that gate. Let us now rework the above example as shown in Fig. 21–4. The only difference is the way the gate-defining set $\mathcal{G}^{(2)}$ is chosen: $\mathcal{G}^{(2)} = \mathcal{K}_{x_1 x_2 1}^{(2)} \setminus \mathcal{G}^{(1)}$. By this choice, see Fig. 21–4b, the output of gate number 1 is used as input to gate number 2. Now the final step, although formally the same as for Fig. 21–3c, leads to the slightly more complex solution of Fig. 21–4c. You can verify the circuit by the following calculation:

$$Y \Leftrightarrow \overline{\overline{X}_1 \, \overline{X}_1 X_3 \, X_2} \Leftrightarrow \overline{\overline{X}_1 \, (X_1 \vee \overline{X}_3) \, X_2} \Leftrightarrow \overline{X_1 X_2 \overline{X}_3}.$$

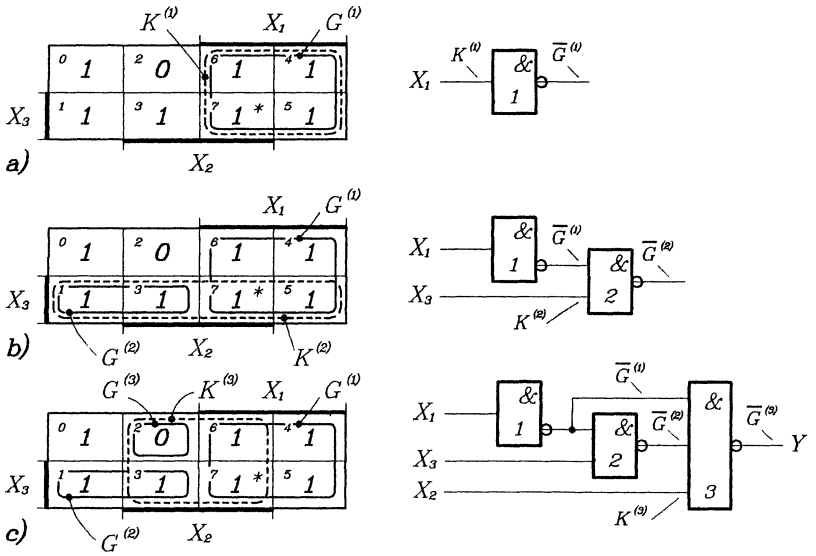


Fig. 21–4 Reworking the example of the previous figure

The last example in this section (see Fig. 21–5) is only marginally more complicated than the previous one. Yet the solution is almost intricate. In the following discussion, let (a), (b), etc., refer to the equally named parts of Fig. 21–5.

The K-map (a) shows our choice for $\mathcal{K}^{(1)}$ and thus for $\mathcal{G}^{(1)} = \mathcal{K}^{(1)}$. In (b) the dashed loop represents $\mathcal{K}^{(2)}$, and as we wish to use $\mathcal{G}^{(1)}$ in determining $\mathcal{G}^{(2)}$, we obtain this set as $\mathcal{G}^{(2)} = \mathcal{K}^{(2)} \setminus \mathcal{G}^{(1)}$ and with it the circuit of (b). In (c) we choose to use none of the previous gate-defining sets $\mathcal{G}^{(1)}$ or $\mathcal{G}^{(2)}$ so that $\mathcal{G}^{(3)}$ equals the anchored K-set $\mathcal{K}^{(3)}$, i.e., $\mathcal{G}^{(3)} = \mathcal{K}^{(3)}$. Now, by choosing $\mathcal{K}^{(4)}$ — as shown on (d) — *without* $\mathcal{G}^{(1)}$ or $\mathcal{G}^{(3)}$, we get $\mathcal{G}^{(4)} = \mathcal{K}^{(4)} \setminus (\mathcal{G}^{(1)} \cup \mathcal{G}^{(3)})$. Now note that the sets $\mathcal{G}^{(2)}$ and $\mathcal{G}^{(4)}$ together contain all the zeroes and only zeroes, the termination condition we are looking for. But, while we cannot describe their union directly, we can describe their complement — which we do in (e). Choosing $\mathcal{K}^{(5)}$ as the universe (i.e., all cells of the K-map) we define $\mathcal{G}^{(5)}$ as $\mathcal{K}^{(5)}$ without $\mathcal{G}^{(2)}$ and $\mathcal{G}^{(4)}$, so that $\mathcal{G}^{(5)} = \mathcal{K}^{(5)} \setminus (\mathcal{G}^{(2)} \cup \mathcal{G}^{(4)})$. Complementing $\mathcal{G}^{(5)}$ as shown in (f) provides us with the final result.

If you feel it necessary to verify the circuit of Fig. 21–5f, evaluate its algebraic equivalent:

$$Y \Leftrightarrow \overline{\underbrace{\underbrace{(\overline{X_1X_2}}_{\mathcal{G}^{(1)}}\ X_1X_3)}_{\mathcal{G}^{(2)}}\ \underbrace{\underbrace{(\overline{X_2X_3}}_{\mathcal{G}^{(3)}}\ X_2\,\overline{X_1X_2})}_{\mathcal{G}^{(4)}}},$$

noting that the top two negations (the double negation) are due to $\mathcal{G}^{(5)}$ and

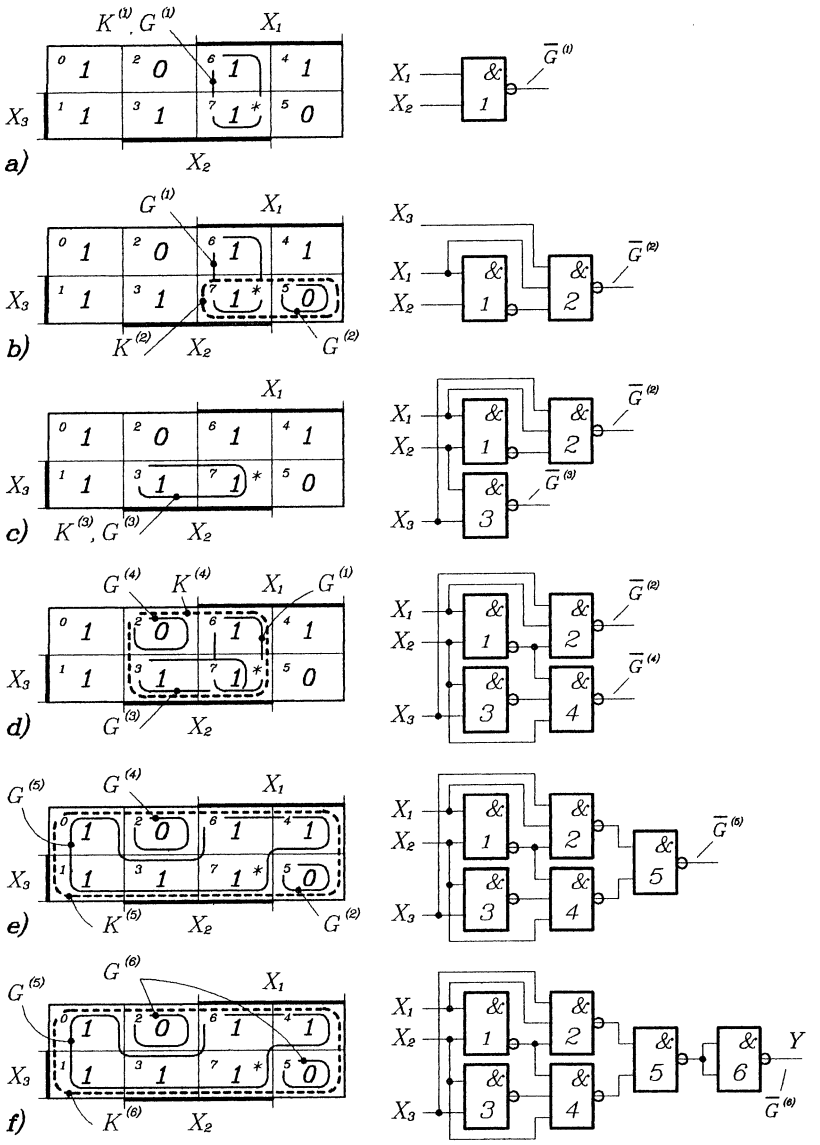


Fig. 21–5 Another simple example on the use of (21.19)

to $\mathcal{G}^{(6)}$. Dropping the double negation and carrying out the negations of the parentheses leads to

$$Y \Leftrightarrow (\underbrace{X_1X_2 \vee \overline{X}_1}_{\overline{X}_1 \vee X_2} \vee \overline{X}_3)(\underbrace{X_2X_3 \vee \overline{X}_2}_{\overline{X}_2 \vee X_3} \vee X_1X_2)$$

$$Y \Leftrightarrow (\overline{X}_1 \vee X_2 \vee \overline{X}_3)(\underbrace{X_1X_2 \vee \overline{X}_2}_{X_1 \vee \overline{X}_2} \vee X_3)$$

$$Y \Leftrightarrow (\overline{X}_1 \vee X_2 \vee \overline{X}_3)(X_1 \vee \overline{X}_2 \vee X_3)$$

As the above expression is the conjunctive normal form that can be read directly from a K-map of Fig. 21–5, the circuit of Fig. 21–5f is verified.

21.4 Multilevel NOT-OR Circuits

A slightly modified form of the map-factoring method of the previous section can be used to obtain a multilevel NOT-OR circuit. Firstly, we need to redefine anchored K-sets so that they are anchored around the n-tuple $(0, 0, \ldots, 0)$.

(21.20) An **anchored K-set** $\mathcal{K}_{\varphi_1 \ldots \varphi_n}$ is a K-set of which the n-tuple $(0, 0, \ldots, 0)$ — consisting exclusively of 0s — is an element.

Then, we need to re-specify steps 2, 5 and 6 of algorithm (21.19) as stated here:

Step 2. Draw a NOT-OR gate with the inputs of the generalised maxterm $D_{\varphi_1 \ldots \varphi_n}(x_1, \ldots, x_n) \Leftrightarrow \bigvee_{i=1}^{n}(x_i \neq \varphi_i)$ in which $\varphi_i \in \{0, 1, x_i\}$, according to (17.13), giving the NOT-OR gate a gate number of 1.

Step 5. Draw a NOT-OR gate (associated with the gate-defining set $\mathcal{G}^{(g)}$) whose input variables are determined as follows. **Firstly**, the variables that define $\mathcal{D}_{\varphi}^{(g)}$ (with $\mathcal{D}_{\varphi}^{(g)} \Leftrightarrow \mathbf{x} \in \overline{\mathcal{K}}_{\varphi}^{(g)} \Leftrightarrow \bigvee_{i=1}^{n}(x_i \neq \varphi_i)$) are inputs to the NOT-OR gate. **Secondly**, the output of every gate assigned to a gate-defining set $\mathcal{G}^{(g')}$, $\mathcal{G}^{(g'')}$, etc., used in $\mathcal{G}^{(g)} = \mathcal{K}_{\varphi}^{(g)} \setminus (\mathcal{G}^{(g')} \cup \mathcal{G}^{(g'')} \cup \cdots)$ is taken as input to the NOT-OR gate.

Step 6. If the loop specifying the gate-defining set $\mathcal{G}^{(g)} \subseteq \mathcal{K}_{\varphi}^{(g)}$ chosen in step 4 has at least one symbol inscribed that is not 1, or does not contain all the ones of the K-map, then increment g by 1 ($g \hookleftarrow g + 1$) and go to step 4. If the loop contains *all the ones of the K-map and only ones*, terminate the algorithm.

The connection of gates, as expressed in the above step 5, is governed by the *negation* of (21.19b),

$$\overline{\mathcal{G}}^{(g)} = \overline{\mathcal{K}}_{\varphi}^{(g)} \cup \mathcal{G}^{(g')} \cup \mathcal{G}^{(g'')} \cup \cdots. \tag{21.21}$$

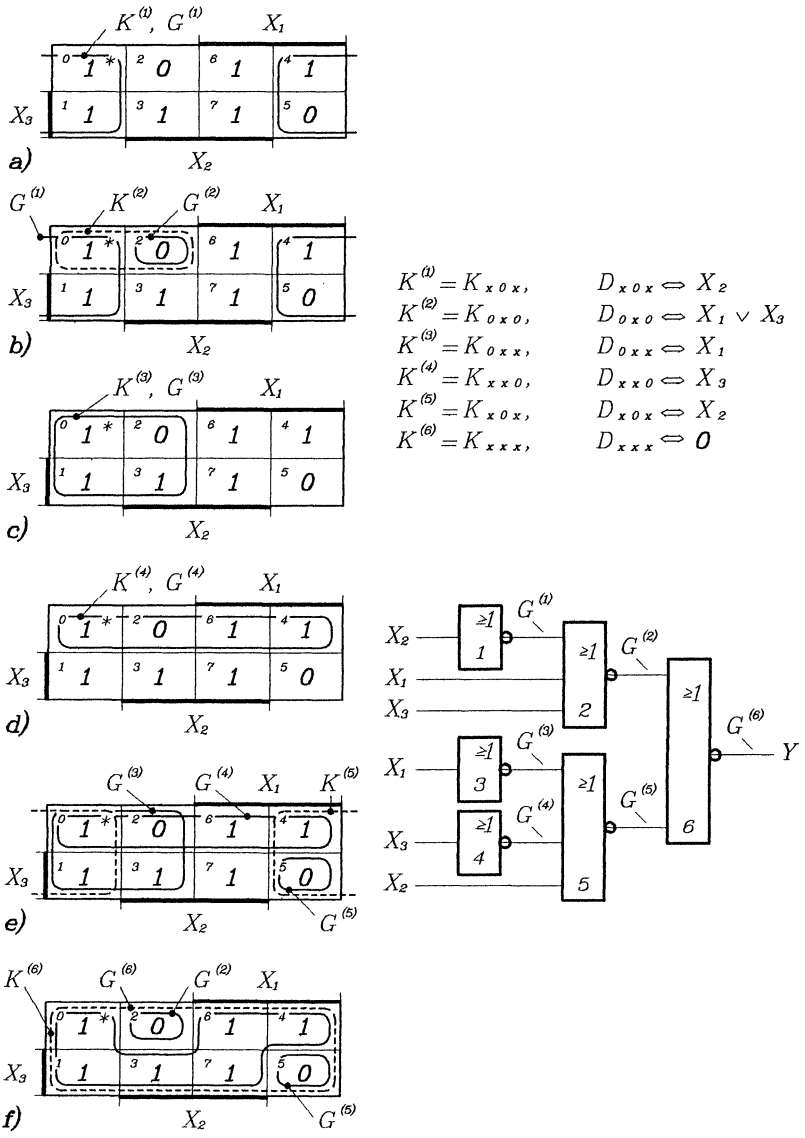


Fig. 21–6 Applying the map-factoring method to a multilevel NOT-OR circuit

If you have worked through the NOT-AND examples of the previous section, you should have no difficulty in following the NOT-OR example of Fig. 22-6.

An alternative method (of at least theoretical interest) to obtain a multi-level NOT-OR circuit is to develop the NOT-AND circuit for the *dual* function $\widetilde{F}$. Then, by replacing the NOT-AND gates by NOT-OR gates you obtain the desired NOT-OR circuit of the given function F. In Section 7.6 you can find a detailed discussion on how to obtain the K-map of the dual function $\widetilde{F}$, given the K-map of F.

Chapter 22

Composition of Circuits *

In a bird's-eye view, the synthesis methods discussed so far require each new circuit (each new *'molecule'*) to be put together from elementary gates (from *'atoms'*). In Chapter 14 the *'atoms'* were AND, OR, and NOT gates, in the previous chapter they were NOT-AND gates or NOT-OR gates. Yet, just as in nature new molecules are put togrther, or *composed*, from already existing ones and from atoms, it would seem reasonable to develop an analogous strategy in synthesising circuits. **Composing** a circuit in this way, i.e., *designing it by combining previously developed (and usually well-tested) subcircuits, and elementary gates (when necessary)*, is the topic of this chapter. Alternatively, see **Ashenhurst**'s [1957] decomposition method.

As simple as the composition method is, its practical application relies (at present) completely on your intuition, insight and experience. Obviously, further development of the method is necessary to make it practicable for computer use.

22.1 The Basic Concept

To keep things simple to start with, we consider using only a single function as an *'atom'*. As our main considerations turn on this function, I choose to call it a **hinge function**. For example, say we want to develop a circuit that realises the EQUIVALENCE specified in Fig. 22–1a. Using NOR as a hinge function, we could (with luck) develop the circuit pictured in Fig. 22–1b. On the other hand, taking the *normally closed* pneumatic valve as a hinge function, we could realise the EQUIVALENCE as shown on Fig. 22–1c. Of course, using a *single* hinge function is only possible if no other functions are needed to realise a circuit (as discussed in Section 15.6). For simpler representation of the procedure and easier verification of the results we shall follow-up the use of NOR as a hinge function.

The basic idea is to develop the circuit *from the output to the inputs.* This is emphasised in Fig. 22–2a where the **specified function** f is decomposed into the **hinge function** h at the output of the circuit, and the so-called **generic function** g the outputs of which are used as inputs to the hinge

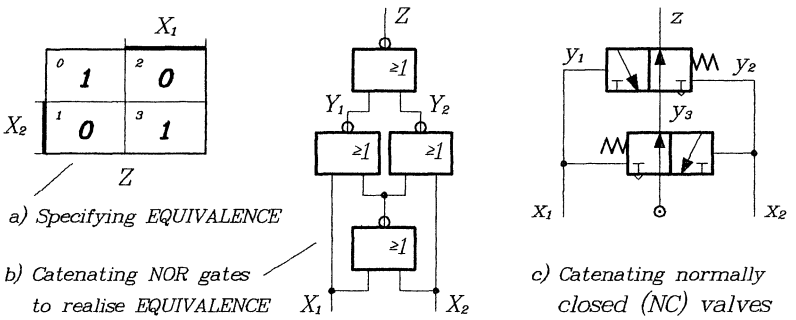
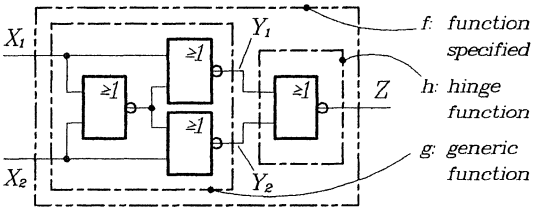


Fig. 22–1 Realisations by single hinge functions

function. The name *generic function* is chosen to remind us that there is not a singe, uniquely defined function g that we need only calculate or determine, but rather, g is a function we can choose from a whole class of functions that together with the hinge function h are a realisation of the specified function f.



X_1 =	0	0	1	1
X_2 =	0	1	0	1
y_1 =	0	1	0	0
y_2 =	0	0	1	0
Z =	1	0	0	1

Fig. 22–2 Decomposing the specified function f into the hinge function h and the generic function g

The generic function g is a multiple-output function. In fact, it always has as many outputs as the hinge function h has inputs. The output values of the generic function g of our example are specified in the table of asserted events (the TAE) of Fig. 22–2. We shall later consider each output y_1, y_2 of the partial generic function independently, so that each output represents an individual (generic) function which in turn can be decomposed using the chosen hinge function.

The mathematician refers to the functions h and g of Fig. 22–2 as being *catenated*, or chained together, to create the function f. Catenation can be pictured very descriptively using mappings so that we can do all our reasoning graphically.

22.2 Catenation

The word *catenation* originates from *'catena'*, the Latin word for *chain*. As a technical term, *catenation* refers to the chaining together of two functions. To get a feeling for this chaining together of functions consider the example shown in Fig. 22–3. The functions we wish to catenate (not identical with the equally named functions of the previous section), $g : \mathcal{X} \mapsto \mathcal{Y}$ and $h : \mathcal{Y} \mapsto \mathcal{Z}$, are defined in Fig. 22–3a, this representation being possible because $\mathcal{Y}$ is simultaneously the co-domain of $g : \mathcal{X} \mapsto \mathcal{Y}$ and the domain of $h : \mathcal{Y} \mapsto \mathcal{Z}$. While Fig. 22–3a visualises the process of catenating, Fig. 22–3b shows the result of the catenation, the composite function $hg : \mathcal{X} \mapsto \mathcal{Z}$. The composite function hg is constructed by *following the arrows* from $\mathcal{X}$ via $\mathcal{Y}$ to $\mathcal{Z}$. For instance, if we follow the arrow in Fig. 22–3a from x_1 we come to y_2, and from here via the next arrow to z_1. The composite function hg summarises these successive mappings, simply stating that x_1 maps to z_1.

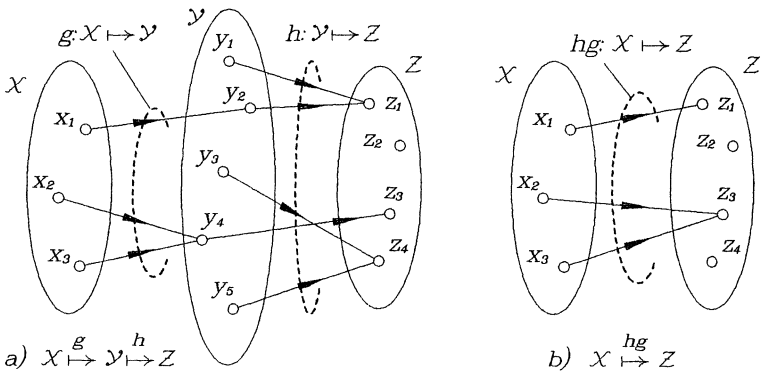


Fig. 22–3 An example demonstrating the catenation of functions

In a formal definition of catenation, we want to make sure that arrows are correctly chained by only using those elements of $\mathcal{Y}$ that are images of g, i.e., those y for which $y = g(x)$ holds. We thus formulate

(22.1) Given arbitrary functions $g : \mathcal{X} \mapsto \mathcal{Y}$ and $h : \mathcal{Y} \mapsto \mathcal{Z}$, their **catenation** $hg : \mathcal{X} \mapsto \mathcal{Z}$ (hg being read as 'h follows g') is specified as the set hg of ordered pairs (x, z) such that

$$(x,z) \in hg \overset{df}{\Leftrightarrow} \bigvee_{y \in \mathcal{Y}} \big((x,y) \in g \wedge (y,z) \in h\big) \qquad (22.1a)$$

for any substitution instances $x \in \mathcal{X}$ and $z \in \mathcal{Z}$.

The *whole* definition (22.1) is often abbreviated by the picturesque notation

$$\mathcal{X} \overset{hg}{\mapsto} \mathcal{Z} = \mathcal{X} \overset{g}{\mapsto} \mathcal{Y} \overset{h}{\mapsto} \mathcal{Z}, \tag{22.2}$$

which emphasises that $\mathcal{Y}$ is simultaneously the co-domain of g and the domain of h. Without further proof, let me state that the above definition allows us to deduce the following explanation for the conventional notation $h(g(x))$:

$$h\big(g(x)\big) = (hg)(x). \tag{22.3}$$

The principle property of catenation is *associativity*, without which the catenation of circuits would not work. Associativity means, given three functions such that

$$\mathcal{X} \overset{g}{\mapsto} \mathcal{Y} \overset{h}{\mapsto} \mathcal{Z} \overset{r}{\mapsto} \mathcal{W},$$

the sequence in which catenation is carried out is irrelevant:

$$rhg = r(hg) = (rh)g. \tag{22.4}$$

22.3 Choosing a Generic Function

As a step towards understanding how to obtain a generic function g, let us draw our example circuit of Fig. 22–2 as a **catenation diagram**, Fig. 22–4. This diagram pictures the catenation

$$\mathcal{X} \overset{f}{\mapsto} \mathcal{Z} = \mathcal{X} \overset{g}{\mapsto} \mathcal{Y} \overset{h}{\mapsto} \mathcal{Z}.$$

The arrows are drawn according to the table of asserted events of Fig. 22–2. The diagram contains more information than the TAE of Fig. 22–2. The TAE does not say what the output z would be if $(y_1, y_2) = (1, 1)$, but this information is contained in Fig. 22–4 in the form of an arrow pointing from $(1, 1) \in \mathcal{Y}$ to $0 \in \mathcal{Z}$. This reflects the necessity to specify a value in the co-domain for *each* element of the domain of a function. Most importantly, the catenation diagram puts the finger on the fact that the co-domain $\mathcal{Y}$ of the generic function g is a set of ordered tuples, and that we are *not* at this stage working with individual generic functions each with its own single output.

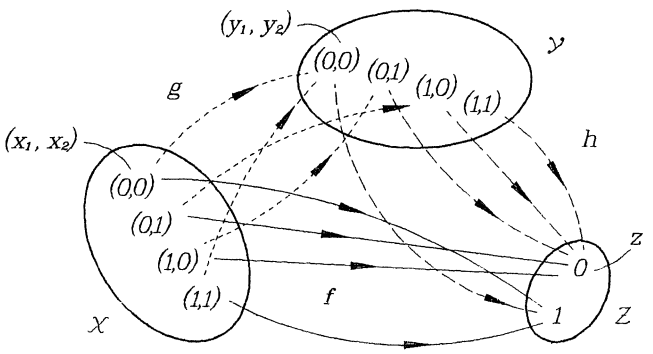


Fig. 22–4 Catenation diagram of the circuit of Fig. 22–2

In the next step we choose an output value $z \in \mathcal{Z}$ to be able to define equivalents $\mathcal{X}_z$ and $\mathcal{Y}_z$ of the specified function $f : \mathcal{X} \mapsto \mathcal{Z}$ and the hinge function $h : \mathcal{Y} \mapsto \mathcal{Z}$,

$$\mathcal{X}_z \stackrel{df}{=} \{\mathbf{x}|f(\mathbf{x}) = z\}, \qquad \mathcal{Y}_z \stackrel{df}{=} \{\mathbf{y}|h(\mathbf{y}) = z\}, \tag{22.5}$$

and use them as the domains of **partial functions**

$$f_z : \mathcal{X}_z \mapsto \{z\}, \quad \text{and} \quad h_z : \mathcal{Y}_z \mapsto \{z\}. \tag{22.6}$$

Consider the case in which $z \hookleftarrow 0$. Restricting our view to those arrows of Fig. 22–4 that have their source in the equivalents $\mathcal{X}_0$ and $\mathcal{Y}_0$ leads to Fig. 22–5. This figure contains the **partial generic function** $g_0 : \mathcal{X}_0 \mapsto \mathcal{Y}_0$ which results from the analysis of the circuit of Fig. 22–2. But, quite clearly, any of the $3^2 = 9$ functions $\mathcal{X}_0 \mapsto \mathcal{Y}_0$ will have the desired property of mapping all $\mathbf{x} \in \mathcal{X}_0$ via $\mathcal{Y}_0$ to $0 \in \mathcal{Z}$. This can be stated more formally: Any partial generic function $g_0 : \mathcal{X}_0 \mapsto \mathcal{Y}_0$ satisfies the catenation $g_0 h_0 = f_0$, or, for any $z \in Z$

$$g_z : \mathcal{X}_z \mapsto \mathcal{Y}_z \Rightarrow g_z h_z = f_z. \tag{22.7}$$

From an algebraic point of view, there are no preferences for any of the partial generic functions g_{zj} that fulfil the catenation $g_{zj} h_z = f_z$ allowing us to choose the destinations $g_{zj}(\mathbf{x}) \in \mathcal{Y}$ independently from one-another and quite arbitrarily.

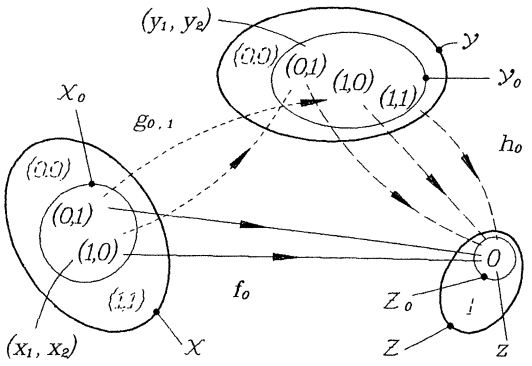


Fig. 22–5 Catenation diagram of the circuit of Fig. 22–2

22.4 Hinge Sets

By (22.7), the equivalents $\mathcal{Y}_z = \{\mathbf{y}|h(\mathbf{y}) = z\}$ of the hinge function h together with the equivalents $\mathcal{X}_z = \{\mathbf{x}|f(\mathbf{x}) = z\}$ of the specified function f define the partial generic functions g_z. To emphasise the importance of $\mathcal{Y}_z$ we give it a name, calling it a **hinge set** because it is defined by the hinge function h.

Hinge sets can easily be determined from the K-map of a given hinge function. *Those cells* $\mathbf{y}$ *which contain a* 0 *belong to the hinge set* $\mathcal{Y}_0$*, while those cells containing a* 1 *belong to hinge set* $\mathcal{Y}_1$. To illustrate, consider the K-maps of Fig. 22–6 which represent the hinge functions for the NOR gate and the normally closed valve used in the circuits of Fig. 22–1. From these K-maps we deduce for the NOR gate:

$$\mathcal{Y}_0 = \{(0,1),(1,0),(1,1)\},$$
$$\mathcal{Y}_1 = \{(0,0)\},$$

and for the NC-valve

$$\mathcal{Y}_0 = \{(0,0,0),(0,1,0),(1,1,0),(1,0,0),(1,0,1)\},$$
$$\mathcal{Y}_1 = \{(0,0,1),(0,1,1),(1,1,1)\}.$$

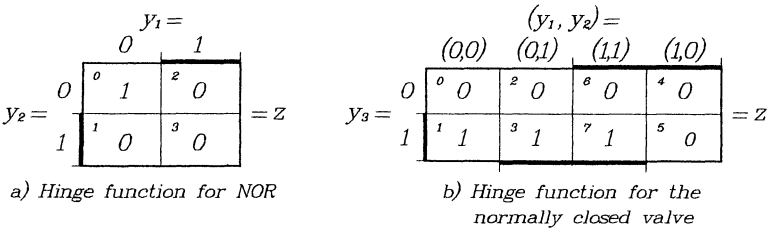


Fig. 22–6 K-maps of the hinge functions used in Fig. 22–1

22.5 Composing a Circuit: Example 1

The introductory example of this chapter, Fig. 22–1, shows how the EQUIVALENCE function can be realised using normally closed valves. We now take a detailed look at this example. The NC-valve is a realisation of a composite logic function of slightly enhanced complexity. What I want to show with this example is the use of a hinge function of above normal complexity. Not the valve is of interest, its logic function is. Nevertheless, Fig. 22–7 shows the

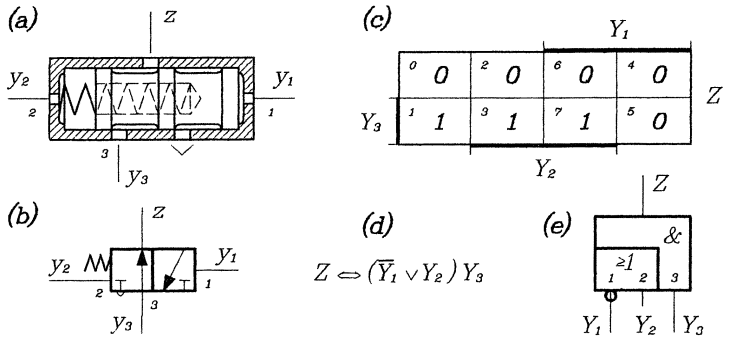


Fig. 22–7 The NC-valve, its K-map and composite logic symbol

valve, its symbol, and its input-output behaviour in the form of a K-map. This valve is a slight extension of the NC-valve discussed in Section 2.1. The K-map of Fig. 22–7c leads to the logic formula $Z \Leftrightarrow (\overline{Y}_1 \vee Y_2)Y_3$ and this to the composite logic symbol (the packed logic symbols) pictured in Fig. 22–7e.

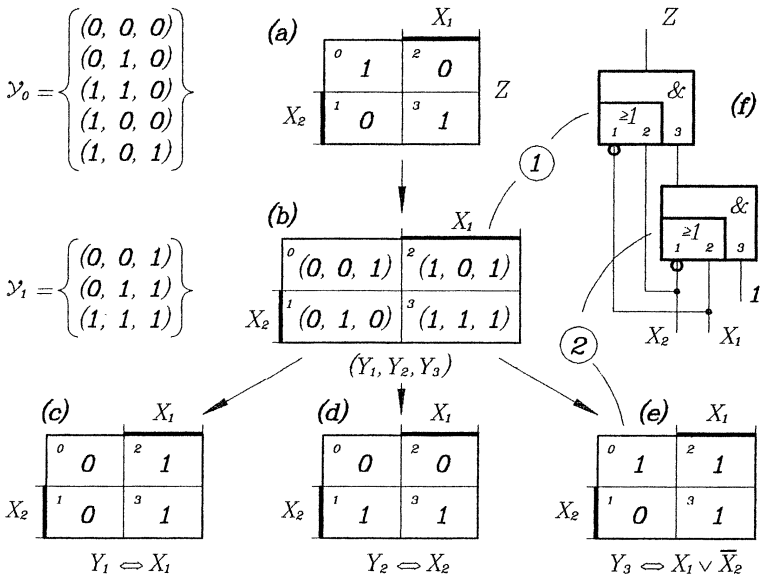


Fig. 22–8 From specified function (a) via $\mathcal{Y}_0$ and $\mathcal{Y}_1$ to circuit (f)

The process of composing the EQUIVALENCE on the basis of the hinge function of Fig. 22–7c is developed in Fig. 22–8. We start with the K-map of the function to be synthesised, Fig. 22–8a, and with the hinge sets $\mathcal{Y}_0$ and $\mathcal{Y}_1$. The **1**s in the K-map of Fig. 22–8a are replaced, independently from

one-another, by n-tuples of the hinge set $\mathcal{Y}_1$. Likewise, the 0s in the K-map of Fig. 22–8a are replaced, independently from one-another, by n-tuples of the hinge set $\mathcal{Y}_0$. By this replacement process we transform the K-map of Fig. 22–8a into that of Fig. 22–8b which represents the generic function. This triple-output K-map is split into the three K-maps of the last row. The K-maps assigned encircled numbers represent composite logic symbols allowing us to draw the circuit of Fig. 22–8f.

Actually, the above example was terminated prematurely. The processing of a K-map should continue unless the K-map represents an IDENTITY or a constant, i.e., FALSITY or TRUTH. The K-map of Fig. 22–8e represents an IMPLICATION, so, why terminate processing? Clearly, because our composite logic function can realise the IMPLICATION. Nevertheless, let us continue the process in a formal way where it was interrupted, using the opportunity to take a closer look at how one might choose n-tuples from the hinge sets $\mathcal{Y}_0$ and $\mathcal{Y}_1$.

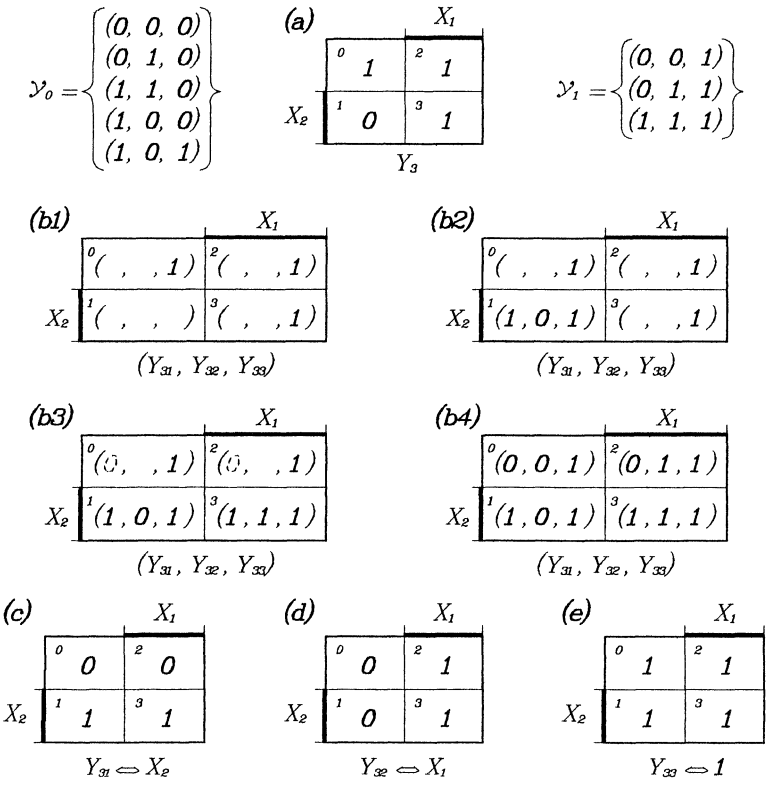


Fig. 22–9 Discussing the choice of n-tuples from $\mathcal{Y}_0$ and $\mathcal{Y}_1$

In Fig. 22–9 we continue developing the function specified in the K-map of Fig. 22–8e. In the following paragraph, (a), (b1),..., (e) refer to the thus specified sub-figures of Fig. 22–9. The composition procedure requires us to replace the **1**s of (a) by elements, i.e., ordered triples, of $\mathcal{Y}_1$. As each triple has **1** as its last element, we have no choice but to begin with a K-map for the generic function as the one shown in (b1). If we could place a **1** in the third position of cell number 1 of (b1), we would be able to express Y_{33} as TRUTH. The triple to be inscribed in cell 1 of (b1) must be taken from $\mathcal{Y}_0$ (why?). There is only one such triple which has **1** as a last element, $(\mathbf{1},\mathbf{0},\mathbf{1})$, which brings us to (b2). Now having a **1** in position 1 of cell 1, it would be advantageous to also have a **1** in position 1 of cell 3. We could then describe Y_{31} as an INDENTITY (i.e., $Y_{31} \Leftrightarrow X_2$) — assuming, of course, that we could place **0**s in the first positions of cell 0 and 2. These aspirations are expressed in (b3) where the second position of cell 3 by necessity becomes **1**. This leads to the desire to describe Y_{32} as an IDENTITY, i.e., as $Y_{32} \Leftrightarrow X_1$, which necessitates us to fill the K-map as shown in (b4). This triple-output K-map is then split into the single-output K-maps of the last row of Fig. 22–9.

22.6 Composing a Circuit: Example 2

Suppose we want to compose a circuit specified in Fig. 22–10a. As a hinge function let us accept either an OR or the composite function $Z \Leftrightarrow (\overline{Y}_1 \vee Y_2)Y_3$ of the NC-valve used in the previous section. After the explanations of that section, I leave you to pondering the possible solution shown in Fig. 22–10. The following calculation, which models the circuit of Fig. 22–10k, verifies the composed circuit because the result of the calculation can be read directly from the K-map of Fig. 22–10a.

$$
\begin{aligned}
Z &\Leftrightarrow (\overline{Y}_1 \vee Y_2)Y_3 \\
&\Leftrightarrow (\overline{X_1 \vee X_3} \vee (\overline{X}_1 \vee X_2)Y_{23})(\overline{X}_1 \vee X_3) \\
&\Leftrightarrow (\overline{X}_1\overline{X}_3 \vee (\overline{X}_1 \vee X_2)(X_1 \vee \overline{X}_2))(\overline{X}_1 \vee X_3) \\
&\Leftrightarrow (\overline{X}_1\overline{X_1\overline{X}_1}_{\mathbf{0}} \vee X_1X_2 \vee \overline{X}_1\overline{X}_2 \vee \underbrace{X_2\overline{X}_2}_{\mathbf{0}})(\overline{X}_1 \vee X_3) \\
&\Leftrightarrow \overline{X}_1\overline{X}_1\overline{X}_3 \vee \overline{X}_1X_1X_2 \vee \overline{X}_1\overline{X}_1\overline{X}_2 \vee \\
&\qquad \overline{X}_1\overline{X}_3X_3 \vee X_1X_2X_3 \vee \overline{X}_1\overline{X}_2X_3 \\
&\Leftrightarrow \overline{X}_1\overline{X}_3 \vee \overline{X}_1\overline{X}_2 \vee X_1X_2X_3 \vee \overline{X}_1\overline{X}_2X_3 \\
&\Leftrightarrow \overline{X}_1\overline{X}_3 \vee \overline{X}_1\overline{X}_2 \vee X_1X_2X_3
\end{aligned}
$$

22.7 Composing a Circuit: Example 3

To finish on a simpler note, you might like to see how to obtain the NOR circuit of the EQUIVALENCE shown in Fig. 22–1b. If so, follow the development laid out in Fig. 22–11.

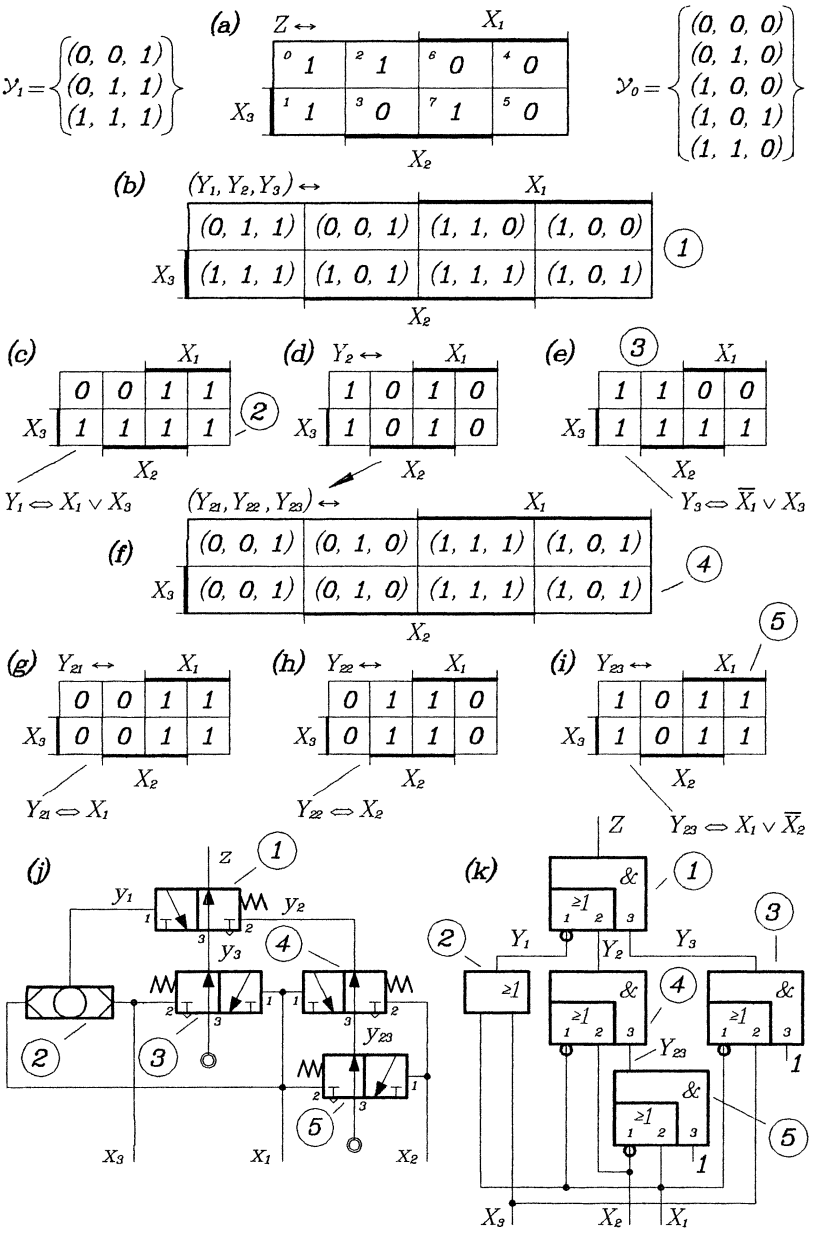


Fig. 22–10 Example 2

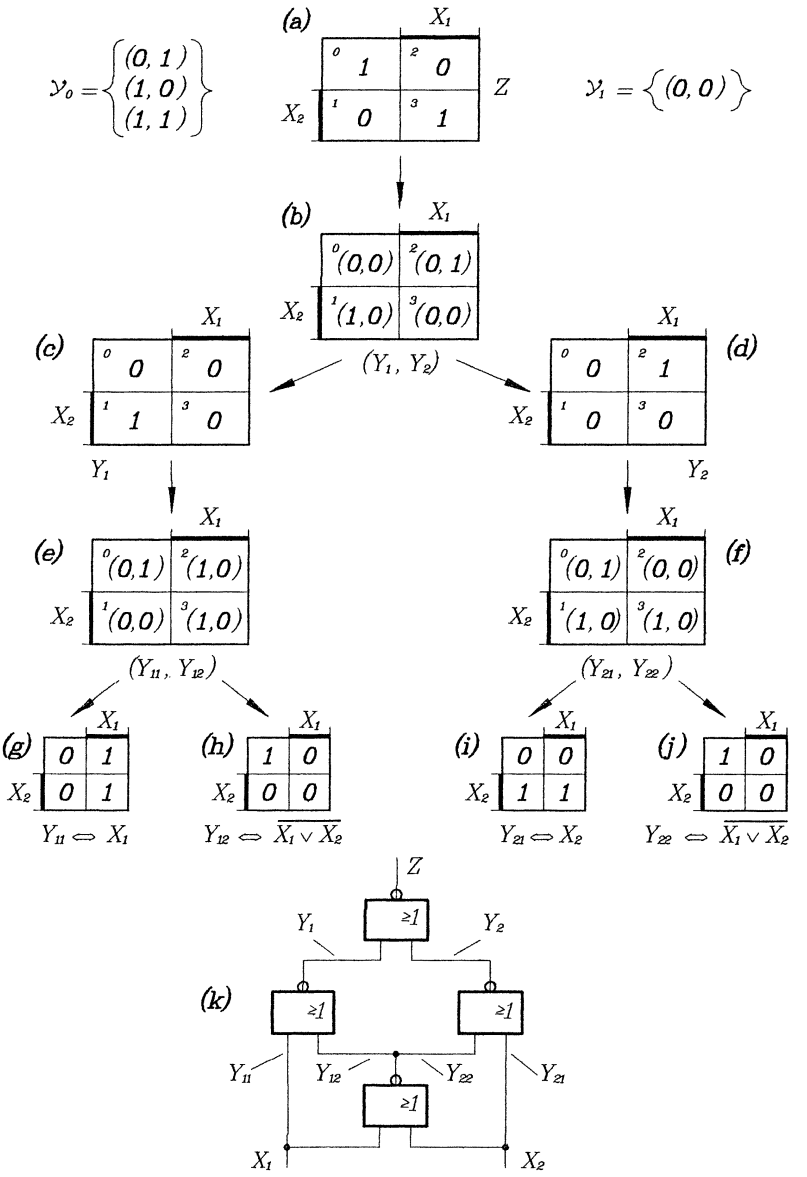


Fig. 22–11 Example 3

Chapter 23

Hazards

Contrary to *gates*, which are idealised switching devices, the *output* values of real-world switching devices are always *delayed* with respect to any changes in *input* values. Such delays can, under certain circumstances, cause illegitimate or unwanted — i.e., **spurious** — output values which by their very nature are of limited duration and are thus called **transient**. The *chance* or *danger* of a **transient spurious output-value** occurring is called a **hazard**.

transient spurious output-values are usually of little or no importance in combinational circuits when their outputs are used as inputs to actuators such as valves or motors. Due to the inertia of an actuator it cannot react to fleeting input fluctuations. But, whenever the outputs of combinational circuits are used as inputs to circuits with memorising abilities, such as latches and sequential circuits, a transient output value may unwantedly be memorised thus perpetuating an incorrect output value. Considering this danger, it is not surprising that it was **Huffman** [1957] who, after initiating the theory of sequential circuits, was the first to analyse the causes and effects of hazards.

23.1 Hazards on Stationary Signals

An a-equivalent $\mathcal{E}_a = \{x|f(x) = a\}$ (as introduced in Section 17.5), in which a is either 0 or 1, is of course the set of those input events x which the switching function f maps to the output value a. Changing the perspective slightly, we emphasise that the output value a, be it 0 or 1, remains *invariant* when switching from an input event $x \in \mathcal{E}_a$ to any other input event $x^* \in \mathcal{E}_a$ of the same equivalent. The output value a of the equivalent $\mathcal{E}_a$ is therefore said to be **stationary**[*)] or a **steady-state output-value**.

---

*) Referring to a as **static** seems inappropriate as *static* implies an unchanging position in *space*. **Stationary**, on the other hand, implies an unchanging value or property in *time*. The mutual antipode of *static* and *stationary* is **dynamic** — this referring equally to a changing position in space or to a changing value in time.

We only consider idealised forms of transient spurious output-values such as the **pulse** and **trough** of Fig. 23–1a,b. The summary expression for a *pulse* or a *trough* is **salient** which refers to a brief projection in an otherwise steady-state. In our idealised model, a salient begins when an input event $x \in \mathcal{E}_a$ changes to some other input event $x^* \in \mathcal{E}_a$ of the same equivalent. The salients last for only a relatively short period of time, say, for Δt. The chance or danger of a pulse occurring is called a **0-hazard**, while the chance or danger of a trough occurring is called a **1-hazard**.

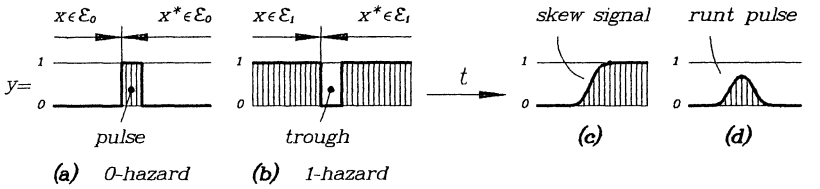


Fig. 23–1 Idealised description of salients

In reality, salients are comprised of *skew* signals, and often enough consist of *runt* pulses or troughs. A signal that does not 'jump', but 'creeps', from one value to the other is said to be **skew**. A skew signal that is forced to change its value prematurely, forms a **runt** (i.e., undersized and skewed) pulse or trough. The cause for skew signals are delays in switching devices as well as in transmission lines.

23.2 Single-Input Hazards

A **single-input hazard** (conventionally called a **static hazard**) is a hazard on a stationary output caused by a *single input-variable* changing its value. The methods that have been developed for finding and eliminating hazards in fact apply only to single-input hazards. These, therefore, are the only ones we consider.

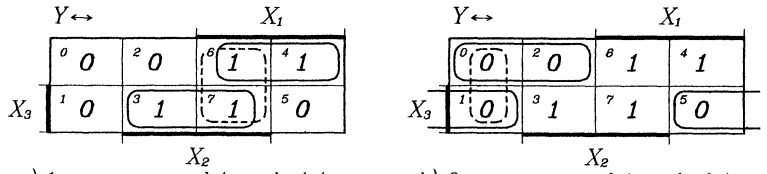


Fig. 23–2 Specifying a circuit to be analysed for hazards

To get a feeling for the mechanism that leads to a single-input hazard, let us design and analyse example circuits derived from the K-map of Fig. 23–2.

The minimal disjunctive and minimal conjunctive forms

$$Y \Leftrightarrow X_1\overline{X}_3 \vee X_2X_3, \qquad Y \Leftrightarrow (X_1 \vee X_3)(\overline{X}_3 \vee X_2),$$

are obtained from the minimal covers of Fig. 23–2a,b — i.e., omitting the loops drawn with dashed lines. The circuit for the minimal disjunctive form $Y \Leftrightarrow X_1\overline{X}_3 \vee X_2X_3$ is shown on Fig. 23–3. Before taking a detailed look at the circuit's behaviour, note that the expression $X_1\overline{X}_3$ has been realised by *'extracting'* the negation, i.e., by realising a NEGATION N ($\overline{X}_3 \Leftrightarrow Z$) the output of which is fed into the AND $A1$ (X_1Z). This gives us the possibility to consider the circuit's behaviour under the assumption that the output of the NEGATION is *delayed* with respect to its input. It is precisely this assumption on which we shall subsequently build the theory of analysing circuits for the occurrence of hazards.

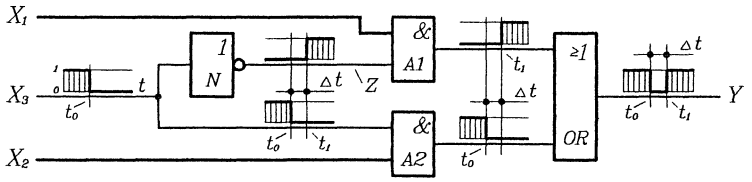


Fig. 23–3 Analysing the minimal disjunctive form

The analysis shown on Fig. 23–3 depicts the case in which input event $7 = (1,1,1)$ switches to input event $6 = (1,1,0)$. The thick lines for X_1 and X_2 are meant to indicate that the value of these two variables is **1** before and after the transition from input event 7 to 6. Let us refer to the instant in which the transition takes place as t_0. Then, a short time interval Δt later, at t_1, the output of the NEGATION N will have responded by switching to **1**. X_3 and its delayed and inverted form, Z, propagate through the circuit as indicated in the figure, causing a trough (or at least the danger of a trough, i.e., a 1-hazard) at the output Y during t_0 to t_1.

It is important to realise that, in the model of the circuit of Fig. 23–3, a transition from $6 = (1,1,0)$ to $7 = (1,1,1)$ will *not* cause a 1-hazard. I strongly encourage you to draw the signal transitions for this case to see the truth of the statement.

There are other single-input transitions that can occur within $\mathcal{E}_1$ — between 4 and 6, and between 3 and 7. When switching between $4 = (1,0,0)$ and $6 = (1,1,0)$, only X_2 is required to change its value. A pondering glance at the circuit of Fig. 23–3 will convince you that during a transition from 4 to 6, or back, the output of $A1$ and thus of the output Y remain unwaveringly at **1** without the slightest danger of a trough. Similarly, no hazard exists when switching between 3 and 7 because the output of $A2$ is permanently **1**.

The insight obtained above in analysing the transitions 4 and 6, and 3 and 7 shows that **we can avoid 1-hazards in disjunctive forms for**

those single-input transitions that are covered by K-sets. In our example, we would add a third AND gate *A3* for the realisation of X_1X_2 feeding its output into the OR gate. The AND gate *A3* effectively eliminates the 1-hazard when switching between 7 and 6.

The next step in our analysis is to check for single-input 0-hazards in the circuit of Fig. 23–3. This means we must check the transitions between $0 = (0,0,0)$ and $1 = (0,0,1)$, $0 = (0,0,0)$ and $2 = (0,1,0)$, and $1 = (0,0,1)$ and $5 = (0,1,1)$. In all these cases, the outputs of *A1* and *A2* remain unwaveringly at **0** so that the circuit is free of 0-hazard. We shall pick up this train of thought in a subsequent section.

You should now have no difficulty in developing the same kind of analysis for the conjunctive form $Y \Leftrightarrow (X_1 \vee X_3)(\overline{X}_3 \vee X_2)$ read from Fig. 23–2b.

23.3 Gates and Delays

Gates are graphic symbols for logic connectives and as such show no delay in transmitting a signal from a gate's inputs to its output. Real-world switching devices do show such a delay. It is customary to model a switching device by placing an idealised delay element at the *output* of an appropriate gate or logic circuit, as pictured in Fig. 23–4, thus averaging the delays of the inputs.

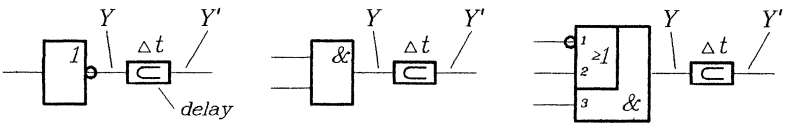


Fig. 23–4 Delays are always placed at the output of a gate

Brzozowski & **Seger** [1994] discuss a number of delay models of which we consider only the **fixed inertial delay-model**. Its basic properties are visualised in Fig. 23–5.

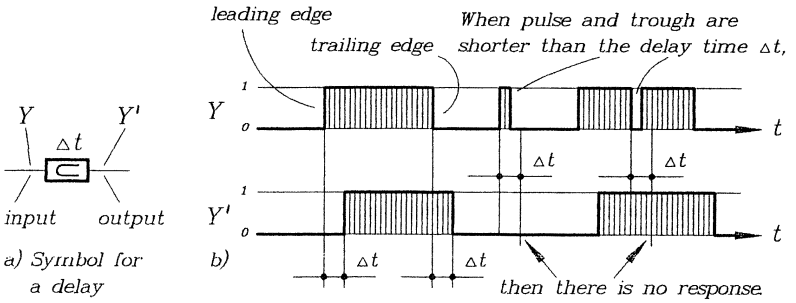


Fig. 23–5 Characterising the fixed inertial delay-model

In the *fixed* inertial delay-model the leading and trailing edges of an input signal are delayed equally by the constant delay-time Δt. *Inertial* refers to pulses and troughs shorter than the delay time Δt having no effect on the output. In this respect you can view a delay element as an *energy buffer* compensating brief and/or runt salients.

Any **primed logic variable** X' is the delayed version of the Unprimed variable X. In other words, if X is the input to a delay element, then X' denotes the delay's output. Considering a NEGATION with input X, how should we write the output: as $(\overline{X})'$, or as $\overline{(X')}$? The former expression, $(\overline{X})'$, refers to the delay element being placed at the output of the NEGATION, while $\overline{(X')}$ refers to the delay element being placed at the input of the NEGATION. As both models behave identically, which is expressed as $(\overline{X})' \Leftrightarrow \overline{(X')}$, we shall simply write $\overline{X}'$ for either $(\overline{X})'$ or $\overline{(X')}$:

$$\overline{X}' \stackrel{df}{\Leftrightarrow} (\overline{X})', \qquad \overline{X}' \Leftrightarrow \overline{(X')}. \tag{23.1}$$

23.4 Detecting Single-Input Hazards

Any theory of single-input hazards (however limited its scope) is based on a model used to explain the hazard's occurrence. Such a model is pictured in Fig. 23–6 and is expressed by the following

(23.2) Postulate on the occurrence of single-input hazards

(a) A **0-hazard** occurs at the output of an AND gate the inputs of which are (or may be reduced to) an input signal X and its delayed and inverted form $\overline{X}'$.

(b) A **1-hazard** occurs at the output of an OR gate the inputs of which are (or may be reduced to) an input signal X and its delayed and inverted form $\overline{X}'$.

Note that this postulate states the *only* cause we are willing to accept for the occurrence of hazards. It is due to this stringent restriction that we are able to develop a theory determining whether a circuit develops hazards. Let me also point out that X_i and $\overline{X}'_i$ are independent of one another, so that neither $X_i\overline{X}'_i \leftrightarrow \mathbf{0}$ nor $X_i \vee \overline{X}'_i \leftrightarrow \mathbf{1}$ are tautologies, meaning that these formulas do not hold for all t.

Actually, it is quite straight forward to apply the above postulate to the output Y of a switching function $f : \{0,1\}^n \mapsto \{0,1\}$. To this end, consider the multiple input-event

$$\mathbf{e}_{X_i} \stackrel{df}{=} (e_1, \ldots, e_{i-1}, x_i, e_{i+1}, \ldots, e_n) \tag{23.3}$$

which stands for the two adjacent individual events

$$(e_1, \ldots, e_{i-1}, 0, e_{i+1}, \ldots, e_n), \quad \text{and} \quad (e_1, \ldots, e_{i-1}, 1, e_{i+1}, \ldots, e_n).$$

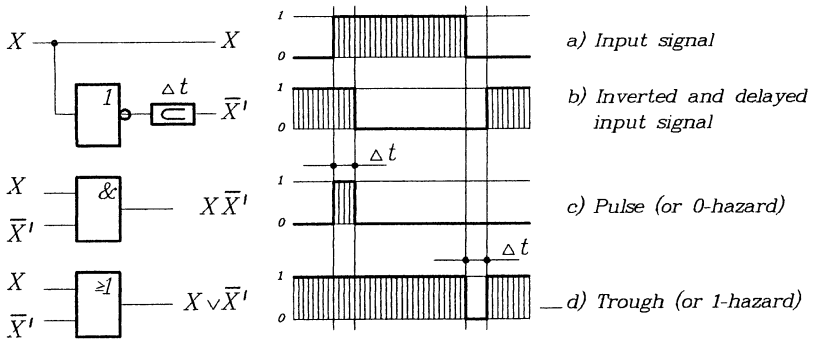


Fig. 23–6 The model explaining single-input hazards

in which each $e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n$ is a constant, either 0 or 1. The individual value we choose for each of the $e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n$ is, at the moment, not up for debate. With the multiple input-event (23.3) we reformulate (23.2) to be read as:

(23.4) Theorem on the existence of single-input hazards in a logic formula and its circuit. Given a switching function $f : \{0,1\}^n \mapsto \{0,1\}$, let $Y(\mathbf{x})$ refer to its logic formula.

(a) The logic formula $Y(\mathbf{x})$ and its associated switching circuit have a single-input **0-hazard** *iff* a multiple input-event $\mathbf{e}_{X_i}$ exists for which the logic formula $Y(\mathbf{x})$ reduces to $Y(\mathbf{e}_{X_i}) = X_i\overline{X}_i'$.

(b) The logic formula $Y(\mathbf{x})$ and its associated switching circuit have a single-input **1-hazard** *iff* a multiple input-event $\mathbf{e}_{X_i}$ exists for which the logic formula $Y(\mathbf{x})$ reduces to $Y(\mathbf{e}_{X_i}) = X_i \vee \overline{X}_i'$.

Next, consider the following theorem which halves the burden of calculation when searching for single-input hazards in two-level circuits:

(23.5) Theorem on single-input hazards in two-level circuits.

(a) A **disjunctive** formula or circuit, as would be obtained from the evaluation formula

$$Y(\mathbf{x}) \overset{(17.30a)}{\Longleftrightarrow} \bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_1)} C_\varphi(\mathbf{x}),$$

is always free of single-input 0-hazards (whether the formula or circuit has a 1-hazard or not).

(b) A **conjunctive** formula or circuit, as would be obtained from the evaluation formula

$$Y(\mathbf{x}) \overset{(17.31b)}{\Longleftrightarrow} \bigwedge_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_0)} D_\varphi(\mathbf{x}),$$

is always free of single-input 1-hazards (whether the formula or circuit has a 0-hazard or not).

Let us prove (23.5a). According to (23.4a) the formula $Y(\mathbf{e}_{X_i})$ or its circuit can only have a 0-hazard if $Y(\mathbf{e}_{X_i})$ can be reduced to $X_i\overline{X}_i'$ for some multiple input-event $(e_1, \ldots, e_{i-1}, x_i, e_{i+1}, \ldots, e_n)$. Such a hazard-causing expression $X_i\overline{X}_i'$ would have to come from one of the generalised minterms, which is impossible because no generalised minterm contains a variable *and* its negation. Thus, no disjunctive form can be reduced to $X_i\overline{X}_i'$ hereby proving (23.5a).

(23.6) Theorem on hazard-free circuits. A disjunctive or conjunctive formula or circuit is completely free of single-input hazards *iff* it is developed using a full cover $\mathcal{P}_{max}(\mathcal{E}_1)$ or $\mathcal{P}_{max}(\mathcal{E}_0)$, whichever is appropriate when applying one of the evaluation formulas (17.30) or (17.31).

By using *full covers*, the above theorem (proved in **Brzozowski & Yoeli** [1976]), allows us to design two-level disjunctive or conjunctive circuits that are completely free of single-input hazards — thus *qualifying the importance of minimisation.*

23.5 Hazard-Detecting Formulas

Theorem (23.6) does not help us decide whether a (multilevel) circuit, designed by the methods put forth in Section 20.4, or in Chapters 21 and 22, is free of single-input hazards or not. A general method for detecting single-input hazards is based on the analysis of what I choose to call a *hazard-detecting formula.* The hazard-detecting formula is obtained in its three guises $Y_H(\mathbf{x})$, $Y_{HD}(\mathbf{x})$, $Y_{HC}(\mathbf{x})$ as explained in the following constructive definition.

(23.7) From the logic formula $Y(\mathbf{x})$ of a switching function you obtain various forms of the associated **hazard-detecting formula** $Y_H(\mathbf{x})$, $Y_{HC}(\mathbf{x})$, $Y_{HD}(\mathbf{x})$ in the following steps:

(a) Using *only* DeMorgan, and double negation, $\overline{\overline{P(\mathbf{x})}} \Leftrightarrow P(\mathbf{x})$, transform any negations (if necessary) so that they are restricted to single variables.

(b) To obtain the **hazard-detecting formula** $Y_H(\mathbf{x})$, write each inverted variable $\overline{X}$ in its delayed form $\overline{X}'$.

(c) From $Y_H(\mathbf{x})$, you obtain the hazard-detecting formula in its **disjunctive form** — denoting it as $Y_{HD}(\mathbf{x})$ — by employing logic theorems appropriately to obtain a *disjunction of pure conjunctions.*

(d) From $Y_H(\mathbf{x})$, you obtain the hazard-detecting formula in its **conjunctive form** — denoting it as $Y_{HC}(\mathbf{x})$ — by employing

logic theorems appropriately to obtain a *conjunction of pure disjunctions.*

We make practical use of the disjunctive and conjunctive hazard-detecting formulas $Y_{HD}(\mathbf{x})$ and $Y_{HC}(\mathbf{x})$ by way of the following

(23.8) Theorem on hazard-equivalent formulas.

(a) The formulas $Y(\mathbf{x})$ and $Y_{HD}(\mathbf{x})$ behave identically with respect to **0**-hazards.

(b) The formulas $Y(\mathbf{x})$ and $Y_{HC}(\mathbf{x})$ behave identically with respect to **1**-hazards.

By the above theorem (see **Brzozowski & Yoeli** [1976]) we can, according to (23.4), use $Y_{HC}(\mathbf{x})$ to determine **1**-hazards, and $Y_{HD}(\mathbf{x})$ to determine **0**-hazards. But contrary to disjunctive and conjunctive forms, for which (23.5) holds, arbitrary multilevel circuits *can have* **0**- *as well as* **1**-hazards.

23.6 Hazards in a NOT-AND Circuit

Let us use the circuit of Fig. 23–7b as an example circuit by which to demonstrate how to detect hazards in a multilevel circuit.

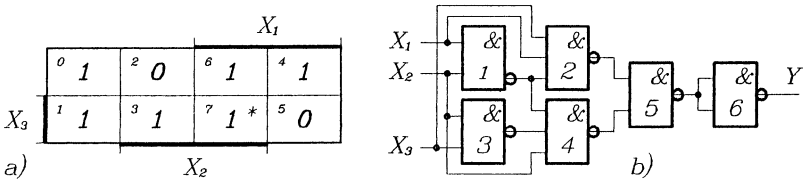


Fig. 23–7 A NOT-AND circuit developed in Fig. 21–5

To analyse the circuit, we cannot develop the logic formula as a disjunctive or conjunctive form derived from the K-map of Fig. 23–7a specifying the circuit. Instead, we must read the logic formula directly from the circuit of Fig. 23–7b:

$$Y \Leftrightarrow \overline{\overline{(\overline{X_1X_2}\; X_1X_3)}\,\overline{(\overline{X_2X_3}\; X_2\; \overline{X_1X_2})}}. \tag{23.9}$$

We now apply double negation, $\overline{\overline{A}} \Leftrightarrow A$ and DeMorgan, as required in (23.7a), to restrict the negations to single variables:

$$Y \Leftrightarrow (X_1X_2 \vee \overline{X}_1 \vee \overline{X}_3)(X_2X_3 \vee \overline{X}_2 \vee X_1X_2). \tag{23.10}$$

By writing any inverted variable $\overline{X}_i$ in its delayed form, $\overline{X}_i'$, we obtain the *hazard-detecting formula*

$$Y_H \Leftrightarrow (\underbrace{X_1}_{A}\,\underbrace{X_2}_{B} \vee \underbrace{\overline{X}_1' \vee \overline{X}_3'}_{C})(\underbrace{X_2X_3}_{} \vee \overline{X}_2' \vee \underbrace{X_1X_2}_{}) \tag{23.11}$$

according to (23.7b). Now, applying $AB \vee C \Leftrightarrow (A \vee C)(B \vee C)$, and $X_2X_3 \vee X_1X_2 \Leftrightarrow X_2(X_1 \vee X_3)$, we get

$$Y_H \Leftrightarrow (X_1 \vee \overline{X}_1' \vee \overline{X}_3')(\overline{X}_1' \vee X_2 \vee \overline{X}_3')(X_2(X_1 \vee X_3) \vee \overline{X}_2').$$

Transforming this into the conjunctive hazard-detecting formula,

$$Y_{HC} \Leftrightarrow (X_1 \vee \overline{X}_1' \vee \overline{X}_3')(\overline{X}_1' \vee X_2 \vee \overline{X}_3')(X_2 \vee \overline{X}_2')(X_1 \vee \overline{X}_2' \vee X_3), \quad (23.12)$$

allows us to detect **1**-hazards. Let us first look for a multiple input-event (x_1, e_2, e_3) that will reduce (23.12) to $Y_{HC} \Leftrightarrow X_1 \vee \overline{X}_1'$. The notation (x_1, e_2, e_3) refers to the variables x_2 and x_3 being **stable**, meaning that their values e_2 and e_3 remain unchanged when x_1 changes its value. For a stable variable X the value of $\overline{X}'$ is that of $\overline{X}$, allowing us to *drop the primes for stable variables* (note that $X_2 \vee \overline{X}_2 \Leftrightarrow \mathbf{1}$):

$$Y_{HC}(x_1, e_2, e_3) \Leftrightarrow (X_1 \vee \overline{X}_1' \vee \overline{X}_3)(\overline{X}_1' \vee X_2 \vee \overline{X}_3)(X_2 \vee \overline{X}_2)(X_1 \vee \overline{X}_2 \vee X_3)$$

$$Y_{HC}(x_1, e_2, e_3) \Leftrightarrow (X_1 \vee \overline{X}_1' \vee \overline{X}_3)(\overline{X}_1' \vee X_2 \vee \overline{X}_3)(X_1 \vee \overline{X}_2 \vee X_3).$$

To be able to obtain $Y_{HC}(x_1, e_2, e_3) \Leftrightarrow X_1 \vee \overline{X}_1'$ from the above formula, we must choose $e_2 \hookleftarrow 1$ and $e_3 \hookleftarrow 1$. Then, according to (23.8b), the circuit in Fig. 23–7b and its logic formula (23.9) have a **1**-hazard for the multiple input-event $(X_1, \mathbf{1}, \mathbf{1})$. An analogous analysis for X_2 leads to

$$Y_{HC}(1, x_2, 0) \Leftrightarrow Y_{HC}(0, x_2, 1) \Leftrightarrow X_2 \vee \overline{X}_2',$$

showing that there are two further multiple input-events, $(\mathbf{1}, X_2, \mathbf{0})$ and $(\mathbf{0}, X_2, \mathbf{1})$, for which (23.9) has a **1**-hazard. But, there is *no* **1**-hazard for a multiple input-event (e_1, e_2, x_3), as you can immediately see, because none of the disjunctions of (23.12) contain the expression $X_3 \vee \overline{X}_3'$.

To detect **0**-hazards *systematically*, it is advantageous to use the *disjunctive* hazard-detecting formula. This we can develop from

$$(23.11) \qquad Y_H \Leftrightarrow (X_1X_2 \vee \overline{X}_1' \vee \overline{X}_3')(X_2X_3 \vee \overline{X}_2' \vee X_1X_2)$$

by distributing AND over OR:

$$\begin{aligned} Y_{HD} \Leftrightarrow & X_1X_2X_3 \vee \overline{X}_1'X_2X_3 \vee X_2X_3\overline{X}_3' \\ & \vee X_1X_2\overline{X}_2' \vee \overline{X}_1'\overline{X}_2' \vee \overline{X}_2'\overline{X}_3' \\ & \vee X_1X_2 \vee X_1\overline{X}_1'X_2 \vee X_1X_2\overline{X}_3'. \end{aligned} \quad (23.13)$$

To see if a multiple input-event (x_1, e_2, e_3) can cause a **0**-hazard, we have to find a conjunction in (23.13) containing $X_1\overline{X}_1'$. The penult $X_1\overline{X}_1'X_2$

is such a conjunction. Thus, if (x_1, e_2, e_3) causes a hazard, it is caused by $(x_1, \mathbf{1}, e_3)$, for which (23.13) reduces to

$$Y_{HD}(x_1, 1, e_3) \Leftrightarrow (X_1X_3 \vee \overline{X}_1'X_3 \vee \mathbf{0}) \vee (\mathbf{0} \vee \mathbf{0} \vee \mathbf{0}) \vee (X_1 \vee X_1\overline{X}_1' \vee X_1\overline{X}_3).$$

Now testing this formula for both possible values of e_3, we get:

$$Y_{HD}(x_1, 1, 0) \Leftrightarrow X_1 \vee X_1\overline{X}_1' \Leftrightarrow X_1, \qquad \ldots \text{no hazard}$$
$$Y_{HD}(x_1, 1, 1) \Leftrightarrow X_1 \vee \overline{X}_1' \vee X_1 \vee X_1\overline{X}_1' \Leftrightarrow X_1\overline{X}_1'. \qquad \ldots \text{a } \mathbf{0}\text{-hazard}$$

No other **0**-hazards exist in (23.13), as you are invited to verify:

$$(Y_{HD}(1, x_2, 0) \Leftrightarrow X_2 \vee \overline{X}_2'), \qquad Y_{HD}(1, x_2, 1) \Leftrightarrow X_2 \vee X_2\overline{X}_2' \Leftrightarrow X_2.$$
$$Y_{HD}(1, 1, x_3) \Leftrightarrow \mathbf{1}, \qquad Y_{HD}(0, 1, x_3) \Leftrightarrow X_3 \vee X_3\overline{X}_3' \Leftrightarrow X_3.$$

23.7 Hazards in a Composed Circuit

As the results of the example of the last section were not encouraging, let us look for hazards in another multilevel circuit, that of Fig. 23–8.

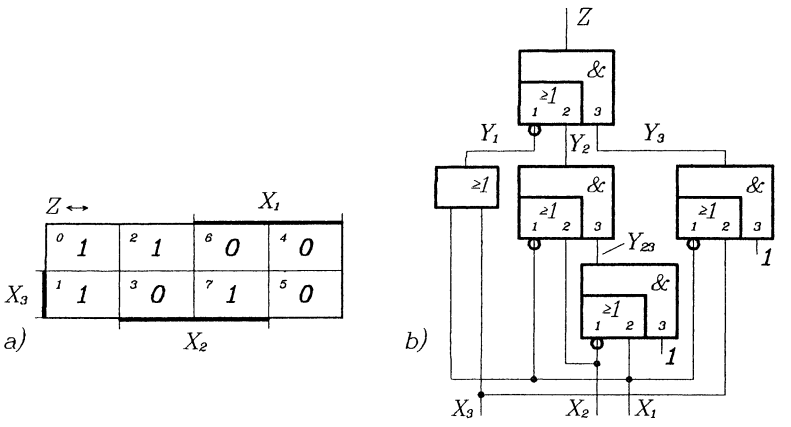


Fig. 23–8 The circuit composed in Section 22.6

After having been quite explicit in the explanations of the previous section, forgive me for merely presenting the results of the hazard analysis by stating the following formulas. These formulas are of course only meant as a guide for your own paper and pencil calculation. As you can see, this circuit, interestingly, has no hazards.

You first write the formula associated with the given circuit,

$$Z(x_1, x_2, x_3) \Leftrightarrow \big(\overline{X_1 \vee X_3} \vee (\overline{X}_1 \vee X_2)(X_1 \vee \overline{X}_2)\big)(\overline{X}_1 \vee X_3),$$

transforming it into its hazard-detecting form according to (23.7ab),

$$Z_H(x_1, x_2, x_3) \Leftrightarrow \big(\overline{X}'_1\overline{X}'_3 \vee (\overline{X}'_1 \vee X_2)(X_1 \vee \overline{X}'_2)\big)(\overline{X}'_1 \vee X_3).$$

The **1**-hazards are determined from the conjunctive hazard-detecting formulas Z_{HC} developed from Z_H as stated in (23.7d),

$$Z_{HC}(x_1, x_2, x_3) \Leftrightarrow (\overline{X}'_1 \vee X_2)(\overline{X}'_1 \vee X_1 \vee \overline{X}'_2)(X_1 \vee \overline{X}'_2 \vee \overline{X}'_3)(\overline{X}'_1 \vee X_3).$$

As demonstrated in the previous section, we let only one input variable at a time change its value, giving the other variables *stable* values:

$$Z_{HC}(x_1, 1, 0) \Leftrightarrow (\overline{X}'_1 \vee X_1)(X_1 \vee \mathbf{1})(\overline{X}'_1 \vee \mathbf{0}) \Leftrightarrow \overline{X}'_1.$$
$$Z_{HC}(x_1, 1, 1) \Leftrightarrow (\overline{X}'_1 \vee X_1)(X_1 \vee \mathbf{0})(\overline{X}'_1 \vee \mathbf{1}) \Leftrightarrow X_1.$$

To determine the **0**-hazards, you develop the disjunctive hazard-detecting formulas Z_{HD} from Z_H as stated in (23.7c),

$$Z_{HD}(x_1, x_2, x_3) \Leftrightarrow \overline{X}'_1\overline{X}'_3 \vee \overline{X}'_1X_1 \vee \overline{X}'_1\overline{X}'_2 \vee \overline{X}'_1X_2\overline{X}'_2 \vee X_1X_2X_3 \vee X_2\overline{X}'_2X_3.$$

Here too, letting one input variable after another vary, allows you to determine any hazard.

$$Z_{HD}(x_1, 0, 0) \Leftrightarrow \overline{X}'_1 \vee \overline{X}'_1X_1 \vee \overline{X}'_1 \vee \mathbf{0} \vee \mathbf{0} \vee \mathbf{0} \Leftrightarrow \overline{X}'_1,$$
$$Z_{HD}(x_1, 0, 1) \Leftrightarrow \mathbf{0} \vee \overline{X}'_1X_1 \vee \overline{X}'_1 \vee \mathbf{0} \vee \mathbf{0} \vee \mathbf{0} \Leftrightarrow \overline{X}'_1,$$
$$Z_{HD}(x_1, 1, 0) \Leftrightarrow \overline{X}'_1 \vee \overline{X}'_1X_1 \vee \mathbf{0} \vee \mathbf{0} \vee \mathbf{0} \vee \mathbf{0} \Leftrightarrow \overline{X}'_1,$$
$$Z_{HD}(x_1, 1, 1) \Leftrightarrow \mathbf{0} \vee \overline{X}'_1X_1 \vee \mathbf{0} \vee \mathbf{0} \vee X_1 \vee \mathbf{0} \Leftrightarrow X_1.$$

$$Z_{HD}(0, x_2, 0) \Leftrightarrow \mathbf{1} \vee \mathbf{0} \vee \overline{X}'_2 \vee X_2\overline{X}'_2 \vee \mathbf{0} \vee \mathbf{0} \Leftrightarrow \mathbf{1},$$
$$Z_{HD}(0, x_2, 1) \Leftrightarrow \mathbf{0} \vee \mathbf{0} \vee \overline{X}'_2 \vee X_2\overline{X}'_2 \vee \mathbf{0} \vee X_2\overline{X}'_2 \Leftrightarrow \overline{X}'_2,$$
$$Z_{HD}(1, x_2, 1) \Leftrightarrow \mathbf{0} \vee \mathbf{0} \vee \mathbf{0} \vee \mathbf{0} \vee X_2 \vee X_2\overline{X}'_2 \Leftrightarrow X_2,$$

As there are no multiple input-events $\mathbf{e}_{X_i}$ for which $Y_{HC}(\mathbf{e}_{X_i})$ reduces to $X_i \vee \overline{X}'_i$, or $Y_{HD}(\mathbf{e}_{X_i})$ to $X_i\overline{X}'_i$, the given circuit of Fig. 23–8b and its formula Y are completely free of hazards.

Division Four

Latches

24 Memorising by Feedback 255 (12 pages)
25 Basic Theory of Latches 267 (12 pages)
26 Optimised Latches 279 (9 pages)
27 Elementary Latches 288 (12 pages)
28 Composition of Latches 300 (8 pages)

There are two kinds of circuits, *combinational* and *sequential*, the former having no memorising ability, the latter having such an ability to various degrees. Those sequential circuits with the least memorising ability are called *latches*.

There are two good reasons to consider latches in depth. Firstly, memorisation is a phenomenon apart from everything we have encountered in combinational circuits. So, if we want to understand how memorisation works, and how to describe it mathematically, we are well advised to examine it in its most elementary form—the latch. Secondly, latches play a similar role in building sequential circuits to that of the elementary logic functions in building combinational circuits. That is, the memory of a sequential circuit consists of (multiple) latches that work in unison. In this respect, latches, together with elementary logic functions, are the building blocks of sequential circuits.

Chapter 24 is an introduction not only intended for the novice, for it is the quintessence of this chapter that enables us to formulate the *definition of a latch* in Chapter 25, a chapter containing an essential minimum of latch theory. Chapter 26 takes a more in-depth view on how to optimise latches with feedback, concentrating on a hazard-free design. In Chapter 27 a system of symbols, analogous to gates, is developed to represent all 28 (or 36, if you wish) elementary latches. This will be of little interest to most readers, and can simply be skipped. The last chapter, Chapter 28, on how to design a latch on the basis of a given (elementary) latch develops an important technique to be used with sequential circuits.

Chapter 24

Memorising by Feedback

Latches are usually introduced as *cross-coupled* NOR and NAND gates (such as the circuits shown in Figs. 27–10 and 27–11). These circuits are, at best, an approximation of a general principle of *memorisation.* To make such a general principle theoretically and practically applicable it must be formulated as a function. This we do in the next chapter where we define it as a function we call **memory**. In the present chapter we collect the information that allows us to formulate the definition of memory, information necessary to get an intuitive grasp of the concept of memory as used in switching systems. In this presentation, the word **latch** refers to a technical or logical realisation of a specific memory (i.e., a specific function).

24.1 What is Memorisation?

A memory has a quality not present in a (combinational) switching function: *At least one input event exists for which the output value is sometimes 1 and sometimes 0.* This statement is rather ambiguous, for, it does not mean that while an input event of the stated property is stationary, the output value is allowed to switch back and forth between 0 and 1 (if it could, it would not be *well behaved*). Nor does it mean that it is uncertain or even unpredictable which of the output values, 0 or 1, will be present. This is discussed for a typical sample memory the transitional behaviour of which is shown in the diagram of successive events (DSE) of Fig. 24–1.

First of all, you will notice that the DSE of Fig. 24–1 depicts a *well behaved* circuit, i.e., one whose output will only change if an input event changes. Next, to the details. The input event (1,0) — for which we can conveniently write the minterm it defines: $(x_1, x_2) = (1, 0) \Leftrightarrow (x_1 = 1) \wedge (x_2 = 0) \Leftrightarrow X_1\overline{X}_2$ — occurs only once (during the time interval T_1), as does the input event $\overline{X}_1X_2$ (during T_3). *If an input event occurs only once in a DSE, it is assumed that the output value assigned to such an input event would be the same whenever the input event were to occur again.* It is customary to call an input event that always causes the output to be 1 a **setting input event**, while a **resetting input event** always causes the output to be 0.

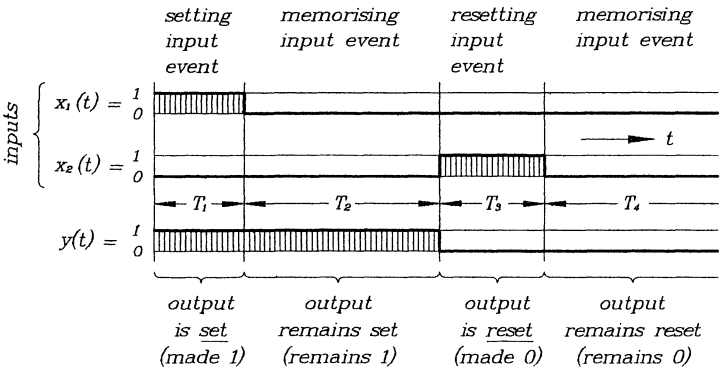


Fig. 24–1 The basic behaviour of a sample memory

The input event $\overline{X}_1\overline{X}_2$, on the other hand, occurs multiple times (during T_2, and T_4), *the associated output values varying.* Nevertheless, we can state unequivocally when the input event $\overline{X}_1\overline{X}_2$ causes the output to be 1, and when it causes the output to be 0.

To find the rule by which the output value is 1 during T_2, and 0 during T_4, compare the output values during these time intervals with the output values occurring during the (immediately) preceding input events, i.e., with those present during T_1 and T_3, respectively. The mutual explanation for the differing output values is: *The output value incurred by the input event $\overline{X}_1\overline{X}_2$ is identical with the output value caused by the immediately preceding input event, whichever input event may have been the preceding one.* As the input event $\overline{X}_1\overline{X}_2$ maintains the previous output value, it is said to be a **memorising input event.**

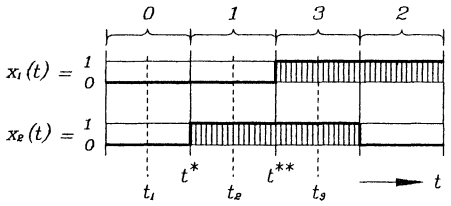


Fig. 24–2 Picturing the sequence (0,1,3,2) of input events $x(t)$

To facilitate further explanations, we introduce the concept of *successive input events.* Consider the arbitrary sequence (0,1,3,2) of input events $x(t)$ pictured in Fig. 24–2, written as the decimal equivalents of $\big(x_1(t), x_2(t)\big)$, with $x_1(t), x_2(t) \in \{0, 1\}$. Let us agree to view the input events 0 and 1, as well as 1 and 3, or 3 and 2 in the above sequence (0,1,3,2) as being *successive.* But in contrast, let us take the input events (0,3), (0,2) and (1,2) of the sequence

(0,1,3,2) *not to be* successive. The concept behind this example is formalised in the following

(24.1) **Definition of successive input events** $\mathbf{x}(t_1)$ and $\mathbf{x}(t_2)$ in which $t_1 < t_2$ and $\mathbf{x}(t) = \big(x_1(t), \ldots, x_n(t)\big)$:
(a) If $\mathbf{x}(t_1) \neq \mathbf{x}(t_2)$, then there exists *exactly one* instant t^* so that

$$\mathbf{x}(t) = \begin{cases} \mathbf{x}(t_1) & \text{for } t_1 \leq t < t^*, \\ \mathbf{x}(t_2) & \text{for } t^* \leq t \leq t_2. \end{cases}$$

(b) If $\mathbf{x}(t_1) = \mathbf{x}(t_2)$, then $\mathbf{x}(t) = \mathbf{x}(t_1)$ for an arbitrary t for which $t_1 \leq t \leq t_2$.

$\mathbf{x}(t_1)$ is called the **previous**, $\mathbf{x}(t_2)$ the **present** input event[1)], and t^* the **instant of transition**.

Characteristic for a memory is that a memorising input event $\mathbf{x}(t)$ will cause the output $y(t)$ to take on a value according to the

(24.2) **Postulate of memorisation and stationarity**. Assume $\mathbf{x}(t)$ to be a memorising input event, and $y(t)$ to be the output it invokes.
(a) Given successive and *distinct* input events $\mathbf{x}(t - \Delta t)$ and $\mathbf{x}(t)$, **memorisation** ensures that the present output $y(t)$ of a memory (or latch) will equal the value $y(t - \Delta t)$ caused by the previous input event $\mathbf{x}(t - \Delta t)$:

$$\mathbf{x}(t) \neq \mathbf{x}(t - \Delta t) \Rightarrow y(t) = y(t - \Delta t).$$

(b) Given successive and *equal* input events $\mathbf{x}(t - \Delta t)$ and $\mathbf{x}(t)$, **stationarity** requires that the present output $y(t)$ equals the value $y(t - \Delta t)$ caused by the previous input event $\mathbf{x}(t - \Delta t)$, as long as the present input-value $\mathbf{x}(t)$ remains unchanged:

$$\mathbf{x}(t) = \mathbf{x}(t - \Delta t) \Rightarrow y(t) = y(t - \Delta t).$$

The above postulate is visualised in Fig. 24–3 for our sample memory. Applying the theorem $(A \rightarrow B) \wedge (\overline{A} \rightarrow B) \Leftrightarrow B$ to (24.2a) and (24.2b) enables us to state that for a stationary and memorising input event $\mathbf{x}(t)$ the associated output value $y(t)$ equals $y(t - \Delta t)$:

$$y(t) = y(t - \Delta t). \tag{24.3}$$

[1)] But it is common in the literature to use the adjectives *present* and *next* for $\mathbf{x}(t_1)$ and $\mathbf{x}(t_2)$, respectively.

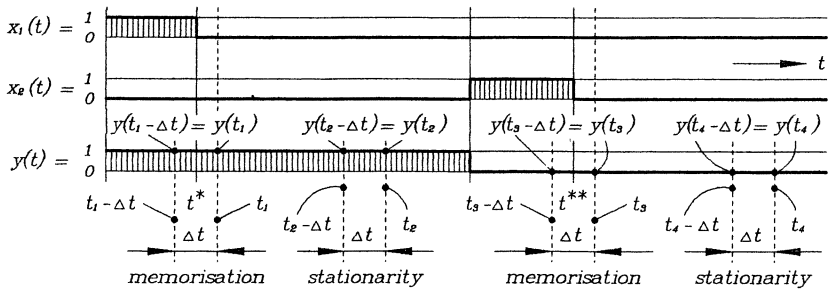


Fig. 24–3 Visualising memorisation and stationarity

24.2 The Feedback Model of Memory

The concept of memorisation, as laid down in (24.2) stating that the output $y(t)$ of a memory equals the delayed output $y(t - \Delta t)$ for a stationary and *memorising* input event $\mathbf{x}(t)$, gives a strong hint as how to define memory as a function or how to realise a latch: *By using the delayed output* $y(t - \Delta t)$ *as a further input to the memory.*

Let us take the delayed output $y(t - \Delta t)$ as an *idealised description* of an **inertial delay**[1] as introduced in Section 23.3, its symbol and transitional behaviour pictured in Fig. 23–5. In other words, not taking inertia into account, we describe the delay's output $y'(t)$ as equalling the time-delayed version $y(t - \Delta t)$ of the delay's input $y(t)$.

$$y'(t) \stackrel{df}{=} y(t - \Delta t). \tag{24.4}$$

If we introduce y' as a further input, as in Fig. 24–4, we must make sure that this is compatible with the original declaration of the latch: Irrespective of any value of the delayed output y', the effect that the input values x_1, x_2 of an input event (x_1, x_2, y') have on the output y of the extended DSE (Fig. 24–4) must be identical with the effect the same input values x_1, x_2 have on the output of the original DSE of Fig. 24–1.

Specifically, the multiple input-event $(1, 0, y')$ causes the output in the extended DSE to be 1, just as $(1, 0)$ does in the original DSE. Likewise $(0, 1, y')$ and $(0, 1)$ cause 0-outputs in the extended and the original DSEs. The same principle holds for the memorising input event $(0, 0)$ of the original DSE (Fig. 24–1) and the multiple input event $(0, 0, y')$ of the extended DSE (Fig. 24–4): Both cause the output y to be either 0 or 1. But while in the

[1] A delay is *inertial* if its output does not respond to input changes shorter than Δt.

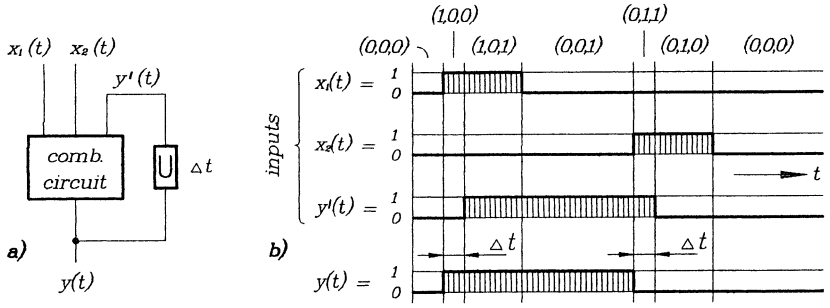


Fig. 24–4 Explaining the behaviour of the sample latch by introducing $y'(t) \stackrel{df}{=} y(t - \Delta t)$ as a further *input*

original DSE the *single* input event $(0,0)$ maps to both output values, 0 and 1, the multiple input event $(0,0,y')$ stands for *two* distinct individual input events, one of which — $(0,0,0)$ — maps to 0, the other — $(0,0,1)$ — to 1.

Thus, by introducing the delayed output y' as a further input, we obtain the memory as a **function** $f : \{0,1\}^3 \mapsto \{0,1\}$, each element (x_1, x_2, y') of its domain $\{0,1\}^3$ mapping to exactly *one* value, either 0 or 1. We call the elements (x_1, x_2, y') of the function's domain $\{0,1\}^3$ the **(input) condition** of the memory, as contrasted with (x_1, x_2) which we call the **input event** of the memory. Expressing memories as functions of the form $\{0,1\}^n \mapsto \{0,1\}$ opens the door to using *all* results pertinent to combinational circuits. As a first such step, we express the extended DSE of Fig. 24–4 in a standard K-map — see Fig. 24–5a.

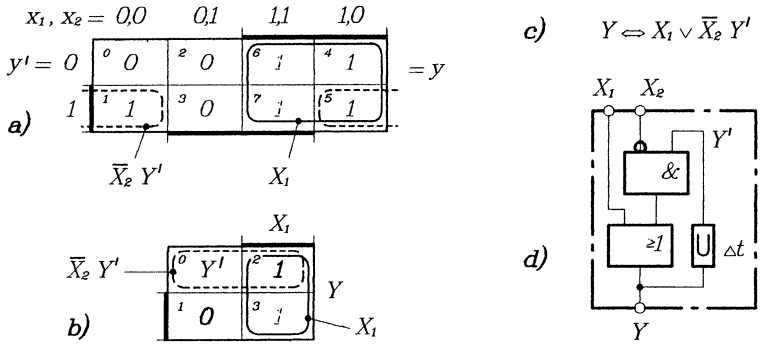


Fig. 24–5 Specification and evaluation of the sample latch

The latch pictured in the extended DSE of Fig. 24–4 is incompletely specified, as, the diagram contains neither of the input events $(1,1,y')$. Obviously we *don't care* what output values are assigned to $(1,1,y')$ giving us the liberty to choose them, say, as in cells 6 and 7 of the K-map of Fig. 24–5a.

Using the input variable y' as a *map-entered variable* we can transform the K-map of Fig. 24–5a into the *reduced K-map* of Fig. 24–5b. This K-map has the intuitive advantage of placing only the so-called *'independent'* input variables x_1, x_2 on the perimeter of the map, while y', being the time-delayed version of the output and thus *'dependent'*, is used in the prominent position of a map-entered variable.

Both K-maps let us calculate a like logic formula for the latch: $Y \Leftrightarrow X_1 \vee \overline{X}_2 Y'$. The circuit obtained is shown in Fig. 24–5d. One input to the AND-gate is Y'. This input is obtained by **feedback** — a connection from the output Y, via an inertial delay Δt, back to the Y'-input of the AND-gate.

Using the above example, let me point out a marked **difference between memory and latch**. Memory, referring to the function specified in Fig. 24–5a, has a domain the elements of which are the ordered triple (x_1, x_2, y'). The logical realisation of the memory, the latch of Fig. 24–5d, is a circuit that is said to have only two input variables, x_1 and x_2 (as contrasted with the three input variables x_1, x_2, y' of the memory), for we can choose the values of only these variables at will. The third variable, y', on the other hand, is called the **state variable**, and any of its values an **(internal) state**, the word *'internal'* indicating that we have no direct influence on its value. In fact, it is the **hallmark of the feedback model** of a latch that $y'(t) = y(t - \Delta t)$ in every instant and independently of any present input event (x_1, x_2) whether *setting*, *resetting*, or *memorising*.

Although any memory with the three inputs x_1, x_2, y' and one output can be described by one of the $2^{(2^3)} = 2^8 = 256$ functions $f : \{0,1\}^3 \mapsto \{0,1\}$, only 36 of these describe latches (as we discuss in a later chapter). What earmarks a K-map for three inputs — x_1, x_2, y' — as the K-map of a memory? We must be able to partition the set $\mathcal{E}$ of input events (x_1, x_2) into **set**, **reset**, and **memory equivalents** $\mathcal{E}_1$, $\mathcal{E}_0$, and $\mathcal{E}_q$, respectively:

$$\mathcal{E}_1 = \{(x_1, x_2) | \bigwedge_{y'} (f(x_1, x_2, y') = 1)\}, \tag{24.5a}$$

$$\mathcal{E}_0 = \{(x_1, x_2) | \bigwedge_{y'} (f(x_1, x_2, y') = 0)\}, \tag{24.5b}$$

$$\mathcal{E}_q = \{(x_1, x_2) | \bigwedge_{y'} \underbrace{(f(x_1, x_2, y')}_{y} = y')\}. \tag{24.5c}$$

These equivalents are visualised in Fig. 24–6a for our sample memory. The cells of $\mathcal{E}_1$ contain only the value 1 for the output variable $y \stackrel{df}{=} f(x_1, x_2, y')$ irrespective of the value of y'. Similarly, the cells of $\mathcal{E}_0$ have only 0s inscribed for any value of y'. Each cell of the memory equivalent $\mathcal{E}_q$[1)] contains an output value y identical to the previous output value y'.

[1)] The index q (or Q) is used to characterise the memory equivalent $\mathcal{E}_q$ because

The equivalents $\mathcal{E}_1$, $\mathcal{E}_0$, and $\mathcal{E}_q$ are most conveniently depicted in a reduced K-map where all cells of $\mathcal{E}_q$ have Y' (or y') inscribed.

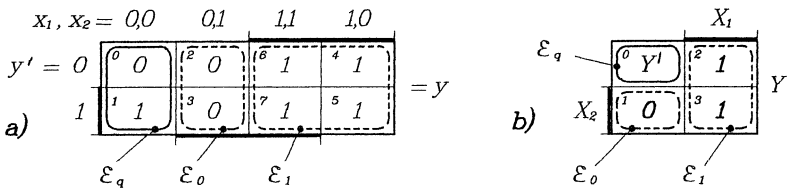


Fig. 24–6 Picturing set, reset, and memory partitions

24.3 Transient versus Stationary Behaviour

The extended DSE of Fig. 24–7b (basically a repetition of Fig. 24–4b) expresses the behaviour of our sample memory, Fig. 24–1, as a function $f : \{0,1\}^3 \mapsto \{0,1\}$ which maps each triple (x_1, x_2, y') to the output y. This function f is specified in the standard K-map of Fig. 24–7a (basically a repetition of Fig. 24–5a) from which the latch or circuit pictured in Fig. 24–5d was calculated — only the encircled output values are part of the analysis to follow.

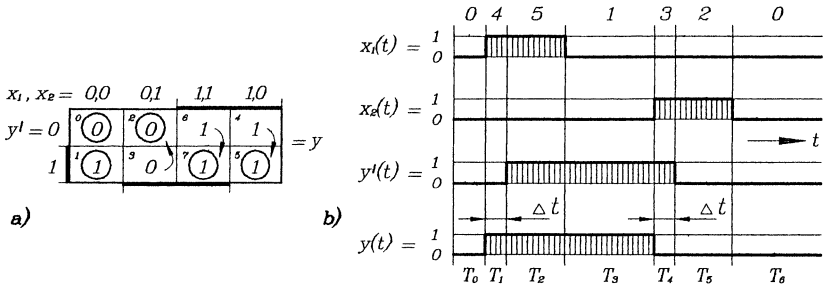


Fig. 24–7 Analysing the transient behaviour

Let us start the analysis of the circuit by *setting* the latch. The associated signal flow is shown in Fig. 24–8a: The moment X_1 becomes **1**, this signal passes through the OR-gate (remember, gates are assumed to have *no delay*) to the circuit's output, and to the delay Δt. This situation remains stationary during Δt, i.e., during T_1 of Fig. 24–7b. After Δt has elapsed (Fig. 24–8b), the **1**-signal entering the delay, passes through it, entering and traversing the

it is quite common to refer to the internal state y' (or Y') by q (or Q) so as to be independent of the name of the output variable. If you choose, you can even substitute (the bound variable) y' of (24.5) by, say, q.

AND-gate to enter the OR-gate. This situation remains unchanged during the rest of the setting input-event $X_1\overline{X}_2$. The circuit in the condition pictured in Fig. 24–8a is said to be in a **transient condition**, meaning that after Δt elapses the internal flow of signals will **change autonomously**, i.e., while the values of the input variables remain unchanged. Fortunately, our circuit enters a **stationary condition** (one in which no internal signals change any more) after Δt has elapsed — this is not always the case for some sequential circuits.

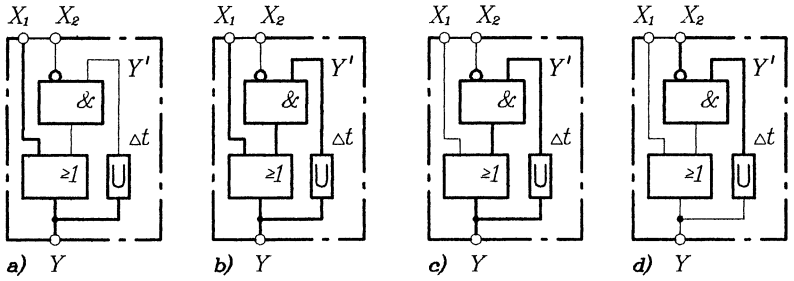


Fig. 24–8 The first steps in analysing the sample latch

The *transient condition* pictured in Fig. 24–8a corresponds to the signal flow during T_1, Fig. 24–7b, and to cell 4 of Fig. 24–7a. The *stationary condition* pictured in Fig. 24–8b corresponds to the signal flow during T_2, Fig. 24–7b, and to cell 5 of Fig. 24–7a. In the K-map, cells in which the output is encircled represent stationary conditions — these are characterised by y' and y having identical values. Cells in which the output is not encircled represent transitory conditions — these are characterised by y' and y being distinct.

The next step is to switch from the setting input-event $X_1\overline{X}_2$, pictured in Fig. 24–8b, to the memorising input-event $\overline{X}_1\overline{X}_2$, the signal flow for which is pictured in Fig. 24–8c. While the OR-element is now fed a **0**-signal from X_1, the output Y remains **1**, this value coming from the **closed feedback loop**. Note that it is necessary in the practical realisation of a feedback design, that at least one of the gates in the closed loop is an **active device** — i.e., it must have its own power supply, and it must act as a power amplifier.

Switching from the stationary setting input-event $X_1\overline{X}_2$, Fig. 24–8b, to the memorising input-event $\overline{X}_1\overline{X}_2$, Fig. 24–8c, corresponds to switching from cell 5 to the stationary cell 1 of the K-map of Fig. 24–7a.

To reset the latch, X_2 must now be switched to **1** (Fig. 24–8d). The moment this happens, the output of the AND-gate is turns to **0**, and with it the output of the OR-gate (i.e., the circuit's output Y), and the input to the delay Δt. But, do note that the output Y' of the delay still remains **1** during the time interval Δt. (This is the complementary situation to the one shown

in Fig. 24–8a). Resetting the latch by switching from Fig. 24–8c to Fig. 24–8d corresponds to switching from cell 1 to the transient cell 3, Fig. 24–7a, where the output y immediately becomes 0. After Δt the delayed output y' is turned to 0 so that we are then autonomously moved on to the stationary cell 2 (this last step not pictured in Fig. 24–8).

When speaking of memories in later chapters, we mainly specify them in **reduced K-maps** in which the internal state y' is used as the map-entered variable. The reduced K-maps are easier to read (i.e., interpret) than their counterparts, the standard K-maps, *if we take the reduced K-maps to* **express only the stationary conditions** *of a memory* as demonstrated in Fig. 24–9. The only question is: How is the value of the map-entered state variable y' determined?

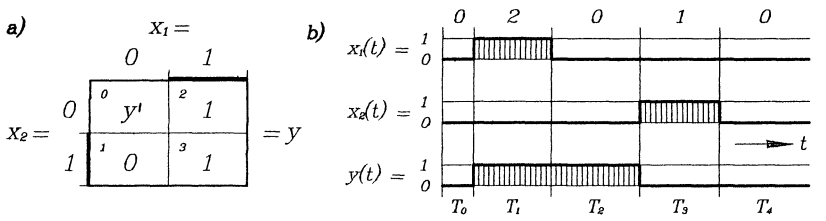


Fig. 24–9 Expressing the *stationary behaviour* in a *reduced* K-map

By the postulate of memorisation and stationarity, (24.2), the state variable $y'(t)$ for any given instant t equals the output $y(t - \Delta t)$ caused by the previous input event $x(t - \Delta t)$. **Thus, in a reduced K-map, y' takes on the output value contained in the cell describing the previous input event.** For instance, if the previous input event was $X_1\overline{X}_2$, described by cell 2 of Fig. 24–9a, then its output value, the 1 inscribed in cell 2, is the value which y' takes on when switching from $X_1\overline{X}_2$ to the present input event $\overline{X}_1\overline{X}_2$ (cell 0).

24.4 Latches without Feedback

The foregoing sections give the *false* impression that latches are necessarily realised by feedback circuits, as in Fig. 24–4a. To counteract this impression, note that the following devices are latches, but that they do not contain feedback: a thyristor (electronics), a polarised electrical relay, a pneumatic spool valve without a return spring.

Let us look at just one of these devices, the spool valve pictured in Fig. 24–10a. To set the latch, a pressure pulse is applied to input x_1 propelling the spool to the right, and connecting the air supply with the output. When both inputs, x_1 and x_2, are pressureless, the spool remains static, thus maintaining (memorising) the output value just set. A pressure pulse

applied to x_2 will reset the latch. You will notice that the valve has *two* memorising input events — $\overline{X}_1\overline{X}_2$ and X_1X_2, as in both cases the spool is in equilibrium.

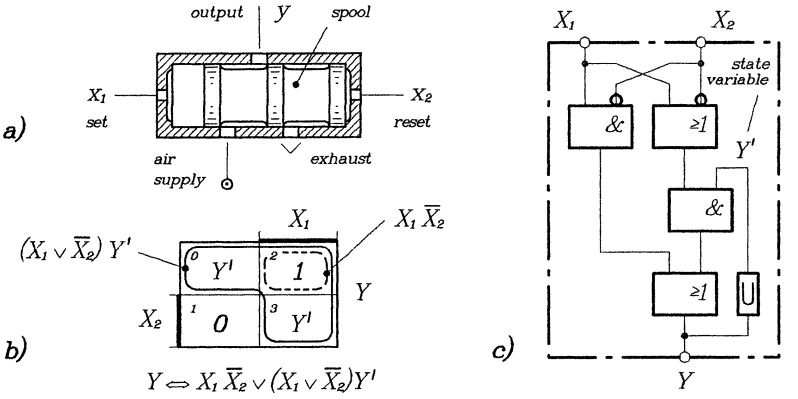


Fig. 24–10 A spool valve without a return spring as an example of a latch without feedback

The *stationary* behaviour of the spool valve is conveniently described in a reduced K-map, see Fig. 24–10b. $X_1\overline{X}_2$ is clearly the only setting input-event so that cell 2 (and only this cell) has the output value **1** inscribed. The only resetting input-event is $\overline{X}_1X_2$ so that cell 1 gets **0** inscribed. The two input-events X_1X_2 and $\overline{X}_1\overline{X}_2$ must be memorising input-events because the spool, being in equilibrium, ensures that the previous output value is retained, i.e., memorised.

If we calculate a logic formula $Y \Leftrightarrow X_1\overline{X}_2 \vee (X_1 \vee \overline{X}_2)Y'$ from the reduced K-map of Fig. 24–10b, we can express it as the feedback circuit of Fig. 24–10c the stationary behaviour of which is identical with the stationary behaviour of the spool valve. Thus, a great **advantage of the feedback model** *is to enable us to study the stationary behaviour of a latch without having to know its physical design.* Furthermore, the feedback model allows us to design a latch by physically employing gates, a time delay and a feedback. The advantage of such designs is that the feedback formulas we develop from the K-maps tell us unequivocally how to design any desired latch as a feedback circuit.

24.5 Is a Delay in the Feedback necessary? *

The opinion is sometimes voiced that reducing the time delay in the feedback of a latch to zero leads to a contradiction. The argument in support of this opinion is simple and, superfluously viewed, persuasive. Consider our sample latch of Fig. 24–5 the equation of which was determined as $Y \Leftrightarrow X_1 \vee \overline{X}_2Y'$.

For $\Delta t = 0$ we can set $Y' = Y$ so that the logic formula for the latch reduces to $Y \Leftrightarrow X_1 \vee \overline{X}_2 Y$. Now, if we subject this formula to a truth-table analysis for each of the triples (X_1, X_2, Y), as in Fig. 24–11, you immediately see that the formula is not tautological, or to be more precise, for the triples $(\mathbf{0},\mathbf{1},\mathbf{1})$, $(\mathbf{1},\mathbf{0},\mathbf{0})$, $(\mathbf{1},\mathbf{1},\mathbf{0})$ of (X_1, X_2, Y) the formula $Y \Leftrightarrow X_1 \vee \overline{X}_2 Y$ is contradictory.

Y	$\leftrightarrow$	$(X_1$	$\vee$	$(\neg X_2$	$\wedge$	$Y))$
0	1	0	0	1	0	0
1	1	0	1	1	1	1
0	1	0	0	0	0	0
1	0	0	0	0	0	1
0	0	1	1	1	0	0
1	1	1	1	1	1	1
0	0	1	1	0	0	0
1	1	1	1	0	0	1

Fig. 24–11 This truth-table analysis is not valid. But why?

One problem with the above argument is that proving $Y \Leftrightarrow X_1 \vee \overline{X}_2 Y$ is not tautological does not infer that its time-delayed version, $Y \Leftrightarrow X_1 \vee \overline{X}_2 Y'$, is tautological. In fact, a truth-table analysis for all quadruples (X_1, X_2, Y, Y') of the time-delayed formula equally shows this formula to be non-tautological. Generalising, we could by this approach, and this is the crux of the argument, even falsify the normal forms and all formulas derived from them (the latch formulas among them). At this point, you might like to try for yourself to disprove the validity of the truth-table approach taken in Fig. 24–11 before I attempt an answer ...

For a formal explanation, we must reach back to the *theorem of identical functions*,

$$(13.33) \qquad (x_1, \ldots, x_n) = (e_1, \ldots, e_n) \Rightarrow f(x_1, \ldots, x_n) = f(e_1, \ldots, e_n),$$

from which the normal forms were derived. This theorem assumes that the value $f(x_1, \ldots, x_n)$ of a function f is uniquely determined by the *'independent'* variables $x_1, \ldots, x_n$. In other words, the function's value $f(x_1, \ldots, x_n)$ cannot be chosen, it is specified by the mapping f. The symbol y is only used as an abbreviation for $f(x_1, \ldots, x_n)$. Unhappily, using y without arguments, i.e., writing y instead of $y(x_1, \ldots, x_n)$, falsely gives the impression that y is an *'independent'* variable the value of which we can choose (independently of $x_1, \ldots, x_n$). Keeping this in mind, the logic formula $Y \Leftrightarrow X_1 \vee \overline{X}_2 Y$ must be understood as $Y(X_1, X_2) \Leftrightarrow X_1 \vee \overline{X}_2 Y(X_1, X_2)$ in which the value of $Y(X_1, X_2)$ is determined by the formula as shown in the table of Fig. 24–12.

The evaluations carried out in the table of Fig. 24–12 assign constants to $Y(X_1, X_2)$ for all pairs of (X_1, X_2) other than for $(X_1, X_2) \hookleftarrow (\mathbf{0},\mathbf{0})$.

In this case, $Y(X_1, X_2)$ is **undetermined** — i.e., $Y(\mathbf{0},\mathbf{0}) \Leftrightarrow Y(\mathbf{0},\mathbf{0})$ — **but not contradictory**. (There is only one evaluation result we would accept as contradictory in this case, and that would be if we were to obtain $Y(\mathbf{0},\mathbf{0}) \Leftrightarrow \overline{Y(\mathbf{0},\mathbf{0})}$).

$Y(X_1,X_2)$	$\Leftrightarrow$	$\big(X_1$	$\vee$	$(\overline{X}_2$	$\wedge$	$Y(X_1,X_2))\big)$
$Y(\mathbf{0},\mathbf{0})$	$\Leftrightarrow$	**0**	$Y(\mathbf{0},\mathbf{0})$	**1**	$Y(\mathbf{0},\mathbf{0})$	$Y(\mathbf{0},\mathbf{0})$
$Y(\mathbf{0},\mathbf{1})$	$\Leftrightarrow$	**0**	**0**	**0**	**0**	$Y(\mathbf{0},\mathbf{1})$
$Y(\mathbf{1},\mathbf{0})$	$\Leftrightarrow$	**1**	**1**	**1**	$Y(\mathbf{1},\mathbf{0})$	$Y(\mathbf{1},\mathbf{0})$
$Y(\mathbf{1},\mathbf{1})$	$\Leftrightarrow$	**1**	**1**	**0**	**0**	$Y(\mathbf{1},\mathbf{1})$

Fig. 24–12 Evaluating $Y(X_1, X_2)$ for all pairs (X_1, X_2)

Reconsidering the above result, I wish to state quite frankly that **there is no algebraic need for a delay in the feedback**, but to avoid an undetermined output value for a memorising input event **we must have a delay with $\Delta t > 0$ in the feedback loop** of a latch that is realised according to the feedback model of a memory.

For a memorising input event, the output $Y(X_1, X_2)$ of any latch will reduce to $Y(X_1, X_2) \Leftrightarrow Y'(X_1, X_2)$ in which $Y'(X_1, X_2)$ is a variable the value of which is chosen freely. I leave it to you to argue why $Y'(X_1, X_2)$ may, in fact must, be chosen independently of X_1 and X_2.

Chapter 25

Basic Theory of Latches

Any formal discussion of latches must start with a definition of a latch, i.e., must state *formally* what a latch is. The challenge is greater than one might think, for it is not a question of not having a generally accepted definition — the problem is that no definition has yet been proposed. This is astonishing because a latch is the basic binary memory element, and is thus conceptually and materially needed in designing and building sequential circuits.

Only after having defined the general concept of a latch is it possible to find an adequate way of specifying individual latches, my choice quite clearly being a reduced K-map the map-entered variable of which is the internal state. One advantage of this choice is that (in Section 20.3) we have already developed a general procedure for evaluating circuits specified in reduced K-maps. All we now need to do is to adapt this procedure to the evaluation of latches.

We compare latches by three different criteria as expressed in the following questions: firstly, *'when are latches identical?'*, secondly, *'when are latches isomorphic?'*, and thirdly, *'when are they complementary?'*.

25.1 What is a Latch? *

If you were to ask *'what is a combinational circuit'*, I would hardly expect you to accept an *example* of a combinational circuit as the general concept. If you search the literature for an explanation of the concept *latch*, you will find platitudinous phrases such as *'a latch is a circuit with memorising ability'*, or *'a latch is a circuit of the following kind'*, and you will be referred to an *example*. The definition given here is the quintessence of the intuitive approach to latches given in the previous chapter.

(25.1) Defining a latch. A latch with n inputs $x_1, \ldots, x_n$ and one output y is a function f from the $(n+1)$-fold Cartesian product $\{0,1\}^{n+1}$ of the $(n+1)$-tuples $(x_1, \ldots, x_n; q)$ into $\{0,1\}$,

$$f : \{0,1\}^{n+1} \mapsto \{0,1\},$$

$$y(t) \stackrel{df}{=} f\big(x_1(t), \ldots, x_n(t); q(t)\big)$$

which has the following two properties:

(a) The n-plus-first input value q is called the **internal state** of the latch and is **postulated** to equal the function's time-delayed output,

$$q(t) \stackrel{df}{=} y'(t), \qquad y'(t) \stackrel{df}{=} y(t - \Delta t).$$

(b) The set $\mathcal{E}$ of input events $(x_1, \ldots, x_n)$ is partitioned into the so-called **set**, **reset**, and **memory equivalents** $\mathcal{E}_1(f)$, $\mathcal{E}_0(f)$, $\mathcal{E}_q(f)$ and only into these, i.e., $\mathcal{E} = \mathcal{E}_1(f) \cup \mathcal{E}_0(f) \cup \mathcal{E}_q(f)$, and neither of the equivalents may be empty:

$$\mathcal{E}_1(f) \stackrel{df}{=} \big\{(x_1, \ldots, x_n) \big| \bigwedge_{q \in \{0,1\}} \big(f(x_1, \ldots, x_n; q) = 1\big)\big\},$$

$$\mathcal{E}_0(f) \stackrel{df}{=} \big\{(x_1, \ldots, x_n) \big| \bigwedge_{q \in \{0,1\}} \big(f(x_1, \ldots, x_n; q) = 0\big)\big\},$$

$$\mathcal{E}_q(f) \stackrel{df}{=} \big\{(x_1, \ldots, x_n) \big| \bigwedge_{q \in \{0,1\}} \big(f(x_1, \ldots, x_n; q) = q\big)\big\}.$$

In most circumstances it will suffice to write $\mathcal{E}_1$, $\mathcal{E}_0$, $\mathcal{E}_q$ instead of $\mathcal{E}_1(f)$, $\mathcal{E}_0(f)$, $\mathcal{E}_q(f)$, respectively. Due to the postulate $q(t) = y'(t)$, (25.1a), the above definition (25.1) is said to be that of the **feedback model of a latch**. The equivalents $\mathcal{E}_1$, $\mathcal{E}_0$, $\mathcal{E}_q$ defined in (25.1b) are **mutually disjoint**,

$$\mathcal{E}_1 \cap \mathcal{E}_0 = \emptyset, \qquad \mathcal{E}_1 \cap \mathcal{E}_q = \emptyset, \qquad \mathcal{E}_0 \cap \mathcal{E}_q = \emptyset, \tag{25.2}$$

which is important to point out, as discussed in the next section. Here, we prove only $\mathcal{E}_0 \cap \mathcal{E}_q = \emptyset$, asking you to prove the other two along the same lines. Using $\mathbf{x} \stackrel{df}{=} (x_1, \ldots, x_n)$, and rewriting the definitions for $\mathcal{E}_0$ and $\mathcal{E}_q$ of (25.1b),

$$\mathbf{x} \in \mathcal{E}_0 \Leftrightarrow \bigwedge_{q \in \{0,1\}} \big(f(\mathbf{x}; q) = 0\big) \Leftrightarrow \big(f(\mathbf{x}; 0) = 0\big) \wedge \big(f(\mathbf{x}; 1) = 0\big),$$

$$\mathbf{x} \in \mathcal{E}_q \Leftrightarrow \bigwedge_{q \in \{0,1\}} \big(f(\mathbf{x}; q) = q\big) \Leftrightarrow \big(f(\mathbf{x}; 0) = 0\big) \wedge \big(f(\mathbf{x}; 1) = 1\big),$$

we can express the intersection $\mathcal{E}_0 \cap \mathcal{E}_q$ as

$$\mathbf{x} \in (\mathcal{E}_0 \cap \mathcal{E}_q) \stackrel{(6.6)}{\Longleftrightarrow} (\mathbf{x} \in \mathcal{E}_0) \wedge (\mathbf{x} \in \mathcal{E}_q) \Leftrightarrow$$
$$\big(f(\mathbf{x}; 0) = 0 \wedge \underbrace{f(\mathbf{x}; 1) = 0}\big) \wedge \big(f(\mathbf{x}; 0) = 0 \wedge \underbrace{f(\mathbf{x}; 1) = 1}\big) \Leftrightarrow \mathbf{0}.$$

The intersection $\mathcal{E}_0 \cap \mathcal{E}_q$ is $\emptyset$ due to the underbraced expressions, above, being contradictory: $f(\mathbf{x};1) = 0 \wedge f(\mathbf{x};1) = 1 \Leftrightarrow \mathbf{0}$.

25.2 Specifying Latches in Reduced K-maps *

The definition of a latch, (25.1), as a *function* $f : \{0,1\}^{n+1} \mapsto \{0,1\}$ enables us to use any specification appropriate to combinational circuits, such as a Table of Asserted Events (TAE) for $n+1$ inputs, a (standard) K-map for $n+1$ inputs, and a Reduced K-map for n inputs, preferably with the internal state as the map-entered variable. I advocate the reduced K-map which uses the *internal state* as the map-entered variable as the most appropriate way to specify a latch.

Using the example of an arbitrary latch with four (independent) inputs x_1, x_2, x_3, x_4, see Fig. 25–1a, let us see how the reduced K-map corresponds to the definition of a latch, (25.1). As the definition requires, the area $\mathcal{E}$ of the reduced K-map is partitioned into the equivalents $\mathcal{E}_1$, $\mathcal{E}_0$, $\mathcal{E}_q$, meaning that their union covers the whole area of the reduced K-map, and that the areas representing the equivalents $\mathcal{E}_1 = \{3,5,7,14,15\}$, $\mathcal{E}_0 = \{0,1,2,4,6,8,10\}$, $\mathcal{E}_q = \{9,11,12,13\}$ do not overlap, these aspects being stressed in Fig. 25–1b.

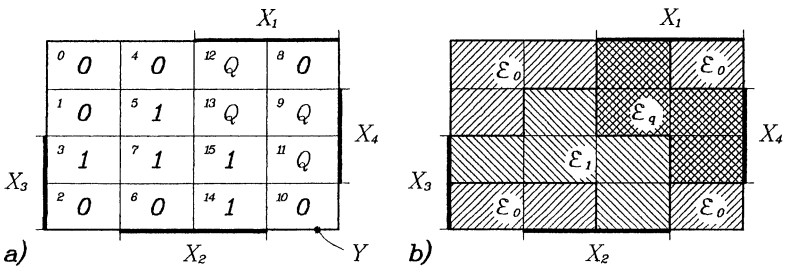


Fig. 25–1 Specifying a latch in a reduced K-map

Equally important is the fact that **the reduced K-maps for latches are easy to read**. Let us look at only one (albeit, non-trivial) example by asking *'what value the present output* $y(e)$ *will have for the memorising input-event* $X_1X_2\overline{X}_3\overline{X}_4$ *of cell* 12'. Assume the previous input-event, call it e', will have been an adjacent one, either 4, 8, 13, or 14, i.e., differs in the value of only one input variable from those of the present input-event 12. For a transition to the present input-event 12 from previous input-events 4, or 8 — which we write as the ordered pairs $(12,4)$ or $(12,8)$, or, generally, as (e,e') — the present output $y(12)$ takes on the value 0. The transition $(12,14)$ leads to $y(12)$ becoming 1. But the transition $(12,13)$ only leads to the question *'what output value or internal state was present in cell* 13'. To

answer questions like this, you need to go back in the sequence of transitions until you come across a setting or resetting input-event. For instance, $y(12)$ is 1 for the transitions $(12, 13, 15)$, whereas $y(12)$ would be 0 for $(12, 13, 9, 1)$.

25.3 Evaluation Formulas and Generic Latches *

At this point, you might find it advantageous to reread Sections 18.3 and 18.4 to bring to mind the notations for *don't cares*, as well as Section 20.3 on the Evaluation of Reduced K-maps. The general evaluation formulas (20.9) to (20.12) assume that there are one or *more* map-entered variables Y_k. The only map-entered variable we are presently considering when speaking of latches is the internal state Q. So, to obtain evaluation formulas for latches, all we need to do is to adapt (20.9) to (20.12) for the internal state Q, or Y' if you choose.

Let us start with the **disjunctive and conjunctive evaluation formula** for a non-inverted output, say, Y:

$$Y(\mathbf{x}) \Leftrightarrow U_{1;\delta}(\mathbf{x}) \vee U_{q;1\delta^*}(\mathbf{x})Y', \tag{25.3a}$$

$$U_{1;\delta}(\mathbf{x}) \Leftrightarrow \bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_1;\mathcal{E}_{\delta S})} C_\varphi(\mathbf{x}), \tag{25.3b}$$

$$U_{q;1\delta^*}(\mathbf{x}) \Leftrightarrow \bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_q;\mathcal{E}_{1S} \cup \mathcal{E}_{\delta S^*})} C_\varphi(\mathbf{x}). \tag{25.3c}$$

$$Y(\mathbf{x}) \Leftrightarrow V_{0;\delta}(\mathbf{x})\big(V_{q;0\delta^*}(\mathbf{x}) \vee Y'\big), \tag{25.4a}$$

$$V_{0;\delta}(\mathbf{x}) \Leftrightarrow \bigwedge_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_0;\mathcal{E}_{\delta S})} D_\varphi(\mathbf{x}), \tag{25.4b}$$

$$V_{q;0\delta^*}(\mathbf{x}) \Leftrightarrow \bigwedge_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_q;\mathcal{E}_{0S} \cup \mathcal{E}_{\delta S^*})} D_\varphi(\mathbf{x}). \tag{25.4c}$$

Do note that $\mathcal{E}_1$ in the above formulas is the 1-nucleus $\mathcal{E}_{1\nu}$, and $\mathcal{E}_0$ the 0-nucleus $\mathcal{E}_{0\nu}$ of Section 18.3. As before, the index S indicates a subset so that $\mathcal{E}_{1S} \subseteq \mathcal{E}_1$ and $\mathcal{E}_{0S} \subseteq \mathcal{E}_0$. Two different indices, S and S^*, are used in the case of two mutually independent subsets, as for $\mathcal{E}_{\delta S} \subseteq \mathcal{E}_\delta$ and $\mathcal{E}_{\delta S^*} \subseteq \mathcal{E}_\delta$. Furthermore, the subsets $\mathcal{E}_{\delta S}$ and $\mathcal{E}_{\delta S^*}$ in (25.3) are — if you will please forgive the inadequacy of the notation — independent of equally denoted subsets in (25.4).

All latches calculated according to (25.3) have a common design, that of Fig. 25–2a. Each latch consists of three distinct circuits. The actual feedback circuit is *common to all latches*, and as its output is a disjunction I call it

the **generic disjunctive latch**. The generic latch is a two-input latch, and is solely responsible for the memorising behaviour of the given latch. The two inputs of the generic latch come from two combinational circuits which I refer to as the **prelogic** *to* the generic latch, or as the *prelogic* of *the given latch*. The output of the **setting prelogic** realises the function $U_{1;\delta}(\mathbf{x})$, while the output of the **memorising prelogic** is $U_{q;1\delta^*}(\mathbf{x})$. An important aspect to note is that two different latches f and g, as specified by two different reduced K-maps, can never be identical in their setting prelogic AND their memorising prelogic. The terminology **generic conjunctive latch** and **resetting prelogic** apply to latches derived from (25.4).

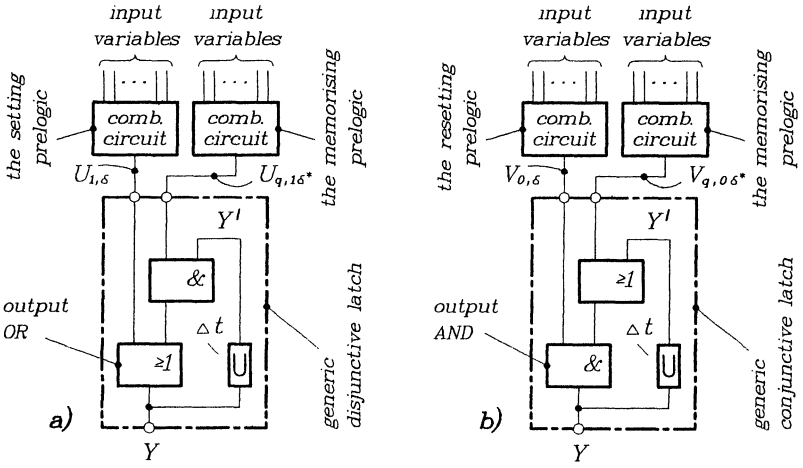


Fig. 25–2 The generic disjunctive and conjunctive latches

The above formulas, (25.3) and (25.4), refer to non-inverted outputs of latches, that is, they answer the question *'when is the numeric output y one?'*. If we concentrate on the question *'when does the numeric output y equal zero?'*, we have to develop formulas for $\overline{Y}$. That means writing (20.10) and (20.12) for the single map-entered variable $\overline{Y}'$ which is the delayed version of the latch's output $\overline{Y}(\mathbf{x})$.

$$\overline{Y}(\mathbf{x}) \Leftrightarrow U_{0;\delta}(\mathbf{x}) \vee U_{q;0\delta^*}(\mathbf{x})\overline{Y}', \tag{25.5a}$$

$$U_{0;\delta}(\mathbf{x}) \Leftrightarrow \bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_0;\mathcal{E}_{\delta S})} C_\varphi(\mathbf{x}), \tag{25.5b}$$

$$U_{q;0\delta^*}(\mathbf{x}) \Leftrightarrow \bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_q;\mathcal{E}_{0S} \cup \mathcal{E}_{\delta S^*})} C_\varphi(\mathbf{x}). \tag{25.5c}$$

$$\overline{Y}(\mathbf{x}) \Leftrightarrow V_{1;\delta}(\mathbf{x})\big(V_{q;1\delta^*}(\mathbf{x}) \vee \overline{Y}'\big), \tag{25.6a}$$

$$V_{1;\delta}(\mathbf{x}) \Leftrightarrow \bigwedge_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_1;\mathcal{E}_{\delta S})} D_\varphi(\mathbf{x}), \tag{25.6b}$$

$$V_{q;1\delta^*}(\mathbf{x}) \Leftrightarrow \bigwedge_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_q;\mathcal{E}_{1S} \cup \mathcal{E}_{\delta S^*})} D_\varphi(\mathbf{x}). \tag{25.6c}$$

This is an all too formalistic approach, so we shall take a more detailed look at inverting latches in a later section.

25.4 An Evaluation Example

Given the latch in Fig. 25–3a (see the next section), let us calculate a disjunctive feedback circuit, using (25.3), which satisfies the input-output behaviour of the given K-map. As the K-map contains no *unassigned input events* ($\mathcal{E}_\delta = \emptyset$), the equations (25.3) reduce to

$$Y_f(\mathbf{x}) \Leftrightarrow U_1(\mathbf{x}) \vee U_{q;1}(\mathbf{x})Y_f', \tag{a}$$

$$U_1(\mathbf{x}) \Leftrightarrow \bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_1)} C_\varphi(\mathbf{x}), \tag{b}$$

$$U_{q;1}(\mathbf{x}) \Leftrightarrow \bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_q;\mathcal{E}_{1S})} C_\varphi(\mathbf{x}). \tag{c}$$

To evaluate $Y_f(\mathbf{x})$ by equation (a) we need to determine the *set inclusion* $U_1(\mathbf{x})$, and the *memory inclusion* $U_{q;1}(\mathbf{x})$. To calculate the set inclusion $U_1(\mathbf{x})$, which is done according to equation (b), we have to determine a set of prime sets, the cover $\mathcal{P}(\mathcal{E}_1)$, that cover $\mathcal{E}_1$. In our example, there is only one such prime set, i.e.,

$$\mathcal{P}(\mathcal{E}_1) = \{\mathcal{P}_{0x0}\},$$

as $\mathcal{P}_{0x0} = \mathcal{E}_1$. Remembering that $\varphi : \mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_1)$ is read: *'those φ for which $\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_1)$'*, as introduced in (17.25), we can now calculate

$$U_1(\mathbf{x}) \overset{(b)}{\Longleftrightarrow} \bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_1)} C_\varphi(\mathbf{x}) \Leftrightarrow C_{0x0}(\mathbf{x}) \Leftrightarrow \overline{X}_1\overline{X}_3.$$

Calculating a memory inclusion $U_{q;1}(\mathbf{x})$ according to (c) requires us to determine a cover $\mathcal{P}(\mathcal{E}_q;\mathcal{E}_{1S})$ for which we can choose $\mathcal{E}_{1S}$ arbitrarily to be $\emptyset \subseteq \mathcal{E}_{1S} \subseteq \mathcal{E}_1$. Visual inspection of the K-map of Fig. 25–3a shows that by

choosing $\mathcal{E}_{1S} = \mathcal{E}_1$ we can maximise the prime set that covers the memorising cells 1 and 3. As the loops of the K-map show, the memory cover is then

$$\mathcal{P}(\mathcal{E}_q; \mathcal{E}_{1S}) = \{\mathcal{P}_{0xx}, \mathcal{P}_{x01}\},$$

enabling us to calculate

$$U_{q;1}(\mathbf{x}) \overset{(c)}{\Longleftrightarrow} \bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}(\mathcal{E}_q;\mathcal{E}_{1S})} C_\varphi(\mathbf{x}) \Leftrightarrow C_{0xx}(\mathbf{x}) \vee C_{x01}(\mathbf{x}) \Leftrightarrow \overline{X}_1 \vee \overline{X}_2 X_3.$$

We can now write the formula for the feedback circuit as

$$Y_f(\mathbf{x}) \overset{(a)}{\Longleftrightarrow} U_1(\mathbf{x}) \vee U_{q;1}(\mathbf{x}) Y_f' \Leftrightarrow \overline{X}_1 \overline{X}_3 \vee (\overline{X}_1 \vee \overline{X}_2 X_3) Y_f'.$$

The circuit implied by this formula is shown in Fig. 25–4.

25.5 Identical and Isomorphic Latches *

Algebraically, latches are functions, functions are sets, and sets are identical if they contain identical elements. Latches are special functions, functions which partition the set of input events, $\mathcal{E}$, into the equivalents $\mathcal{E}_1$, $\mathcal{E}_0$, $\mathcal{E}_q$ specified in (25.1b). As it is assumed that none of the equivalents are empty, that their union equals $\mathcal{E}$, meaning $\mathcal{E} = \mathcal{E}_1 \cup \mathcal{E}_0 \cup \mathcal{E}_q$, and that they are mutually disjoint, (25.2), any two of the equivalents specify the third. The above considerations allow us to formulate the

(25.7) Theorem on the identity of latches. Two latches, $f : \{0,1\}^{n+1} \mapsto \{0,1\}$, from the $(n+1)$-fold Cartesian product $\{0,1\}^{n+1}$ of the $(n+1)$-tuples $(x_1, \ldots, x_n; q)$ into $\{0,1\}$, and $g : \{0,1\}^{n+1} \mapsto \{0,1\}$, from the $(n+1)$-fold Cartesian product $\{0,1\}^{n+1}$ of the $(n+1)$-tuples $(x_1^*, \ldots, x_n^*; q)$ into $\{0,1\}$, are identical *iff* their *set equivalents* $\mathcal{E}_1(f)$ and $\mathcal{E}_1(g)$ are equal, and their *reset equivalents* $\mathcal{E}_0(f)$ and $\mathcal{E}_0(g)$ are equal,

$$f = g \Leftrightarrow \mathcal{E}_1(f) = \mathcal{E}_1(g) \wedge \mathcal{E}_0(f) = \mathcal{E}_0(g).$$

Clearly, the latches specified in the K-maps of Fig. 25–3 are not identical, for, $\mathcal{E}_1(f)$ and $\mathcal{E}_1(g)$ are not equal ($\mathcal{E}_1(f) = \{0,2\}$ and $\mathcal{E}_1(g) = \{0,1\}$) — this being sufficient to show the inequality of the latches.

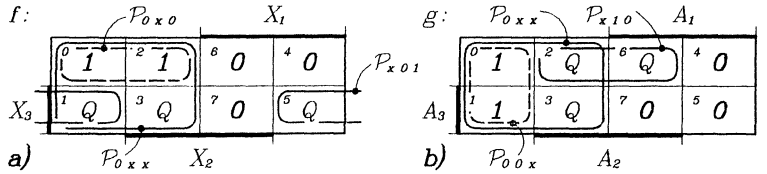


Fig. 25–3 An example of isomorphic latches

Nevertheless, the latches f and g of Fig. 25–3 hold a minor surprise for us. If we calculate the circuits using, say, the disjunctive formulas of (25.3) by employing the loops shown in the K-maps, we get

$$Y_f(X_1, X_2, X_3) \Leftrightarrow \overline{X}_1\overline{X}_3 \vee (\overline{X}_1 \vee \overline{X}_2 X_3)Q,$$
$$Y_g(A_1, A_2, A_3) \Leftrightarrow \overline{A}_1\overline{A}_2 \vee (\overline{A}_1 \vee A_2\overline{A}_3)Q.$$

These two formulas demonstrate a striking similarity which is emphasised by drawing the associated circuit (note the singular: *circuit*, not *circuits*) as in Fig. 25–4.

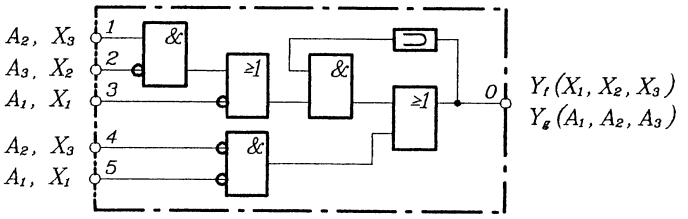


Fig. 25–4 A circuit common to both latches of Fig. 25–3

Two latches f and g are said to be **isomorphic** if you can develop a common circuit for both latches; only the ways their input variables connect to the input terminals of the circuit differ. One refers to g as being *an isomorph* of f (and *vice versa*). The noun *'isomorph'* literally means *equal-structure*, as the two combining forms imply: *'iso-'* meaning *equal* or *identical*, while *'-morph'* indicates *form* or *structure*.

You can quite easily transform a given latch f into a latch isomorphic to it. Firstly, develop all latches identical to f by *permuting* the K-map of f, see for instance Fig. 25–5. **Permuting a K-map** refers to rearranging the input variables around the perimeter of the K-map, and entering an output value $y(e)$ of the original K-map into an equally numbered cell of the permuted K-map. Secondly, rename the input variables of the permuted K-map so that their positions coincide with the positions of the original input-variables (thereby also returning to the numbering of the cells as used in the original K-map). By this procedure, as there are $n!$ permutations of n input variables, you easily see that for any given latch with n inputs there are $n!-1$ isomorphic latches.

As an example for finding a latch isomorphic to a given latch, consider the given latch to be specified by the K-map of Fig. 25–3a. Rearrange the input variables and the cells with their contents according to Fig. 25–5b. In this *permuted* K-map rename the input variables, $(X_1, X_2, X_3) \hookleftarrow (A_2, A_1, A_3)$, and renumber the cells to coincide with the placement of the new input variables A_1, A_2, A_3 — see Fig. 25–3b.

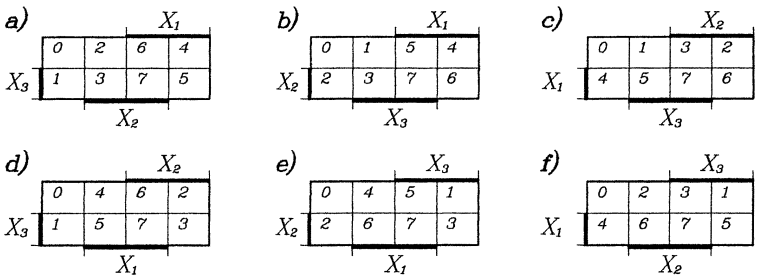


Fig. 25–5 Permuted K-maps represent identical latches

25.6 Inverting a Sample Memory-Circuit

There are two ways to invert a combinational circuit, either by appending an inverter to the output of the circuit, or by designing a new circuit the output of which is the negation of the given circuit. The same principle is true for latches as a simple example demonstrates.

A thyristor (a silicon rectifier controlled by the gate voltage x_2) — see Fig. 25–6a — behaves as the diagram of successive events (DSE) for x_1, x_2 and y of Fig. 25–6d indicates. This is the DSE of a latch (with $X_1\overline{X}_2$ as the memorising input event), its K-map pictured in Fig. 25–6b. Appending an inverter to the output y of the thyristor allows us to augment $\overline{y}$ to the DSE. This, too, specifies a latch, again with $X_1\overline{X}_2$ as the memorising input event, its K-map pictured in Fig. 25–6c. Not surprisingly, the output values in equally placed cells e of the two K-maps are inverted.

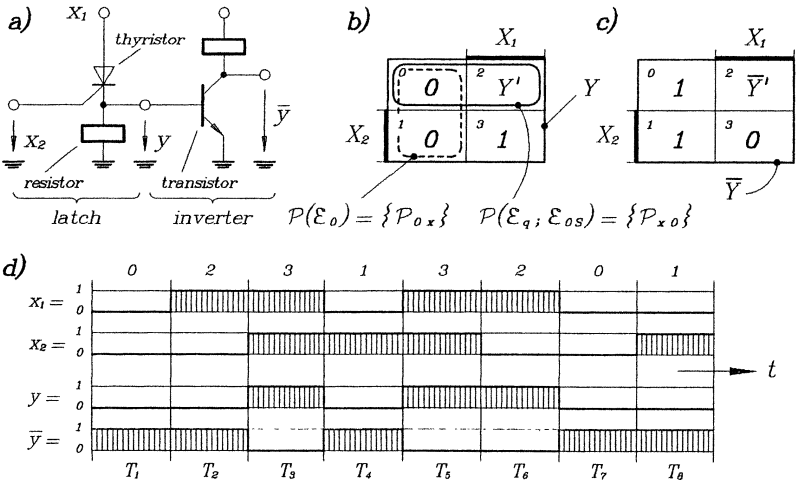


Fig. 25–6 Inverting the output of a latch

Next, let us calculate a feedback circuit equivalent in its behaviour to the thyristor latch, specified in the K-map of Fig. 25–6b, by using the conjunctive formulas (25.4). As the loops in the K-map show, we may write

$$V_0(\mathbf{x}) \Leftrightarrow \bigwedge_{\varphi:\mathcal{P}_\varphi \in \{\mathcal{P}_{0x}\}} D_\varphi(\mathbf{x}) \Leftrightarrow D_{0x}(\mathbf{x}) \overset{(17.13)}{\Longleftrightarrow} \underbrace{(x_1 \neq 0)}_{x_1=1} \vee \underbrace{(x_2 \neq x_2)}_{0} \Leftrightarrow X_1,$$

$$V_{q;0}(\mathbf{x}) \Leftrightarrow \bigwedge_{\varphi:\mathcal{P}_\varphi \in \{\mathcal{P}_{x0}\}} D_\varphi(\mathbf{x}) \Leftrightarrow D_{x0}(\mathbf{x}) \Leftrightarrow X_2,$$

$$Y(\mathbf{x}) \Leftrightarrow V_0(\mathbf{x})\big(V_{q;0}(\mathbf{x}) \vee Y'\big) \Leftrightarrow X_1(X_2 \vee Y').$$

This thyristor-equivalent feedback-circuit $Y \Leftrightarrow X_1(X_2 \vee Y')$ is pictured in Fig. 25–7a. Fig. 25–7 shows how to obtain the inverted latch in a semi-graphical way. First invert the given circuit by appending a negation to the output (... b). Then, transfer the feedback to the inverted output $\overline{Y}$, inserting a negation into the feedback to make sure that the signal fed back remains unchanged (... c). Finally, perform the negations (... d, and ... e).

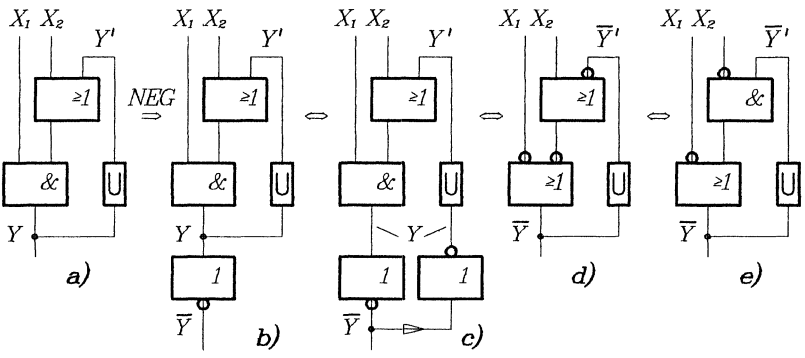


Fig. 25–7 Inverting the output of a latch semi-graphically

The same result (i.e., Fig. 25–7e) is, of course, also obtained algebraically:

$$\begin{aligned} \overline{Y}(\mathbf{x}) &\Leftrightarrow \overline{X_1(X_2 \vee Y')} && \text{(see Fig. 25–7b)} \\ &\Leftrightarrow \overline{X}_1 \vee \overline{(X_2 \vee Y')} && \text{(see Fig. 25–7d)} \\ \overline{Y}(\mathbf{x}) &\Leftrightarrow \overline{X}_1 \vee \overline{X}_2\overline{Y}'. && \text{(see Fig. 25–7e)} \end{aligned}$$

This formula, $\overline{Y}(\mathbf{x}) \Leftrightarrow \overline{X}_1 \vee \overline{X}_2\overline{Y}'$, is also directly obtainable from the K-map of Fig. 25–6c using the disjunctive formulas (25.3). When first working with latches, it is quite common to be a bit shy of applying algebra to their formulas. Therefore, I would like to encourage you to take the time to work through all aspects of this example. In the next section, we shall consider the topic more formally.

25.7 General Negation of Latches *

In the context of this section, the prime question is *'When are two given latches (mutually) inverted?'*. Prompted by the example in the previous section, we answer the question by

(25.8) Defining when latches are inverted. Two latches $f : \{0,1\}^{n+1} \mapsto \{0,1\}$ and $g : \{0,1\}^{n+1} \mapsto \{0,1\}$ are said to be inverted *iff* $\mathcal{E}_1(f) = \mathcal{E}_0(g)$ and $\mathcal{E}_0(f) = \mathcal{E}_1(g)$, while their memory equivalents are then of necessity equal $(\mathcal{E}_q(f) = \mathcal{E}_q(g))$,

$$(f = \overline{g}) \Leftrightarrow \big(\mathcal{E}_1(f) = \mathcal{E}_0(g)\big) \wedge \big(\mathcal{E}_0(f) = \mathcal{E}_1(g)\big).$$

By this criterion, the two K-maps of Fig. 25–6 specify inverted latches. Clearly, it is not important which symbols are used in defining the memory equivalents.

As latches are defined as combinational circuits (albeit, special ones — see (25.1)), all rules of switching algebra are applicable to them, especially those on negation. For actual calculations, we need to know how the inclusions $U_S(\mathbf{x}) \Leftrightarrow \mathbf{x} \in \mathcal{E}_S$, (17.27), and exclusions $V_S(\mathbf{x}) \Leftrightarrow \mathbf{x} \notin \mathcal{E}_S$, (17.29), interact. We first detail these formulas for each of the setting, resetting, and memorising equivalent $\mathcal{E}_1$, $\mathcal{E}_0$, $\mathcal{E}_q$:

$$U_1(\mathbf{x}) \Leftrightarrow \mathbf{x} \in \mathcal{E}_1, \quad U_0(\mathbf{x}) \Leftrightarrow \mathbf{x} \in \mathcal{E}_0, \quad U_q(\mathbf{x}) \Leftrightarrow \mathbf{x} \in \mathcal{E}_q, \tag{25.9}$$

$$V_1(\mathbf{x}) \Leftrightarrow \mathbf{x} \notin \mathcal{E}_1, \quad V_0(\mathbf{x}) \Leftrightarrow \mathbf{x} \notin \mathcal{E}_0, \quad V_q(\mathbf{x}) \Leftrightarrow \mathbf{x} \notin \mathcal{E}_q, \tag{25.10}$$

allowing us to state the following relationships:

$$\overline{U}_1(\mathbf{x}) \Leftrightarrow V_1(\mathbf{x}), \quad \overline{U}_0(\mathbf{x}) \Leftrightarrow V_0(\mathbf{x}), \quad \overline{U}_q(\mathbf{x}) \Leftrightarrow V_q(\mathbf{x}). \tag{25.11}$$

$$U_1(\mathbf{x}) \vee U_0(\mathbf{x}) \vee U_q(\mathbf{x}) \Leftrightarrow \mathbf{1} \qquad V_1(\mathbf{x})V_0(\mathbf{x})V_q(\mathbf{x}) \Leftrightarrow \mathbf{0}. \tag{25.12}$$

$$U_1(\mathbf{x}) \vee U_0(\mathbf{x}) \Leftrightarrow V_q(\mathbf{x}), \qquad V_1(\mathbf{x})V_0(\mathbf{x}) \Leftrightarrow U_q(\mathbf{x}), \tag{25.13}$$

$$U_1(\mathbf{x}) \vee U_q(\mathbf{x}) \Leftrightarrow V_0(\mathbf{x}), \qquad V_1(\mathbf{x})V_q(\mathbf{x}) \Leftrightarrow U_0(\mathbf{x}), \tag{25.14}$$

$$U_0(\mathbf{x}) \vee U_q(\mathbf{x}) \Leftrightarrow V_1(\mathbf{x}), \qquad V_0(\mathbf{x})V_q(\mathbf{x}) \Leftrightarrow U_1(\mathbf{x}). \tag{25.15}$$

$$U_1(\mathbf{x})U_0(\mathbf{x}) \Leftrightarrow \mathbf{0}, \qquad V_1(\mathbf{x}) \vee V_0(\mathbf{x}) \Leftrightarrow \mathbf{1}, \tag{25.16}$$

$$U_1(\mathbf{x})U_q(\mathbf{x}) \Leftrightarrow \mathbf{0}, \qquad V_1(\mathbf{x}) \vee V_q(\mathbf{x}) \Leftrightarrow \mathbf{1}, \tag{25.17}$$

$$U_0(\mathbf{x})U_q(\mathbf{x}) \Leftrightarrow \mathbf{0}, \qquad V_0(\mathbf{x}) \vee V_q(\mathbf{x}) \Leftrightarrow \mathbf{1}, \tag{25.18}$$

Let us take at least a cursory look at some of the proofs. Firstly, to prove $U_1(\mathbf{x}) \vee U_0(\mathbf{x}) \vee U_q(\mathbf{x}) \Leftrightarrow \mathbf{1}$ of (25.12) we use $\mathcal{E} = \mathcal{E}_1 \cup \mathcal{E}_0 \cup \mathcal{E}_q$, (25.1b), to write

$$\underbrace{(\mathbf{x} \in \mathcal{E})}_{\mathbf{1}} \Leftrightarrow \underbrace{(\mathbf{x} \in \mathcal{E}_1)}_{U_1(\mathbf{x})} \vee \underbrace{(\mathbf{x} \in \mathcal{E}_0)}_{U_0(\mathbf{x})} \vee \underbrace{(\mathbf{x} \in \mathcal{E}_q)}_{U_q(\mathbf{x})}.$$

Secondly, to prove $U_1(\mathbf{x}) \vee U_0(\mathbf{x}) \Leftrightarrow V_q(\mathbf{x})$ of (25.13) we again remember that $\mathcal{E} = \mathcal{E}_1 \cup \mathcal{E}_0 \cup \mathcal{E}_q$, and that the equivalents $\mathcal{E}_1$, $\mathcal{E}_0$, and $\mathcal{E}_q$ are mutually disjoint so that we may write

$$\mathcal{E}_1 \cup \mathcal{E}_0 = \mathcal{E} \setminus \mathcal{E}_q,$$

$$\underbrace{\mathbf{x} \in \mathcal{E}_1}_{U_1(\mathbf{x})} \vee \underbrace{\mathbf{x} \in \mathcal{E}_0}_{U_0(\mathbf{x})} \Leftrightarrow \mathbf{x} \in (\mathcal{E} \setminus \mathcal{E}_q) \overset{df}{\Leftrightarrow} \underbrace{\mathbf{x} \in \mathcal{E}}_{1} \wedge \underbrace{\mathbf{x} \notin \mathcal{E}_q}_{V_q(\mathbf{x})}.$$

Finally, $U_1(\mathbf{x})U_0(\mathbf{x}) \Leftrightarrow \mathbf{0}$ of (25.16) follows directly from $\mathcal{E}_1$ and $\mathcal{E}_0$ being mutually disjoint, as stated in (25.2).

Now, writing the evaluation formulas (25.3a) to (25.6a) in their basic form, i.e., for $\mathcal{E}_{1S} = \emptyset$, $\mathcal{E}_{0S} = \emptyset$, $\mathcal{E}_{\delta S} = \emptyset$, $\mathcal{E}_{\delta S^*} = \emptyset$,

$$Y(\mathbf{x}) \overset{(25.3a)}{\Longleftrightarrow} U_1(\mathbf{x}) \vee U_q(\mathbf{x})Y', \tag{25.19}$$

$$Y(\mathbf{x}) \overset{(25.4a)}{\Longleftrightarrow} V_0(\mathbf{x})(V_q(\mathbf{x}) \vee Y'), \tag{25.20}$$

$$\overline{Y}(\mathbf{x}) \overset{(25.5a)}{\Longleftrightarrow} U_0(\mathbf{x}) \vee U_q(\mathbf{x})\overline{Y}', \tag{25.21}$$

$$\overline{Y}(\mathbf{x}) \overset{(25.6a)}{\Longleftrightarrow} V_1(\mathbf{x})(V_q(\mathbf{x}) \vee \overline{Y}'), \tag{25.22}$$

the above relationships between inclusions U_1, U_0, U_q and exclusions V_1, V_0, V_q, (25.11) to (25.18), allow us to calculate the two inverted formulas of any given latch formula, e.g.,

$$\begin{aligned}
\overline{Y}(\mathbf{x}) &\Longleftrightarrow \overline{U_1(\mathbf{x}) \vee U_q(\mathbf{x})Y' \ldots (25.19)} \\
&\Longleftrightarrow \overline{U}_1(\overline{U}_q \vee \overline{Y}') \\
&\overset{(25.11)}{\Longleftrightarrow} V_1(V_q \vee \overline{Y}') \ldots (25.22) \\
&\overset{(25.15)}{\Longleftrightarrow} (U_0 \vee U_q)(V_q \vee \overline{Y}') \\
&\overset{(25.13)}{\Longleftrightarrow} (U_0 \vee U_q)(U_0 \vee U_1 \vee \overline{Y}') \\
&\Longleftrightarrow U_0(U_0 \vee U_1 \vee \overline{Y}') \vee U_q(U_0 \vee U_1 \vee \overline{Y}') \\
&\Longleftrightarrow U_0 \vee \underbrace{U_0U_1}_{\mathbf{0}} \vee U_0\overline{Y}' \vee \underbrace{U_oU_q}_{\mathbf{0}} \vee \underbrace{U_1U_q}_{\mathbf{0}} \vee U_q\overline{Y}' \\
&\Longleftrightarrow \underbrace{U_0 \vee U_0\overline{Y}'}_{U_0} \vee U_q\overline{Y}' \\
&\Longleftrightarrow U_0 \vee U_q\overline{Y}' \ldots (25.21)
\end{aligned}$$

Chapter 26

Optimised Latches

Optimising a feedback-latch designed according to either of the evaluation formulas (25.3) to (25.6) reduces to *optimising the prelogic to the generic latch* — see Section 25.3. Essentially, optimisation covers two aspects, firstly, *obtaining a minimised circuit*, and secondly, *obtaining a hazard-free circuit.* To limit explanations, we concentrate on latches with a *standard* or non-inverted output Y. These are designed according to the evaluation formulas (25.3) and (25.4).

To express formally how optimisation is carried out, we need the concepts of minimal covers $\mathcal{P}_{min}(\mathcal{E}_1)$, $\mathcal{P}_{min}(\mathcal{E}_0)$ as developed in (17.21), and of minimal irredundant covers (as $\mathcal{P}^{\overline{r}}_{min}(\mathcal{E}_1;\mathcal{E}_{\delta S})$, $\mathcal{P}^{\overline{r}}_{min}(\mathcal{E}_0;\mathcal{E}_{\delta S})$, or $\mathcal{P}^{\overline{r}}_{min}(\mathcal{E}_q;\mathcal{E}_{1S}\cup\mathcal{E}_{\delta S^*})$, $\mathcal{P}^{\overline{r}}_{min}(\mathcal{E}_q;\mathcal{E}_{0S}\cup\mathcal{E}_{\delta S^*})$), a concept developed in Section 18.3, and specifically in (18.1) to (18.3).

26.1 Designing Minimised Latches *

Minimised latches are obtained by minimising a latch's prelogic:

$$Y(\mathbf{x}) \Leftrightarrow U_{1;\delta}(\mathbf{x}) \vee U_{q;1\delta^*}(\mathbf{x})Y', \tag{26.1a}$$

$$U_{1;\delta}(\mathbf{x}) \Leftrightarrow \bigvee_{\varphi:\mathcal{P}_\varphi\in\mathcal{P}^{\overline{r}}_{min}(\mathcal{E}_1;\mathcal{E}_{\delta S})} C_\varphi(\mathbf{x}), \tag{26.1b}$$

$$U_{q;1\delta^*}(\mathbf{x}) \Leftrightarrow \bigvee_{\varphi:\mathcal{P}_\varphi\in\mathcal{P}^{\overline{r}}_{min}(\mathcal{E}_q;\mathcal{E}_{1S}\cup\mathcal{E}_{\delta S^*})} C_\varphi(\mathbf{x}). \tag{26.1c}$$

$$Y(\mathbf{x}) \Leftrightarrow V_{0;\delta}(\mathbf{x})\big(V_{q;0\delta^*}(\mathbf{x}) \vee Y'\big), \tag{26.2a}$$

$$V_{0;\delta}(\mathbf{x}) \Leftrightarrow \bigwedge_{\varphi:\mathcal{P}_\varphi\in\mathcal{P}^{\overline{r}}_{min}(\mathcal{E}_0;\mathcal{E}_{\delta S})} D_\varphi(\mathbf{x}), \tag{26.2b}$$

$$V_{q;0\delta^*}(\mathbf{x}) \Leftrightarrow \bigwedge_{\varphi:\mathcal{P}_\varphi\in\mathcal{P}^{\overline{r}}_{min}(\mathcal{E}_q;\mathcal{E}_{0S}\cup\mathcal{E}_{\delta S^*})} D_\varphi(\mathbf{x}). \tag{26.2c}$$

If the set of unassigned input events $\mathcal{E}_\delta$ is empty, $\mathcal{E}_\delta = \emptyset$, then the above formulas reduce to

$$Y(\mathbf{x}) \Leftrightarrow U_1(\mathbf{x}) \vee U_{q;1}(\mathbf{x})Y', \tag{26.3a}$$

$$U_1(\mathbf{x}) \Leftrightarrow \bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}_{min}(\mathcal{E}_1)} C_\varphi(\mathbf{x}), \tag{26.3b}$$

$$U_{q;1}(\mathbf{x}) \Leftrightarrow \bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}_{min}(\mathcal{E}_q;\mathcal{E}_{1S})} C_\varphi(\mathbf{x}). \tag{26.3c}$$

$$Y(\mathbf{x}) \Leftrightarrow V_0(\mathbf{x})\big(V_{q;0}(\mathbf{x}) \vee Y'\big), \tag{26.4a}$$

$$V_0(\mathbf{x}) \Leftrightarrow \bigwedge_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}_{min}(\mathcal{E}_0)} D_\varphi(\mathbf{x}), \tag{26.4b}$$

$$V_{q;0}(\mathbf{x}) \Leftrightarrow \bigwedge_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}^{\overline{r}}_{min}(\mathcal{E}_q;\mathcal{E}_{0S})} D_\varphi(\mathbf{x}). \tag{26.4c}$$

The above formulas of this section, of course, give you complete freedom as to the method you employ in determining the necessary *minimal* covers $\mathcal{P}_{min}(\ldots)$. The two methods discussed in previous chapters are *minimisation by K-map*, Section 18.5, and *tabular minimisation considering don't cares*, as discussed in Section 19.5.

Further note that the disjunctive formulas (26.1), and the conjunctive formulas (26.2) usually provide circuits of differing degrees of minimisation. As it is not possible to predict which of the formulas provides the simpler circuit, you need to evaluate both formulas and choose the simplest result.

26.2 A Minimised Latch — an Example

The latch we wish to design is specified in the K-maps of Fig. 26–1. As they contain no unassigned input events δ, we use the evaluation formulas as expressed in (26.3) and (26.4).

To calculate $U_1(\mathbf{x})$ of (26.3a) according to (26.3b) we need to determine a minimal (irredundant) cover $\mathcal{P}_{min}(\mathcal{E}_1)$. For our example, such a cover $\mathcal{P}_{min}(\mathcal{E}_1) = \{\mathcal{P}_{0x11}, \mathcal{P}_{01x1}, \mathcal{P}_{111x}\}$ is determined in Fig. 26–1a, and for this cover we may write

$$\begin{aligned} U_1(\mathbf{x}) &\Leftrightarrow C_{0x11}(\mathbf{x}) \vee C_{01x1}(\mathbf{x}) \vee C_{111x}(\mathbf{x}) \\ &\Leftrightarrow \overline{X}_1 X_3 X_4 \vee \overline{X}_1 X_2 X_4 \vee X_1 X_2 X_3. \end{aligned} \tag{26.5}$$

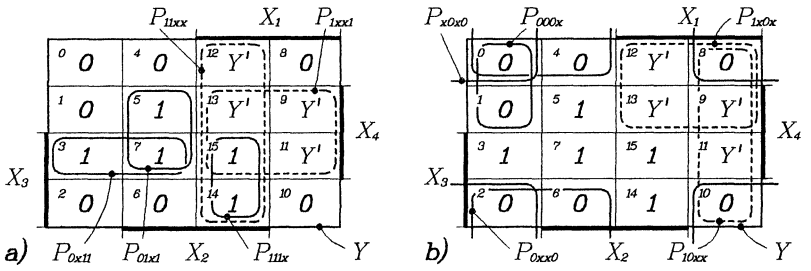


Fig. 26–1 Two latch-evaluation examples

We next need to calculate $U_{q;1}(\mathbf{x})$ of (26.3a) which is done according to (26.3c). In this formula, we note that $\mathcal{E}_{1S} \subseteq \mathcal{E}_1$ is an arbitrary subset of $\mathcal{E}_1$, the cover $\mathcal{P}(\mathcal{E}_q; \mathcal{E}_{1S}) = \{\mathcal{P}_{\varphi(1)}, \ldots, \mathcal{P}_{\varphi(L)}\}$ being a set of prime sets which cover $\mathcal{E}_q \cup \mathcal{E}_{1S}$ meaning that $\mathcal{P}_{\varphi(1)} \cup, \ldots, \cup \mathcal{P}_{\varphi(L)} = \mathcal{E}_q \cup \mathcal{E}_{1S}$. All **1**-cells, i.e., all input events $\mathbf{x}$ of $\mathcal{E}_1$ are used analogously to *don't cares* (how to cover *don't cares* with prime sets was discussed in Section 18.3). In our case, the subset $\{14, 15\}$ of $\mathcal{E}_1$, of all possible choices of subsets, leads to a set $\mathcal{E}_q \cup \mathcal{E}_{1S}$ that has the smallest cover possible $\mathcal{P}^{\overline{r}}_{min}(\mathcal{E}_q; \mathcal{E}_{1S}) = \{\mathcal{P}_{11xx}, \mathcal{P}_{1xx1}\}$ — no other choice of a subset $\mathcal{E}_{1S}$ leads to a smaller cover of $\mathcal{E}_q \cup \mathcal{E}_{1S}$. The inclusion $U_{q;1}(\mathbf{x})$ of this cover is, according to (26.3c),

$$U_{q;1}(\mathbf{x}) \Leftrightarrow C_{11xx}(\mathbf{x}) \vee C_{1xx1}(\mathbf{x}) \Leftrightarrow X_1X_2 \vee X_1X_4. \tag{26.6}$$

Our evaluation formula for latches, (26.3a), then leads to

$$\begin{aligned} Y(\mathbf{x}) &\Leftrightarrow U_1(\mathbf{x}) \vee U_{q;1}(\mathbf{x})Y' \\ &\Leftrightarrow (\overline{X}_1X_3X_4 \vee \overline{X}_1X_2X_4 \vee X_1X_2X_3) \vee (X_1X_2 \vee X_1X_4)Y'. \end{aligned} \tag{26.7}$$

The above formula gives rise to the disjunctive circuit of Fig. 26–2a. The conjunctive circuit of Fig. 26–2b is obtained from the same specification, but applying to it the formulas (26.4). To calculate $V_0(\mathbf{x})$ we first determine $\mathcal{P}_{min}(\mathcal{E}_0) = \{\mathcal{P}_{000x}, \mathcal{P}_{0xx0}, \mathcal{P}_{x0x0}\}$ as indicated in Fig. 26–1b.

$$\begin{aligned} V_0(\mathbf{x}) &\Leftrightarrow D_{000x}(\mathbf{x})D_{0xx0}(\mathbf{x})D_{x0x0}(\mathbf{x}) \\ &\Leftrightarrow (X_1 \vee X_2 \vee X_3)(X_1 \vee X_4)(X_2 \vee X_4). \end{aligned} \tag{26.8}$$

As visual inspection of Fig. 26–1b shows, we obtain a smallest possible cover of $\mathcal{E}_q$ by choosing $\mathcal{P}^{\overline{r}}_{min}(\mathcal{E}_q; \mathcal{E}_{0S}) = \{\mathcal{P}_{1x0x}, \mathcal{P}_{10xx}\}$ for which

$$V_{q;0}(\mathbf{x}) \Leftrightarrow D_{1x0x}(\mathbf{x})D_{10xx}(\mathbf{x}) \Leftrightarrow (\overline{X}_1 \vee X_3)(\overline{X}_1 \vee X_2). \tag{26.9}$$

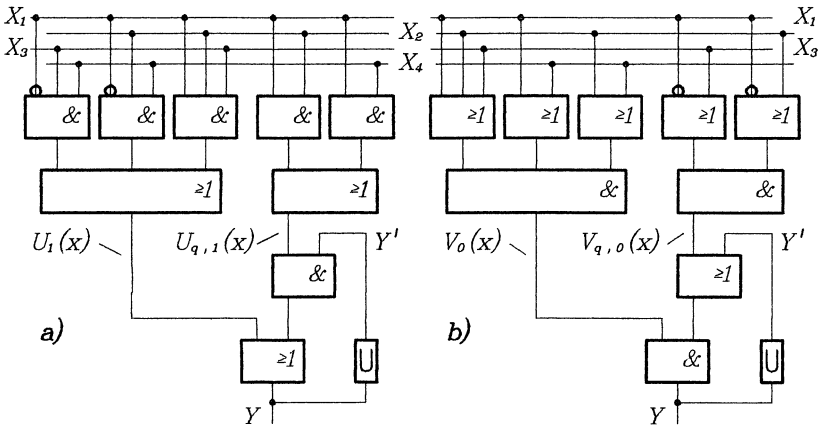


Fig. 26–2 The latches specified in Fig. 26–1

Having determined $V_0(\mathbf{x})$ and $V_{q;0}(\mathbf{x})$, we can now write (26.4a) as

$$Y(\mathbf{x}) \Leftrightarrow V_0(\mathbf{x})(V_{q;0}(\mathbf{x}) \vee Y') \tag{26.10}$$
$$\Leftrightarrow \underbrace{(X_1 \vee X_2 \vee X_3)(X_1 \vee X_4)(X_2 \vee X_4)}_{V_0(\mathbf{x})}\big(\underbrace{(\overline{X}_1 \vee X_3)(\overline{X}_1 \vee X_2)}_{V_{q;0}(\mathbf{x})} \vee Y'\big)$$

which is pictured in Fig. 26–2b. Seldom are circuits, which are designed according to (26.3) and (26.4) (or (26.1) and (26.2), to state it generally), minimised to the same degree. In the example of this section, (26.4) provides the slightly simpler solution as shown in Fig. 26–2b.

26.3 Designing Hazard-Free Latches *

It is far more important to design latches that are free of hazards than to design minimised latches. To explain hazards in latches, we must slightly extend the concept of hazards as developed for combinational circuits in Section 23.1. There, we concentrated solely on switching from one input-event to another *with both belonging to the same equivalent.*

With latches the problem is this: When switching from an arbitrary setting, resetting, or even memorising input event *to a memorising input event* (or a *new* memorising input event, in the latter case), the output, the value of which should remain unchanged, may develop a hazard. **Such a hazard poses a serious danger** because its erroneous value can be memorised thus leading to *a stable but incorrect output.* To view the mechanism in detail, let us consider the signal flow in the generic disjunctive latch of Fig. 26–3.

Assume we have designed a combinational circuit for $U_1(\mathbf{x})$ and another for $U_q(\mathbf{x})$, and that a **1**-hazard with a duration of $\Delta_H t$ *smaller* than Δt

occurs when switching from $\mathbf{x} \in \mathcal{E}_1$ to $\mathbf{x} \in \mathcal{E}_q$ — this is the situation the input signals U_0, U_1, and U_q of Fig. 26–3a depict. As you can deduce from the signal flow in the generic circuit of Fig. 26–3c, the trough ends after $\Delta_H t$ because Y' is still **1** (and would remain so until the feedback delay-time Δt were to elapse). In itself, this situation is no more serious than a hazard in a combinational circuit.

A **permanently wrong output value** is realised if the duration $\Delta_H t$ of the **1**-hazard is *larger* than the delay Δt in the feedback. As the signal flow in Fig. 26–3b shows, Y' will have already switched to **0** before U_q becomes **1** letting the output Y falsely remain **0**. An awkward situation arises when $\Delta_H t$ and Δt are roughly equal — it is then a matter of chance whether a hazard is perpetuated or not.

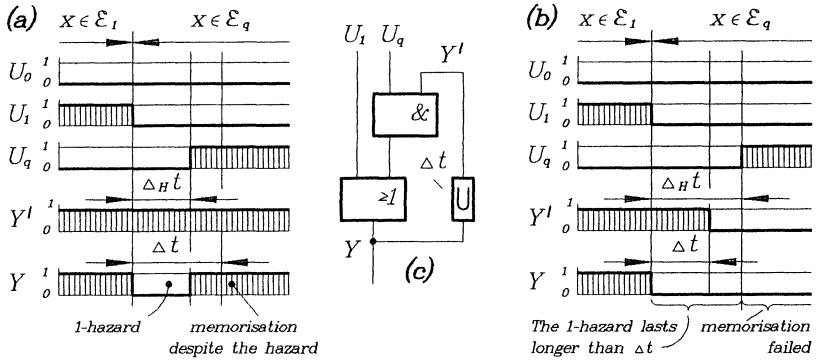


Fig. 26–3 Occurrence of a **1**-hazard and its possible perpetuation

To avoid hazards of the kind discussed above, we obviously need to ensure a hazard-free design of circuits for the unions $\mathcal{E}_1 \cup \mathcal{E}_q$ and $\mathcal{E}_0 \cup \mathcal{E}_q$. But, we need not cover the whole union $\mathcal{E}_1 \cup \mathcal{E}_q$ or $\mathcal{E}_0 \cup \mathcal{E}_q$, for, those prime sets of $\mathcal{P}(\mathcal{E}_q; \mathcal{E}_1)$ or $\mathcal{P}(\mathcal{E}_q; \mathcal{E}_0)$ that contain only input-events $\mathbf{x} \in \mathcal{E}_1$ or $\mathbf{x} \in \mathcal{E}_0$ need not be considered, i.e., can be dropped. We thus define the concept of *hazard-free memory-covers*:

(26.11) A **hazard-free memory-cover** $\mathcal{P}^{\overline{h}}_{max}(\mathcal{E}_q; \mathcal{E}_1)$ is *the* full cover of $\mathcal{E}_q \cup \mathcal{E}_1$ from which those prime sets have been omitted that contain only elements of $\mathcal{E}_1$.

A **hazard-free memory-cover** $\mathcal{P}^{\overline{h}}_{max}(\mathcal{E}_q; \mathcal{E}_0)$ is *the* full cover of $\mathcal{E}_q \cup \mathcal{E}_0$ from which those prime sets have been omitted that contain only elements of $\mathcal{E}_0$.

With these concepts, we can specialise the evaluation formulas (26.1) to (26.4) to obtain **evaluation formulas for hazard-free circuits**. Rewriting (26.1) and (26.2) we get:

$$Y(\mathbf{x}) \Leftrightarrow U_{1;\delta}(\mathbf{x}) \vee U_{q;1\delta^*}(\mathbf{x})Y', \tag{26.12a}$$

$$U_{1;\delta}(\mathbf{x}) \Leftrightarrow \bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}^{\overline{r}}_{max}(\mathcal{E}_1;\mathcal{E}_{\delta S})} C_\varphi(\mathbf{x}), \tag{26.12b}$$

$$U_{q;1\delta^*}(\mathbf{x}) \Leftrightarrow \bigvee_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}^{\overline{h}}_{max}(\mathcal{E}_q;\mathcal{E}_{1S} \cup \mathcal{E}_{\delta S^*})} C_\varphi(\mathbf{x}). \tag{26.12c}$$

$$Y(\mathbf{x}) \Leftrightarrow V_{0;\delta}(\mathbf{x})\big(V_{q;0\delta^*}(\mathbf{x}) \vee Y'\big), \tag{26.13a}$$

$$V_{0;\delta}(\mathbf{x}) \Leftrightarrow \bigwedge_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}^{\overline{r}}_{max}(\mathcal{E}_0;\mathcal{E}_{\delta S})} D_\varphi(\mathbf{x}), \tag{26.13b}$$

$$V_{q;0\delta^*}(\mathbf{x}) \Leftrightarrow \bigwedge_{\varphi:\mathcal{P}_\varphi \in \mathcal{P}^{\overline{h}}_{max}(\mathcal{E}_q;\mathcal{E}_{0S} \cup \mathcal{E}_{\delta S^*})} D_\varphi(\mathbf{x}). \tag{26.13c}$$

How hazard-free are hazard-free latches that are calculated by the above evaluation formulas (26.12) or (26.13)? Let me answer this question in two steps, firstly stating the following

(26.14) Theorem on partially hazard-free latches.
(a) Any *disjunctive* latch calculated by (26.12) is *per se* free of single-input **0**-hazards (whether it has a **1**-hazard of not).
(b) Any *conjunctive* latch calculated by (26.13) is *per se* free of single-input **1**-hazards (whether it has a **0**-hazard of not).

Let us verify (26.14a). Take $\mathbf{x}$ to be a resetting input-event. Then, not only is $\mathbf{x} \in \mathcal{E}_0 \Leftrightarrow \mathbf{1}$, $\mathbf{x} \in \mathcal{E}_1 \Leftrightarrow \mathbf{0}$, and $\mathbf{x} \in \mathcal{E}_q \Leftrightarrow \mathbf{0}$, but also $\mathbf{x} \in (\mathcal{E}_q \cup \mathcal{E}_{1S}) \Leftrightarrow \mathbf{0}$. With $U_S(\mathbf{x}) \Leftrightarrow \mathbf{x} \in \mathcal{E}_S$, (17.27), the above expressions show the inclusions to have the following values: $U_0(\mathbf{x}) \Leftrightarrow \mathbf{1}$, $U_1(\mathbf{x}) \Leftrightarrow \mathbf{0}$, $U_q(\mathbf{x}) \Leftrightarrow \mathbf{0}$, and $U_{q;1}(\mathbf{x}) \Leftrightarrow \mathbf{0}$. Using the values for $U_1(\mathbf{x})$ and $U_{q;1}(\mathbf{x})$ in (26.12a), the output Y is **0**, as is Y'. When switching to a memorising input-event $\mathbf{x}^* \in \mathcal{E}_q$, for which $U_q(\mathbf{x}^*)$ is **1**, and therefore also $U_{q;1}(\mathbf{x})$, the conjunction $U_{q;1}(\mathbf{x}^*)Y'$ remains **0**, as Y' is **0**. The output Y remaining **0** excludes the existence of a pulse thus proving that the circuit is *per se* free of **0**-hazards.

Secondly, an immediate consequence of the above theorem is the important

(26.15) Theorem on completely hazard-free latches:
(a) A *disjunctive* latch evaluated by (26.12) so as to be free of single-input **1**-hazards is also completely free of single-input hazards as it is *per se* free of single-input **0**-hazards.
(b) A *conjunctive* latch evaluated by (26.13) so as to be free of single-input **0**-hazards is also completely free of single-input hazards as it is *per se* free of single-input **1**-hazards.

To see that (26.15a) is correct, we need only show that switching from a setting input-event $\mathbf{x} \in \mathcal{E}_1$ to a memorising input event $\mathbf{x}^* \in \mathcal{E}_q$ does not cause a **1**-hazard. If $\mathbf{x} \in \mathcal{E}_1 \Leftrightarrow \mathbf{1}$, then $U_1(\mathbf{x})$, $U_{q;1}(\mathbf{x})$, Y, and Y' are all **1**. When switching to a memorising input-event $\mathbf{x}^* \in \mathcal{E}_q$, for which $U_q(\mathbf{x}^*)$ is **1**, and therefore $U_{q;1}(\mathbf{x}^*)$, the conjunction $U_{q;1}(\mathbf{x}^*)Y'$ remains **1** as Y' is **1**. The output Y remaining **1** excludes the existence of a trough which means that no **1**-hazard occurs when switching from a setting to a memorising input event. No doubt, you will have no difficulty in proving (26.15b) along the lines sketched above.

26.4 A Hazard-Free Latch — an Example

It is instructive to follow the signal flow in an example latch to see how a hazard-free output-signal is brought about. In the example latch specified in Fig. 26–4 the full cover

$$\mathcal{P}_{max}(\mathcal{E}_1) = \{\mathcal{P}_{0x11}, \mathcal{P}_{01x1}, \mathcal{P}_{x11x}\}, \tag{26.16}$$

by which to calculate U_1 or V_1, is determined in the K-map of Fig. 26–4a by visual inspection.

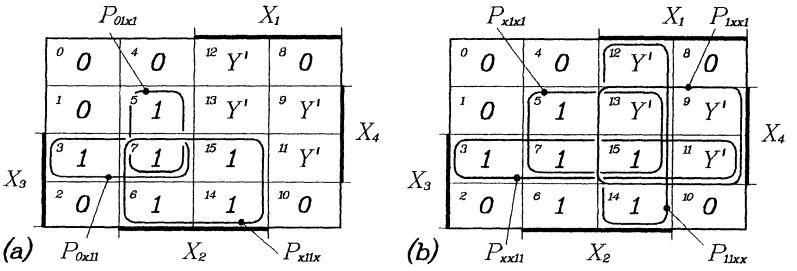


Fig. 26–4 An example latch. (a) $\mathcal{P}_{max}(\mathcal{E}_1)$, (b) $\mathcal{P}^{\overline{h}}_{max}(\mathcal{E}_q; \mathcal{E}_1)$

The hazard-free memory-cover $\mathcal{P}^{\overline{h}}_{max}(\mathcal{E}_q; \mathcal{E}_1)$ is comprised of the prime sets of Fig. 26–4b,

$$\mathcal{P}^{\overline{h}}_{max}(\mathcal{E}_q; \mathcal{E}_1) = \{\mathcal{P}_{1xx1}, \mathcal{P}_{11xx}, \mathcal{P}_{x1x1}, \mathcal{P}_{xx11}\}. \tag{26.17}$$

Each prime set contains at least one Y'-cell thus allowing for all possible single-input transitions from a **1**-cell to a Y'-cell to take place within a prime set encompassing both cells. This explains why the **1**-cell $\{6\}$ need not be an element of a prime set contained in $\mathcal{P}^{\overline{h}}_{max}(\mathcal{E}_q; \mathcal{E}_1)$: Hazard-free behaviour refers to single-input transitions, i.e., transitions between adjacent cells, and there is no adjacent Y'-cell to the **1**-cell $\{6\}$.

The covers $\mathcal{P}_{max}(\mathcal{E}_1)$ and $\mathcal{P}^{\overline{h}}_{max}(\mathcal{E}_q;\mathcal{E}_1)$, from (26.16) and (26.17) enable us to write the hazard-free formula

$$\begin{aligned} Y(\mathbf{x}) &\overset{(26.12)}{\Longleftrightarrow} U_1(\mathbf{x}) \vee U_{q;1}(\mathbf{x})Y' \\ &\Longleftrightarrow (X_2X_3 \vee \overline{X}_1X_3X_4 \vee \overline{X}_1X_2X_4) \\ &\qquad \vee (\underbrace{X_1X_2 \vee X_1X_4}_{minimal} \vee \underbrace{X_2X_4 \vee X_3X_4}_{hazard-free})Y'. \end{aligned} \tag{26.18}$$

The circuit which is equivalent to this formula is shown in Fig. 26–5. The K-map of Fig. 26–4, specifying the latch, allows us to recognise all transitions from any **1**-cell (or setting input-event) to adjacent Y'-cells (or memorising input-events). These are the single-variable transitions (3-to-11), (5-to-13), (7-to-15), (14-to-12), (15-to-11), (15-to-13) which might cause hazards — hazards that could be perpetuated.

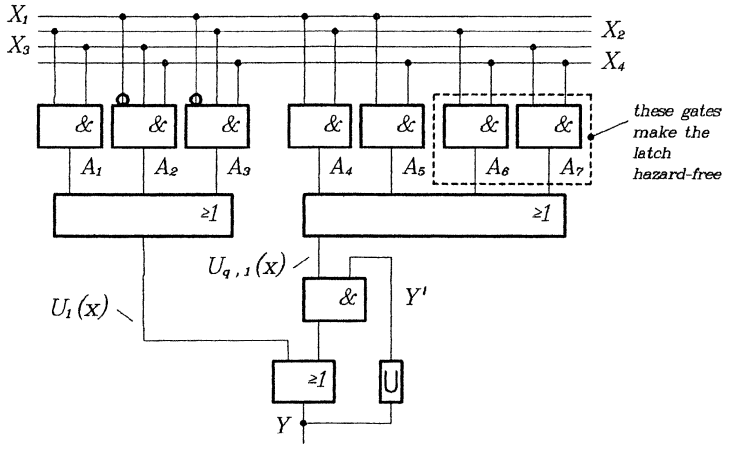


Fig. 26–5 The hazard-free circuit of Fig. 26–4 according to (26.18)

We take a closer look at the signal flow for the transition from (3-to-11). The signal flow for the setting input-event $(3)_{10} = (0,0,1,1)_2$ is shown in Fig. 26–6a. The latch is set via the AND-output A_2. But the thing to note is that the AND-output A_7 is also **1**, this making $U_{q;1}(\mathbf{x}) \Leftrightarrow \mathbf{1}$. The output value **1** is fed back via the delay to the loop's AND-gate, the output of which now becomes **1** ($U_{q;1}(\mathbf{x}) \wedge Y' \Leftrightarrow \mathbf{1}$). One says *'the loop has been closed'*.

Now, additionally, letting X_1 become **1**, as pictured in Fig. 26–6b, we note that the latch's setting signal, $U_1(\mathbf{x})$, is no longer **1**. But, as this has no influence on the value of the signal in the closed loop, the output value of the latch remains unfalteringly **1**, guaranteeing that the output is hazard-free. The **1**-signal circling in the closed loop when a memorising input event is present, is said to have been **pre-established**, meaning it was established during the presents of the preceding setting input-event.

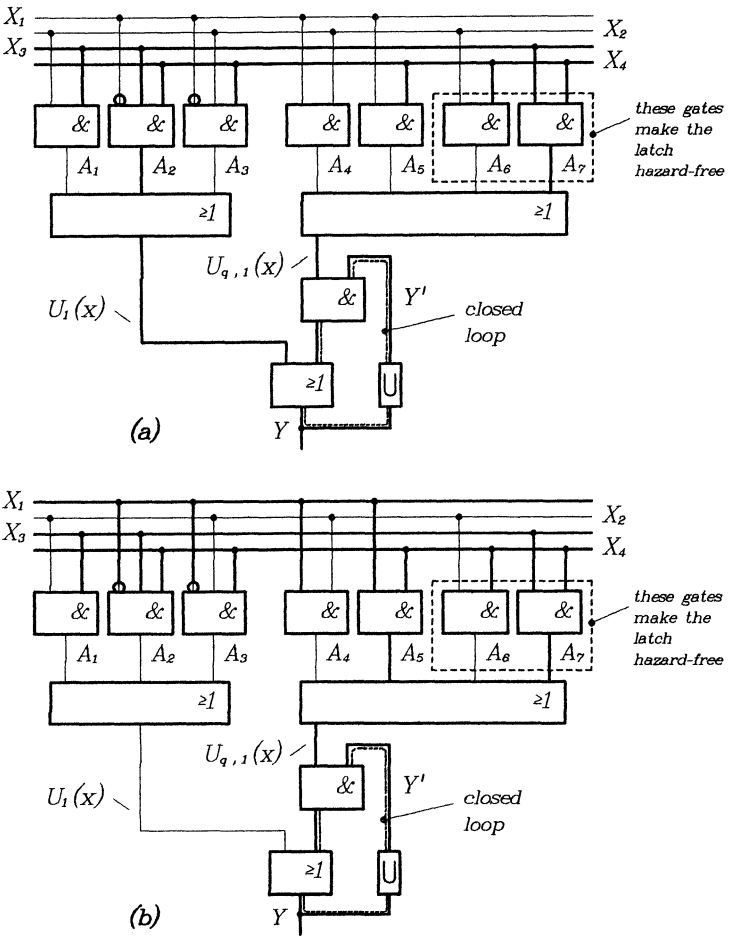


Fig. 26–6 Showing the signal flow for the input transition from (a) $(\mathbf{0},\mathbf{0},\mathbf{1},\mathbf{1})$ to (b) $(\mathbf{1},\mathbf{0},\mathbf{1},\mathbf{1})$

Pre-establishing a closed feedback-loop theoretically enables a latch to react immediately and correctly to a memorising input-event of arbitrarily short duration. Allow me to expound this. Suppose you have designed a disjunctive latch according to (26.12) making it free of **1**-hazards. Then, applying a setting input-event for longer than Δt (the time lag of the delay element in the feedback), you will have a **1**-signal circling in the closed feedback-loop. This signal is maintained or memorised the moment you switch to a memorising input-event thereby providing a correct output value, however brief the duration of the memorising input is. An analogous behaviour holds for a conjunctive latch designed according to (26.13). Let us refer to this behaviour as **rapid reaction**.

Chapter 27

Elementary Latches

An **elementary latch** is a latch with two inputs and one output. There are 36 such latches, as compared with 10 combinational functions of two inputs and one output. It seems wise to discard 8 of the 36 latches as too risky to be technically feasible, still leaving us with 28 elementary latches. Any of these can appear in the realisation of sequential circuits — how many and which ones depends, for instance, on the so-called *'word-recognition tree'* specifying the circuit.

This chapter presents an overview over the elementary latches, first classifying them, and then developing a system of latch symbols. The few latch symbols standardised in IEC 117-15 simply don't cover the problem.

27.1 Classification of Elementary Latches

A latch can be described by a table of asserted events (TAE). The header of a TAE for an elementary latch, see Fig. 27–1, contains the two binary inputs x_1 and x_2. The table has four columns which we number by the decimal equivalent x of the input event (x_1, x_2). The columns are *weighted* from left to right by decreasing powers of 2. The output values $y(0)$, $y(1)$, $y(2)$, $y(3)$ assigned to the four columns stand for one of the symbols 0, 1, or q, i.e., $y(x) \in \{0, 1, q\}$. As each symbol 0, 1, or q must occur at least once in any given row, one of the symbols must occur exactly *twice*. A row containing two 1s is said to describe a **predominantly setting latch**. A row with two 0s describes a **predominantly resetting latch** while a **predominantly memorising latch** is described by a row in which the symbol q occurs twice.

Each latch $f_{s,r}$, and its output $y_{s,r}(\mathbf{x})$, is specified by *two* indices. The first index, denoted as s and called the **set index**, describes the position of 1s (i.e., the columns containing the 1s), while the second index, the so-called **reset index** r describes the columns containing 0s. They are determined as follows:

$$s = \sum_{x=0}^{2^2-1} s_x \cdot 2^{(2^2-1)-x}, \qquad \begin{cases} s_x = 1 & \text{if } y(x) = 1 \\ s_x = 0 & \text{if } y(x) \neq 1 \end{cases} \tag{27.1}$$

$x_1 =$	0	0	1	1	
$x_2 =$	0	1	0	1	
$x \ldots$	0	1	2	3	$\ldots$ input event x
$(2^2-1)-x \ldots$	3	2	1	0	$\ldots$ complement of x
$2^{(2^2-1)-x} \ldots$	2^3	2^2	2^1	2^0	$\ldots$ *weight* of $y(x)$
$y_{s,r}(x) =$	$y(0)$	$y(1)$	$y(2)$	$y(3)$	$\ldots$ output value $y(x)$

Fig. 27–1 The full header of a TAE for two inputs

$$r = \sum_{x=0}^{2^2-1} r_x \cdot 2^{(2^2-1)-x}, \qquad \begin{cases} r_x = 1 & \text{if } y(x) = 0 \\ r_x = 0 & \text{if } y(x) \neq 0 \end{cases} \tag{27.2}$$

The predominantly memorising latches are summarised in the two tables of Fig. 27–2. The latches in equal rows are mutually inverted. Using the set and reset indices together with definition (25.8), we can state that **two latches are negated** if the set and reset indices of one latch equal, respectively, the reset and set indices of the other:

$$f_{s,r} = \overline{f}_{r,s} \tag{27.3}$$

$x_1 =$	0	0	1	1	
$x_2 =$	0	1	0	1	
$x =$	0	1	2	3	
$y_{2,1}(x) =$	q	q	1	0	a
$y_{4,1}(x) =$	q	1	q	0	a
$y_{4,2}(x) =$	q	1	0	q	c
$y_{8,1}(x) =$	1	q	q	0	d
$y_{8,2}(x) =$	1	q	0	q	e
$y_{8,4}(x) =$	1	0	q	q	e

$x_1 =$	0	0	1	1	
$x_2 =$	0	1	0	1	
$x =$	0	1	2	3	
$y_{1,2}(x) =$	q	q	0	1	b
$y_{1,4}(x) =$	q	0	q	1	b
$y_{2,4}(x) =$	q	0	1	q	c
$y_{1,8}(x) =$	0	q	q	1	d
$y_{2,8}(x) =$	0	q	1	q	f
$y_{4,8}(x) =$	0	1	q	q	f

Fig. 27–2 The predominantly memorising latches

Latches marked with equal alphabetic letters 'a' to 'f' in Fig. 27–2 are isomorphic (see Section 25.5). When considering only two inputs, as with elementary latches, we can equally well refer to **isomorphic latches** as **transposed latches** — latches you obtain by interchanging the input variables to the circuits.

The predominantly setting and resetting latches are summarised in the two tables of Fig. 27–3. Here too, the latches in equal rows are complementary. The isomorphic latches that are predominantly setting are marked 'a1' to 'd1', those of the predominantly resetting latches are marked 'a0' to 'd0'.

Predominantly setting latches

$x_1 =$	0	0	1	1	
$x_2 =$	0	1	0	1	
$x =$	0	1	2	3	
$(y_{6,1}(x) =)$	q	1	1	0	
$y_{10,1}(x) =$	1	q	1	0	a1
$y_{12,1}(x) =$	1	1	q	0	a1
$y_{5,2}(x) =$	q	1	0	1	b1
$(y_{9,2}(x) =)$	1	q	0	1	
$y_{12,2}(x) =$	1	1	0	q	c1
$y_{3,4}(x) =$	q	0	1	1	b1
$(y_{9,4}(x) =)$	1	0	q	1	
$y_{10,4}(x) =$	1	0	1	q	c1
$y_{3,8}(x) =$	0	q	1	1	d1
$y_{5,8}(x) =$	0	1	q	1	d1
$(y_{6,8}(x) =)$	0	1	1	q	

Predominantly resetting latches

$x_1 =$	0	0	1	1	
$x_2 =$	0	1	0	1	
$x =$	0	1	2	3	
$(y_{1,6}(x) =)$	q	0	0	1	
$y_{1,10}(x) =$	0	q	0	1	a0
$y_{1,12}(x) =$	0	0	q	1	a0
$y_{2,5}(x) =$	q	0	1	0	b0
$(y_{2,9}(x) =)$	0	q	1	0	
$y_{2,12}(x) =$	0	0	1	q	c0
$y_{4,3}(x) =$	q	1	0	0	b0
$(y_{4,9}(x) =)$	0	1	q	0	
$y_{4,10}(x) =$	0	1	0	q	c0
$y_{8,3}(x) =$	1	q	0	0	d0
$y_{8,5}(x) =$	1	0	q	0	d0
$(y_{8,6}(x) =)$	1	0	0	q	

Fig. 27–3 The predominantly setting and resetting latches

Those latches — $f_{6,1}$, $f_{9,2}$, $f_{9,4}$, $f_{6,8}$, and $f_{1,6}$, $f_{4,9}$, $f_{2,9}$, $f_{8,6}$ — whose outputs $y_{s,r}(x)$ are written in parentheses lead to **risky circuits** as explained for the examples $f_{6,1}$ and $f_{1,6}$ of Fig. 27–4. Consider the predominantly resetting latch of Fig. 27–4a. Its characteristic property is that the only setting input event, cell 3, is *not* adjacent to the (only) memorising input event, cell 0.

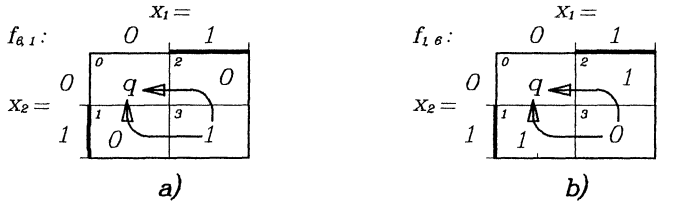


Fig. 27–4 Technically *risky* elementary latches

To memorise the output value of 1, caused by the setting input event $(1,1)$, both input variables must change their values *simultaneously* from 1 to 0. If simultaneousness is not achieved, then switching from $(1,1)$ to $(0,0)$ does not occur directly, rather $(3)_{10} = (1,1)_2$ switches via $(1)_{10} = (0,1)_2$ to $(0)_{10} = (0,0)_2$ if x_1 changes its value first (lower path), or $(3)_{10} = (1,1)_2$ switches via $(2)_{10} = (1,0)_2$ to $(0)_{10} = (0,0)_2$ if x_2 changes its value first (upper path). The input events $(0,1)$ and $(1,0)$ are both *resetting* input

events and when *previous* to the present and memorising input event $(0,0)$ for longer than Δt cause the output value of 0 they invoke to be memorised and not the value of 1 invoked by the setting input event $(1,1)$.

27.2 Two Systems of Symbolic Latches *

Remember that *gates* were introduced as graphic symbols that represent logic connectives (such as NEGATION, AND, OR), or, more precisely, certain logic functions. Graphic symbols for latches have not been given a class name, so I take the liberty of referring to them as *symbolic latches*. That is, a **symbolic latch** is a graphic symbol that represents a switching function that is a latch according to (25.1). Gates and symbolic latches thus stand for an *idealised* behaviour that can be *approximated* by physical realisations.

The two systems of symbolic latches put forth in the following sections apply only to elementary latches. Extending them to three or more inputs is theoretically possible but leads to unwieldy symbols. The systems of symbolic latches proposed here are an open criticism of the symbolism put forth in IEC 117-15.

To get a first impression, let us consider a symbol for the latch $f_{1,10}$ the latter represented by the reduced K-map of Fig. 27–5a. The associated symbolic latch has two input lines (numbered 1 and 2) and a single output line. Simply as a matter of choice, let input line 1 carry the value of input variable X_1, and input line 2 carry the value of input variable X_2. The output line is associated with the *setting* input event $(1,1)$. The input events $(0,0)$ and $(1,0)$, dissociated from the output line, comprise the *resetting* input events. Numbering the input lines is essential in defining the input events used in the symbolic latch: The value in the first position of an input event is the value of the variable carried on input line 1; the value in the second position of an input event is the value of the variable carried on input line 2. The symbol of Fig. 27–5b can be simplified if the two individual resetting input events $(0,0)$ and $(1,0)$ are replaced by the *generalised* input event $(x,0)$ leading to the symbolic latch of Fig. 27–5c.

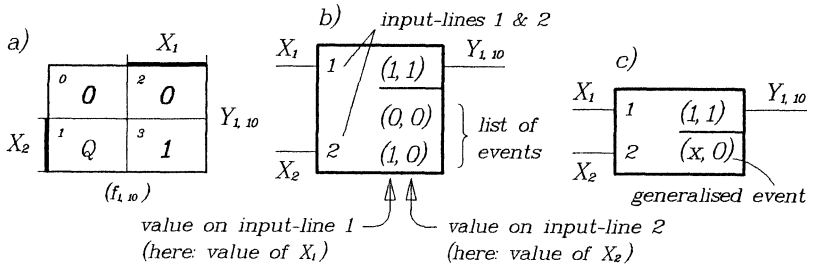


Fig. 27–5 Introducing symbolic latches by example

Every symbolic latch represents two *mutually isomorphic* (or possibly self-isomorphic) latches obtained by transposing the input variables. You can verify this for our example by comparing Figs. 27–5 and 27–6.

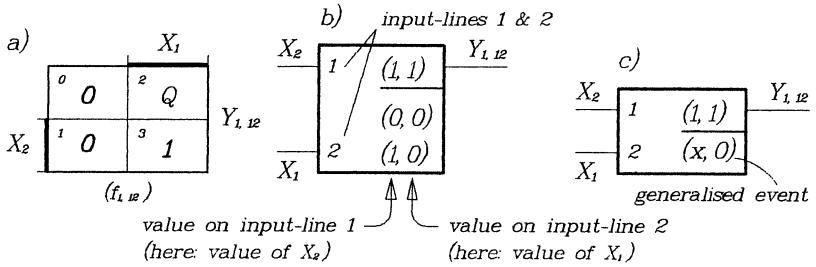


Fig. 27–6 Transposing the input variables leads to isomorphic latches

We now expand the basic concept for symbolic latches by introducing a second output. In the **first model** the two outputs are taken to be *complementary*, see Fig. 27–7a. This is a consequence of the fact that the setting input events $\mathcal{E}_1(f)$ of a latch f are the resetting input events $\mathcal{E}_0(g)$ of a complementary latch g, i.e., $\mathcal{E}_0(g) = \mathcal{E}_1(f)$, and *vice versa*, the resetting input events $\mathcal{E}_0(f)$ of f equal the setting input events of g, i.e., $\mathcal{E}_1(g) = \mathcal{E}_0(f)$. Then, associating the output Y_f of f with $\mathcal{E}_1(f)$, and the output Y_g of g with $\mathcal{E}_0(f)$ ensures that $Y_f = \overline{Y}_g$.

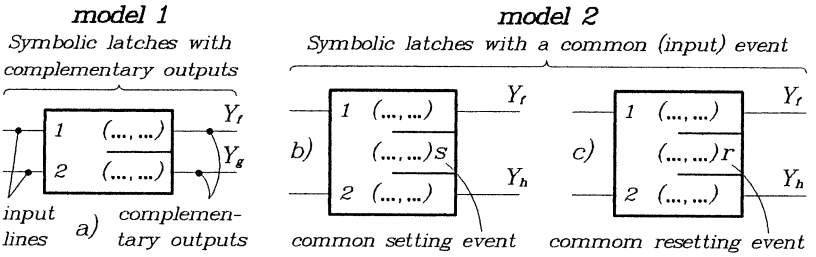


Fig. 27–7 Two basic models for symbolic latches

The **second model** is not as obvious as the one discussed above. It requires combining two specific, non-negated latches. First of all, it applies only to latches with a 1-element memory-equivalent $\mathcal{E}_q(f)$, i.e., to predominantly setting and resetting latches (under omission of the *risky* latches discussed in connection with Fig. 27–4). The outputs Y_f and Y_h of the second model (see Fig. 27–7b,c) are related in the following way. The (single) input event associated with Y_f sets this output and resets the second output Y_h; the (single) input event associated with Y_h sets this output and resets the first output Y_f. The third and also non-memorising input event either *sets both outputs* (as in the basic latch of Fig. 27–7b), or *resets both outputs* (as in the basic latch of Fig. 27–7c). These symbolic latches, as is shown later, can

be **approximated** by so-called *cross-coupled feedback circuits*, which is the actual reason for introducing this model.

The *'internal functionality'* of a symbolic latch (just as with any *gate*) is not changed by inverting any inputs or outputs.

27.3 Symbols with Complementary Outputs *

All the 16 predominantly setting and resetting latches summarised in the two tables of Fig. 27–3 (excluding the *risky* ones written in parentheses) are represented by one of the *four* symbolic latches of Fig. 27–8. That is, each symbolic latch can represent four latches. For instance, the symbolic latch in the first row is shown to represent the two latches $f_{4,3}$ and $f_{3,4}$. If the input variables X_1 and X_2 are transposed, the same symbol represents the isomorphic latches $f_{2,5}$ and $f_{5,2}$. In this case, the input event $(0,1)$ — the first value (the 0) now standing for X_2, the second (the 1) standing for X_1 — would set the output $Y_{2,5}$ and resets $Y_{5,2}$. Or, viewing it conversely, the generalised input event $(1,x)$ would set $Y_{5,2}$, and reset $Y_{2,5}$. You might like to sketch the situation isomorphic to the first row of Fig. 27–8 so as not to become confused.

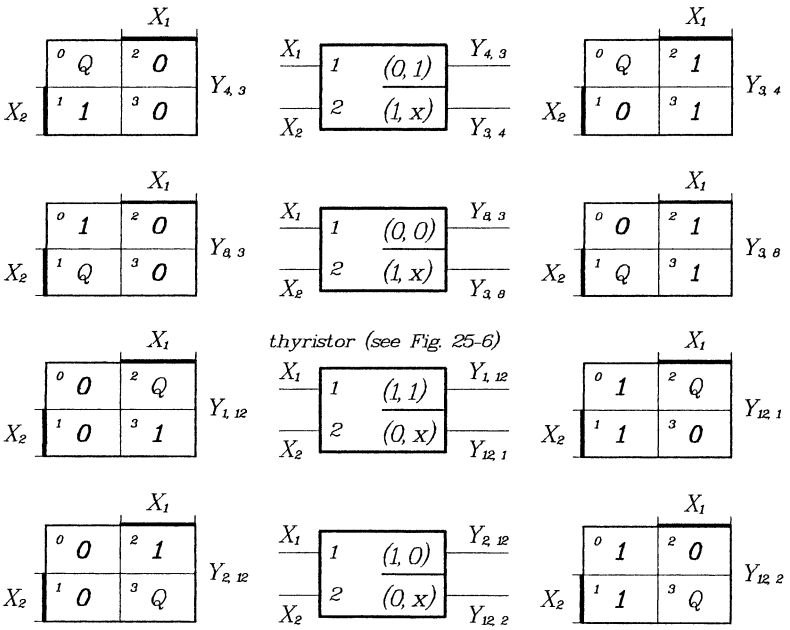


Fig. 27–8 Four symbolic latches that represent all 16 predominantly setting and resetting latches

There are no *risky* latches in the sense discussed in Fig. 27–4 among the memorising latches listed in the table of Fig. 27–2. The *four* symbolic latches of Fig. 27–9 represent all 12 memorising latches. The symbolic latch in the first row, when used as shown, represents the latches $f_{2,1}$ and $f_{1,2}$; transposing its inputs, it represents $f_{4,1}$ and $f_{1,4}$ as Fig. 27–2 shows. In a like manner, the symbolic latch in the fourth and last row represents $f_{8,2}$ and $f_{2,8}$ when used as drawn, and represents $f_{8,4}$ and $f_{4,8}$ when the input variables are transposed (as you can see from Fig. 27–2). Thus the two symbols just discussed allow us to represent 8 of the 12 memorising latches.

The remaining latches — $f_{4,2}$ with $f_{2,4}$, and $f_{8,1}$ with $f_{1,8}$ — are self-isomorphic. This means, transposing the input variables likewise causes the output variables to be transposed. Thus, the symbolic symbols in the second and third rows of Fig. 27–9 each represent only two latches, the ones shown.

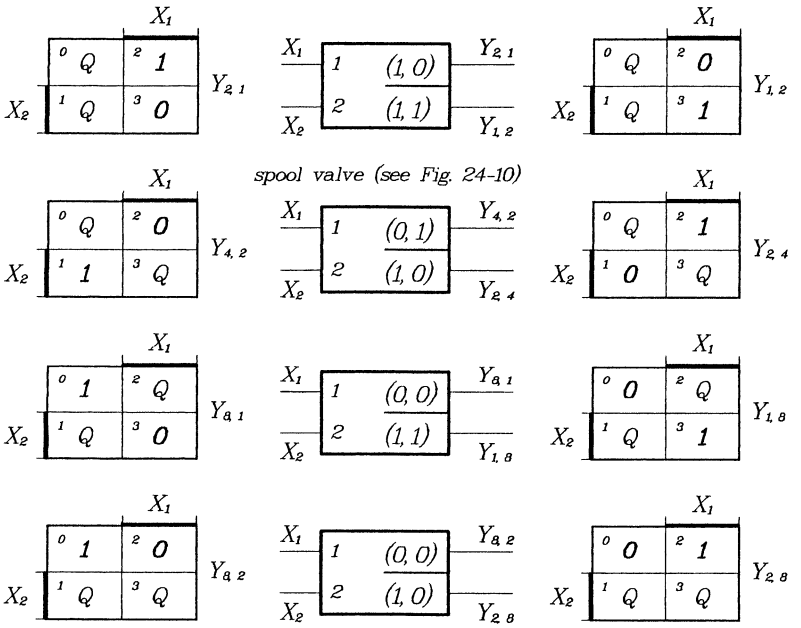


Fig. 27–9 Four symbolic latches that represent all 12 predominantly memorising latches

27.4 Symbolic Latches with a Common Event *

In this section we consider **model 2** of the symbolic latches introduced in Section 27.2 and Fig. 27–7. It is characteristic to both models of symbolic latches presented in this chapter that any input event associated with one

of the outputs sets that output and resets the other. The symbolic latches discussed in this section associate exactly one input event to each of the two outputs of the symbol while a third input event is not associated uniquely with a single output in the way just discussed, but, rather, is assumed to either *set both outputs simultaneously* or to *reset both outputs simultaneously.* This input event is said to be the **common (input) event** because its effect is common to both outputs.

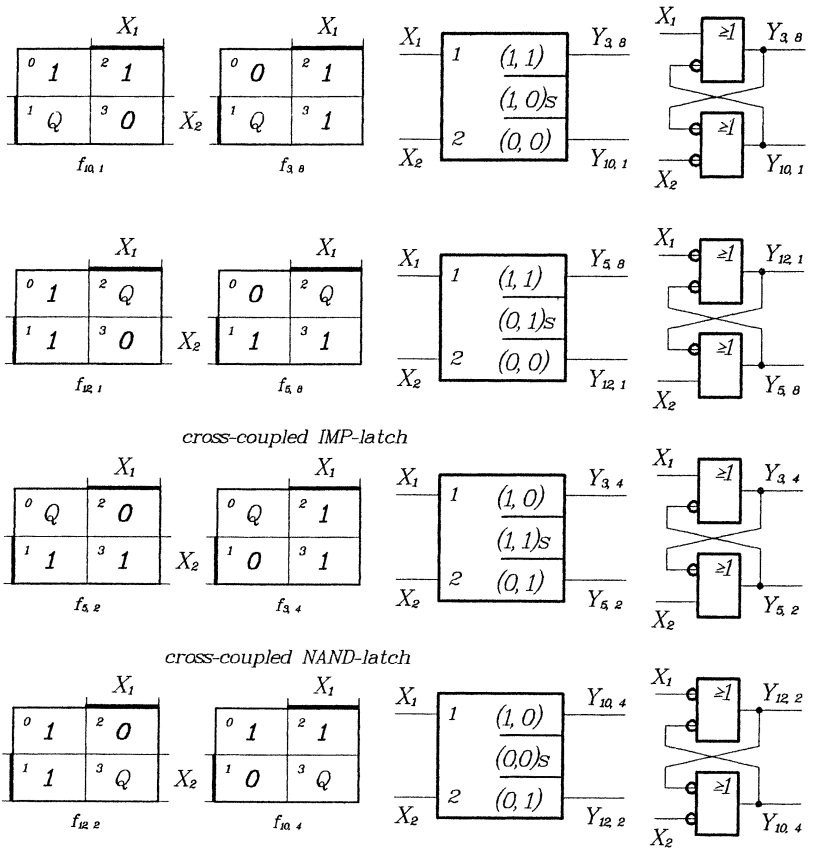


Fig. 27–10 The symbolic latches with a common setting-event

Let us first take a detailed look at the **symbolic latches with a common setting-event** summarised in Fig. 27–10. Obviously, as the common event is a *setting* event, all the latches represented by the symbolic latches in the figure are *predominantly setting latches* so that the two outputs of a symbolic latch are *not* complementary. Now, look at the two K-maps in any given row. They both have identical *memorising* input-events, each flanked by an adjacent setting and resetting input-event, these being complementary

in the two K-maps. The input event non-adjacent to the memorising input-event is always a setting input-event — it is the *common setting-event*. The outputs of a symbolic latch are associated with an input event adjacent to the memorising input-event. The common event is dissociated from the outputs and marked with an *'s'*.

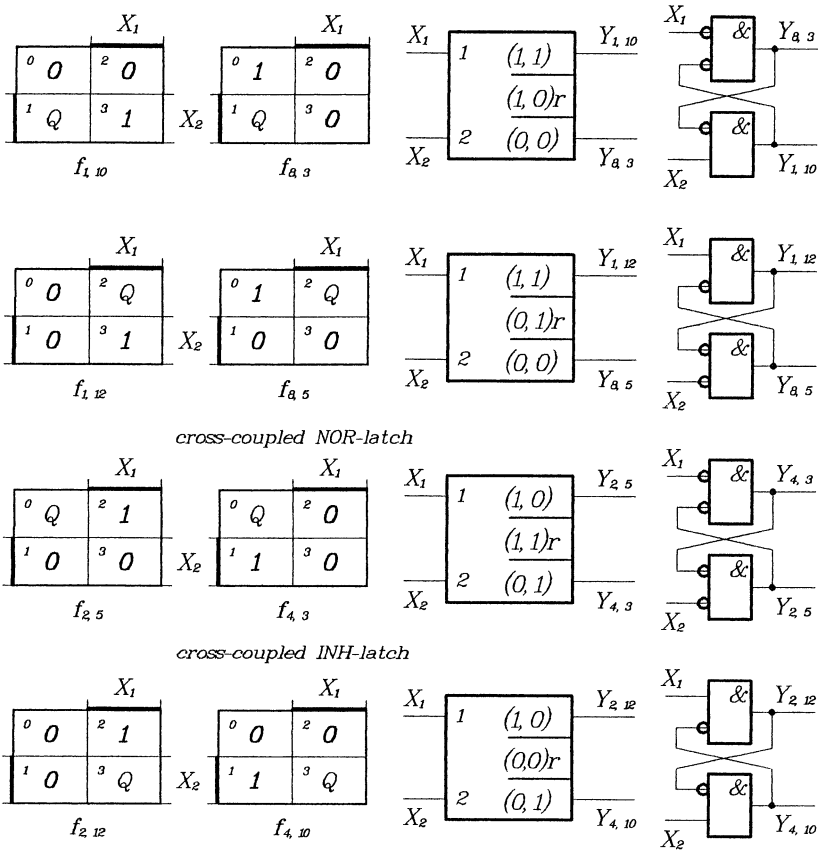


Fig. 27–11 The symbolic latches with a common resetting-event

The reason for introducing symbolic latches with a common event is that their theoretical behaviour is approximated by **delay-free cross-coupled feedback circuits** convenient to realise (electronically, by NOR and NAND). Let us calculate the circuit in the first row of Fig. 27–10 using (25.3).

$$Y_{10,1} \Leftrightarrow \overline{X}_2 \vee \overline{X}_1 Y_{10,1} \Leftrightarrow \overline{X}_2 \vee \overline{(\overline{\overline{X}_1 Y_{10,1}})} \Leftrightarrow \overline{X}_2 \vee (X_1 \vee \overline{Y}_{10,1}),$$

$$Y_{3,8} \Leftrightarrow X_1 \vee X_2 Y_{3,8} \Leftrightarrow X_1 \vee \overline{(\overline{X_2 Y_{3,8}})} \Leftrightarrow X_1 \vee (\overline{X}_2 \vee \overline{Y}_{3,8}).$$

Allow me to encourage you to compare the above formulas carefully with the circuit in the first row of Fig. 27–10. The circuit can only be drawn

because the time delays in the above formulas have been omitted. On the necessity of a delay in the feedback see Section 24.5.

The **symbolic latches with a common resetting-event** are summarised in Fig. 27–11. A detailed explanation of this figure seems superfluous. Its circuits are evaluated using (25.4) as shown for the circuit of the first row.

$$Y_{1,10} \Leftrightarrow X_2(X_1 \vee Y_{1,10}) \Leftrightarrow X_2\overline{(\overline{X_1 \vee Y_{1,10}})} \Leftrightarrow X_2\overline{(\overline{X}_1\overline{Y}_{1,10})}.$$

$$Y_{8,3} \Leftrightarrow \overline{X}_1(\overline{X}_2 \vee Y_{8,3}) \Leftrightarrow \overline{X}_1\overline{(\overline{X}_2 \vee Y_{8,3})} \Leftrightarrow \overline{X}_1\overline{(X_2\overline{Y}_{8,3})},$$

Finally, you might like to note that the outputs $Y_{s,r}$ and $Y_{r,s}$ of the latches in equal rows of the Figs. 27–10 and 27–11 are complementary as follows from a comparison of their K-maps or from (27.3).

27.5 Standard Symbols for Elementary Latches

The IEC 117-15 standard has developed five symbols for latches with two inputs and two outputs. One such latch is given the name *gated D-latch* (in this text, simply called *D-latch*), the four others are summarily called *set-reset latches*, or *SR-latches*. The IEC-symbols for these latches are the left-most symbols in the Figs. 27–12 and 27–13. For an explanation of the inscribed letters and numbers (which comprise the so-called *dependency notation*) I refer you to the IEC standard.

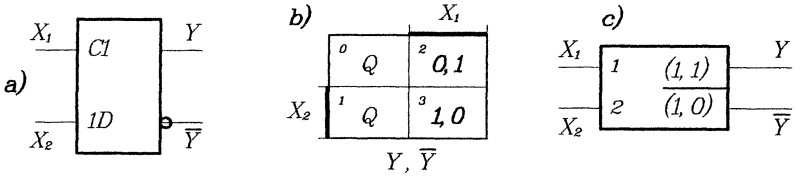


Fig. 27–12 The D-latch

The **D-latch**, defined in the K-map of Fig. 27–12b, was originally conceived to synchronise input Data with the Clock signal of a synchronous sequential circuit. The outputs of the latch are assumed to be complementary as is indicated by the negation symbol on the lower output of Fig. 27–12a.

Characteristic for the **SR-latches**, listed in Fig. 27–13, is that the input event $\overline{X}_1\overline{X}_2$ is *the* memorising input event. For convenience, the adjacent setting and resetting input events are taken to be identical for all four latches.

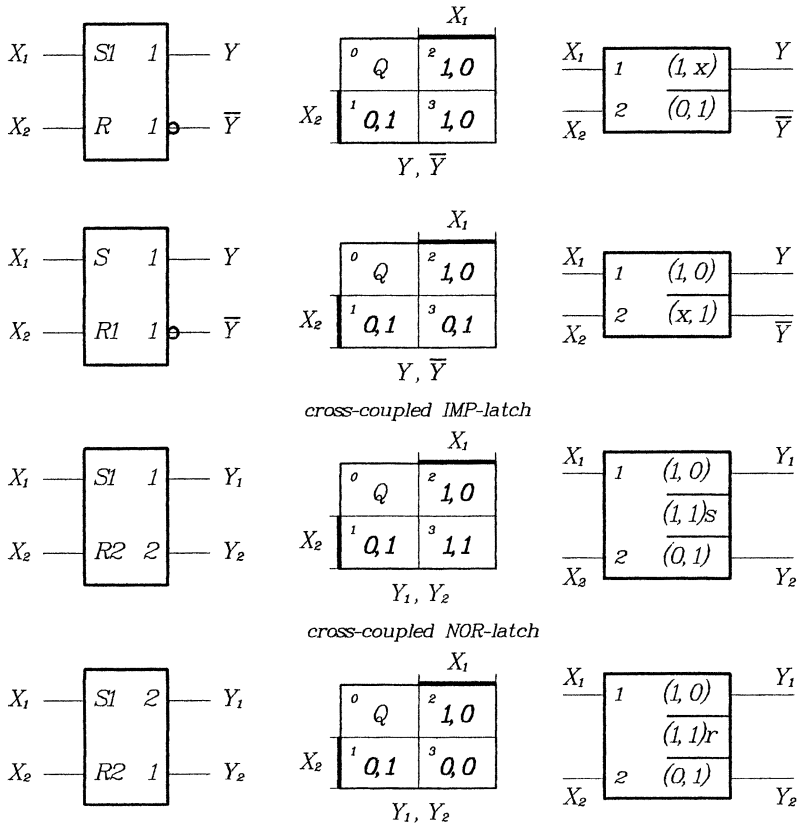


Fig. 27–13 The four set-reset latches

The four SR-latches differ only in the output values invoked by the input event X_1X_2 [1].

The IEC standard 117-15 states explicitly that all output lines of an element that is not subdivided (such as an IEC latch element) carry identical logic values except when otherwise indicated by associated *qualifying symbols or labels* inside the element. For instance, in the top two rows of Fig. 27–13 both output lines of a latch symbol are assigned the number 1 as a qualifying symbol indicating that the outputs are identical. To make the

[1] With the knowledge we have acquired about latches, I would be inclined to add a fifth latch to the list of SR-latches, i.e., the latch characterised by the symbol in the second row of Fig. 27–9 with the output variables $Y_{2,4}, Y_{4,2}$. For this latch, the input event X_1X_2 is a memorising input event. Latches with exactly this function are common as pneumatic spool valves with two outputs. But, if I am not mistaken, this latch cannot be represented in the symbolism of the IEC standard 117-15.

outputs complementary, an inverter symbol (a circle) is assigned to one of the outputs. On the other hand, the output values of the bottom two latch symbols (Fig. 27–13) are unequivocally specified by the associated qualifying symbols 1 and 2 — adding an inverter symbol to one of the outputs (as is done in IEC 117-15) poses an error.

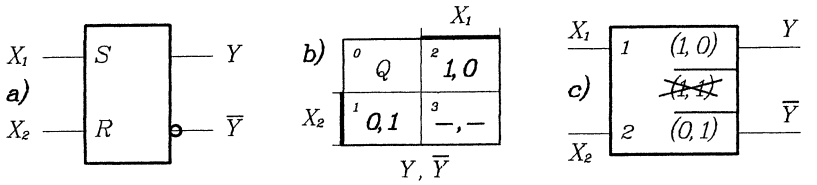


Fig. 27–14 The generic SR-latch

Frequently, one works with what I would call a *generic SR-latch*, symbolised in Fig. 27–14. Characteristic for the generic SR-latch is the assumption that the output values are *not defined* for the input event X_1X_2. When using this latch in designing a circuit, the desire is that the circuit should function properly for any of the actual SR-latches. To achieve this, you need to make sure that the generic latch is never supplied the input event $(\mathbf{1},\mathbf{1})$ crossed out in the symbolic latch of Fig. 27–14c. How to achieve requirements of this kind is one of the topics of the next chapter.

Chapter 28

Composition of Latches *

Methodically, the composition of latches, as presented here, is identical with the composition of combinational circuits as developed in Chapter 22. I therefore dispense with any repetition of basic principles, referring you for these, as for points of terminology, to Chapter 22.

When composing a given latch f it quickly becomes obvious that the hinge function h must be a latch, while the generic function g must be a (multiple-output) combinational circuit. In fact, any latch can be chosen as hinge function h — the composition procedure then enables us to development appropriate generic functions.

The composition of latches provides a second method for their design, the first one being the evaluation of a latch as a feedback circuit. Whether a latch that is designed by composition is hazard-prone or hazard-free is a question that, to my knowledge, cannot (yet?) be decided generally. But, we can apply our knowledge of hazards in latches (see Sections 26.3 and 26.4) to decide if a specific latch designed by composition is hazard-prone or not.

28.1 Catenation Diagram for Latches

The problem we want to look at is *'how can we design a specified latch f on the basis of a given latch — a* **hinge latch** *— h?'*. This problem is elaborated in Fig. 28–1 for a given example. Don't be deterred by the figure's seeming complexity, I shall explain it step by step.

First of all, consider the specified latch f represented by the arrows mapping the elements of $\mathcal{X} = \{(0,0),(0,1),(1,0),(1,1)\}$ to $\mathcal{Z} = \{0,1,q\}$. This mapping expresses exactly the same as the reduced K-map denoted by the function name f and drawn below the mapping of f. The reduced K-map allows us to recognise the specified latch f as the D-latch. An input to the D-latch is an input event $(x_1,x_2) \in \mathcal{X}$, its output is $z \in \mathcal{Z}$.

Any composite function hg, where h *follows* g, i.e., $hg : \mathcal{X} \overset{g}{\mapsto} \mathcal{Y} \overset{h}{\mapsto} \mathcal{Z}$, and that equals a specified function $f : \mathcal{X} \mapsto \mathcal{Z}$, i.e., $f = hg$, requires the *'following function'* h to have the same co-domain, $\mathcal{Z}$, as the specified function f. By this token, *if we want h to be a switching function it must be a latch,*

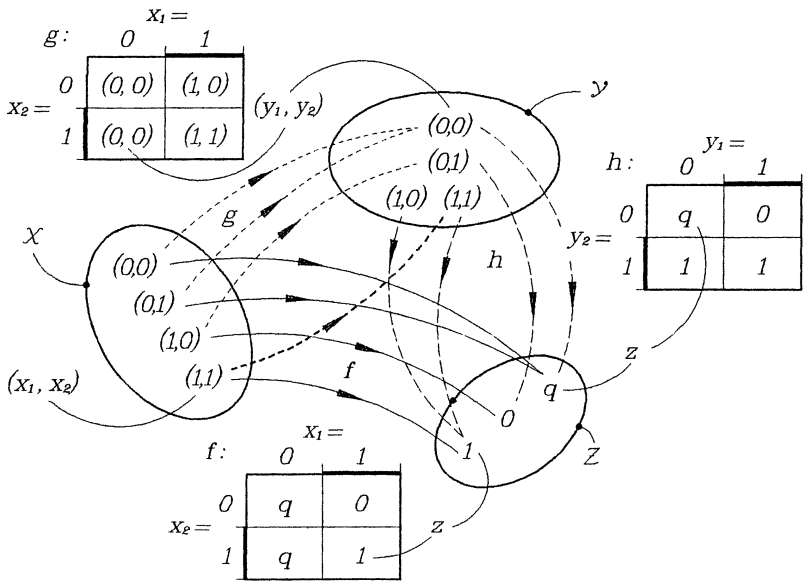


Fig. 28–1 Composing the D-latch f on the basis of a SR-latch as hinge latch h

while g must be a multiple-output combinational circuit. It is important to realise that h may in fact be *any* latch so that the composition procedure allows us to choose, for instance, an arbitrary *elementary latch* to be the basis on which to compose any specified latch f. Now let us look at the functions h and g in detail for our example of Fig. 28–1.

The **hinge latch** h we choose to use — in our example, the SR-latch specified in the K-map named h — is represented by the arrows mapping the elements of $\mathcal{Y} = \{(0,0),(0,1),(1,0),(1,1)\}$ to $\mathcal{Z} = \{0,1,q\}$. An input to the SR-latch is an input event $(y_1,y_2) \in \mathcal{Y}$, its output is $z \in \mathcal{Z}$.

The **prelogic**, as we call the multiple-output combinational logic g, is *any* function from $\mathcal{X}$ into $\mathcal{Y}$, for wich $\mathcal{X} \overset{f}{\mapsto} \mathcal{Z} = \mathcal{X} \overset{g}{\mapsto} \mathcal{Y} \overset{h}{\mapsto} \mathcal{Z}$ holds. In our example there are two such functions, one which maps $(1,1) \in \mathcal{X}$ to $(1,1) \in \mathcal{Y}$ (as pictured in Fig. 28–1), the other which would map $(1,1) \in \mathcal{X}$ to $(1,0) \in \mathcal{Y}$. I know of no criterion by which to choose between these two mappings. We have to rely on a later analysis to decide which was the better choice. The prelogic defined in the mapping from $\mathcal{X}$ into $\mathcal{Y}$ is pictured in the multiple-output K-map g.

Analogous to the catenation of combinational circuits, the present hinge function, the hinge latch $h : \mathcal{Y} \mapsto \mathcal{Z}$, with $\mathcal{Z} = \{0,1,q\}$, allows us to partition its domain $\mathcal{Y}$ into the equivalents, the so-called **hinge-sets** $\mathcal{Y}_z = \{\mathbf{y}|h(\mathbf{y}) = z\}$, with $z \in \mathcal{Z}$. The hinge-sets for the SR-hinge-latch in the example of

Fig. 28–1 are

$$\mathcal{Y}_0 = \{(0,1)\},$$
$$\mathcal{Y}_1 = \{(1,0),(1,1)\},$$
$$\mathcal{Y}_q = \{(0,0)\}.$$

The hinge-sets $\mathcal{Y}_z = \{\mathbf{y}|h(\mathbf{y}) = z\}$ of the hinge latch are used to develop the K-map of the prelogic g from the K-map of the specified latch f by **(a)** replacing each 0-output in the K-map of h by an element of $\mathcal{Y}_0$, **(b)** replacing each 1-output in the K-map of h by an element of $\mathcal{Y}_1$, and **(c)** replacing each symbol q in the K-map of h by an element of $\mathcal{Y}_q$. The two solutions that can be obtained for the example of Fig. 28–1 are developed in the next section.

28.2 Composing the D-Latch using a SR-Latch

It is long and well established knowledge how to compose a D-latch on the basis of a SR-latch, see Fig. 28–2. I only use this circuit as a running example for the rest of this chapter because it is so well understood allowing you more easily to check and compare the results.

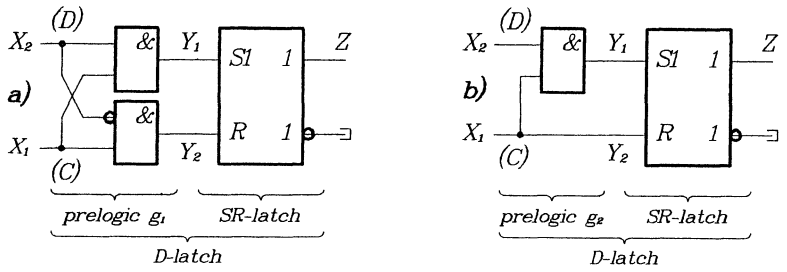


Fig. 28–2 The two possible generic or prelogic functions g_1 and g_2 that let a SR-latch work as a D-latch

To compose a latch by catenation as explained in the previous section, you must specify the K-map of the latch to be composed — in our case, the D-latch — and the K-map of the hinge-latch — in our case, the set-dominant SR-latch — see Fig. 28–3. The K-map of the hinge-latch is only used to be able to specify the hinge-sets $\mathcal{Y}_0$, $\mathcal{Y}_1$, and $\mathcal{Y}_q$. The K-map of the latch to be composed, the D-latch f, is transformed into the K-map of a multiple-output prelogic, say g_1 by **(a)** substituting the 0-output-value of f by the element $(\mathbf{0},\mathbf{1})$ of $\mathcal{Y}_0$, **(b)** substituting both symbols for the internal state Q of f by the single element $(\mathbf{0},\mathbf{0})$ of $\mathcal{Y}_q$, and finally by **(c)** substituting the 1-output-value of f by one of the elements of $\mathcal{Y}_1$, arbitrarily choosing $(\mathbf{1},\mathbf{0})$ in the case of g_1. To obtain g_2, we alternatively substitute the 1-output-value of f by $(\mathbf{1},\mathbf{1})$ of $\mathcal{Y}_1$ — the other element of $\mathcal{Y}_1$. You can read the logic formulas $Y_1 \Leftrightarrow X_1X_2$,

$Y_2 \Leftrightarrow X_1\overline{X}_2$ for the prelogic g_1, and $Y_1 \Leftrightarrow X_1X_2$, $Y_2 \Leftrightarrow X_1$ for the prelogic g_2, from the K-maps for g_1 and g_2 of Fig. 28–3.

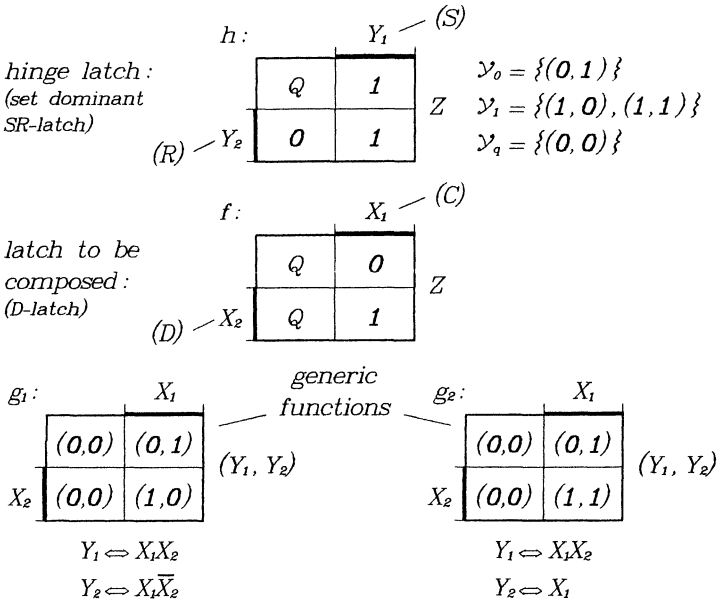


Fig. 28–3 Developing the two prelogic functions g_1 and g_2 used in Fig. 28–2 to compose the D-latch on the basis of the set-dominant SR-hinge-latch

The circuits calculated in Fig. 28–3 are shown in Fig. 28–2. Viewing them, it seems quite valid to ask the following questions. How can we prove or verify that the composed circuits have the input-output behaviour of the D-latch? Are the circuits hazard-free? And finally: Can the set-dominant SR-latch be substituted by any SR-latch of Fig. 27–13? In the next section, we consider the first two questions together.

28.3 Checking for a Hazard-Free Solution

Checking the input-output behaviour of a latch obtained by composition goes hand in hand with the assessment of whether the obtained latch is hazard-free or hazard-prone. Consider, for instance, our running example of a D-latch as obtained in Fig. 28–3 and shown in Fig.28–2a,b. The D-latch is specified by either K-map of Fig. 28–4, these showing that there are two possible evaluation solutions — one hazard-free, the other hazard-prone.

$$Z \Leftrightarrow X_1X_2 \vee (\overline{X}_1 \vee X_2)Z', \qquad \ldots \text{hazard-free D-latch}$$
$$Z \Leftrightarrow X_1X_2 \vee \overline{X}_1Z'. \qquad \ldots \text{hazard-prone D-latch}$$

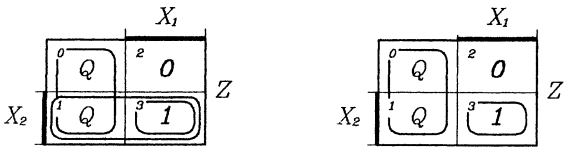


Fig. 28–4 Two possible solutions for the D-latch

To obtain cohesive formulas for the D-latch from the circuits of Fig. 28–2ab requires us to express the SR-latches of those circuits by formulas. Let us first assume the SR-hinge-latch to be hazard-free — see Fig. 28–5:

$$Z \Leftrightarrow Y_1 \vee \overline{Y}_2 Z'. \qquad \ldots \text{hazard-free SR-latch}$$

Now, to verify the circuit of Fig. 28–2a, we substitute the input variables Y_1 and Y_2 of the hazard-free SR-latch by $Y_1 \Leftrightarrow X_1X_2$, and $Y_2 \Leftrightarrow X_1\overline{X}_2$ of the prelogic g_1,

$$Z \Leftrightarrow Y_1 \vee \overline{Y}_2 Z' \Leftrightarrow X_1X_2 \vee \overline{X_1\overline{X}_2}Z' \Leftrightarrow X_1X_2 \vee (\overline{X}_1 \vee X_2)Z',$$

verifying that the obtained solution is not only a D-latch, but also that it is a hazard-free D-latch. If we use the same hazard-free SR-latch together with the prelogic g_2 of Fig. 28–2,

$$Z \Leftrightarrow Y_1 \vee \overline{Y}_2 Z' \Leftrightarrow X_1X_2 \vee \overline{X}_1 Z' \qquad \ldots \text{hazard-prone D-latch}$$

we again get a D-latch, but it is *not* hazard-free.

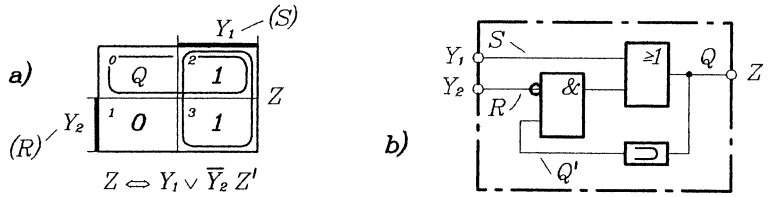


Fig. 28–5 The hazard-free set-dominant SR-latch

To complete the analysis, we shall want to see how the two prelogic functions g_1 and g_2 work together with the *hazard-prone* version of the set-dominant SR-hinge-latch calculated and shown [1] in Fig. 28–6.

[1] Of course, nobody would dream of designing, buying, or using such a latch — for whatever purpose. Except, as here, to point out that hinge latches can by mishap be hazard-prone, especially, and more generally, when they are neither D- nor SR-latches nor even elementary.

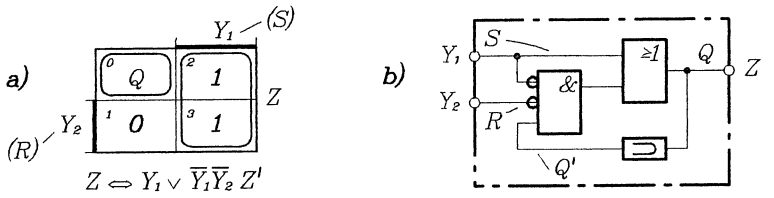


Fig. 28–6 The hazard-prone set-dominant SR-latch

We get the two following results for g_1 and g_2, respectively:

$$
\begin{aligned}
Z &\Leftrightarrow Y_1 \vee \overline{Y}_1\overline{Y}_2 Z' && \text{... hazard-prone SR-latch}\\
&\Leftrightarrow X_1X_2 \vee (\overline{X_1X_2})(\overline{X_1\overline{X}_2})Z' && \text{... with prelogic } g_1\\
&\Leftrightarrow X_1X_2 \vee (\overline{X}_1 \vee \overline{X}_2)(\overline{X}_1X_2)Z'\\
&\Leftrightarrow X_1X_2 \vee (\overline{X}_1 \vee \underbrace{\overline{X}_1\overline{X}_2 \vee \overline{X}_1X_2}_{\overline{X}_1} \vee \underbrace{\overline{X}_2X_2}_{0})Z'\\
&\Leftrightarrow X_1X_2 \vee \overline{X}_1Z'. && \text{... hazard-prone D-latch}
\end{aligned}
$$

$$
\begin{aligned}
Z &\Leftrightarrow Y_1 \vee \overline{Y}_1\overline{Y}_2 Z' && \text{... hazard-prone SR-latch}\\
&\Leftrightarrow X_1X_2 \vee (\overline{X_1X_2})\overline{X}_1Z' && \text{... with prelogic } g_2\\
&\Leftrightarrow X_1X_2 \vee (\overline{X}_1 \vee \overline{X}_2)\overline{X}_1Z'\\
&\Leftrightarrow X_1X_2 \vee \underbrace{(\overline{X}_1 \vee \overline{X}_1\overline{X}_2)}_{\overline{X}_1} Z'\\
&\Leftrightarrow X_1X_2 \vee \overline{X}_1Z'. && \text{... hazard-prone D-latch}
\end{aligned}
$$

Both above results show that the two circuits of Fig. 28–2 are hazard-prone if the hinge latch is hazard-prone. On the basis of the above verifications allow me to **conjecture** that *'for a latch f, obtained by composition, to be hazard-free, its hinge latch h must be hazard-free'.*

For all practical purposes, the above conjecture only states that a hazard-free hinge latch h is a necessity in obtaining a hazard-free composed latch f, but is not in itself sufficient.

28.4 Avoiding Undesirable Input-Events

Take a few minutes to ponder the circuit design of the set-dominant SR-latch f pictured in Fig. 28–7. This latch, designed by composition, is based on a hinge latch h which is itself a set-dominant SR-latch. Why this redundancy?

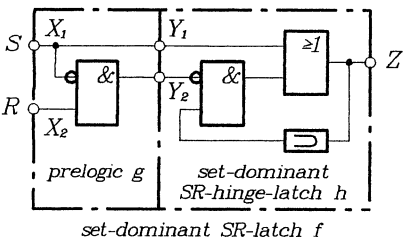


Fig. 28–7 A SR-latch on the basis of a SR-latch

The idea behind a design as the one above is *to avoid presenting a latch with input events that might lead to a malfunction of the latch.* To illustrate, consider the situation pictured in the K-map h of Fig. 28–8. If you want to memorise the output value **1** invoked by the input event Y_1Y_2, both input values must switch simultaneously from **1** to **0**. If Y_1 were to change its value before Y_2 does (illustrated by the bottom arrow), you would have a good chance of seeing the output turn incorrectly and permanently to **0**. Now, referring to Fig. 28–1, you avoid the problem mentioned in the previous sentence if every $\mathbf{x} \in \mathcal{E}_1$ is mapped by g to the input event $Y_1\overline{Y}_2$ adjacent to the memorising input event $\overline{Y}_1\overline{Y}_2$. You can achieve this by retaining the setting input event $(\mathbf{1},\mathbf{0})$ in $\mathcal{Y}_1$, and dropping $(\mathbf{1},\mathbf{1})$ from $\mathcal{Y}_1$.

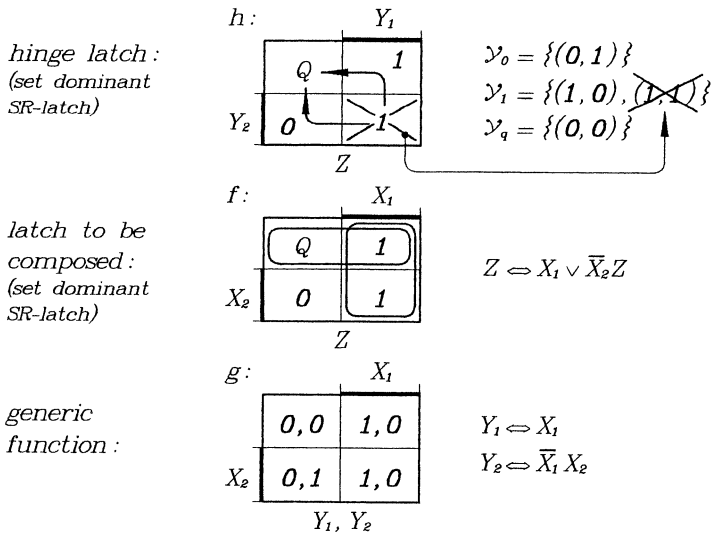


Fig. 28–8 Composing a SR-latch on the basis of a SR-latch

As each of the equivalents $\mathcal{Y}_0$, $\mathcal{Y}_1$, $\mathcal{Y}_q$ contain only a single input event (after $(\mathbf{1},\mathbf{1})$ has been dropped from $\mathcal{Y}_1$), the K-map f of the latch to be composed — itself a set-dominant SR-latch — is transformed uniquely into

a single K-map g for the generic function. Evaluating this K-map, lets us write $Y_1 \Leftrightarrow X_1$, $Y_2 \Leftrightarrow \overline{X}_1 X_2$.

If you now take another look at the SR-latch of Fig. 28–7, you will notice that both input events $(\mathbf{1},\mathbf{1})$ and $(\mathbf{1},\mathbf{0})$ applied to f are passed on as $(\mathbf{1},\mathbf{0})$ to h and that, for this input event, the feedback loop (of h) is pre-established so that h is hazard-free. Although we cannot prevent the undesirable input event $(\mathbf{1},\mathbf{1})$ being presented to f, $(\mathbf{1},\mathbf{1})$ is never presented to h so that for this latch the potentially dangerous transition from $(\mathbf{1},\mathbf{1})$ to $(\mathbf{0},\mathbf{0})$ can never occur. The set-dominant SR-latch of Fig. 28–7 is thus the safest design that theory implies.

Division Five

Sequential Circuits
with
Continuously Read Inputs

29 Automata and Programs * 311 (14 pages)
30 Word-Recognition Tables * 325 (10 pages)
31 Catenation Model * 335 (14 pages)
32 Toggle Circuits 349 (11 pages)
33 Triggering and Synchronising 360 (10 pages)
34 Verifying a Logic Design * 370 (15 pages)
35 Discussing Huffman's Theory 385 (13 pages)
36 State Encoding Techniques 398 (13 pages)

In 1938 **Shannon** initiated the use of propositional logic to calculate combinational switching circuits, creating quite a flurry. Despite this breakthrough in the theory of memoryless circuits, no headway was made in designing circuits with memorising abilities, the sequential circuits. It took 16 years for the next quantum jump to occur: In 1954 **Huffman** presented his ideas on *'The Synthesis of Sequential Switching Circuits'*. His synthesis procedure was exclusively concerned with so-called *asynchronous circuits*, but has put its stamp on all subsequent work in the field. As with combinational circuits, an **asynchronous (sequential) circuit** is one *whose input signals are read continuously*, and whose output can only change when an input event changes. With the advent of the integrated circuit in 1961 it became feasible to operate sequential circuits in a *synchronous* mode in which all parts of a circuit switch synchronously with an external clock pulse. More generally, a **synchronous (sequential) circuit** is one *whose input signals are read intermittently*, and whose output can only change when an input event is read.

The reason for introducing the synchronous mode of operation was the belief that to organise large circuits, as were needed in computers, you had to make all their gates and latches switch simultaneously — a conception that prevails to this day as our giga-hertz computer chips show. But obviously, no synchronous circuit can work faster than the clock frequency allows which leads to ever higher frequencies. This has its drawbacks: Clock pulses tend to degenerate to runt pulses; relatively small differences in the distance between communicating sub-circuits can disrupt synchronisation; any sub-circuit operates every time a pulse comes even if its other inputs remain unchanged — this is a source of unnecessary energy consumption; high energy consumption in ever smaller chips causes problems of overheating.

Asynchronous circuits (the topic of this division) pose none of the above mentioned problems — they have their own. But, nevertheless, they are potentially the fastest sequential circuits, and those with the smallest energy consumption. Especially the latter property is an asset when developing circuits for battery driven devices. Huffman's synthesis procedure for asynchronous circuits is plagued by one major problem — how to design the circuit's memorising ability. This is usually referred to by its technical term of *'encoding the internal states of the circuit'*. To this day there is no algorithmic solution to this problem making Huffman's asynchronous design cumbersome. The synthesis procedure put forth in the first three Chapters (29 to 31) breaks with tradition to circumvent (but not solve) the encoding problem. The examples in Chapters 32 and 33 might be of general interest. Chapter 34, which introduces verification, will hopefully prove to be of principle importance. For comparison, Huffman's procedure is discussed in Chapter 35 while Chapter 36 should give you an idea of the encoding problem. Chapters 29, 30, 31, 33,and 34 are based on **Vingron** [1983].

Chapter 29

Automata and Programs *

As we already know, the prime characteristic of a sequential circuit is that at least one input event sometimes causes the output to be 0 and sometimes to be 1. Nevertheless there is a general principle that states what output value to expect whenever a new input event is entered. This principle is best explained by the behaviour of a simple tape-reading device (an **'automaton'**) that emits an output value (0 or 1) contained in a program line written on tape and being read, while the line being read is determined by an input event to the automaton, *and* by the program itself. **Huffman** [1954], who invented these programs, called them **flow tables**. In principle, each flow table defines a different sequential circuit whereas the automaton always remains the same.

The rules for writing a flow table and the basic operation of the automaton are discussed in Section 1. In Section 2 we consider how to transpose the verbal specification of a sequential circuit into a flow table. The flow table is usually *'encoded'* more or less arbitrarily to obtain the logic formulas for the sequential circuit. In contrast, we shall use the flow table as a starting point from which to develop a *tree representation* of the circuit. This so-called **word-recognition tree** not only depicts a possible structure of the circuit, but also enables an *'encoding-free'* synthesis. The word-recognition tree is discussed in Section 3, a synthesis example on its basis is presented in Section 4.

29.1 Continuously reading Automata

A **flow table** consists of a finite number s of code lines written one below the other. Each of these code lines is referred to as an **(internal) state** of the automaton to be described. The symbols used to label the code lines or *states* (as we shall more often call them) are chosen arbitrarily, but we usually use positive integers $\{1, 2, \ldots, s\}$, or letters of the alphabet $\{a, b, c, \ldots\}$. A calligraphic $\mathcal{Q}$ refers to the **set of states**, e.g., $\mathcal{Q} = \{1, 2, \ldots, s\}$ or $\mathcal{Q} = \{a, b, c, \ldots\}$. A variable $q \in \mathcal{Q}$ that stands for any element of $\mathcal{Q}$ is referred to as a **state variable**.

Each line of code consists of $2^n + 2$ cells. The first cell contains the **line number** or state label q, the last cell an **output value** $y(q) \in \{0, 1\}$. The

2^n cells between the first and the last cell of a code line are each associated with one of the input events $x \in \mathcal{E}$, with $\mathcal{E} = \{0, 1, \ldots, 2^n - 1\}$[1]. These cells comprise the **body of a code line**, and each such cell contains a *'GOTO line-number'* command — and nothing else. That is, in each such cell you will find a state label for mnemonic reasons designated as $q^+ \in \mathcal{Q}$, and usually referred to as the **next state** — this name only being understandable in conjunction with the usage of a flow table as a program governing the behaviour of an *automaton*.

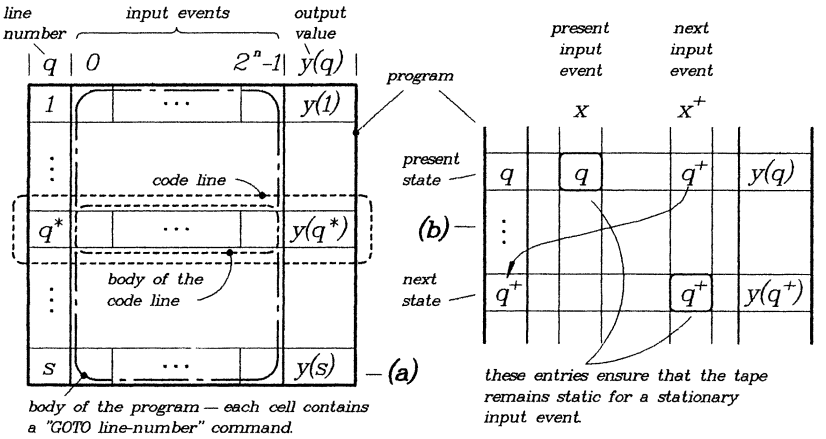


Fig. 29–1 Organisation of a **flow table**

Our **automaton** is an abstract machine that accepts an input event $x \in \mathcal{E}$ as input signal, and emits the binary output signal $y \in \{0, 1\}$. The dependence of the output y on an input event x is governed by a program, the flow table, which the automaton reads and acts upon. Think of the flow table as having been written to a tape which the automaton can move back and forth under two reading heads aligned so as to read a single line of code (or row of the flow table). If the line presently being read is q, then the so-called **output head** reads and emits the output value $y(q)$. The other reading head, the **next-state head** can, in principle, be moved to and fro across the body of the code line q being read. The next-state head is always positioned over the cell (q, x), whose elements q and x are called the **present state** and **present input**, respectively, and whose content is the **next state** $q^+ \in \mathcal{Q}$. When the next-state head reads the next state q^+, the automaton moves the tape so that the code line labelled q^+ comes to rest under the reading heads of our automaton.

To ensure that the automaton does not move the tape as long as the present input event remains stationary, **the flow table must abide by the**

[1] Any $x \in \mathcal{E}$ is of course the decimal equivalent of the n-tuple $(x_1, \ldots, x_n)$ of binary inputs $x_i \in \{0, 1\}$, i.e., $x = \sum_1^n x_i . 2^{n-i}$.

following principle (see Fig. 29–1b): If we are sent to a cell (q^+, x^+) due to a GOTO-q^+ command in a cell (q, x^+) of a present state q, with $q \neq q^+$, then the cell (q^+, x^+) must have q^+ inscribed so that the new code line q^+ remains static as long as the input event x^+ remains stationary. We now summarise the above concepts in the following definition.

> **(29.1)** A **flow table** is a program — with a finite number, s, of program lines — that governs the way the input events $x \in \mathcal{E}$ effect the binary output variable $y \in \{0,1\}$. The program abides by the following rules:
>
> **(a)** An *unique* output value $y(q)$, either a 0 or a 1, is assigned to each program line q.
>
> **(b)** The program remains at any line q it may be in (referred to as the *present line*) as long as the present input event $x \in \mathcal{E}$ remains stationary.
>
> **(c)** Whenever the present input event $x \in \mathcal{E}$ changes to a next and different input event $x^+ \in \mathcal{E}$, the content q^+ of the new cell (q, x^+) of the present program line q is read, the program going to the program line labelled q^+. If $q^+ = q$, then *'going to the program line* q^+*'* means staying in program line q.
>
> **(d)** It is always possible to reach any line of the program in a finite number of input-event changes.

The first problem we now face is to show how a verbal specification of the behaviour of a sequential circuit can be expressed by a flow table.

29.2 Developing an Example Flow Table

Developing a flow table, i.e., a program for our automaton, is an art, and not a science. The only guiding principle I can offer is to express any problem by a number of successive input events where each such sequence of input events uniquely defines an output value. If you are new to the field, please take your time in working through the example discussed in this section — it should help clarify how to employ the above principle. But, only after you have worked through a number of problems yourself will you start developing your own individual approach (or approaches) to the process.

To facilitate explanations in connection with flow tables we use the following terminology. While the rows of a flow table are generally called the **(internal) states** q, and the columns x are the **input events**, this text refers to the cells (q, x) of the body of the flow table as the **conditions** of the flow table. A **stable condition** is a cell (q, x) that has a 'GOTO-q' command inscribed, thus causing the tape to remain *static* under the reading heads of the automaton. A GOTO-q command in the cell or stable condition (q, x) is

often referred to as a **stable state**. It is common to emphasize the stable states by encircling or underlining them. You will see that this convention will help you navigate the flow table.

The development of a flow table is often made easier if we develop it as a **primitive flow table**, i.e., a flow table that has only a single stable condition per row. A **non-primitive flow table** can have a number of stable conditions per row. To see how to develop a (primitive) flow table let us proceed with a verbal specification of an example of a sequential circuit:

> **Example** The single output, y, of a sequential circuit with two inputs, x_1 and x_2, is 0 when x_1 is 0, and when x_1 is 1 takes on the value x_2 had the last time x_1 was 1.

While the above example fulfils no technical or practical purpose, it has the advantage that formulating its flow table is a non-trivial process.

When you start drawing a flow table for a problem with a stated number of inputs, you know how many columns the flow table must have, but generally you cannot say how many lines or rows you will need. Our example has two inputs, so the flow table must have $2^n + 2 = [n \hookleftarrow 2] = 2^2 + 2 = 6$ columns. As shown in the figures of this section, we shall always make a point of it to organise the input events according to a Gray code for later use with K-maps.

(a)

	(x_1, x_2)				
	0	1	3	2	
q	(0,0)	(0,1)	(1,1)	(1,0)	$y(q)$
a			(a)		0
b			(b)		1
c				(c)	0
d				(d)	1
e		(e)			0
f	(f)				0

(b)

	(x_1, x_2)				
	0	1	3	2	
q	(0,0)	(0,1)	(1,1)	(1,0)	$y(q)$
a			(a)		0
b			(b)		1
c				(c)	0
d				(d)	1
e		(e)	a	c	0
f	(f)		a	c	0
g		(g)	b	d	0
h	(h)		b	d	0

Fig. 29–2 A first attempt at finding the number of rows needed

Fig. 29–2a: The example states that *whenever* $x_1 = 0$ *the output* $y = 0$. The two input events (x_1, x_2) for which $x_1 = 0$ are $(0, 0)$ and $(0, 1)$, see columns 0 and 1. In these columns we arbitrarily choose rows, e and f, to contain a stable state to which we assign the output value 0. On the other hand the example specifies that *when* $x_1 = 1$ *the output* y *takes on the value* x_2 *had the last time* x_1 *was 1.* For the time being the only thing we want to deduce from this formulation is that whenever $x_1 = 1$, i.e., in either of the columns 2 and 3, the output y will sometimes be 0, and sometimes

be 1, necessitating each column to have (at least) two stable states — one to output 0, the other to output 1. The number of stable states so far specified is not necessarily enough to express the above example in a flow table as we discuss next.

Fig. 29–2b: Still considering Fig. 29–2a, for the moment, let us ask which of the GOTO commands, GOTO-a or GOTO-b, should we enter into cell $(e, 3)$? Whichever we enter, we exclude the possibility of going to the alternative stable condition in column 3. As shown in Fig. 29–2b, if we choose to enter GOTO-a into cell $(e, 3)$ we need to introduce a second stable condition in column 1, in our example the stable g, from which we can GOTO-b when switching to column 3 — see cell $(g, 3)$. These considerations necessitate the introduction of the states or rows g and h together with the GOTO commands a to d in the bottom right quadrant of the flow table of Fig. 29–2b.

(a)

	(x_1, x_2)				
	0	1	3	2	
q	(0,0)	(0,1)	(1,1)	(1,0)	$y(q)$
a			(a)	(3 2) d	0
b			(b)	(3 2) d	1
c			(2 3) a	(c)	0
d			(2 3) a	(d)	1
e		(e)	a	c	0
f	(f)		a	c	0
g		(g)	b	d	0
h	(h)		b	d	0

(b)

	(x_1, x_2)				
	0	1	3	2	
q	(0,0)	(0,1)	(1,1)	(1,0)	$y(q)$
a	([3 0] 2) ([3 0] 3) h	([3 1] 2) ([3 1] 3) g	(a)	d	0
b	([3 0] 2) ([3 0] 3) h	([3 1] 2) ([3 1] 3) g	(b)	d	1
c	([2 0] 2) ([2 0] 3) f	([2 1] 2) ([2 1] 3) e	a	(c)	0
d	([2 0] 2) ([2 0] 3) f	([2 1] 2) ([2 1] 3) e	a	(d)	1
e		(e)	a	c	0
f	(f)		a	c	0
g		(g)	b	d	0
h	(h)		b	d	0

Fig. 29–3 GOTO commands defined by direct and double transitions

Fig. 29–3a: In this figure we consider the GOTO commands for the top right quadrant of the flow table where $x_1 = 1$. For input transitions within this quadrant let us reformulate the original specification *'when $x_1 = 1$ the output y takes on the value x_2 had the last time x_1 was 1'* to be read as *'when x_2 changes its value from 0 to 1 (x_1 remaining 1), the new output value must be 0, this being the previous value of x_2'*, and as *'when x_2 changes*

its value from 1 to 0 (with x_1 remaining 1), the new output value must be 1, this being the previous value of x_2'. Thus when switching from column 2 to column 3 — written as the sequence $(2,3)$ of input events — we must GOTO-a, while when switching from column 3 to column 2 — written as $(3,2)$ — we must GOTO-d.

Fig. 29–3b: Let us now concentrate on the GOTO commands to be entered into the top left quadrant of the flow table. When considering only the direct transition from column 2 to column 0 it is not possible to decide whether to enter a GOTO-f or a GOTO-h command in the cells $(c,0)$ or $(d,0)$ because both stable states, f and h, correctly let the output become 0. But, considering the *double* transitions from column 2 to column 0 and back from column 0 to column 2 (or on to column 3), which we write as $(2,0,2)$ and $(2,0,3)$, we see clearly that GOTO-f (as contrasted with GOTO-h) must be entered into both cells $(c,0)$ and $(d,0)$ to let the output become 0 when switching from column 0 to columns 2 or 3. The above considerations are documented by entering the double transition $(2,0,2)$ into the cells $(c,0)$ and $(d,0)$ as $([2,0],2)$. The brackets, $[\ldots]$, mark the direct transition being considered within the multiple transition. The subsequent direct transition in our example would be denoted as $(2,[0,2])$, and would be entered into cell $(f,2)$. But, transitions (whether direct or multiple transitions) are only entered into cells *not* containing a GOTO command, and only when the transition is used to motivate or explain a GOTO entry. Allow me to ask you to ponder the GOTO commands for the remaining cells of the top left quadrant with the help of the double transitions written into the cells.

Fig. 29–4a: The GOTO commands to be entered onto the bottom left quadrant cannot be defined by considering direct or even double transitions. For instance, if you try to specify the GOTO command for cell $(e,0)$ you *must* start in column 2 to be able to GOTO-e — i.e., the stable e in cell $(e,1)$ — from where you can switch to cell $(e,0)$. But whether we must enter a GOTO-f or a GOTO-h command into cell $(e,0)$ cannot yet be decided. Considering as third transition — after $(2,1)$ and $(1,0)$ — the transition $(0,2)$, we know from the verbal specification of our example that the output y must be 0, this being *'the value x_2 had the last time x_1 was 1'*. This third transition requires us to switch from column 0 to column 2. If we do so from the stable h, the GOTO-d in cell $(h,2)$ sends us to row d where the output y is 1, and 0 as required. On the other hand, switching from the stable f of column 0 to column 2 sends us to row c where the output y is 0, as desired. We must therefore enter GOTO-f into cell $(e,0)$.

As cell $(e,0)$ defines the direct transition from column 1 to column 0, the triple transition $(2,1,0,2)$ when entered into cell $(e,0)$ is denoted as $(2,[1,0],2)$. The transitions $([2,1],0,2)$ would be entered in the cells $(c,1)$ and $(d,1)$, while $(2,1,[0,2])$ would be entered in cell $(f,2)$. Again, allow me to ask you to take the time to ponder the remaining entries in the bottom left quadrant.

(a)

q	0 (0,0)	1 (0,1)	3 (1,1)	2 (1,0)	y(q)
a	h	g	(a)	d	0
b	h	g	(b)	d	1
c	f	e	a	(c)	0
d	f	e	a	(d)	1
e	(2 [1 0] 2) (2 [1 0] 3) f	(e)	a	c	0
f	(f)	(2 [0 1] 2) (2 [0 1] 3) e	a	c	0
g	(3 [1 0] 2) (3 [1 0] 3) h	(g)	b	d	0
h	(h)	(3 [0 1] 2) (3 [0 1] 3) g	b	d	0

(b)

q	0 (0,0)	1 (0,1)	3 (1,1)	2 (1,0)	y(q)
a	h	g	(a)	d	0
b	h	g	(b)	d	1
c	f	e	a	(c)	0
d	f	e	a	(d)	1
e	f	(e)	a	c	0
f	(f)	e	a	c	0
g	h	(g)	b	d	0
h	(h)	g	b	d	0

Fig. 29–4 GOTO commands defined by triple transitions

Fig. 29–4b: In the final flow table we usually enter only the GOTO commands, omitting all transitions that led to the GOTO entries.

A remark is in place on the specification of the behaviour of a sequential circuit when two (or more) input variables change their values simultaneously. When two or more input variables change their values simultaneously we speak of **non-adjacent successive input events**, i.e., successive input events[1] whose hamming distance[2] is 2 or greater ($h_d \geq 2$). It is usually argued that it is hard to impossible to guarantee that two (let alone more than two) input variables change their values simultaneously when desired, so that specifying a circuit behaviour for such cases is folly. On the other hand, I choose to argue that in those cases when two or more input variables change their values simultaneously — however small the likelihood for this may be — it may well be desirable or even necessary to specify the output value you want invoked. If this is not done, the process of *merging* — discussed in Chapter 31 — may impose an unforeseen or even undesired circuit behaviour.

---

[1] On successive input events see Section 24.1 and definition (24.1).

[2] Hamming distance was introduced in the last paragraph of Chapter 3. But also see definition (19.3) on the adjacency vector.

29.3 The Word-Recognition Tree

The primitive flow table completed in Fig. 29–4b is the specification *par excellence* of the example expressed only verbally in the previous section. The question now is: How can we employ the primitive flow table to develop a sequential circuit? **Huffman**'s [1954] solution to this problem (consisting in representing the internal states by binary state-variables which could then be encoded more or less arbitrarily) was so brilliantly simple and convincing that it impeded alternative developments (which was not his fault) but has left us with formidable encoding problems. To avoid them, allow me to break with tradition in presenting a tree representation of a sequential circuit: the **word-recognition tree** as I choose to call it. This tree is obtained by a simple and formal analysis of a flow table. But, as there are a number of different aspects to this analysis, we shall leave its presentation for the next chapter. In this chapter, let us assume we have done the necessary analysis for our example, obtaining from the primitive flow table of Fig. 29–4b the word-recognition tree of Fig. 29–5.

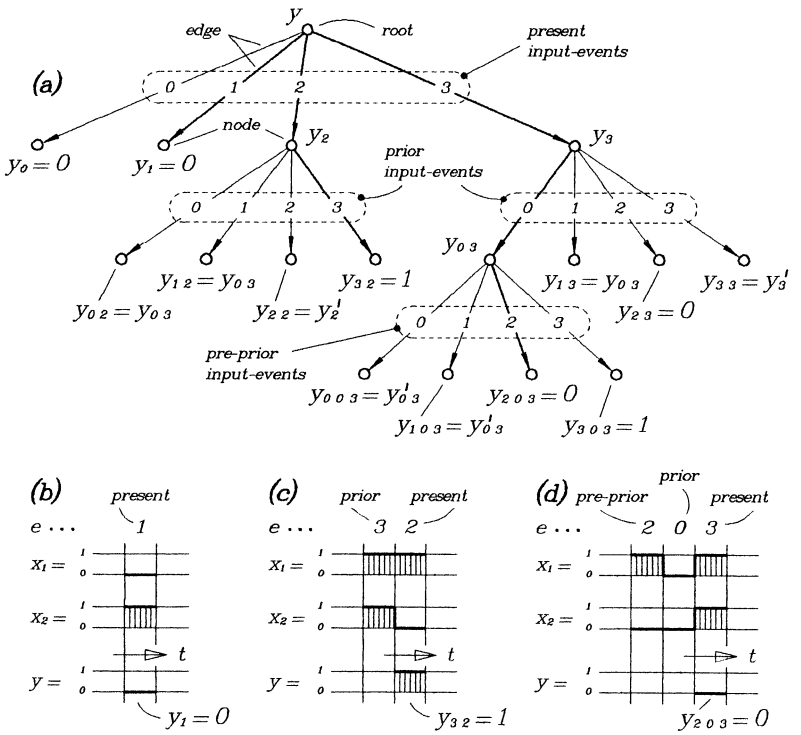


Fig. 29–5 The word-recognition tree for the flow table of Fig. 29–4b

A tree consists of **nodes** (the circles), and **(directed) edges** (the arrows). One distinguishes between an edge that leads *to* a node, a so-called **input edge**, and one that leads *away from* a node, an **output edge**. Each node has only *one* input edge, with the exception of a special node called the **root** which has *no* input edge. Some nodes, referred to as **end nodes**, or **leaves**, have *no output edges*. Those nodes that *do have* output edges are referred to as **internal nodes**. The root is an internal node. In actual usage the arrow heads are often not drawn, leaving us with simple lines instead of arrows.

The *edges* of a **word-recognition tree** are associated with input events $x \in \{0, 1, \ldots, 2^n - 1\}$. The number of output edges leading away from an internal node is 2^n, one for each input event. A succession of connected edges from the root to a leaf represents a succession of input events from present to past. Every *node* is specified by an unique **input word** $w(\ell)$, i.e., the ℓ-tuple of input events whose edges lead from that node to the root. We look at the concept of a word more closely.

As in previous cases, we use x (without a subscript) as an abbreviation for the input event $(x_1, \ldots, x_n)$ in which each $x_i \in \{0, 1\}$ is an input variable. To distinguish between input events, we use *superscribed* indices (mostly integers $1, 2, \ldots$) writing them in parentheses to avoid interpreting them as exponents:

$$x^{(j)} \stackrel{df}{=} (x_1^{(j)}, x_2^{(j)}, \ldots, x_n^{(j)}), \qquad j \in \{1, 2, \ldots\} \qquad x_i^{(j)} \in \{0, 1\}. \tag{29.3}$$

An **input word** $w(\ell)$ is a concept used to characterise a succession of input events,

$$w(\ell) \stackrel{df}{=} (x^{(1)}, x^{(2)}, \ldots, x^{(\ell)}). \tag{29.4}$$

The **length** ℓ of an input word specifies the number of its input events. Any two neighbouring input events, $x^{(j)}$ and $x^{(j+1)}$, are successive, in the sense of (24.1), the left one, $x^{(j)}$, always *preceding* the right one, $x^{(j+1)}$. The rightmost input event is always the present one. For future use let us agree on referring to an arbitrary but fixed input event $x^{(j)}$ of a given input word $(x^{(1)}, \ldots, x^{(j)}, \ldots, x^{(\ell)})$ as a **reference input-word**, the ones successively to its left as the **prior**, **pre-prior**, **pre-pre-prior** input events, etc., and the ones successively to its right as the **next**, **post-next**, **post-post-next** input events, etc. An input word of length 1 is identical with its (single) input event, $w(1) = (x) \stackrel{df}{=} x$. For the input word $w(0)$ of zero length, the **empty word**, we define concatenation to be $w(\ell), w(0) \stackrel{df}{=} w(\ell)$, and the output $y_{w(0)} \stackrel{df}{=} y$.

Each node of the word-recognition tree is assigned a **filtered output variable** $y_{w(\ell)} \in \{0, 1\}$, the word-index $w(\ell)$ acting as the filter. These filtered variables are interpreted as follows:

y is the output *per se*, i.e., is the non-filtered output you can observe at any instant and *irrespective of the present input event* x.

y_x is the output y invoked by *any* occurrence of the input event x.

$y_{x^{(1)},x^{(2)}}$ is the output y caused by only those occurrences of $x^{(2)}$ that are preceded by $x^{(1)}$.

$y_{x^{(1)},x^{(2)},x^{(3)}}$ is the output y caused by those occurrences of $x^{(3)}$ that are preceded by those occurrences of $x^{(2)}$ that are themselves preceded by an occurrence of $x^{(1)}$. Or, if you prefer:

$y_{x^{(1)},x^{(2)},x^{(3)}}$ is the output y caused by those occurrences of $x^{(3)}$ that are preceded by an occurrence of the input word $(x^{(1)}, x^{(2)})$.

$y_{w(\ell)}$ is said to be the output y caused by the input word $w(\ell)$. Or, more precisely: given that $x^{(\ell)}$ is the present input event of $w(\ell)$ — i.e., $w(\ell) \stackrel{df}{=} \left(w(\ell-1), x^{(\ell)}\right)$ — $y_{w(\ell)}$ is the output y caused by those occurrences of $x^{(\ell)}$ that are preceded by an occurrence of the input word $w(\ell-1)$.

To make sure you understand the above concepts, you might like to ponder the examples for the filtered outputs y_1, $y_{3,2}$, and $y_{2,0,3}$ shown in Fig. 29–5. Most of all, do note that the values of the variables are always the *present* values, i.e., those observed for the present input event.

Every node (x, x, w) whose first two input events are identical is an end-node, and the associated output $y_{x,x,w}$ is identical with the output $y_{x,w}$ of node (x, w). In fact, to guarantee memorisation we define

$$y_{x,x,w} \stackrel{df}{=} y'_{x,w} \qquad Y_{x,x,w} \stackrel{df}{\Leftrightarrow} Y'_{x,w}. \tag{29.5}$$

Each end-node of the word-recognition tree represents an input word recognised — hence the name of the tree. But, note that word recognition is *not* interpreted as the word's output being 1. It is interpreted as the associated output always reacting in the same way (either always being 0, or always being 1), or by the same rule (e.g., being equal or unequal to some other filtered output).

29.4 Evaluating a Sequential Circuit

The word-recognition tree lends itself to the direct evaluation of the sequential circuit it describes. To demonstrate this, consider Fig. 29–6. Its tree is that of Fig. 29–5 without the burdening details, and the variables and values are expressed in their logical form. As you can see, the only information retained is the numbering of the edges (without which the tree would be meaningless), and the specifications of the nodes.

To see how the word-recognition tree corresponds to a whole group of reduced K-maps, all four **'elementary sub-trees'** of our example tree have been emphasized by triangular enclosures (but the concept sub-tree is not formally defined here — which you will hopefully excuse). Each such elementary sub-tree corresponds in an obvious way to a reduced K-map: the edges of a sub-tree correspond to the cells of a K-map, the output values and variables associated with the nodes to which the edges are input edges are entered into these cells of the K-maps as values and map-entered variables; the output described by the K-map is the output associated with the root of the sub-tree. Before continuing, do make sure that you understand each entry into a cell of the reduced K-maps of Fig. 29–6.

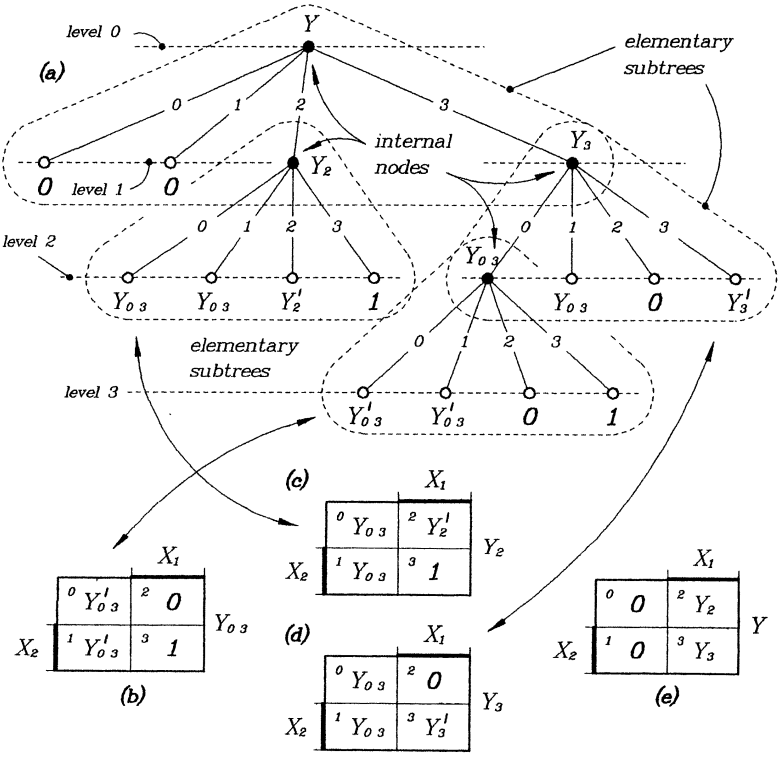


Fig. 29–6 Evaluating the word-recognition tree of Fig. 29–5a

The advantage of these reduced K-maps is obvious: We can interpret them as latches using the equations (25.3) to (25.6), or, where appropriate, the equations (20.9) to (20.12) to evaluate them. Assuming that you are by now well acquainted with the evaluation procedure, allow me to write the evaluation results of the K-maps of Fig. 29–6b to 6e without any further explanations.

$$Y_{0,3} \Leftrightarrow X_1X_2 \vee X_2Y'_{0,3} \vee \overline{X}_1Y'_{0,3}, \tag{29.6a}$$

$$Y_{0,3} \Leftrightarrow (\overline{X}_1 \vee X_2)(X_2 \vee Y'_{0,3})(X_1 \vee Y'_{0,3}). \tag{29.6b}$$

$$Y_2 \Leftrightarrow \underbrace{X_1X_2 \vee X_2Y_{0,3} \vee \overline{X}_1Y_{0,3}}_{?:\ Y_{0,3}\ldots(29.6a)} \vee X_1Y'_2. \tag{29.7}$$

$$Y_3 \Leftrightarrow \underbrace{(\overline{X}_1 \vee X_2)(X_2 \vee Y_{0,3})(X_1 \vee Y_{0,3})}_{?:\ Y_{0,3}\ldots(29.6b)}(\overline{X}_1 \vee Y'_3). \tag{29.8}$$

$$Y \Leftrightarrow X_1(X_2 \vee Y_2)(\overline{X}_2 \vee Y_3). \tag{29.9}$$

But just in case you do not yet feel comfortable with the direct evaluation of the reduced K-maps of Fig. 29–6 let me point out that you can transform them for one map-entered variable after another into standard K-maps[1]. For the internal latches Y_3, Y_2, and $Y_{0,3}$ this transformation is shown in Fig. 29–7. The equations (29.6a), (29.6b), (29.7), and (29.8) can then, if you prefer, be read from the standard K-maps of Figs. 29–7fdb, respectively.

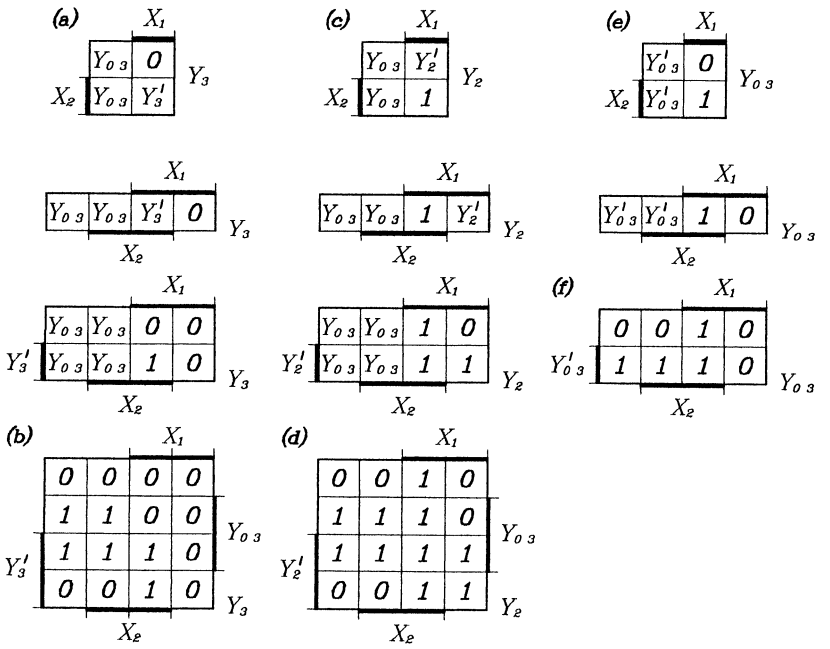


Fig. 29–7 Transforming reduced K-maps to standard K-maps

The formulas for Y_2, (29.7), and Y_3, (29.8), can be markedly simplified if the indicated substitutions can be proved to be valid. This is shown for

[1] You might like to refer to Sections 20.1 and 20.2.

one of them:

$$
\begin{aligned}
Y_2 &\overset{(29.7)}{\Longleftrightarrow} X_1X_2 \vee X_2Y_{0,3} \vee \overline{X}_1Y_{0,3} \vee X_1Y_2' \\
&\Leftrightarrow X_1X_2 \vee X_2\underbrace{(X_1X_2 \vee X_2Y_{0,3}' \vee \overline{X}_1Y_{0,3}')}_{Y_{0,3}\ldots(29.6a)}\vee \\
&\qquad \overline{X}_1\underbrace{(X_1X_2 \vee X_2Y_{0,3}' \vee \overline{X}_1Y_{0,3}')}_{Y_{0,3}\ldots(29.6a)} \vee X_1Y_2' \\
&\Leftrightarrow X_1X_2 \vee (X_1X_2X_2 \vee X_2X_2Y_{0,3}' \vee \overline{X}_1X_2Y_{0,3}')\vee \\
&\qquad (\overline{X}_1X_1X_2 \vee \overline{X}_1X_2Y_{0,3}' \vee \overline{X}_1\overline{X}_1Y_{0,3}') \vee X_1Y_2' \\
&\Leftrightarrow X_1X_2 \vee X_2Y_{0,3}' \vee \underbrace{\overline{X}_1Y_{0,3}' \vee \overline{X}_1X_2Y_{0,3}'}_{\overline{X}_1Y_{0,3}'} \vee X_1Y_2' \\
&\Leftrightarrow \underbrace{X_1X_2 \vee X_2Y_{0,3}' \vee \overline{X}_1Y_{0,3}'}_{Y_{0,3}\ldots(29.6a)} \vee X_1Y_2' \\
Y_2 &\Leftrightarrow Y_{0,3} \vee X_1Y_2'
\end{aligned}
$$

Employing the substitutions indicated in (29.7) and (29.8) allows us to use the following formulas to design the circuit — the circuit itself is shown in Fig. 29–8.

$$
\begin{aligned}
Y_{0,3} &\Leftrightarrow X_1X_2 \vee (\overline{X}_1 \vee X_2)Y_{0,3}', && (29.10a)\\
Y_2 &\Leftrightarrow Y_{0,3} \vee X_1Y_2', && (29.10b)\\
Y_3 &\Leftrightarrow Y_{0,3}(\overline{X}_1 \vee Y_3'), && (29.10c)\\
Y &\Leftrightarrow X_1(X_2 \vee Y_2)(\overline{X}_2 \vee Y_3). && (29.10d)
\end{aligned}
$$

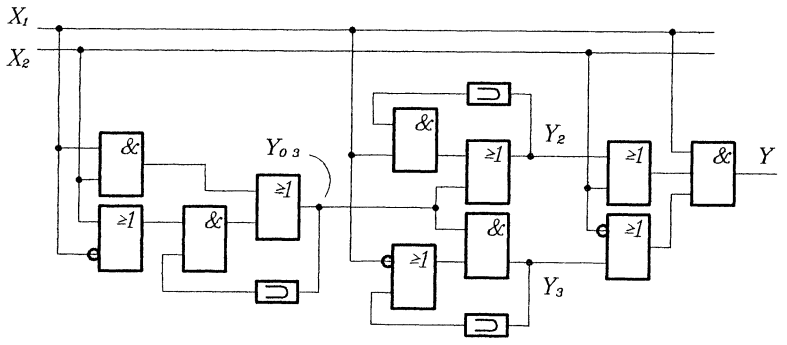


Fig. 29–8 Circuit for Fig. 29–6, derived from equations (29.10)

There is a noteworthy aspect to the equations for the internal latches Y_2 and Y_3, (29.10bc): Neither can $Y_2 \Leftrightarrow Y_{0,3} \vee X_1Y_2'$ be deduced from the K-map

of Fig. 29–7d, nor can $Y_3 \Leftrightarrow Y_{0,3}(\overline{X}_1 \vee Y_3')$ be calculated from the K-map of Fig. 29–7b. Likewise they cannot be calculated from the reduced K-maps Fig. 29–7ac. This is due to the fact that the variables arranged around the perimeter of the K-maps are *not independent of one another* —specifically: the value of the variable $Y_{0,3}$, defined as $Y_{0,3} \Leftrightarrow X_1X_2 \vee (\overline{X}_1 \vee X_2)Y_{0,3}'$, (29.10a), depends on the values of X_1 and X_2.

After this first impression of the usage of the word-recognition tree, we next take a close look at how to obtain the word-recognition tree by a process of formal analysis of a flow table.

Chapter 30

Word-Recognition Tables *

The procedure for designing a sequential switching circuit can be summarised as follows: From a flow table which is the specification of the switching circuit to be designed, you develop a whole set or cluster of interconnected *'word-recognition tables'*. Each such table allows you to develop an associated reduced K-map. The evaluation of the reduced K-maps provides logic formulas that, together, describe the circuit.

The first section, Section 30.1, states what a word-recognition table is, and presents an algorithm for developing such a table from a flow table. The prime problem is how to terminate the successive development of word-recognition tables. A rather simple approach is presented in Section 30.2. The general case is discussed in Section 30.3. The procedures presented are formal processes, so that they can be programmed for a computer.

30.1 Developing Word-Recognition Tables

Each row of a *word-recognition table* corresponds to an internal state q, while each column corresponds to an input event x giving the word-recognition table the same arrangement of cells as that of the body of a flow table. In fact, the values entered into the cells (q, x) of a word-recognition table are obtained from the output column of an associated flow table. Let me refer to this output column as $y_{w(\ell)}$, and to the flow table as FT-$y_{w(\ell)}$. The corresponding word-recognition table WRT-$y_{w(\ell)}$ is then obtained by the following process:

(30.1) Developing a **word-recognition table** WRT-$y_{w(\ell)}$ from a given flow table FT-$y_{w(\ell)}$.

(a) Every cell (q, x) in a word-recognition table WRT-$y_{w(\ell)}$ that corresponds to a stable condition (q, x) in the flow table FT-$y_{w(\ell)}$ is assigned the flow table's output value $y_{w(\ell)}(q)$.

(b) In the flow table FT-$y_{w(\ell)}$ consider any cell (q, x) that contains a GOTO-q^+ command such that $q^+ \neq q$: Then the flow table's output value $y_{w(\ell)}(q^+)$ of the target state q^+ is inscribed into cell (q, x) of the word-recognition table WRT-$y_{w(\ell)}$.

The above procedure applies to primitive as well to non-primitive flow tables, and is illustrated in Fig. 30–1 where ℓ is set to 0 ($y_{w(0)} = y$). Before continuing, please do take the time to understand how the word-recognition tables of Fig. 30.1bc follow from procedure (30.1).

(a) (x_1, x_2)

	0	1	3	2	
q	(0,0)	(0,1)	(1,1)	(1,0)	y
a	e	e	(a)	d	0
b	e	e	(b)	d	1
c	(c)	(c)	a	(c)	0
d	c	c	a	(d)	1
e	(e)	(e)	b	d	0

FT-y

(b) WRT-y

	0	1	3	2
q	(0,0)	(0,1)	(1,1)	(1,0)
a			(0)	
b			(1)	
c	(0)	(0)		(0)
d				(1)
e	(0)	(0)		

Developed from (30.1a)

(c) WRT-y

0	1	3	2
(0,0)	(0,1)	(1,1)	(1,0)
0	0		1
0	0		1
		0	
0	0	0	
		1	1

Developed from (30.1b)

Fig. 30–1 Illustrating procedure (30.1) for obtaining the primary word recognition table from a flow table

The first word-recognition table you will want to develop will always be the word-recognition table WRT-y associated with the original flow table FT-y, and thus called the **primary WRT**. If the values, the 0s and 1s, of a given column x of the primary word-recognition table WRT-y do not — to put it very vaguely for the moment — obey the law of **causality**, i.e., cannot *be predicted or deduced*, then the column x being considered is used to define a new flow table FT-y_x which could be referred to as a **secondary** flow-table of FT-y. Such a secondary flow-table FT-y_x will itself allow us to develop an associated **secondary** word-recognition table WRT-y_x. As is easy to imagine, this process of going from *primary* to *secondary* to *'ternary'* tables, etc., will continue as long as we encounter a column in a successor word-recognition table whose values do not abide by the law of causality. Only when no such columns are encountered does the process terminate. Let us refer to the set of flow tables created in this process (including the original flow-table FT-y) as a **cluster of flow tables**. Similarly, the set of associated word-recognition tables is referred to as a **cluster of word-recognition tables**. To express the above procedure in a formal way, we introduce the following auxiliary definition.

(30.2) Defining **column** or **output vectors.**

(a) The 0 or 1 of any row q of the output column of a flow table FT-$y_{w(\ell)}$ is an element $y_{w(\ell)}(q)$ of an **output vector** $\mathbf{y}_{w(\ell)}$. The elements of the output vector $\mathbf{y}_{w(\ell)}$ constitute the values, 0 or 1, of the **filtered output variable** $y_{w(\ell)}$.

(b) The 0 or 1 of any row q of a column x of a word-recognition table WRT-$y_{w(\ell)}$ is an element $y_{x,w(\ell)}(q)$ of an **output vector** $\mathbf{y}_{w(\ell)}$. The

elements of the output vector $\mathbf{y}_{x,w(\ell)}$ constitute the values, 0 or 1, of the **filtered output variable** $y_{x,w(\ell)}$.

The value $y_{w(\ell)}$ of any output vector $\mathbf{y}_{w(\ell)}$ is either 0 or 1 depending on the internal state q. When speaking of an output vector $\mathbf{y}_{w(\ell)}$ we shall, for the sake of simplicity, often only write its value $y_{w(\ell)}$.

In the formal presentation of the process for developing a cluster of word-recognition tables, (30.3), you will come across the as yet undefined technical term of *determinateness*. In a first reading, please take **determinateness** in the informal sense of **causality** mentioned above. The next two sections of this chapter are devoted to explaining various forms of determinateness.

(30.3) Developing a **cluster of word-recognition tables** from a given mother flow table FT-y.

Step 1. Set ℓ to 0.

Step 2. Develop a word-recognition table WRT-$y_{w(\ell)}$ from a flow table FT-$y_{w(\ell)}$ according to (30.1).

Step 3. Successively, each *'non-determinate'* output vector $\mathbf{y}_{x,w(\ell)}$ of the word-recognition table WRT-$y_{w(\ell)}$ is used to substitute the output vector $\mathbf{y}_{w(\ell)}$ of the flow table FT-$y_{w(\ell)}$ thereby creating a new flow table FT-$y_{x,w(\ell)}$ for each non-determinate output vector $\mathbf{y}_{x,w(\ell)}$.

Step 4. Rename each new flow table FT-$y_{x,w(\ell)}$ developed in the previous step to FT-$y_{w(\ell+1)}$ and go to step 2.

Step 5. The procedure terminates when none of the new flow tables leads to a word-recognition table containing a *'non-determinate'* output vector.

30.2 Primitive Determinateness

We now take a first look at the concept of determinateness. The simpler concept, discussed here, is a special case of the general concept discussed in the next section.

(30.4) Primitive determinateness. A column or output vector $\mathbf{y}_{x,w(\ell)}$ of a word-recognition table WRT-$y_{w(\ell)}$ is, as well as the filtered output variable $y_{x,w(\ell)}$, **determinate**

(a) if all its elements $y_{x,w(\ell)}(q)$ are 0, in which case $y_{x,w(\ell)}$ itself is said to be 0, or if all its elements $y_{x,w(\ell)}(q)$ are 1, in which case $y_{x,w(\ell)}$ is said to be 1.

(b) if it is either *equal* or *complementary* to an output vector $\mathbf{y}_{w_j(\ell_j)}$ of some other flow table FT-$y_{w_j(\ell_j)}$ — with $0 \leq \ell_j \leq \ell$ — of the same cluster. (Note that two binary vectors are *equal* or *complementary* if equally indexed elements are equals or complements, respectively.)

(a) (x_1, x_2)

	0	1	3	2	
q	(0,0)	(0,1)	(1,1)	(1,0)	y
a	h	g	(a)	d	0
b	h	g	(b)	d	1
c	f	e	a	(c)	0
d	f	e	a	(d)	1
e	f	(e)	a	c	0
f	(f)	e	a	c	0
g	h	(g)	b	d	0
h	(h)	g	b	d	0

FT-y

(b) WRT-y

	0	1	3	2
q	(0,0)	(0,1)	(1,1)	(1,0)
a	0	0	0	1
b	0	0	1	1
c	0	0	0	0
d	0	0	0	1
e	0	0	0	0
f	0	0	0	0
g	0	0	1	1
h	0	0	1	1
y_e	y_0 =0	y_1 =0	y_3	y_2

Fig. 30–2 Developing a basic or mother word-recognition table

Let us now develop a cluster of word-recognition tables for our example of Section 29.2 the flow table of which (Fig. 29–4b) is repeated in Fig. 30–2a and referred to as FT-y. The primary word-recognition table WRT-y of Fig. 30–2b is obtained using procedure (30.1). Before continuing with this text, please do make certain you know how to obtain WRT-y from FT-y (Fig. 30–2) — if necessary, it might help to refer to Fig. 30–1.

(a) (x_1, x_2)

	0	1	3	2	
q	(0,0)	(0,1)	(1,1)	(1,0)	y_3
a	h	g	(a)	d	0
b	h	g	(b)	d	1
c	f	e	a	(c)	0
d	f	e	a	(d)	0
e	f	(e)	a	c	0
f	(f)	e	a	c	0
g	h	(g)	b	d	1
h	(h)	g	b	d	1

FT-y_3

(b) WRT-y_3

	0	1	3	2
q	(0,0)	(0,1)	(1,1)	(1,0)
a	1	1	0	0
b	1	1	1	0
c	0	0	0	0
d	0	0	0	0
e	0	0	0	0
f	0	0	0	0
g	1	1	1	0
h	1	1	1	0
$y_{e\,3}$	$y_{0\,3}$	$y_{1\,3}$ $=y_{0\,3}$	$y_{3\,3}$ y_3'	$y_{2\,3}$ $=0$

Fig. 30–3 Developing the word-recognition table for y_3

According to the definition of determinateness, (30.4a), the filtered variables y_0 and y_1 of WRT-y of Fig. 30–2b, both being 0, are determinate, while y_3 and y_2 are non-determinate, neither of them being either equal or complementary to y. For these two variables, y_3 and y_2, step 3 of (30.3) leads

to the secondary flow tables FT-y_3 and FT-y_2 of Fig. 30–3a and Fig. 30–4a, respectively.

(a) (x_1, x_2)

	0	1	3	2	
q	(0,0)	(0,1)	(1,1)	(1,0)	y_2
a	h	g	(a)	d	1
b	h	g	(b)	d	1
c	f	e	a	(c)	0
d	f	e	a	(d)	1
e	f	(e)	a	c	0
f	(f)	e	a	c	0
g	h	(g)	b	d	1
h	(h)	g	b	d	1

FT-y_2

(b) WRT-y_2

	0	1	3	2
q	(0,0)	(0,1)	(1,1)	(1,0)
a	1	1	1	1
b	1	1	1	1
c	0	0	1	0
d	0	0	1	1
e	0	0	1	0
f	0	0	1	0
g	1	1	1	1
h	1	1	1	1
$y_{e\,2}$	$y_{0\,2}$ $=y_{0\,3}$	$y_{1\,2}$ $=y_{0\,3}$	$y_{3\,2}$ $=1$	$y_{2\,2}$ y_2'

Fig. 30–4 Developing the word-recognition table for y_2

According to step 4 of procedure (30.3), each of the secondary flow tables FT-y_3 and FT-y_2 are again analysed (by going to steps 2 of the procedure) to obtain the secondary word-recognition tables, WRT-y_3 and WRT-y_2 respectively, shown in Fig. 30–3b and Fig. 30–4b. Again: Please make sure you know how these word-recognition tables were obtained using (30.1). Let us take a closer look at WRT-y_3 of Fig. 30–3b. Which of its filtered variables $y_{x,3}$ (i.e., $y_{0,3}$, $y_{1,3}$, $y_{3,3}$, and $y_{2,3}$) is determinate according to (30.4)? $y_{2,3}$ is determinate, as all elements $y_{2,3}(q)$ are 0. $y_{3,3}$ is determinate due to $y_{3,3} = y_3$, and by (29.5) we can even write $y_{3,3} = y_3'$. $y_{0,3}$ and $y_{1,3}$ are non-determinate, as they are neither equal nor complementary to y, or to y_3, or to y_2. But, as $y_{0,3}$ and $y_{1,3}$ are equal, it suffices to determine one of them. The word-recognition table WRT-y_2 can be discussed in a like manner.

As the four non-determinate filtered variables $y_{0,3}$, $y_{1,3}$, $y_{0,2}$, and $y_{1,2}$ of WRT-y_3 and WRT-y_2 are mutually identical, it suffices to use *one* of these columns, say $y_{0,3}$, to create the next flow table: FT-$y_{0,3}$, shown in Fig. 30–5a. You might like to argue, why $y_{1,0,3}$ of the associated word-recognition table WRT-$y_{0,3}$ can be equalled to $y'_{0,3}$. All filtered variables $y_{x,0,3}$ of WRT-$y_{0,3}$ are determinate so that procedure (30.3) terminates. The obtained cluster of word-recognition tables is $\{\text{WRT-}y, \text{WRT-}y_2, \text{WRT-}y_3, \text{WRT-}y_{0,3}\}$.

A **word-recognition tree** is a good visual representation of a cluster of **word-recognition tables**. Each elementary sub-tree (see Fig. 29–6a) of the word-recognition tree is associated with a word-recognition table in the following way. The output variable $y_{w(\ell)}$ defining a word-recognition table WRT-$y_{w(\ell)}$ is the root of the elementary sub-tree, while each of its edges x leads to the node $y_{x,w(\ell)}$ taken from column x of the word-recognition

(a) (x_1, x_2)

	0	1	3	2	
q	(0,0)	(0,1)	(1,1)	(1,0)	$y_{0\,3}$
a	h	g	(a)	d	1
b	h	g	(b)	d	1
c	f	e	a	(c)	0
d	f	e	a	(d)	0
e	f	(e)	a	c	0
f	(f)	e	a	c	0
g	h	(g)	b	d	1
h	(h)	g	b	d	1

FT-$y_{0\,3}$

(b) WRT-$y_{0\,3}$

	0	1	3	2
q	(0,0)	(0,1)	(1,1)	(1,0)
a	1	1	1	0
b	1	1	1	0
c	0	0	1	0
d	0	0	1	0
e	0	0	1	0
f	0	0	1	0
g	1	1	1	0
h	1	1	1	0
$y_{e\,0\,3}$	$y_{0\,0\,3}$ $=y'_{0\,3}$	$y_{1\,0\,3}$ $=y'_{0\,3}$	$y_{3\,0\,3}$ $=1$	$y_{2\,0\,3}$ $=0$

Fig. 30–5 Developing the word-recognition table for $y_{0,3}$

table. Similarly, every word-recognition table WRT-$y_{w(\ell)}$ corresponds to a **reduced K-map** RKM-$y_{w(\ell)}$. The output of the reduced K-map is the variable $y_{w(\ell)}$, while the filtered variable $y_{x,w(\ell)}$ of WRT-$y_{w(\ell)}$ is entered into cell x of the RKM. The example used in this section is continued and completed in Section 29.4.

30.3 General Determinateness

The principle idea of *primitive determinateness* of an output vector $\mathbf{y}_{x,w(\ell)}$ is that it can be *'related to'* or be seen as being *'dependent on'* a previously developed output vector of the cluster. In extension, the basic idea of *general determinateness* is that the output vector $\mathbf{y}_{x,w(\ell)}$ can be *'related to all'* or be seen as being *'dependent on all'* previously developed output vectors of the cluster. To give tangible form to this concept, let us take a second look at the possible determinateness of the output variable $y_{0,3}$ of Fig. 30–5a writing it in a so-called **table of output vectors** (TOV) together with the previously developed output variables y, y_3, and y_2 — see Fig. 30–6a.

The depedence of the output vector $\mathbf{y}_{0,3}$ on the previously developed output vectors $\mathbf{y}$, $\mathbf{y}_3$, $\mathbf{y}_2$ is *functional*, meaning that a triple of values (y, y_3, y_2) which occurs more than once (i.e., in more than one row) — such as the triple $(0, 0, 0)$ — always maps to one and the same value of $y_{0,3}$. Generalising, this leads to the

(30.5) Definition of **General determinateness**. An output vector $\mathbf{y}_{x,w(\ell)}$ is **determinate** if its elements $y_{x,w(\ell)}(q) \in \{0, 1\}$ are values

of some **function**[1] that maps all ordered tuples $(\cdots, y_{w_j(\ell_j)}(q), \cdots)$ of one and the same state q of all *non-determinate* output vectors $\mathbf{y}_{w(\ell)}$, with $0 \leq \ell_j \leq \ell$, to $\{0,1\}$. The output vector $\mathbf{y}$ of the original or mother flow table FT-y is always taken to be non-determinate.

To acquaint ourselves with the above definition, let us apply it to the table of output vectors of Fig. 30–6a: The output vector $y_{0,3}$ is determinate if its elements $y_{0,3}(a) = 1$, $y_{0,3}(b) = 1$, $y_{0,3}(c) = 0$, etc., are values of some *function* that maps all ordered triples $\big(y(q), y_3(q), y_2(q)\big)$ of one and the same state q — i.e., all the triples $(0,0,1)$, $(1,1,1)$, $(0,0,0)$, etc. — of all non-determinate output vectors $\mathbf{y}$, $\mathbf{y}_3$, $\mathbf{y}_2$ to $\{0,1\}$. As a visual aid in deciding whether $y_{0,3}$ is determinate, i.e., whether the table of output vectors, Fig. 30–6a, is a *function* or a *relation*, take a look at the *graph of the table of output vectors* shown in Fig. 30–6b.

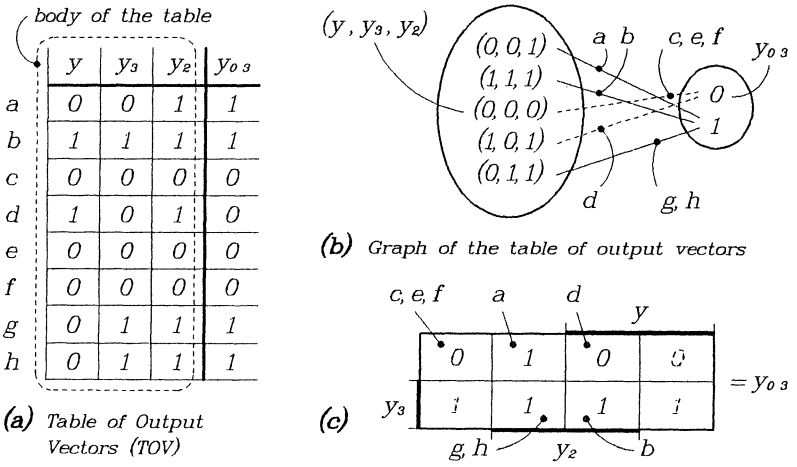


Fig. 30–6 Testing for general determinateness

Each *unique* triple $\big(y(q), y_3(q), y_2(q)\big)$ of the body of the table of output vectors is one element of the domain. Each *unique* element of the output vector $y_{0,3}$ is an element of the co-domain. As each element of the domain maps to one and only one value of the co-domain, as follows from the table of output vectors, the graph represents a *function*, so that $y_{0,3}$ is determinate. Expressing $Y_{0,3} \Leftrightarrow y_{0,3} = 1$ as a logic formula of the variables Y, Y_3, Y_2, as indicated in the K-map of Fig. 30–6c, is discussed in the next section.

In the above example, the variables y_3 and y_2 were asserted to be non-determinate. Normally, you would have started by testing whether this was

[1] which maps each element of its domain once and only once to an element of the co-domain, as contrasted with a **relation** which we take to map at least one element of the domain to two or more elements of its co-domain.

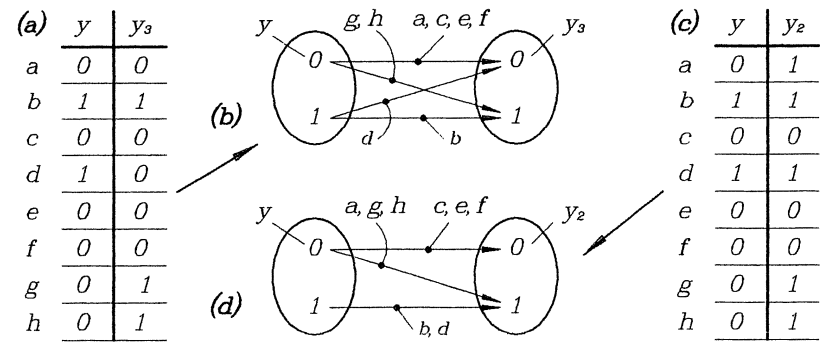


Fig. 30–7 Testing for general determinateness of y_3 and y_2

the case or not. This test is — belatedly — carried out in Fig. 30–7, and as the graphs represent relations, and not functions, y_3 and y_2 are non-determinate.

30.4 Evaluation and Partial Mappings

The function pictured in Fig. 30–6b is incompletely specified in that its domain does not contain *all* ordered triples of (y, y_3, y_2), i.e., $(0,1,0)$, $(1,0,0)$, and $(1,1,0)$ are missing. As these input events do not occur in our circuit, they can (and must) be assigned arbitrary values of $\{0,1\}$ — so-called *'don't cares'*[1]. This allows us to choose the inscribed values shown in the K-map of Fig. 30–6c to evaluate $Y_{0,3}$ as expressed in (30.6a). The formulas for Y_2, Y_3, and Y of (30.6bcd) are equivalent to (29.7), (29.8), and (29.9), respectively.

$$Y_{0,3} \Leftrightarrow Y_3 \vee \overline{Y} Y_2, \tag{30.6a}$$

$$Y_2 \Leftrightarrow X_1 X_2 \vee (\overline{X}_1 \vee X_2) Y_{0,3} \vee X_1 Y_2', \tag{30.6b}$$

$$Y_3 \Leftrightarrow (\overline{X}_1 \vee X_2)(X_1 X_2 \vee Y_{0,3})(\overline{X}_1 \vee Y_3'), \tag{30.6c}$$

$$Y \Leftrightarrow X_1 (X_2 \vee Y_2)(\overline{X}_2 \vee Y_3). \tag{30.6d}$$

The circuit, shown in Fig. 30–8, is equivalent in its stationary behaviour to that of Fig. 29–7. To compare the number of gates employed in their designs, let us count a 3-input gate as two 2-input gates. Then, the circuit of Fig. 30–8 employs 9 AND-gates, 9 OR-gates, and 5 INVERTERs as compared with 6 AND-gates, 6 OR-gates, and 3 INVERTERs for the circuit of Fig. 29–7. Counting the number of signals that are fed back, the comparison is as follows: In Fig. 30–8 five signals are fed back (Y, Y_2, Y_2', Y_3, and Y_3'), while in Fig. 29–7 only three signals are fed back (Y_2', Y_3', $Y_{0,3}'$). But, do note that the number of signals fed back is distinct from the number of

[1] see Section (18.2).

latches needed to realise the circuits. While the circuit of Fig. 29–7 requires three latches (and no further feedback signals), the circuit of Fig. 30–8 can be designed with only two latches (for Y_2 and Y_3 — which internally make use of the feedback signals Y_2' and Y_3'), but the signals Y, Y_2, and Y_3 must still be fed back.

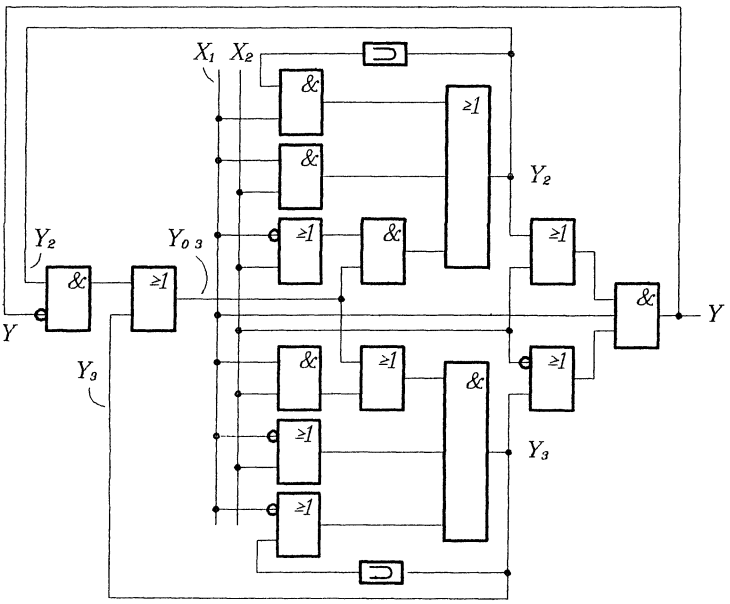


Fig. 30–8 The circuit according to equations (30.6)

A word of caution is appropriate: The result of the above comparison is not typical for a comparison of primitive with general determinateness.

30.5 Testing for General Determinateness

The purpose of the *graph* of a table of output vectors is only to visualise and clarify concepts: (a) Which elements belong to the domain of the mapping, and (b) does the mapping represent a function or a relation? But, whether the mapping is a function or a relation is just as easily deduced directly from the table of output vectors allowing us to omit the development of the table's graph:

(30.7) The output variable $y_{x,w(\ell)}$ of a table of output vectors is **determinate** if all mutually identical ordered tuples of the rows q of the body of the table are mapped to one and the same output value of $y_{x,w(\ell)}$. *Vice versa*, if two identical tuples exist that map to different values, then $y_{x,w(\ell)}$ is **non-determinate**.

For instance: y_3 of the table of output vectors for y_3 of Fig. 30–7a is non-determinate, as in row a, 0 maps to 0, while in row g, 0 maps to 1. On the other hand, $y_{0,3}$ of the table of output vectors $y_{0,3}$ of Fig. 30–6a is determinate because all rows — c, e, and f — containing the ordered tuples $(0, 0, 0)$ map to 0, and all rows — g and h — containing the ordered tuples $(0, 1, 1)$ map to 1.

Chapter 31

Catenation Model*

The explanation of a sequential circuit as an automaton that works in conjunction with a program called a flow table — which is the theme of Section (29.1) — is descriptive but not algebraic. The first attempts to formulate algebraic models were put forth by **Moore** [1956], **Mealy** [1955] and **Medvedev** [1956]. Their models are contrasted with the stringent definition of a sequential circuit as a *composite function* f which itself is the *catenation* of two specified functions — the so-called *transition function* g (conforming to very special properties) followed by the so-called *output function* h. The main body of the chapter is then devoted to analysing and developing the catenation model of a sequential circuit. Especially the connections to the orthodox models, as well as to the methods centring around the word-recognition tables of the previous chapter are studied.

31.1 The Moore and Mealy Models

Huffman's flow table (see, e.g., Fig. 29–1), being so successful, was used as the basis for an algebraic definition of a sequential circuit. The two values determined by a flow table are, on the one hand, the *internal states* q^+ entered into the body of the flow table, and on the other hand, the *output values* y entered into the output column. As before, we refer to the 'GOTO-commands', i.e., the internal states entered into the body of the flow table as q^+, while the internal states labelling the rows of the flow table are denoted as q. Then, q^+ is taken to be a function of q and the input event x, this function being called the *transition function* g. **Moore** [1956] took the output variable y to be a function only of the internal state q, while **Mealy** [1955] preferred to view a seemingly more general case in which the output y is dependent not only on the internal state q but also on the input event x. An example for this way of viewing the output dependence is the circuit of Fig. 30–8. The above concepts are usually expressed in a formal way (which seems to have made them more acceptable). Where it is necessary to distinguish between sets and functions of the following *Moore*, *Mealy*, and *Medvedev models*, the double indices *'mr'* are used for Moore, *'ml'* for Mealy, and *'md'* for Medvedev.

(31.1) The **Moore model** of a sequential circuit is the 'quintuple' $(\mathcal{E}, \mathcal{Q}_{mr}, \{0,1\}, g_{mr}, h_{mr})$ in which
$\mathcal{E}$ is the set of input events $0, 1, \ldots, 2^n - 1$,
$\mathcal{Q}_{mr}$ is the finite non-empty set of states $1, 2, \ldots, s_{mr}$,
$\{0,1\}$ is the set of output values,
g_{mr} is any function $g_{mr} : \mathcal{Q}_{mr} \times \mathcal{E} \mapsto \mathcal{Q}_{mr}$,
h_{mr} is any function $h_{mr} : \mathcal{Q}_{mr} \mapsto \{0,1\}$.

As described in the paragraph above, the Mealy model differs from the Moore model mainly in the output function. But at this point it is still open whether both models necessarily have the same set of internal states — an unwarranted *a priori* assumption.

(31.2) The **Mealy model** of a sequential circuit is the 'quintuple' $(\mathcal{E}, \mathcal{Q}_{ml}, \{0,1\}, g_{ml}, h_{ml})$ in which
$\mathcal{E}$ is the set of input events $0, 1, \ldots, 2^n - 1$,
$\mathcal{Q}_{ml}$ is the finite non-empty set of states $1, 2, \ldots, s_{ml}$,
$\{0,1\}$ is the set of output values,
g_{ml} is any function $g_{ml} : \mathcal{Q}_{ml} \times \mathcal{E} \mapsto \mathcal{Q}_{ml}$,
h_{ml} is any function $h_{ml} : \mathcal{Q}_{ml} \times \mathcal{E} \mapsto \{0,1\}$.

Not quite understandably, the Moore and Mealy models define the so-called **transition functions** g_{mr} and g_{ml}, and **output functions** h_{mr} and h_{ml} **independently** of one another (neither influencing the other). The prime deficiency of the Moore and Mealy models is that, even if you accept that they describe sequential circuits, they do not do so exclusively: They also describe systems that no engineer would accept to be sequential circuits. This is mainly due to the lack of any restrictions imposed upon the transition functions. In the next section a more restrictive model is therefore proposed.

The independence of the functions g and h led **Medvedev** [1956] to propose studying a model consisting exclusively of the transition function. The idea behind this approach is to study the class of sequential circuits that have the same transition function g in common. **Harrison** [1965], in his excellent book, introduced sequential circuits via the Medvedev model.

(31.3) The **Medvedev model** of a sequential circuit *without outputs* is the 'quadruple' $(\mathcal{E}, \mathcal{Q}_{md}, g, q_0)$ in which
$\mathcal{E}$ is the set of input events $0, 1, \ldots, 2^n - 1$,
$\mathcal{Q}_{md}$ is the finite non-empty set of states $1, 2, \ldots, s_{md}$,
g_{md} is any function $g_{md} : \mathcal{Q}_{md} \times \mathcal{E} \mapsto \mathcal{Q}_{md}$,
q_0 is the 'initial' value of the state variable $q \in \mathcal{Q}_{md}$.

In contrast to these established models, you might want to consider the more restrictive catenation model presented next.

31.2 Catenation Model — Feedback and Delays

The algebraic definition of a sequential circuit f, given in (31.4) below, is a summary of Section 29.1. The main aspects of the definition are, **firstly**, that f is the *catenation* of a transition function g followed by an output function h, and that, **secondly**, the transition function g cannot be an arbitrary function from its domain to its co-domain, but must abide by certain rules or restrictions. Each rule is expressed in three ways: a characterising phrase, an explanation in plain English, and finally, the logic formulation (this and only this being binding). Both above aspects (catenation, and imposing restricting rules on g) are missing in the Moore and Mealy models.

(31.4) A sequential circuit is defined as the **catenation** $f = hg$ of an **output function** $h : \mathcal{Q} \mapsto \{0,1\}$ — with $y = h(q^+)$ — whose arguments $q^+ \in \mathcal{Q}$ stem from a **transition function** $g : \mathcal{Q} \times \mathcal{E} \mapsto \mathcal{Q}$ with the output $q^+ = g(q,x)$. Thus, the composite function $f : \mathcal{Q} \times \mathcal{E} \mapsto \{0,1\}$ maps $\mathcal{Q} \times \mathcal{E}$ to $\{0,1\}$. $\mathcal{Q} = \{1,2,\ldots,s\}$ consists of s states. The transition function g complies to the following rules:

(a) Every row q must be accessible: Any row q of a table defining g has at least one 'stable condition' (q,x), and at least one of these is accessible from some other row q^*,

$$\bigvee_{x \in \mathcal{E}} \Big(g(q,x) = q \wedge \bigvee_{q^* \in \mathcal{Q} \setminus \{q\}} g(q^*,x) = q \Big) \Leftrightarrow \mathbf{1}.$$

(b) No subset $\mathcal{R}$ of the states $\mathcal{Q}$ may be a sink. If *all* GOTO-commands of the rows q of a real subset $\mathcal{R}$ of $\mathcal{Q}$ point exclusively to rows of $\mathcal{R}$, then $\mathcal{R}$ is called a **sink**,

$$q \in \mathcal{R} \Leftrightarrow (\mathcal{R} \subset \mathcal{Q}) \wedge (\mathcal{R} \neq \mathcal{Q}) \wedge (g(q,x) \in \mathcal{R}).$$

You can express the content of *'No subset $\mathcal{R} \subset \mathcal{Q}$ may be a sink'* positively by stating: In every real subset $\mathcal{R}$ of states there must be at least one state q that for some input event x maps to a state q^* not in $\mathcal{R}$,

$$\bigvee_{q \in \mathcal{R}} (g(q,x) = q^*) \wedge (q^* \notin \mathcal{R}) \Leftrightarrow \mathbf{1}.$$

(c) Stability*): A GOTO-q^+ command in a column x of the table defining g always points to the 'stable condition' (q^+,x) in that column,

$$g(q,x) = q^+ \Rightarrow g(q^+,x) = q^+.$$

*) On stability also see (29.1c).

The simplest sink is a subset $\mathcal{R} \subset \mathcal{Q}$ consisting of a single state q: Its GOTO-commands must all be q thus representing stable states.

The above catenation model of a sequential circuit, (31.4), is dominated by the concepts of *catenation* and of *stability*. Let us **first** look at the **catenation aspect** as pictured in Fig. 31–1. Each sub-figure emphasizes the table which characterises one of the functions g, h and f. The transition function $g : \mathcal{Q} \times \mathcal{E} \mapsto \mathcal{Q}$ is specified by a table called a **transition table** — see Fig. 31–1a — while the output function $h : \mathcal{Q} \mapsto \{0,1\}$ is specified by the **output table** — see Fig. 31–1b. The composite function $f : \mathcal{Q} \times \mathcal{E} \mapsto \{0,1\}$, with $f = hg$, is then specified by the **primary WRT** as in Fig. 31–1c. That f *is* pictured by the primary WRT follows from the process described in (30.1b) — which in fact holds for all cells of the transition table, whether stable or not — of the previous chapter by which the primary WRT is developed: 'The flow table's output value $y(q^+)$ of the target state q^+ is inscribed into cell (q, x) of the word recognition table WRT'.

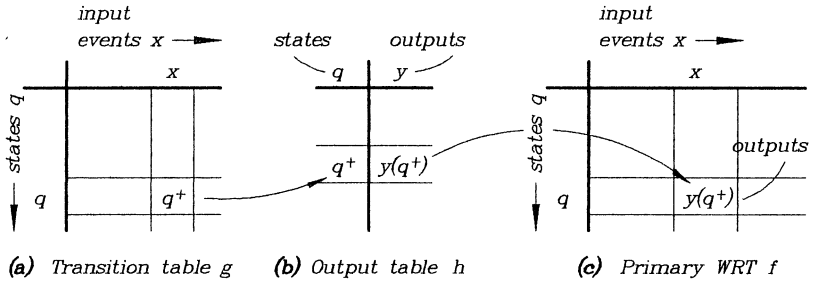


Fig. 31–1 Picturing the catenation model of a sequential circuit
(a) transition table describing $g : \mathcal{Q} \times \mathcal{E} \mapsto \mathcal{Q}$ with $q^+ = g(q, x)$,
(b) output table describing $h : \mathcal{Q} \mapsto \{0,1\}$ with $y = h(q^+)$,
(c) primary WRT describing $f : \mathcal{Q} \times \mathcal{E} \mapsto \{0,1\}$ with $f = hg$,
i.e, $f(q, x) = h(g(q, x)) = h(q^+) = y$.

Just for the moment, take f, g, and h to be any functions such that $\mathcal{X} \overset{f}{\mapsto} \mathcal{Z} = \mathcal{X} \overset{g}{\mapsto} \mathcal{Y} \overset{h}{\mapsto} \mathcal{Z}$. By this token, f is developed from g and h by catenating these functions — $f = hg$. In this general case, in which $f = hg$, there is no way of finding h when f and g have been specified, and there is no way of finding g when f and h have been specified, as even the example of Fig. 22–3 shows. Now, returning to the sequential circuit where f, g, and h comply to (31.4), we shall see that the restrictive rules that govern the transition function g allow us to find the output function h when the transition function g and the composite function f are specified.

As a final remark on catenation, note that according to (31.4), a sequential circuit is a *function* — and not a 'quintuple', or a 'quadruple', or even a 'computer program', these concepts being too vague to build a theory on.

The **second** dominating aspect is that of **stability** as pictured in the transition table of Fig. 31–2a: If q^+ is the output value of the transition

function g for the arguments q and x, i.e., $g(q,x) = q^+$, then the output q^+ remains unchanged or **stable** for an invariate input event x if the *'present'* state q of $g(q,x)$ is replaced by the *'next'* state q^+:

(31.4c) $$g(q,x) = q^+ \Rightarrow g(q^+,x) = q^+.$$

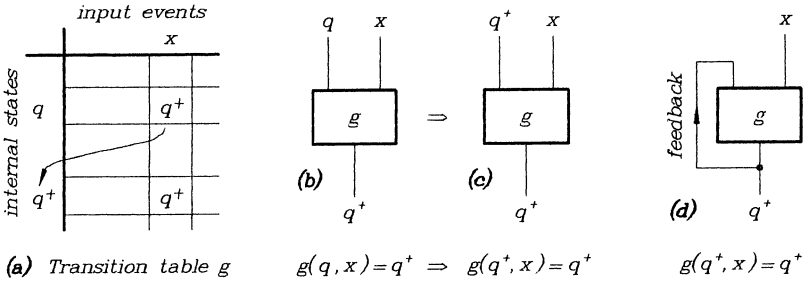


Fig. 31–2 Introducing the concept of feedback for sequential circuits

Suppose the *'black box'* of Fig. 31–2b expresses the input-output behaviour of the transition function g, picturing that when we enter q and x the output value is q^+ — according to $g(q,x) = q^+$. Due to stability, we know that if $g(q,x) = q^+$, we may validly write $g(q^+,x) = q^+$, this step being pictured in Figs. 31–2bc. As the *input* q^+ and *output* q^+ in Fig. 31–2c are identical we can connect them obtaining the **feedback** as shown in Fig. 31–2d. Due to stability, $g(q,x) = q^+ \Rightarrow g(q^+,x) = q^+$, the feedback model of Fig. 31–2d completely replaces the *'open'* model Fig. 31–2b. This follows directly from *modus ponens* 'If $P \Leftrightarrow \mathbf{1}$ and $P \Rightarrow Q$, then $Q \Leftrightarrow \mathbf{1}$', (8.16), if we substitute P by $g(q,x) = q^+$, and Q by $g(q^+,x) = q^+$. The first premise, $g(q,x) = q^+$, is true because by definition we are working with the transition function g. The second premise, $g(q,x) = q^+ \Rightarrow g(q^+,x) = q^+$, is true due to the assumption of stability. Thus the conclusion, $g(q^+,x) = q^+$, is true validating the feedback model of Fig. 31–2d.

To realise binary sequential circuits, each state q of $Q = \{0,1,\ldots,s\}$ must be encoded binarily, meaning, we must introduce, say, p binary state-variables $s_1, s_2, \ldots, s_p \in \{0,1\}$ so that we can map each $q \in Q$ to exactly one of the binary numbers $s_1 2^0 + s_2 2^1 + \ldots + s_p 2^{p-1}$. Just as q is fed back (see Fig. 31–2d) so is each s_j, $j \in \{1,\ldots,p\}$, of the binary number $(s_1,\ldots,s_p)$ representing q allowing us to interpret each s_j as the output variable of an internal latch. For instance, the three logic variables Y_{03}, Y_2, Y_3 of Fig. 29–8 can be associated with three numeric variables s_1, s_2, s_3 as used here. The smallest number $p_{\min}$ of binary state-variables (and thus of internal latches) needed to encode s internal states is the smallest integer $p_{\min}$ to satisfy the inequality $s < 2^{p_{\min}}$. Drawing on our experience with the feedback model of latches (Section 24.2), and the discussion on the necessity of a delay in the

feedback (Section 24.5), it is obvious that each internal latch must contain an inertial delay in the feedback. In its general form this is expressed as

$$s_j = g_j(x_1, \ldots, x_n; \underbrace{s_1, \ldots, s_{j-1}, s'_j, s_{j+1}, \ldots, s_p}_{\text{only } s_j \text{ is delayed}}) \quad \text{with} \quad j \in \{1, \ldots, p\},$$

and pictured in Fig. 31–3a. The inertial time-delay may be arbitrarily small which means that it suffices to rely on the time delays of the physical feedback lines and/or on those of the switching devices in the feedback loop.

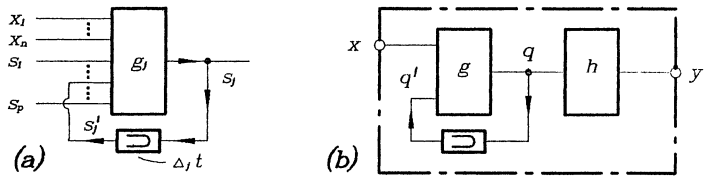


Fig. 31–3 The catenation model of a sequential circuit

Combining *catenation* and *stability* leads to the **catenation model of a sequential circuit** shown in Fig. 31–3. Note that in the binary case — where $q = (s_1, \ldots, s_p)$ — the feedback q of Fig. 31–3b is interpreted as in Fig. 31–3a for each binary state-variable s_j, and that due to catenation of g and h **no delays are needed in the leads from g_j to h.**

31.3 Moore and Mealy Flow Tables

The flow tables used in the previous two chapters are called **Moore flow tables** to distinguish them from a second type called *Mealy flow tables.* Let me first bring the concept of the Moore flow table into context with the models of the previous two sections.

A transition table, as the one in Fig. 31–1a, can be used to specify any of the transition functions g_{mr}, g_{ml}, g_{md}, g of the Moore, Mealy, Medvedev, or catenation models. Due to the restrictions imposed on the transition *function* g of the catenation model, see (31.4abc), its associated transition *table* cannot be used as freely as with the other models. An output table, Fig. 31–1b, can be used to specify the output function h_{mr} of the Moore model, just as well as the output function h of the catenation model.

Combining a transition table, as the one of Fig. 31–1a, and an output table, Fig. 31–1b, in a single table leads to a flow table as the one discussed in Fig. 29–1. In this text we use such a flow table in connection with the catenation model, but it can equally be used to specify a circuit according to the Moore model. For this reason, and to distinguish it from the next type of flow table to be discussed, we call a flow table that combines a transition and an output table a *Moore flow table.* Do note that a Moore flow table can be, but need not be a *primitive* flow table.

q \ x_1, x_2	0 0,0	1 0,1	3 1,1	2 1,0
a	e	e	(a)	d
b	e	e	(b)	d
c	(c)	(c)	a	(c)
d	c	c	a	(d)
e	(e)	(e)	b	d

(a) Transition table g

q	y
a	0
b	1
c	0
d	1
e	0

(b) Output table h

q \ x_1, x_2	0 0,0	1 0,1	3 1,1	2 1,0
a	0	0	0	1
b	0	0	1	1
c	0	0	0	0
d	0	0	0	1
e	0	0	1	1

(c) Primary WRT f

Fig. 31–4 On the way to developing a Mealy type flow table

Commonly, a **Mealy flow table** is a table in which any cell (q, x) contains the ordered pair (q^+, y), where $q^+ = g_{ml}(q, x)$ and $y = h_{ml}(q, x)$ according to the Mealy model of (31.2). The catenation model, (31.4), also lends itself to being described by a *Mealy type flow table* when defining the circuit by its transition function g and its composite function f. As the composite function f is specified by a primary WRT, the **Mealy type flow table** for the catenation model (31.4) is a table each cell (q, x) of which contains the ordered pair $(q^+, h(q^+))$ with $q^+ = g(q, x)$.

To illustrate, let us develop a Mealy type flow table for the sequential circuit specified by the transition table and output table of Fig. 31–4ab (usually, these tables, taken together, comprise the Moore flow table which defines the circuit). According to the catenation model the output values are determined by the primary WRT of Fig. 31–4c.

Then, the Mealy type flow table for our example, see Fig. 31–5c, is obtained by combining the transition table and primary WRT (both repeated in Fig. 31–5ab). As stated above, each cell (q, x) of the Mealy type flow table contains the ordered pair $(q^+, h(q^+))$ with $q^+ = g(q, x)$. For instance, cell $(e, 3)$ of Fig. 31–5c contains $(b, 1)$ because $b = g(e, 3)$, see Fig. 31–5a, and $1 = h(b)$.

q \ x_1, x_2	0 0,0	1 0,1	3 1,1	2 1,0
a	e	e	(a)	d
b	e	e	(b)	d
c	(c)	(c)	a	(c)
d	c	c	a	(d)
e	(e)	(e)	b	d

(a) Transition table g

q \ x_1, x_2	0 0,0	1 0,1	3 1,1	2 1,0
a	0	0	0	1
b	0	0	1	1
c	0	0	0	0
d	0	0	0	1
e	0	0	1	1

(b) Primary WRT f

q \ x_1, x_2	0 0,0	1 0,1	3 1,1	2 1,0
a	(e, 0)	(e, 0)	(a, 0)	(d, 1)
b	(e, 0)	(e, 0)	(b, 1)	(d, 1)
c	(c, 0)	(c, 0)	(a, 0)	(c, 0)
d	(c, 0)	(c, 0)	(a, 0)	(d, 1)
e	(e, 0)	(e, 0)	(b, 1)	(d, 1)

(c) Mealy-type flow table

Fig. 31–5 Developing a Mealy-type flow table

The second element, $h(q^+)$, of the ordered pair $(q^+, h(q^+))$ inscribed into a cell (q, x) of a Mealy type flow table of the catenation model cannot be chosen freely — it is always determined via $q^+ = g(q, x)$. In contrast, the output value $h_{ml}(q, x)$ of a conventional **Mealy flow table** representing a standard Mealy model can, in principle, be chosen freely. Thus, the **Mealy model** *does not have the same output behaviour as the* **catenation model.**

31.4 Merging Moore and Mealy Flow Tables

Frequently, a *Moore* flow table developed to describe the behaviour of a certain sequential circuit contains a number of *rows whose contents are identical.* Do note that ...

(31.5) Rows of a Moore flow table are **equivalent** if they have identical output values and column-wise identical GOTO-commands in the body of the rows.

As equivalent rows are indistinguishable in their behaviour, we ...

(31.6) Substitute *all* occurrences of the names of equivalent rows by a common **surrogate name** not yet used in the flow table.

Having chosen surrogate names where possible, you can now avoid ambiguous behaviour if you ...

(31.7) Delete duplicate rows of the flow table. This process is referred to as **merging.**

To see how the merging of rows is carried out, consider merging the primitive flow table of Fig. 31–6a. This flow table is a repetition of Fig. 29–4b, the only difference being that the GOTO-commands of the stable states are not encircled so as to better be able to see which rows are equivalent.

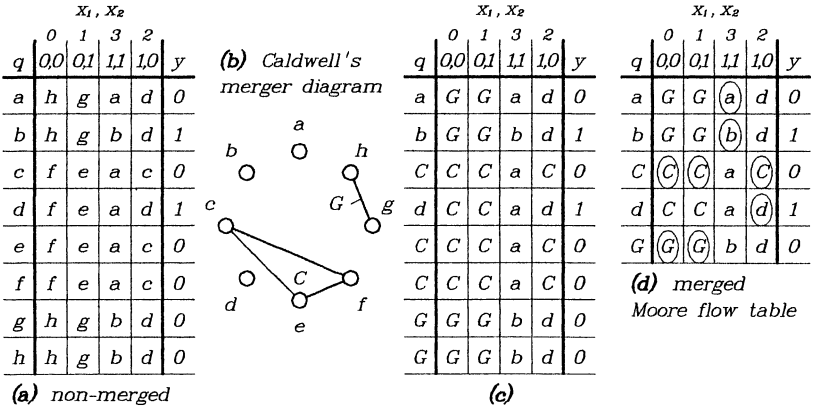


Fig. 31–6 On the merging of rows of a Moore flow table

To facilitate in the process of merging, you can use the **merger diagram** introduced by **Caldwell** [1958], and shown in Fig. 31–6b. Each row of the flow table is represented by a node (drawn as a small circle). Those nodes that represent equivalent rows are connected. Note that equivalence is a transitive relation, so that if the contents of rows c and e, and the contents of rows c and f are identical, the contents of rows e and f must also be identical. Transitivity does not hold in the presence of *don't cares*. Connections in the merger diagram are associated with surrogate names for connected rows. In our example, the line names g and h and all equally named GOTO-commands are replaced by G. Similarly, all occurrences of c, e, and f are replaced by C. This process leads to the flow table of Fig. 31–6c.

Finally, dropping duplicate rows in Fig. 31–6c leads to the **merged flow table** of Fig. 31–6d. For your convenience, the stable states are emphasized. You have encountered this flow table in Fig. 31–4a, but also in Fig. 30–1a where C and G of Fig. 31–6d have been replaced by c and e, respectively.

Merging can make the process of developing the cluster of word recognition tables from a given flow table simpler as there are less rows to consider. *The word-recognition trees obtained are the same as those obtained prior to merging.*

When **merging a Mealy type flow table** we use the following equivalence criterion for the rows of the table.

(31.8) Rows of a Mealy type flow table are **equivalent** if the contents $(q^+, h(q^+))$ of their cells are column-wise identical.

To illustrate, consider merging the Mealy type flow table of Fig. 31–5c repeated in Fig. 31–7a. By the above criterion, the rows b and e are equivalent. Using B as a surrogate name for b as well as for e leads to Fig. 31–7b. Then, dropping duplicate rows, according to (31.7) , we obtain the **merged Mealy type flow table** of Fig. 31–7c.

	x_1, x_2				
	0	1	3	2	
q	0,0	0,1	1,1	1,0	
a	e,0	e,0	a,0	d,1	
b	e,0	e,0	b,1	d,1	B
c	c,0	c,0	a,0	c,0	
d	c,0	c,0	a,0	d,1	
e	e,0	e,0	b,1	d,1	B

(a) Mealy type flow table

	x_1, x_2			
	0	1	3	2
q	0,0	0,1	1,1	1,0
a	B,0	B,0	a,0	d,1
B	B,0	B,0	B,1	d,1
c	c,0	c,0	a,0	c,0
d	c,0	c,0	a,0	d,1
B	B,0	B,0	B,1	d,1

(b) surrogate name B for all b and e

	x_1, x_2			
	0	1	3	2
q	0,0	0,1	1,1	1,0
a	B,0	B,0	a,0	d,1
B	B,0	B,0	B,1	d,1
c	c,0	c,0	a,0	c,0
d	c,0	c,0	a,0	d,1

(c) merged Mealy type flow table

Fig. 31–7 Merging a Mealy type flow table

While any row of a Moore flow table is assigned one and only one output value, a *merged* Mealy flow table, on the other hand, can have (but does not

necessarily have) at least one row in which some stable conditions cause the output to be 0, whereas others cause the output to be 1. An example for this behaviour is shown in row B of Fig. 31–7c: The cells $(B,0)$ and $(B,1)$ cause the output to be 0, but cell $(B,3)$ causes the output to be 1.

It should be pointed out that for certain circuits the fully merged Mealy flow table may have less rows than the fully merged Moore flow table. This difference can, under advantageous circumstances, lead to circuits that employ less internal latches when using Huffman's synthesis method discussed in later chapters. Let me further mention that, although most people prefer using the Moore flow table to specify a circuit, it is possible and sometimes more convenient to specify a circuit directly in a Mealy flow table.

31.5 From Mealy to Moore Flow-Tables

Transforming a Mealy type flow table (such as that of Fig. 31–8a) — in which each row contains one and the same output value in each of its stable conditions or cells — into a Moore flow table requires us to 'extract' all output values from the body of the Mealy type flow table and to write the output value that was associated with a stable condition of a row into that row of a separate output column. As Fig. 31–8b shows, this is the equivalent Moore flow table to the Mealy type flow table of Fig. 31–8a.

	x_1, x_2			
	0	1	3	2
q	0,0	0,1	1,1	1,0
a	e,0	e,0	a,0	d,1
b	e,0	e,0	b,1	d,1
c	c,0	c,0	a,0	c,0
d	c,0	c,0	a,0	d,1
e	e,0	e,0	b,1	d,1

(a) Mealy type flow table

	x_1, x_2				
	0	1	3	2	
q	0,0	0,1	1,1	1,0	y
a	e	e	(a)	d	0
b	e	e	(b)	d	1
c	(c)	(c)	a	(c)	0
d	c	c	a	(d)	1
e	(e)	(e)	b	d	0

(b) Moore flow table

Fig. 31–8 When simple transition from Mealy to Moore is possible

Now, consider transforming a Mealy type flow table (such as that of Fig. 31–7c) that has at least one row (in our case row B) in which some stable conditions cause the output to be 0 — cells $(B,0)$ and $(B,1)$ — while others cause the output to be 1 — cell $(B,3)$. First, each such row is duplicated — in our example, row B is duplicated as shown in Fig. 31–9a. Each of these rows are given surrogate names that do not yet occur as state names — the method of choice being to distinguish the identical rows by giving one of the row names the index 0, and the other the index 1.

	x_1, x_2			
	0	1	3	2
q	0,0	0,1	1,1	1,0
a	(B, 0)	(B, 0)	(a, 0)	(d, 1)
B	(B, 0)	(B, 0)	(B, 1)	(d, 1)
B	(B, 0)	(B, 0)	(B, 1)	(d, 1)
c	(c, 0)	(c, 0)	(a, 0)	(c, 0)
d	(c, 0)	(c, 0)	(a, 0)	(d, 1)

(a) duplicating row B

	x_1, x_2				
	0	1	3	2	
q	0,0	0,1	1,1	1,0	
a	$(B_0, 0)$	$(B_0, 0)$	(a, 0)	(d, 1)	
B_0	$(B_0, 0)$	$(B_0, 0)$	$(B_1, 1)$	(d, 1)	$= e$
B_1	$(B_0, 0)$	$(B_0, 0)$	$(B_1, 1)$	(d, 1)	$= b$
c	(c, 0)	(c, 0)	(a, 0)	(c, 0)	
d	(c, 0)	(c, 0)	(a, 0)	(d, 1)	

(b) renaming duplicate rows so that the stable conditions per row are assigned identical output values

Fig. 31–9 The general transition from Mealy to Moore flow table

In the row indexed 0, gather the stable conditions associated with the 0-output, while in the row indexed 1, gather the stable conditions associated with the 1-output. For our example this leads to the Mealy flow table of Fig. 31–9b, the row names e (for B_0) and b (for B_1) referring to the names used in Fig. 31–5c. Finally, the table obtained is transformed into a Moore flow table as discussed in the paragraph above.

31.6 The Composite Word-Recognition Table

Word-recognition tables of one and the same cluster need not be drawn in separate tables. If you look back at the word-recognition tables of Figs. 30–2 to 30–5, you will notice that they all are developed from one and the same transition table. Repeating the transition table every time you want to develop a new word-recognition table seems and of course is quite unnecessary. You can reduce the writing load if you use a **composite word-recognition tables** as the one shown in Fig. 31–10 its name reminding us that its constituent parts stem from the word-recognition tables. In fact, when using a computer to develop a cluster of word-recognition tables, the cluster is best organised in the form of a composite word-recognition table.

The composite WRT, Fig. 31–10b, consists of a table of output vectors followed by n **main columns** — one for each input event. Both the table of output vectors, just as each main column consist of **sub- or individual columns** — one for each WRT of the cluster. Equally positioned subcolumns k_j are associated with one and the same WRT. It is best to draw a surplus of individual columns because the number needed is not known at the outset (unless you are transferring the values of already developed WRTs).

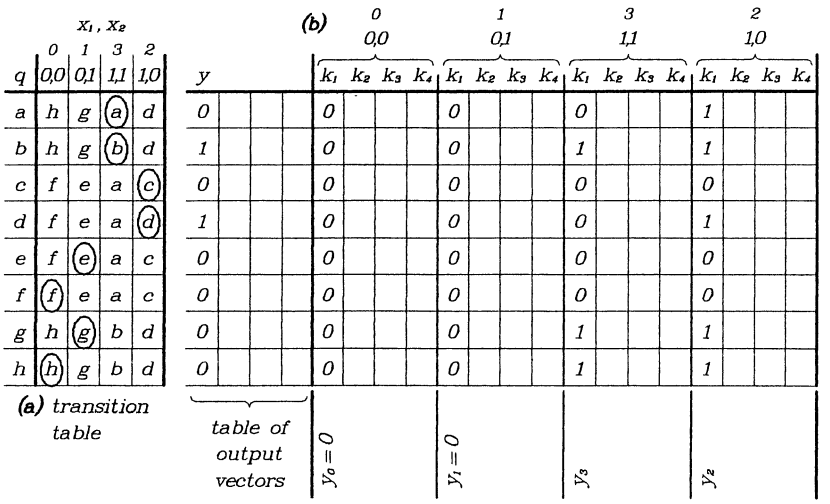


Fig. 31–10 The organisation of a Composite Word-Recognition Table

We discuss the composite WRT using the cluster of WRTs developed in Section 30.2. The transition table contained in the flow table FT-y of Fig. 30–2a is repeated in Fig. 31–10a. The output vector y of the flow table is transferred to the first column of the table of output vectors (Fig. 31–10b). The first sub-columns, all marked k_1, of the main columns of the composite WRT have the columns of the primary WRT, i.e. of WRT-y of Fig. 30–2b, inscribed. The names — y_0, y_1, y_3, and y_2 — used in the primary WRT are assigned to the sub-columns k_1 of the composite WRT.

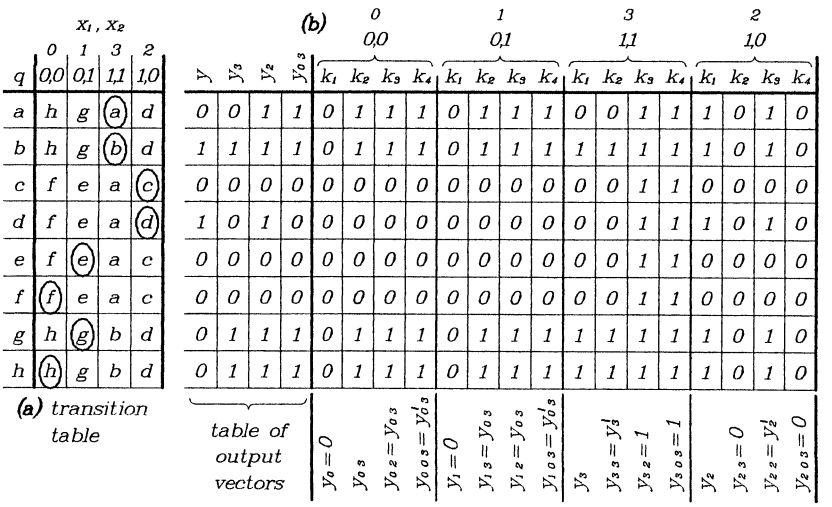


Fig. 31–11 Developing a Composite Word-Recognition Table

The next step is to copy a non-determinate output vector, in our example choosing y_3, from a sub-column k_1 to the table of output vectors, see Fig. 31–11b, and use this output vector to develop the entries in the sub-columns k_2. You continue in a like manner for each (new) non-determinate output vector (in our case for y_2 and $y_{0,3}$).

					0				1				3				2				
					0,0				0,1				1,1				1,0				
	y	y_3	y_2	$y_{0,3}$	k_1	k_2	k_3	k_4	k_1	k_2	k_3	k_4	k_1	k_2	k_3	k_4	k_1	k_2	k_3	k_4	
a	0	0	1	1	0	1	1	1	0	1	1	1	0	0	1	1	1	0	1	0	
b	1	1	1	1	0	1	1	1	0	1	1	1	1	1	1	1	1	0	1	0	
c	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	$=C$
d	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	0	
e	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	$=C$
f	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	$=C$
g	0	1	1	1	0	1	1	1	0	1	1	1	1	1	1	1	1	0	1	0	$=G$
h	0	1	1	1	0	1	1	1	0	1	1	1	1	1	1	1	1	0	1	0	$=G$

Fig. 31–12 A basic composite word-recognition table

The basic form of the composite WRT, Fig. 31–12, is a special **binary encoding** of the transition table of Fig. 31–11a. One characteristic aspect of this binary encoding is that its first element is the output vector y. This ensures that the composite WRT is a Moore and *not* a Medvedev model — as it still describes an *unique circuit* (and not a set of circuits that have a certain transition table in common).

				0				1				3				2			
				0,0				0,1				1,1				1,0			
y	y_3	y_2	$y_{0,3}$	k_1	k_2	k_3	k_4	k_1	k_2	k_3	k_4	k_1	k_2	k_3	k_4	k_1	k_2	k_3	k_4
0	0	1	1	0	1	1	1	0	1	1	1	0	0	1	1	1	0	1	0
1	1	1	1	0	1	1	1	0	1	1	1	1	1	1	1	1	0	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	0
0	1	1	1	0	1	1	1	0	1	1	1	1	1	1	1	1	0	1	0

Fig. 31–13 The non-merged composite word-recognition table

A further aspect of the composite WRT is that simply dropping duplicate rows leads to a merged (transition) table — there is no need to introduce surrogate names. This makes the computer development of a composite word-recognition table an efficient computer merging-procedure.

(a)

	y_3	y_2	$y_{0\,3}$	0 (0,0) k_2	k_3	k_4	1 (0,1) k_2	k_3	k_4	3 (1,1) k_2	k_3	k_4	2 (1,0) k_2	k_3	k_4	
a	0	1	1	1	1	1	1	1	1	0	1	1	0	1	0	
b	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	$=B$
c	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	
d	0	1	0	0	0	0	0	0	0	0	1	1	0	1	0	
e	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	$=B$

(b) WRT-y

y_3	y_2	$y_{0\,3}$	0 0,0	1 0,1	3 1,1	2 1,0
0	1	1	0	0	0	1
1	1	1	0	0	1	1
0	0	0	0	0	0	0
0	1	0	0	0	0	1
1	1	1	0	0	1	1

Fig. 31–14 The Mealy model of the circuit

Note that the merged composite word-recognition table of Fig. 31–13 contains the primary word-recognition table WRT-y as one of its constituent parts. Extracting this table provides us with the two tables of Fig. 31–14 which is a slightly modified way of writing a Mealy model.

(a)

	y_3	y_2	$y_{0\,3}$	0 (0,0) k_2	k_3	k_4	1 (0,1) k_2	k_3	k_4	3 (1,1) k_2	k_3	k_4	2 (1,0) k_2	k_3	k_4
a	0	1	1	1	1	1	1	1	1	0	1	1	0	1	0
b	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0
c	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0
d	0	1	0	0	0	0	0	0	0	0	1	1	0	1	0

(b) WRT-y

y_3	y_2	$y_{0\,3}$	0 0,0	1 0,1	3 1,1	2 1,0
0	1	1	0	0	(0)	1
1	1	1	(0)	(0)	(1)	1
0	0	0	(0)	(0)	0	(0)
0	1	0	0	0	0	(1)
			(y_0	y_1	y_3	y_2)

Fig. 31–15 The merged Mealy model of the circuit

Dropping duplicate rows of both tables of Fig. 31–14 leads to the merged form of the Mealy model pictured in Fig. 31–15. **Huffman** [1954] was the first to develop the binary encoding of transition and output tables, and their subsequent evaluation. We take up Huffman's approach in a later chapter.

Chapter 32

Toggle Circuits

Toggling is the basis for counting. A **toggling command** or **toggling input-event** is an input event the effect of which is to *invert and then memorise* the output value invoked by the previous and distinct input-event. A toggling input-event is said to **toggle the output**. Loosely spoken, I refer to a toggle circuit with two or more inputs as a circuit which (in a likeness to latches) has at least one setting, one resetting, one memorising, and one toggling input-event. In a K-map the respective cells will have the symbols $0, 1, q, \overline{q}$ inscribed (when using numeric variables), or $\mathbf{0}, \mathbf{1}, Q, \overline{Q}$ (when using logic variables). Especially the symbols $\overline{q}$ or $\overline{Q}$ have only symbolic and no algebraic meaning. The simplest toggle circuits, though, have only one input (and one output), and are called T-Flipflops.

32.1 T-Flipflop — its Feedback Design

While there is no latch with one input (you need at least one setting, one resetting, and one memorising input event), it is feasible to design a toggle flipflop which has only a single input. One input event (i.e., one input value) would be a memorising, the other a toggling input-event. A *symbolic* representation of the two possible flipflops is shown in Figs. 32–1ab. If the output of a flipflop is called, say, y then the memorising input-event in the reduced K-map is marked by a cell containing the symbol y, and the toggling input-event is marked by a cell containing the symbol $\overline{y}$. If the symbolic notation were taken at face value we would be able to read the logic formula $Y \Leftrightarrow X\overline{Y} \vee \overline{X}Y$ from the reduced K-map which, when X is true, leads to the contradiction $Y \Leftrightarrow \overline{Y}$. The actual meaning given the symbolic notation expressed in the K-maps of Figs. 32–1ab is expressed in the flow table of Fig. 32–1c while the diagram of successive events (DSE) of Fig. 32–1d is a good visualisation of the circuits' behaviour.
The complete synthesis procedure for the circuit specified in Fig. 32–1 is shown in Fig. 32–2. We start with the defining flow-table FT-y which is the flow table of Fig. 32–2a without the right-most column for y_1. This flow table is used to develop the cluster of word-recognition tables according to (30.3)

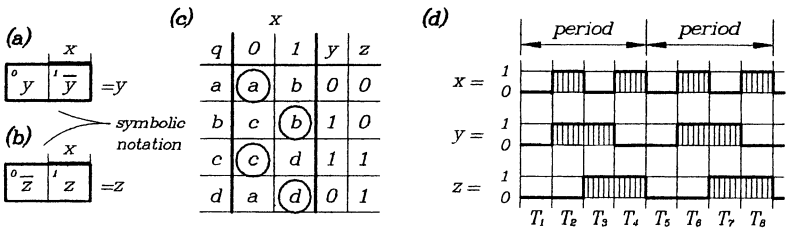


Fig. 32–1 Describing the behaviour of the two T-flipflops

of Section 30.1. The word-recognition table WRT-y of Fig. 32–2b is developed as explained in (30.1) when setting ℓ to 0. The only *non-determinate* output vector of WRT-y is y_1 its values therefore being appended to the flow table FT-y of Fig. 32–2a and used to develop the next word-recognition table WRT-y_1 of Fig. 32–2c. Both output vectors, $y_{0,1}$ and $y_{1,1}$, are determinate according to (30.4), $y_{0,1}$ being complementary to y, and $y_{1,1}$ being equal to y_1 allowing us to write $y_{1,1} \Leftrightarrow y_1'$ according to (29.5).

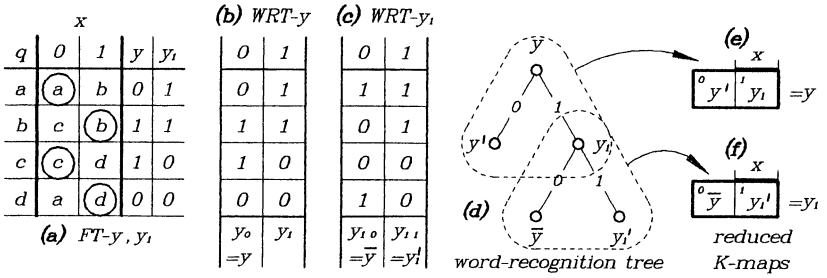


Fig. 32–2 The complete synthesis procedure for a T-flipflop

As explained in the last paragraph of Section 30.2 the word-recognition tables WRT-y and WRT-y_1 allow us to draw the word-recognition *tree* of Fig. 32–2d, each emphasized sub-tree corresponding to a reduced K-map of Figs. 32–2ef. Although it is not necessary, it is advantageous (especially for the novice) to redraw these reduced K-maps so that only the time-delayed output variables are retained as map-entered variables. This is done in Figs. 32–3 and 32–4 (you might want to refer to Chapter 20).

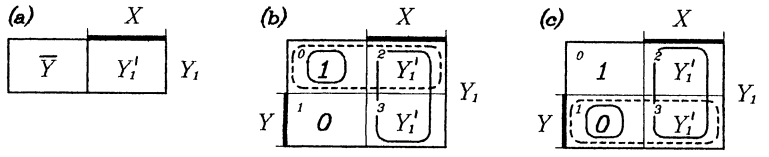


Fig. 32–3 Evaluating the reduced K-map specifying Y_1

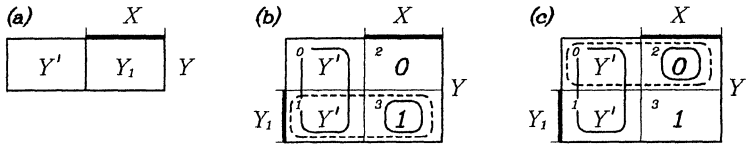


Fig. 32–4 Evaluating the reduced K-map specifying Y

The advantage of these representations is that they comply to those used when developing latch theory allowing us to use the theory without modifications. The dashed loops are responsible for a circuit being hazard-free. Omitting the dashed loops, and using (25.3), the disjunctive solutions are

$$Y_1 \Leftrightarrow \overline{X}\,\overline{Y} \vee XY_1', \qquad Y \Leftrightarrow XY_1 \vee \overline{X}Y', \tag{32.1}$$

while (25.4) allow us to calculate the conjunctive solutions

$$Y_1 \Leftrightarrow (X \vee \overline{Y}) \wedge (\overline{X} \vee Y_1'), \qquad Y \Leftrightarrow (\overline{X} \vee Y_1) \wedge (X \vee Y'). \tag{32.2}$$

The above formulas are pictured in Fig. 32–5.

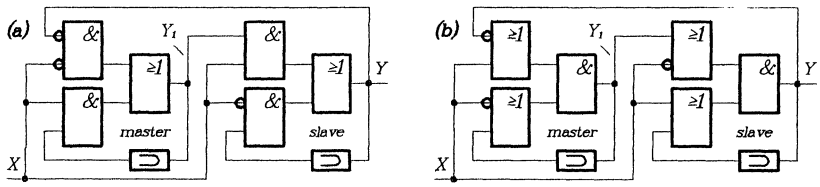


Fig. 32–5 Hazard-prone T-flipflops

The hazard-free solutions, on the other hand, incorporate the dashed loops of Figs. 32–3 and 32–4 which for the disjunctive forms lead to the formulas

$$Y_1 \Leftrightarrow \overline{X}\,\overline{Y} \vee (X \vee \overline{Y})Y_1', \qquad Y \Leftrightarrow XY_1 \vee (\overline{X} \vee Y_1)Y', \tag{32.3}$$

which are expressed in the circuit of Fig. 32–6.

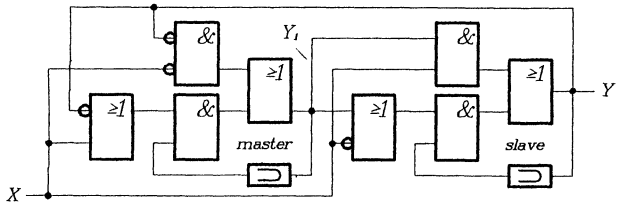


Fig. 32–6 A hazard-free T-flipflop complying with (32.3)

Just this once, let us take a look at the difference in behaviour between the hazard-prone circuit of Fig. 32–5a and its hazard-free version of Fig. 32–6.

These two circuits are drawn next to each other in Fig. 32–7 for the input signal specified in the diagram of successive events of Fig. 32–1d.

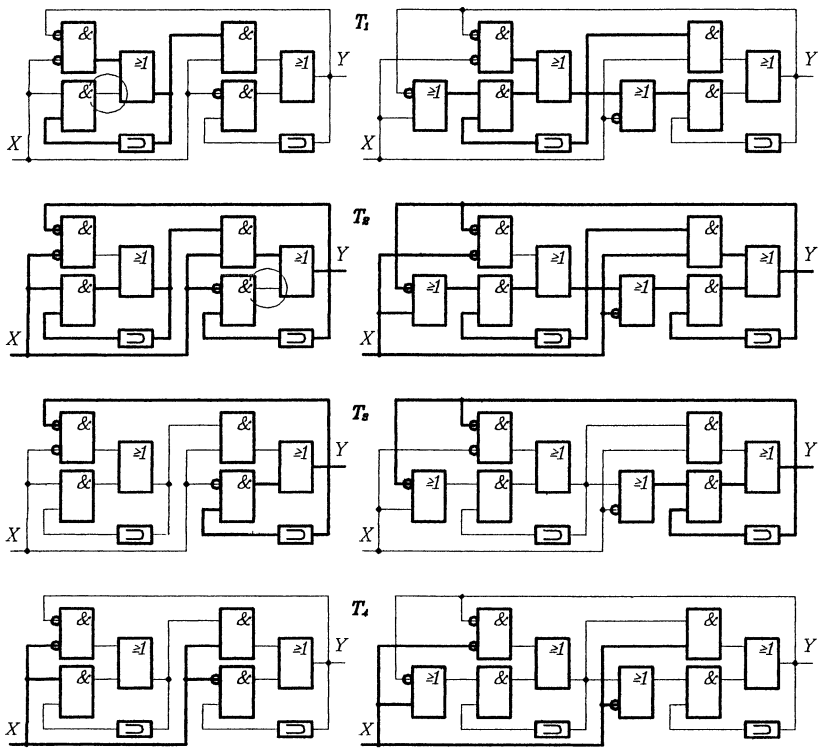


Fig. 32–7 Signal flow in a hazard-prone and hazard-free circuit

Please do take the time to follow and understand the propagation of the signals as shown in the figure (as before, the thick lines represent **1**-signals). Note that the hazard-prone latches in the left-hand circuit do not develop a 'closed' feedback loop when they are set. This is emphasized by the two circles where the signals are interrupted.

32.2 T-Flipflop — its Composition

The value of the method for composing latches (Chapter 28) lies in its application to the composition of internal latches of sequential circuits. If we want to develop the T-flipflop using D-latches as hinge latches, we quickly realise that this is possible without any composition theory as explicated in Fig. 32–8.

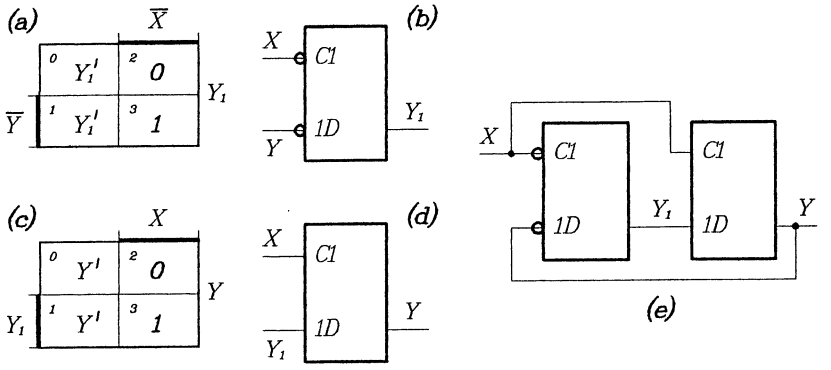


Fig. 32–8 Developing a T-flipflop in the basis of the D-latch

The K-maps of Fig. 32–8 are those of Fig. 32–4 with a slight alteration for the K-map representing Y_1 (Fig. 32–8a) which has been drawn for inverted input variables $\overline{X}$ and $\overline{Y}$. A comparison with Fig. 27–12 shows that the K-maps of Fig. 32–8 represent D-latches (Fig. 32–8b,d). Connecting equally labelled inputs and outputs of the D-latches leads to the circuit of Fig. 32–8e.

Before considering the following composition of a T-flipflop, you might want to reread Chapter 28 on the Composition of Latches. To see how to proceed with our example, first take a look at the result, i.e., at the circuit shown in Fig. 32–9. For each internal latch, you can choose a hinge latch on the basis of which the internal latch may be composed. In our example the cross-coupled NOR-latch was chosen for both internal latches. We choose to label the inputs of the hinge latch for Y_1 as S_1 and R_1, and those of Y as S and R. The logic expressions for these input variables — S_1, R_1, and S, R — are what we want to determine by the composition method. The process by which to do this is explicated in Fig. 32–9.

In the following, the composition procedure is explained for the internal latch Y_1. For this latch you first choose a hinge latch, in our case the cross-coupled NOR-latch, and draw its K-map (Fig. 32–9a). As we decide to prohibit the input event S_1, R_1 — so that the outputs of the cross-coupled NORs are inverted — we can read the hinge sets $\mathcal{Y}_0 = \{(\mathbf{0},\mathbf{1})\}$, $\mathcal{Y}_1 = \{(\mathbf{1},\mathbf{0})\}$, $\mathcal{Y}_q = \{(\mathbf{0},\mathbf{0})\}$ from the K-map of Fig. 32–9a. Note that the first value of an ordered pair of a hinge set stands for S_1, the second value for R_1. The next step, shown in Fig. 32–9b, is to draw the reduced K-map obtained from the synthesis procedure for Y_1 (Fig. 32–2f). To apply the composition method of Chapter 28 the reduced K-map must be redrawn to a standard form for latches in which only the time-delayed output variables are retained as map-entered variables (Fig. 32–9c). To calculate the pre-logic, each resetting cell (one containing a **0**) is replaced by an element of $\mathcal{Y}_0$ — in our case by $(\mathbf{0},\mathbf{1})$. In a like manner, the **1**-cell of Fig. 32–9c is replaced by $(\mathbf{1},\mathbf{0})$, while the Q-cells are replaced by $(\mathbf{0},\mathbf{0})$ (Fig. 32–9d). This K-map contains the values

for the input variables S_1 and R_1 which are read separately from the K-map as $S_1 \Leftrightarrow \overline{X}\,\overline{Y}$ and $R_1 \Leftrightarrow \overline{X}Y$.

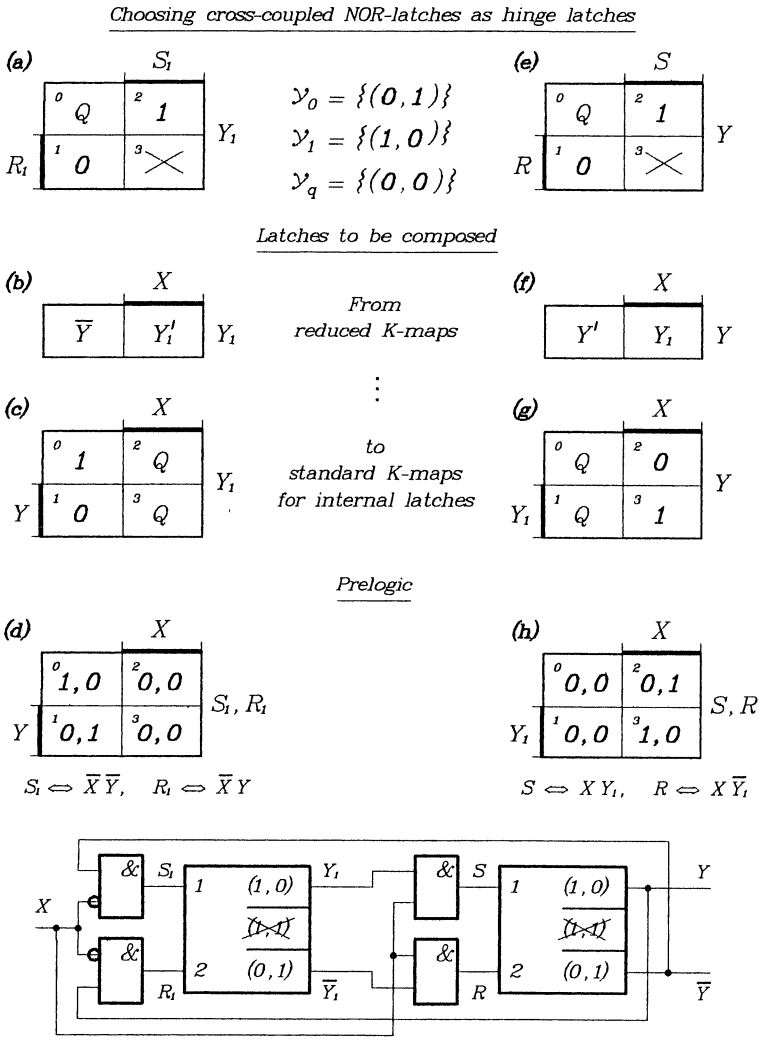


Fig. 32–9 Composing a T-flipflop

The process described above is carried out for each internal latch. In our example the only other internal latch is Y, its input being referred to as S and R. Following the K-maps on the right side of Fig. 32–9 we finally obtain the prelogoc $S \Leftrightarrow XY_1$ and $R \Leftrightarrow X\overline{Y}_1$. As in the previous example, connecting equally labelled inputs and outputs of the pre-logic and the latches leads to the circuit of Fig. 32–9.

32.3 JK-Flipflop — its Feedback Design

A toggle circuit for two inputs can symbolically be specified by a K-map for two inputs, each of its four cells having one of the symbols $0, 1, q, \overline{q}$ inscribed (when using numeric variables), or $\mathbf{1}, \mathbf{0}, Q, \overline{Q}$ (in the case of logic variables). One of the possible permutations of these symbols over the cells of the K-map has been singled out by convention and called the JK-flipflop (Fig. 32–10).

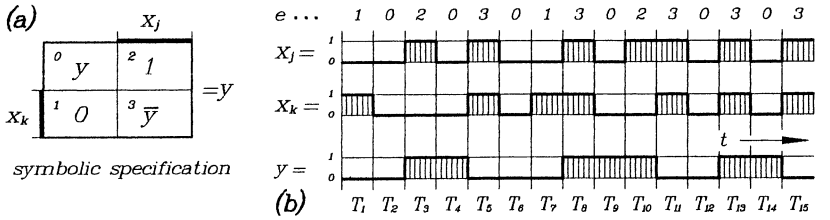


Fig. 32–10 Visualising the behaviour of the JK-flipflop

To a certain degree its behaviour resembles that of a SR-latch whose setting input we call $J \Leftrightarrow x_j = 1$, and whose resetting input is called $K \Leftrightarrow x_k = 1$. That is, $J\overline{K}$ is the setting input-event, $\overline{J}K$ is the resetting input-event, and $\overline{J}\,\overline{K}$ is the memorising input event. The effect the input event JK has is to invert the output value caused by one of the previous *but distinct* input events $J\overline{K}$, $\overline{J}K$, or $\overline{J}\,\overline{K}$, and then to memorise and output this inverted value — as before, this is expressed *symbolically* by entering the inverted output variable y in cell JK.

For the chosen sequence of input events e used in the diagram of successive events (DSE), Fig. 32–10b, the output values for y were read from the symbolic representation of the JK-flipflop of Fig. 32–10a. Please do take the time to duplicate this process of obtaining the DSE.

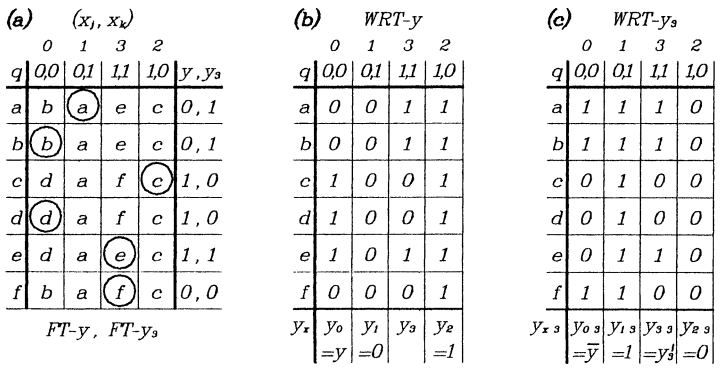


Fig. 32–11 Primitive flow table and derived word-recognition tables

To obtain a circuit for the JK-flipflop we need to develop its primitive flow table, see Fig. 32–11a. Note that its output originally only consists of the Y-column. Please do take the time to develop this primitive flow table on the basis of the information contained in Fig. 32–10. The output column Y of the primitive flow table FT-y, Fig. 32–11a, leads to the word-recognition table WRT-y of Fig. 32–11b. The only non-determinate output variable is Y_3 (please make sure you could argue this) so that the Y_3-column of Fig. 32–11b is taken as next output column of the primitive flow table as shown in Fig. 32–11a. Using this column with the body of the primitive flow table, together they comprise the child flow-table FT-y_3, we obtain the word-recognition table WRT-y_3 of Fig. 32–11c. All output variables $Y_{3,x}$ of this table are determinate. As there are no more non-determinate output variables to account for, the process of developing word-recognition tables terminates.

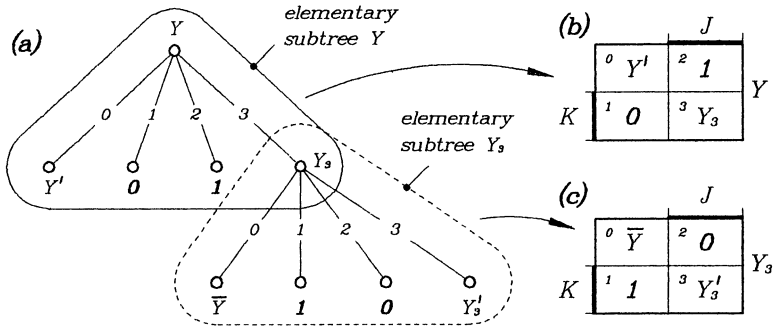


Fig. 32–12 The word-recognition tree and its associated K-maps

The word-recognition table WRT-y of Fig. 32–11b can be expressed as the elementary sub-tree Y of Fig. 32–12a. The non-determinate output variable Y_3 is the root of the elementary sub-tree Y_3 obtained from the word-recognition table WRT-y_3 of Fig. 32–11c. The elementary sub-trees can be expressed as reduced K-maps, as shown in Fig. 32–12bc. The correspondence between the logic variables J and K, and the numeric variables x_j and x_k is $J \Leftrightarrow x_j = 1$ and $K \Leftrightarrow x_k = 1$.

To evaluate reduced K-maps that specify latches it is best to expand the K-maps to their standard form, i.e., extracting all map-entered variables except the delayed output variable. For our example, this leads to the right-hand K-maps of Fig. 32–13.

The standard K-maps of Fig. 32–13bd may be evaluated using the *disjunctive evaluation formulas* (25.3) for which we determine the *inclusions* $U_1(\mathbf{x})$ and $U_{q;1}(\mathbf{x})$,

$$Y_3 \Leftrightarrow \overline{J}(K \vee \overline{Y}) \vee KY_3', \tag{32.4a}$$

$$Y \Leftrightarrow J(\overline{K} \vee Y_3) \vee \overline{K}Y'. \tag{32.4b}$$

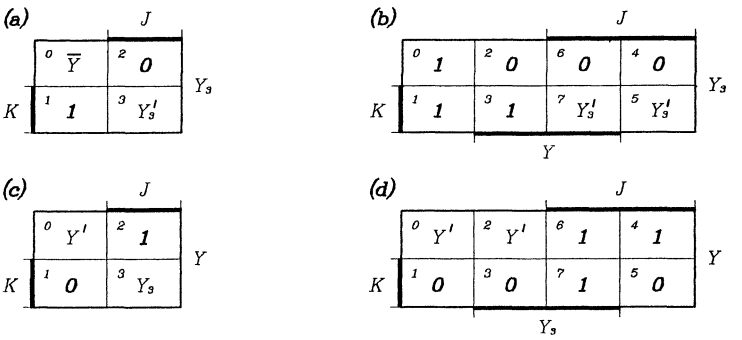


Fig. 32–13 Transition to the standard K-maps of a latch

Alternatively, the K-maps of Fig. 32–13bd may be evaluated using the *conjunctive evaluation formulas* (25.4) for which we determine the *exclusions* $V_0(\mathbf{x})$ and $V_{q;0}(\mathbf{x})$,

$$Y_3 \Leftrightarrow (K \vee \overline{J}\,\overline{K})(\overline{J} \vee Y_3'), \tag{32.5a}$$

$$Y \Leftrightarrow (\overline{K} \vee JY_3)(J \vee Y'). \tag{32.5b}$$

The *disjunctive formulas* (32.4) are expressed graphically in Fig. 32–14 which represents a hazard-free version to a feedback circuit for the JK-flipflop. You are invited to draw the circuit for (32.5).

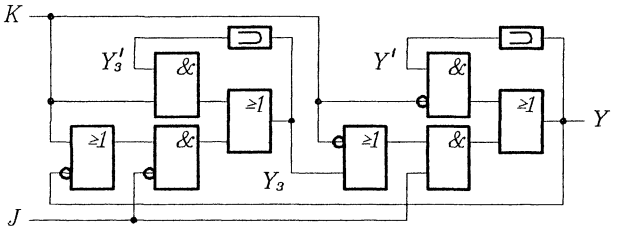


Fig. 32–14 A feedback circuit for the JK-flipflop

32.4 JK-Flipflop — its Composition

If you want to use latches of your choice to realise the internal latches of the JK-flipflop, you must first determine the internal latches. You can either do this by using Huffman's method (discussed in later chapters), or you can follow the procedure of the previous section until you have obtained the standard K-maps for the internal latches shown in Fig. 32–13bd. Then

choosing, say, the D-latch as hinge latches (Fig. 32–15ad), you read from their K-map the hinge sets $\mathcal{Y}_0 = \{(\mathbf{1},\mathbf{0})\}$, $\mathcal{Y}_1 = \{(\mathbf{1},\mathbf{1})\}$, $\mathcal{Y}_q = \{(\mathbf{0},\mathbf{0}),(\mathbf{0},\mathbf{1})\}$. All **0**s in the cells of the K-maps of Y_3 and Y (Fig. 32–15be) are replaced by the (only) element, $(\mathbf{1},\mathbf{0})$, of $\mathcal{Y}_0$. Similarly, all **1**s in the cells of the K-maps of Y_3 and Y are replaced by the $(\mathbf{1},\mathbf{1}) \in \mathcal{Y}_1$. Any of tha symbols Q in the K-maps of Fig. 32–15be can be replaced by any element, either $(\mathbf{0},\mathbf{0})$ or $(\mathbf{0},\mathbf{1})$, of $\mathcal{Y}_q$. As the K-maps of Fig. 32–15cf show, my (arbitrary) choice fell on $(\mathbf{0},\mathbf{1})$. Evaluating these K-maps leads, via the formulas written below them, to the circuit of Fig. 32–15.

If you don't mind running through some simple algebra, we can easily verify the circuit obtained. Let us do this for the K-maps on the right side of Fig. 32–15 used to calculate the output latch Y.

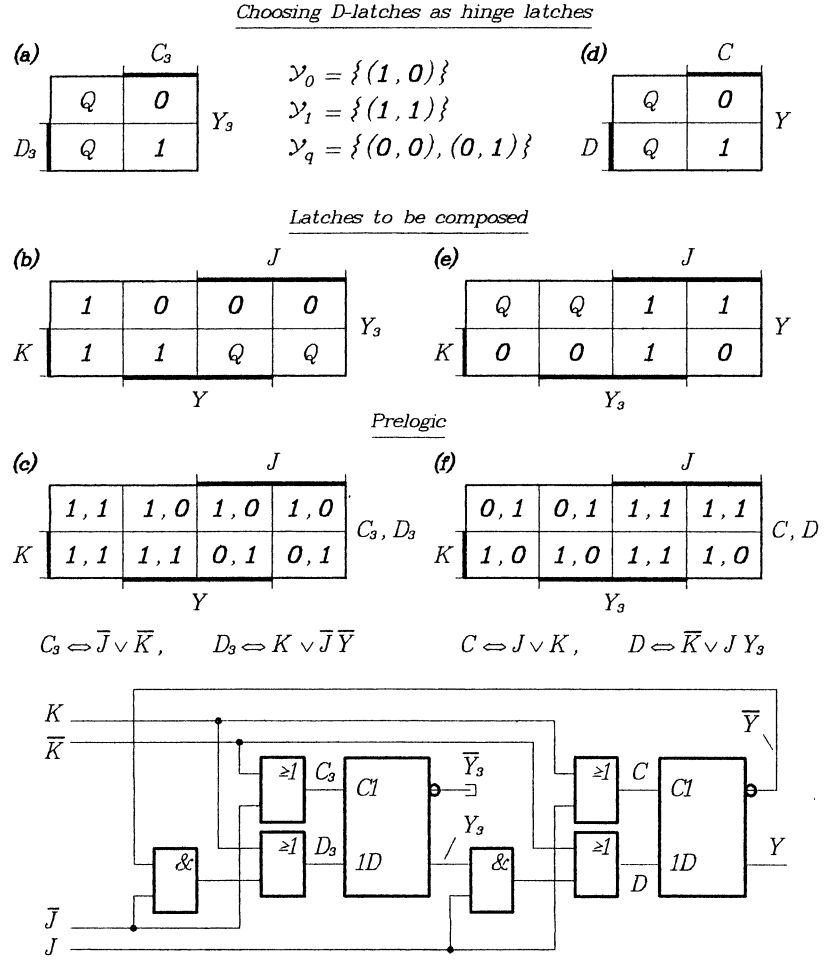


Fig. 32–15 The JK-flipflop on the basis of D-latches

$$
\begin{aligned}
Y &\Leftrightarrow CD \vee \overline{C}Y' \vee DY' \qquad \text{read from Fig. 32–15d} \\
&\Leftrightarrow CD \vee (\overline{C} \vee D)Y' \\
&\qquad C \Leftrightarrow J \vee K, \quad \overline{C} \Leftrightarrow \overline{J}\,\overline{K} \\
&\qquad D \Leftrightarrow \overline{K} \vee JY_3 \\
&\Leftrightarrow (J \vee K)(\overline{K} \vee JY_3) \vee (\underbrace{\overline{J}\,\overline{K} \vee \overline{K}}_{\overline{K}} \vee JY_3)Y' \\
&\Leftrightarrow J\overline{K} \vee \underbrace{K\overline{K}}_{0} \vee JY_3 \vee JKY_3 \vee (\overline{K} \vee JY_3)Y' \\
&\Leftrightarrow J(\overline{K} \vee \underbrace{Y_3 \vee KY_3}_{Y_3}) \vee (\overline{K} \vee JY_3)Y' \\
Y &\Leftrightarrow J(\overline{K} \vee Y_3) \vee (\overline{K} \vee JY_3)Y'.
\end{aligned}
$$

This formula, although not identical with (32.4b), is of course a valid description of the latch specified in Fig. 32–13d (or Fig. 32–15e) as discussed in Section 26.3 (immediately following Fig. 26–3). A like calculation to the one carried out above validates the solution for Y_3.

A word is in place on the use of the initial formula $Y \Leftrightarrow CD \vee \overline{C}Y' \vee DY'$ used above, and which implies a specific design of a feedback latch. The D-latches employed need not be designed as a feedback circuit. But, as discussed in Section 24.4, the *'advantage of the feedback model'* is that it allows us to study the stationary behaviour of a latch without having to know its physical design.

Chapter 33

Triggering and Synchronising

Neither *triggering* nor *synchronisation* are formal concepts. To develop these concepts, we shall try to express the desired behaviour in diagrams of successive events (DSE) these showing intuitively and vividly what we wish to achieve. Then, to design circuits that have the desired behaviour we shall consider a **method of developing a flow table directly from the diagram of successive events**. It is this transition from a DSE to a flow table I wish to draw your attention to — not so much to the circuits developed.

33.1 Triggering

A **triggering circuit** has one input and one output. The shape of the output signal is always a *pulse*. If the pulse is initiated by the input signal rising from 0 to 1 (and only then) it is referred to as a **positive trigger**, see output y of Fig. 33–1a. If, on the other hand, the pulse is initiated by the input signal falling from 1 to 0 (and only then) it is referred to as a **negative trigger**, see output z of Fig. 33–1a. Specifying the desired duration Δt of the pulse is problematic. Frequently, a trigger is used to *set* or *reset* a latch. With such a usage in mind, we would want the pulse to last as long as needed, but no longer than necessary.

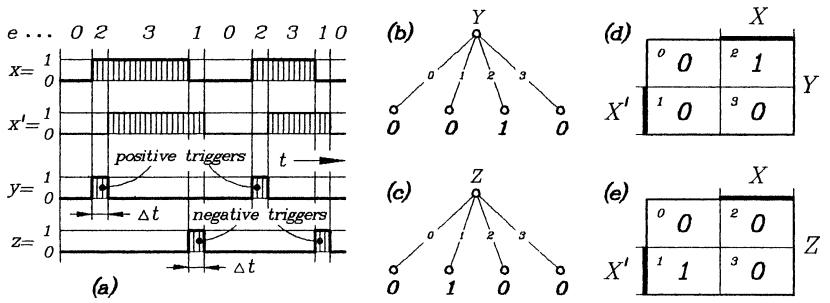


Fig. 33–1 Specifying and designing a trigger circuit

The simplest way to produce a trigger is to employ the input X together with its delayed version X' as shown in Fig. 33–1a. To check whether the

circuit for the positive trigger y is a combinational or a sequential circuit, it can be advantageous to start with a word-recognition tree. To transfer the output y of the DSE of Fig. 33–1a you first draw a word-recognition tree (Fig. 33–1b without the output values). To find the output value to assign to y_0, you turn to the DSE and check the output values caused by the input event 0. These all being 0 (i.e., $y_0 = 0$), you substitute y_0 by 0 in the word-recognition tree. You proceed in a like manner for the other input events. As each occurrence of a given input event in the DSE is always assigned the same value, the DSE describes a combinational circuit.

Transferring the word-recognition tree of Fig. 33–1b to the K-map of Fig. 33–1d allows us to calculate the formula $Y \Leftrightarrow X \wedge \overline{X'}$, which leads to the circuit of Fig. 33–2a as a possible realisation of a trigger circuit. The IEC symbol for a circuit the output of which is a positive trigger is shown in Fig. 33–2b. Inverting the input provides us with a circuit for a negative trigger.

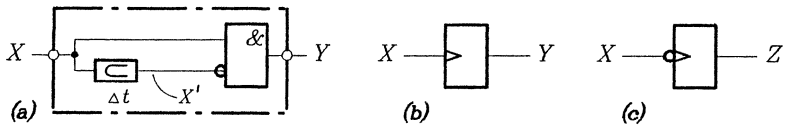


Fig. 33–2 Possible realisation and symbol for a trigger circuit

A disadvantage of the above realisation, Fig. 33–2a, is that the duration of the emitted pulse is *constant*. As it is **not adaptive**, it will always have to last longer than may be necessary. An intuitive attempt to develop an **adaptive circuit** could employ a feedback as shown in Fig. 33–3a, but, one immediately realises that this circuit *oscillates* and thus is useless. Nevertheless, the oscillating circuit can lead to a new idea as expressed in the DSE of Fig. 33–3b: We shall try to design a circuit in which the output fed back maintains its inhibiting effect as long as the circuit's independent input, X, remains 1.

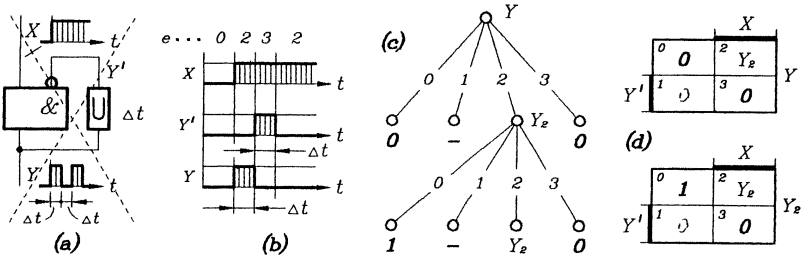


Fig. 33–3 Specifying an adaptive trigger circuit

The DSE of Fig. 33–3b is next used to develop a tree representation of the circuit, Fig. 33–3c. First, drawing only the sub-tree with the output

Y as its root, we read from the DSE that $Y_0 \Leftrightarrow \mathbf{0}$ and $Y_3 \Leftrightarrow \mathbf{0}$, while Y_1 (not occurring) can be chosen freely, and Y_2 is *non-determinate*. To further analyse Y_2, we draw a sub-tree with Y_2 as its root, again Fig. 33–3c. Node $(0,2)$ is assigned the output value $Y_{0,2}$ which is caused by those occurrences of the input event 2 that are preceded by the input event 0[1]. The first occurrence of the input event 2 in the DSE is preceded by the input event 0 enabling us to read the value for $Y_{0,2}$ as $Y_{0,2} \Leftrightarrow \mathbf{1}$. The second occurrence of the input event 2 is preceded by the input 3, allowing us to determine $Y_{3,2} \Leftrightarrow \mathbf{0}$. $Y_{1,2}$ doesn't occur and can be chosen, while $Y_{2,2}$ is chosen as Y_2 instead of Y_2' to reduce the possibility of oscillations.

The K-maps of Fig. 33–3d, developed from the word-recognition tree of Fig. 33–3c, enable us to calculate Y and Y_2 as

$$Y_2 \Leftrightarrow \overline{Y'}\,(\overline{X} \vee Y_2), \tag{33.1a}$$

$$Y \Leftrightarrow X\overline{Y'}\,Y_2 \overset{(33.1a)}{\Longleftrightarrow} X\underbrace{\overline{Y'}\,\overline{Y'}}_{\overline{Y'}}(\overline{X} \vee Y_2)$$

$$\Leftrightarrow X\underbrace{\overline{Y'}\,(\overline{X} \vee Y_2)}_{Y_2} \Leftrightarrow XY_2. \tag{33.1b}$$

The circuit equivalent to the above formulas (33.1) is shown in Fig. 33–4a. Its output Y would oscillate if X and the output of the IMPLICATION were simultaneously **1**. To ensure against oscillation the time delay Δt must be (noteably) greater than any time delay within the internal memory.

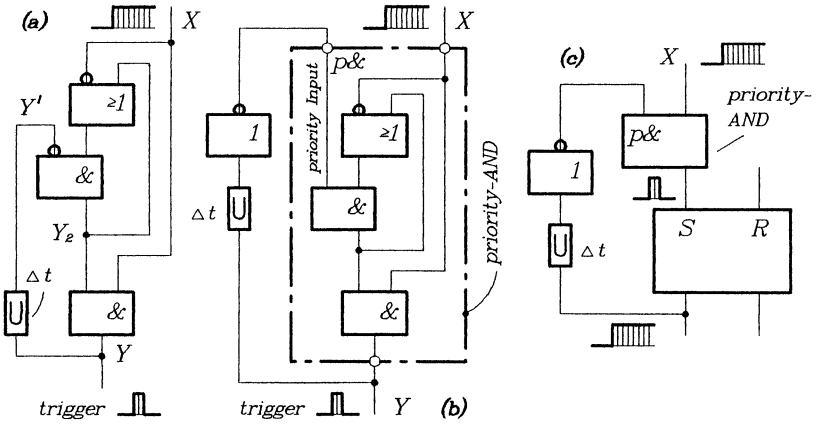


Fig. 33–4 An adaptive trigger circuit

Please follow me in trying to develop this circuit into an adaptive circuit. Extracting the NEGATION from the INHIBITION of the internal latch of

[1] You might like to reread Section 29.3 and especially the definitions of the filtered output variables $y_{w(\ell)}$, i.e., of y, y_x, $y_{x^{(1)},x^{(2)}}$, etc.

Fig. 33–4a, allows us to draw this circuit as shown in Fig. 33–4b. The framed circuit is the circuit we called a *priority-AND* in Fig. 1–8, and discussed quite extensively in Section 1.7. Let us now intuitively employ the circuit of Fig. 33–4b as shown in Fig. 33–4c to trigger the set-input of a SR-latch. Then the trigger's duration is adaptive in the following sense. Although the pulse will have a minimal duration of Δt (what I would *not* call adaptive), the pulse terminates Δt seconds after the latch is set, i.e., the duration of the pulse adapts to the time necessary for the latch to set.

33.2 Sampling

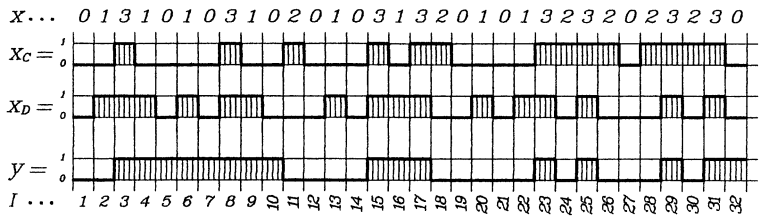
The **sampling circuit** has two inputs and one output. Its prime input, the **data input** x_D, is **sampled**, i.e., its value is passed on to the output, y, during usually brief time intervals defined by the second input x_C, the **clock** or **control signal**, being 1; between sampling, i.e., while x_C is 0, the output y retains the last sampled value. This behaviour is demonstrated in the diagram of successive events (DSE) of Fig. 33–5a.

The sampling circuit is a latch as you can readyly see when describing the sampling circuit's behaviour as follows: Y is *set* by $X_C X_D$, and *reset* by $X_C \overline{X}_D$, while $\overline{X}_C$ retains the output value unchanged. This is the verbal description of the **D-Latch** of Fig. 27–12. But, let us see if we can show more systematically that the sampling circuit is a D-latch.

As it is not possible to develop a circuit systematically directly from a DSE, we first need to develop a flow table from the DSE. This we *can* do systematically although not algorithmically. Let us use the example of Fig. 33–5 to see how a flow table can be obtained from a given DSE. We first determine a **lower bound for the number of stable states** in each column of the flow table: **(a)** If, for a given input event x in the DSE, all occurrences of $y(x)$ are constant (all are either 0, or all are either 1) then we enter (at least as a first assumption) a **single stable state** in column x, and assign to this stable state the output value $y(x)$. **(b)** On the other hand, if at least one occurrence of a given input event x in a DSE invokes a response $y(x)$ of 0 while another invokes an output $y(x)$ of 1, then column x of the flow table must contain (at least) **two stable states**, one, to which the output value 0 is assigned, the other to which the output value 1 is assigned.

The flow table we wish to develop must have two stable states in column 0, for, y_0 is 0 in interval $I = 1$, while y_0 is 1 in interval $I = 5$. Likewise, column 1 must contain two stable states: y_1 is 0 during interval $I = 2$, and y_1 is 1 at $I = 4$. All occurrences of the input event 3 cause the output, y_3, to be 1, so that column 3 has only a single stable state, and to this we assign the output value 1. Likewise, column 2 has only a single stable state.

In the present state of development, all cells, except the ones with encircled entries, are empty. For each of these cells we have to decide on a GOTO



(a) Specifying the behaviour of the sampling circuit in a DSE

	(x_C, x_D)				
	0	1	3	2	
q	(0,0)	(0,1)	(1,1)	(1,0)	$y(q)$
a	(a)	$(y_0, y_1) = (0,0)$ (1,2), (12,13), (19,20), (21,22) c	$(y_0, y_3) = (0,1)$ (14,15) e	$(y_0, y_2) = (0,0)$ (27,28) f	0
b	(b)	$(y_0, y_1) = (1,1)$ (5,6) d	$(y_0, y_3) = (1,1)$ (7,8) e	$(y_0, y_2) = (1,0)$ (10,11) f	1
c	$(y_1, y_0) = (0,0)$ (13,14), (20;21) a	(c)	$(y_1, y_3) = (0,1)$ (2,3), (22,23) e	—	0
d	$(y_1, y_0) = (1,1)$ (4,5), (6,7), (9,10) b	(d)	$(y_1, y_3) = (1,1)$ (16,17) e	—	1
e	$(y_3, y_0) = (1,1)$ (31,32) b	$(y_3, y_1) = (1,1)$ (3,4), (8,9), (15,16) d	(e)	$(y_3, y_2) = (1,0)$ (17,18), (23,24), (25,26), (29,30) f	1
f	$(y_2, y_0) = (0,0)$ (11,12), (18,19), (26,27) a	—	$(y_2, y_3) = (0,1)$ (24,25), (28,29), (30,31) e	(f)	0

(b) Developing the flow table from the DSE

Fig. 33–5 DSE and flow table of the sampling circuit

command, and this decision must be based on information contained in the DSE. I shall only discuss the GOTO commands for column 1, i.e., for cells $(a,1)$, $(b,1)$, $(e,1)$, $(f,1)$, and would encourage you to ponder the GOTO commands for the remaining cells on the basis of the information entered into them.

Cell $(e,1)$**:** All transitions in the DSE from an input event 3 to an input event 1 — they occur at the intervals (3,4), (8,9), and (15,16) — are

associated with output values $(y_3, y_1) = (1,1)$. In the flow table, the output transition $(y_3, y_1) = (1,1)$ is associated with the transition from stable e to stable d; to achieve this transition we must enter a GOTO-d command in cell $(e,1)$.

Cells $(a,1)$ and $(b,1)$: The transitions in the DSE from an input event 0 to an input event 1 do not produce uniform output transitions (y_0, y_1). During the intervals $(1,2)$, $(12,13)$, $(19,20)$, $(21,22)$ the associated output transition is $(y_0, y_1) = (0,0)$, but in the intervals $(5,6)$ the output transition is $(y_0, y_1) = (1,1)$. The output transition $(y_0, y_1) = (0,0)$ — in which $y_0 = 0$ — is associated with cell $(a,1)$ necessitating a GOTO-c command in cell $(a,1)$ — as opposed to a GOTO-d command — to ensure an output value of 0 for y_1. On the other hand, the output transitions $(y_0, y_1) = (1,1)$ in which $y_0 = 1$ necessitate us to enter the GOTO command into cell $(b,1)$, while $y_1 = 1$ requires the GOTO command to be GOTO-d (as opposed to GOTO-c).

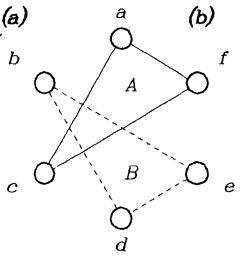
Cell $(f,1)$: This cell describes the reaction of the sampling circuit when the input event 2 changes to input event 1. No intervals $(I, I+1)$ in the DSE depict the input transition $(2,1)$. Therefore the DSE in no way defines the output transition (y_2, y_1). The DSE thus gives no hint on what GOTO command (either GOTO-c or GOTO-d) to enter into cell $(f,1)$ — it is this we express when entering a dash into $(f,1)$.

If the DSE were the exclusive specification of the circuit, we would have two options open to us to enter a GOTO command into $(f,1)$. Firstly, you can choose one of the possible GOTO commands, GOTO-c or GOTO-d, *arbitrarily*. Secondly, and this is the more rational and common thing to do, you choose the GOTO command so as to obtain a minimal number of rows when merging.

In our example, we are in the lucky situation to have a parallel specification. In the word specification (given at the beginning of the Section) we state that *'between sampling, i.e., while x_C is 0, the output y retains the last sampled value'*. If the last sampled value was 0 in the stable f, it must remain 0 when x_C turns 0 which requires us to enter GOTO-c into cell $(f,1)$ as indicated in Fig. 33–6a.

	0	1	3	2	
q	(0,0)	(0,1)	(1,1)	(1,0)	y
a	a	c	e	f	0
b	b	d	e	f	1
c	a	c	e	f	0
d	b	d	e	f	1
e	b	d	e	f	1
f	a	c	e	f	0

(a)



(c)

	0	1	3	2	
q	(0,0)	(0,1)	(1,1)	(1,0)	y
A	(A)	(A)	B	(A)	0
B	(B)	(B)	(B)	A	1

Fig. 33–6 Merging the flow table of the sampling circuit

Before continuing with the text it might be a good idea to consider the remaining cells of the flow table not discussed above. The final result, in which all GOTO commands are entered, is pictured in Fig. 33–6a. This flow table is merged to obtain the flow table of Fig. 33–6c.

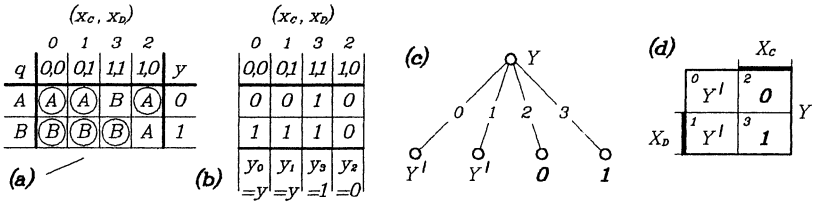


Fig. 33–7 Determining the K-map of the sampling circuit

The merged flow table is used to obtain the word-recognition table, Fig. 33–7b, and from it the word-recognition tree, Fig. 33–7c, and the reduced K-map, Fig. 33–7d. As the reduced K-map of the sampling circuit is identical with the reduced K-map of the D-latch, the sampling circuit is a D-latch.

33.3 Synchronising

Computer circuits are commonly designed as so-called **synchronous sequential circuits**, these being circuits whose input signals are only allowed to change their values the moment a clock or control signal switches *on*. An external input signal is usually referred to as being **asynchronous**, i.e., it can change its value independently of any control signal. External input signals are fed into synchronous sequential circuits via synchronising circuits.

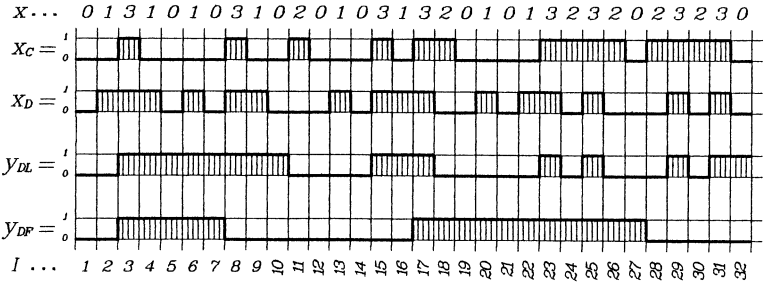
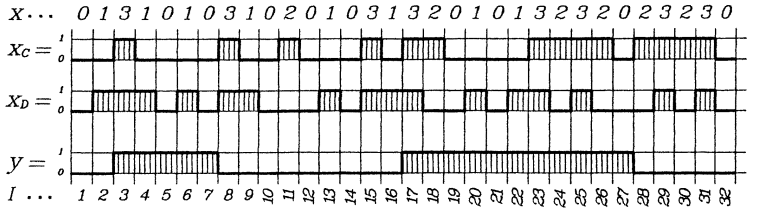


Fig. 33–8 Comparing sampling, y_{DL}, with synchronising, y_{DF}

A **synchronising circuit** has two inputs, the (external) data signal x_D and the clock or control signal x_C, and a single output y_{DF}[1]. Do note the

[1] The index DF of the output y_{DF} refers to the name usually given the synchronising circuit — D-flipflop.

(a)



(b)

q	0 (0, 0)	1 (0, 1)	3 (1, 1)	2 (1, 0)	y(q)
		(x_C, x_D)			
a	(a)	$(y_0, y_1) = (0, 0)$ (1, 2), (12, 13), c	$(y_0, y_3) = (0, 1)$ (14, 15) e	$(y_0, y_2) = (0, 0)$ (10, 11) g	0
b	(b)	$(y_0, y_1) = (1, 1)$ (5, 6) (19, 20), (21, 22) d	$(y_0, y_3) = (1, 1)$ (7, 8) e	$(y_0, y_2) = (1, 0)$ (27, 28) g	1
c	$(y_1, y_0) = (0, 0)$ (9, 10), (13 14) a	(c)	$(y_1, y_3) = (0, 1)$ (2, 3), (16, 17) f	—	0
d	$(y_1, y_0) = (1, 1)$ (4, 5), (6, 7), (20, 21) b	(d)	$(y_1, y_3) = (1, 1)$ (22, 23) f	—	1
e	$(y_3, y_0) = (0, 0)$ (31, 32) a	$(y_3, y_1) = (0, 0)$ (8, 9), (15, 16) c	(e)	$(y_3, y_2) = (0, 0)$ (29, 30) g	0
f	—	$(y_3, y_1) = (1, 1)$ (3, 4) d	(f)	$(y_3, y_2) = (1, 1)$ (17, 18), (23, 24) (25, 26) h	1
g	$(y_2, y_0) = (0, 0)$ (11, 12) a	—	$(y_2, y_3) = (0, 0)$ (28, 29), (30, 31) e	(g)	0
h	$(y_2, y_0) = (1, 1)$ (18, 19), (26, 27) b	—	$(y_2, y_3) = (1, 1)$ (24, 25) f	(h)	1

Fig. 33–9 Deriving the flow table, (b), for the synchronising circuit from its DSE, (a)

difference between the output y_{DL}[1)] of the *sampling circuit*, and the output y_{DF} of a *synchronising circuit*, Fig. 33–8.

The **behaviour of the synchronised output** y_{DF} can be explained as follows: The value of the output y_{DF} equals the value x_D had *prior* to when x_C became or last became 1. Preferably, you might like to use the following equivalent definition: The value of the output y_{DF} equals the last value x_D had the last time x_C was 0. Please do take the time to check that, and how, both definitions explain the output behaviour of y_{DF} shown in Fig. 33–8.

In all the remaining figures of this section I refer to the output of the synchronising circuit as y (instead of y_{DF}) so as to avoid double indices such as y_{DF_3} or $y_{DF_{3,2}}$. Following the same procedure as in the previous section, you should be able to develop the flow table of Fig. 33–9b from the DSE of Fig. 33–9a. As before, it is good practice to first determine a lower bound for the number of stable states — it is easy to see that we need at least two stable states per column. As with the sampling circuit, the GOTO command of every specifiable cell of the flow table (Fig. 33–9b) can be specified by a direct transition from one input event to another. This is certainly not always the case, as the introductory example in Section 29.2 demonstrates.

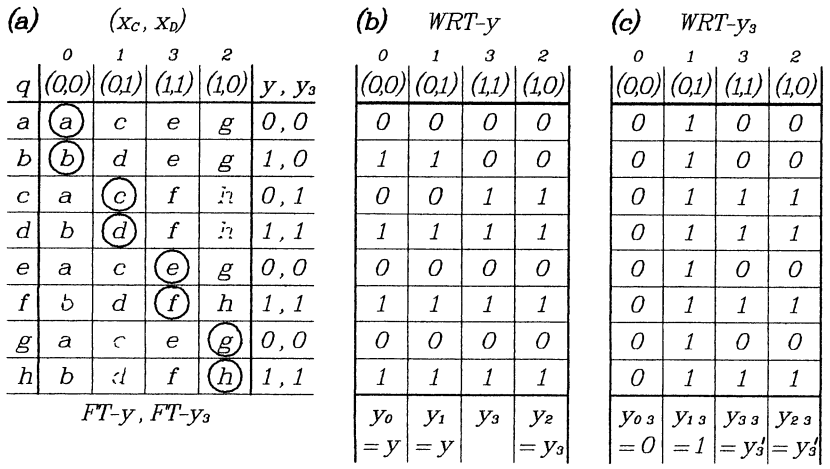


Fig. 33–10 Developing the word-recognition tables

The cells $(c,2)$, $(d,2)$, $(g,1)$, $(h,1)$, and $(f,0)$ cannot be specified from the DSE. In these cases we can and must fall back on the verbal specification. For example, when switching from stable c or stable d in column 1, do we enter GOTO-g or GOTO-h in either of the cells $(c,2)$ and $(d,2)$? In fact, for both transitions, as the output must be 1, we must enter GOTO-h: When inputs (x_C, x_D) switch (simultaneously) from the prior values $(0,1)$ to the

[1)] The index DL used here for the output y_{DL} of the sampling circuit refers to the name usually given the sampling circuit — D-latch.

present values $(1,0)$, the present output y must be 1, this being the prior value of x_D (the value x_D had the last time x_C was 0).

The complete flow table is shown in Fig. 33–10a from which we develop the word-recognition tables of Fig. 33–10bc.

To calculate the synchronising circuit, we first draw the word-recognition tree of Fig. 33–11a, this tree already showing the basic structure of the circuit.

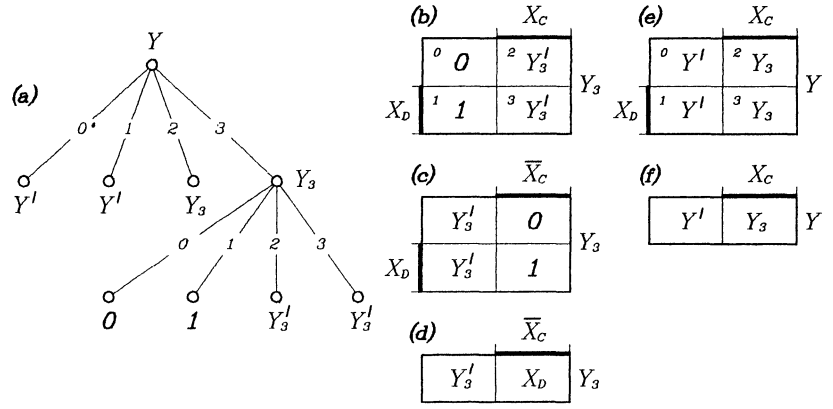


Fig. 33–11 Calculating the synchronising circuit

Sub-tree Y_3 leads to the K-map for Y_3, Fig. 33–11b. This is the K-map of a D-latch the control input of which is inverted as Fig. 33–11c allows us to deduce. In this example it proves practical to further reduce the dimension the K-map of Fig. 33–11c by using X_D as a map-entered variable which leads to the K-map of Fig. 33–11d. Now, consider the K-map, Fig. 33–11e, associated with the sub-tree for Y: Clearly, X_D has no influence on Y so that the K-map of Fig. 33–11e can be drawn as shown in Fig. 33–11f. Comparing this K-map with the K-map of Fig. 33–11d, it becomes clear that the K-map of Fig. 33–11f represents a D-latch *the data input of which* is Y_3 — and this is also the output of the D-latch specified by the K-map of Fig. 33–11d.

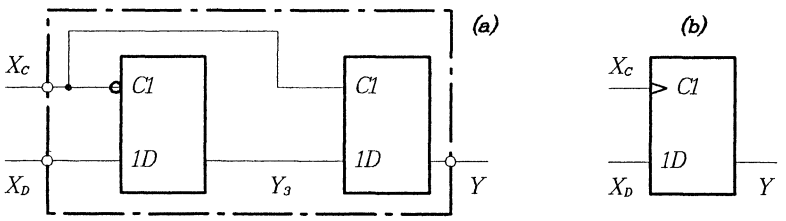


Fig. 33–12 The synchronising circuit is a D-flipflop

Thus, the K-maps of Fig. 33–11d and Fig. 33–11f lead to the circuit of Fig. 33–12a. This circuit is called a **D-flipflop**. Its IEC 117-15 symbol is shown in Fig. 33–12b.

Chapter 34

Verifying a Logic Design *

How can, and why should we verify a logic design of a sequential circuit? First to the *'why'* question. Even if we have developed a sequential circuit by some formal means (such as those set forth in Chapters 29 and 30), we could have erred. But, more importantly, a faultlessly developed sequential circuit can come up with unpleasant surprises when realised with real-world switching-devices so that verifying actual technical solutions is prudent, even necessary. Answering the *'how'* question is the subject of this Chapter. The aim is to design a **verification-DSE**: Given a flow table (FT) that defines the logic behaviour of a sequential circuit, we want to develop a minimal-length diagram of sequential events (DSE) that consists of all sub-sequences of input events (with their implied outputs) necessary and sufficient to design the flow table. I know of no algorithmic solution to the problem, but the following examples (each developed in its own Section) should be able to illustrate the general procedure.

34.1 A Verification-DSE for the D-Latch

Latches being the simplest sequential circuits, we start our study with one of the 28 — the D-latch. Assume we have developed its circuit, shown in Fig. 34–1c, as a realisation of its specification repeated in Fig. 34–1a. Then, to test or *verify* the circuit, we want to develop a sequence of successive input events together with their corresponding output values (i.e., a diagram of successive events, a DSE) so that when applying these input events to the circuit, we can observe whether the circuit's output values are identical with the ones required by the DSE. Of course you will want the DSE to be no longer than necessary, but to be long enough to contain all the information needed to develop the flow table of the D-latch. Only if this latter point is ensured can we be certain that all the sub-sequences of input events necessary to define the circuit will have been tested with the circuit. If you are satisfied with testing the logic of the circuit (instead of its actual hardware realisation) you can have this done by a computer using a logic simulation program. One of the possible verification-DSEs you can use is shown in Fig. 34–1d — it is a far cry from the highly redundant DSE used in Fig. 33–5a.

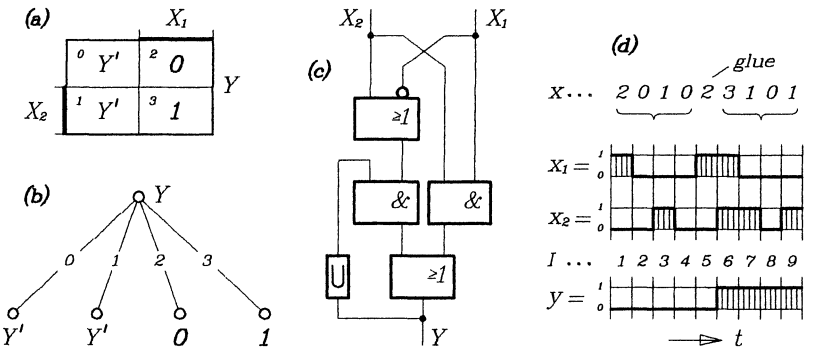


Fig. 34–1 The D-latch (c) and a possible verification-DSE (d)

This section is devoted to showing in detail how the verification-DSE of Fig. 34–1d was developed. The **first step** is to develop the primary WRT (Fig. 34–2b) from the flow table (Fig. 34–2a) originally used to specify and design the circuit (you can read up on this in Section 30.1). This step is identical with the corresponding step of the synthesis process.

(a) (x_1, x_2)

	0	1	3	2	
s	(0,0)	(0,1)	(1,1)	(1,0)	y
a	(a)	c	e	f	0
b	(b)	d	e	f	1
c	a	(c)	e	f	0
d	b	(d)	e	f	1
e	b	d	(e)	f	1
f	a	c	e	(f)	0

FT-y

(b) WRT-y

0	1	3	2
(0,0)	(0,1)	(1,1)	(1,0)
0	0	1	0
1	1	1	0
0	0	1	0
1	1	1	0
1	1	1	0
0	0	1	0
y_0	y_1	y_3 =1	y_2 =0

Fig. 34–2 Developing the primary WRT

In the **second step** we develop all word-recognition tables, WRT-y_0 and WRT-y_1 of Fig. 34–3bc, for those variables, y_0 and y_1, of the primary word-recognition table WRT-y that are non-determinate. But do note an important difference to those word-recognition tables used for synthesis purposes. **When verifying**, we want to make sure that **successive input-events are adjacent**. In the last paragraph of Section 29.2 I argued that, *for synthesis purposes*, specifying the output values for non-adjacent input-events was a necessity however unlikely the occurrence of such input events was. Our present consideration, on the other hand, is *for analysis*, and here *we are responsible* for presenting the circuit with a sequence of input events. As non-adjacent input-events are hard to impossible to produce reliably, it is only prudent to

avoid them. In fact, we shall agree that **no verification sequence may contain non-adjacent input-events**, i.e., two or more input variables may not change their values simultaneously. Furthermore, **when verifying, it is not possible to have an input event be followed by an identical one** — this situation would simply refer to an input event remaining constant. By the above restrictions the outputs $y_{0,0}$, $y_{3,0}$, $y_{1,1}$, and $y_{2,1}$ are not considered when verifying which we document by entering **dashes** into the appropriate columns (see Figs. 34–3bc).

(a) (x_j, x_k)

	0	1	3	2		
q	0,0	0,1	1,1	1,0	$y_{1\,0}$	$y_{0\,1}$
a	(a)	c	e	f	0	0
b	(b)	d	e	f	1	1
c	a	(c)	e	f	0	0
d	b	(d)	e	f	1	1
e	b	d	(e)	f	1	1
f	a	c	e	(f)	0	0

FT-y_0, FT-y_1

(b) WRT-y_0

0	1	3	2
0,0	0,1	1,1	1,0
—	0	—	0
—	1	—	0
—	0	—	0
—	1	—	0
—	1	—	0
—	0	—	0
$(y_{0\,0})$	$y_{1\,0}$	$(y_{3\,0})$	$y_{2\,0}$ =0

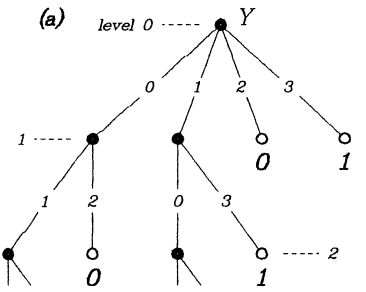
(c) WRT-y_1

0	1	3	2
0,0	0,1	1,1	1,0
0	—	1	—
1	—	1	—
0	—	1	—
1	—	1	—
1	—	1	—
0	—	1	—
$y_{0\,1}$	$(y_{1\,1})$	$y_{3\,1}$ =1	$(y_{2\,1})$

Fig. 34–3 Developing all secondary word-recognition tables

In **step three** we start developing the actual verification procedure by beginning to draw what I choose to call the *verification tree* (Fig. 34–4a) and the *verification table* (Fig. 34–4b). The **verification tree** of Fig. 34–4a is drawn according to the primary word-recognition table WRT-y of Fig. 34–2b and the word-recognition tables WRT-y_0 and WRT-y_1 of Fig. 34–3bc. The verification tree differs from the word-recognition tree in that successive edges of the verification tree always represent adjacent input events.

The organisation of the **verification table** of Fig. 34–4b is identical to that of the original **primitive** flow table that specifies the circuit, in our example with FT-y of Fig. 34–2a. The differences in detail are, firstly, that cells which can only be reached if multiple simultaneous changes of values of input variables are allowed contain a **dash** to indicate that such transitions will not be considered. Secondly, the cells are drawn large enough for us to enter input words that determine the leaves (or end-nodes) of the verification tree. Without much ado let me mention that each unstable condition in column 2 of Fig. 34–4b — i.e., each cell $(a, 2)$, $(b, 2)$, and $(e, 2)$ — represents end-node (2) of Fig. 34–4a, so that we enter the single input event (2) into these cells. Similarly, in all cells of column 3 we enter (3). The end-node $(2, 0)$ of Fig. 34–4a is entered into cell $(f, 0)$ because this is the only cell which describes the transition from the input event 2 to the input event 0. Likewise, the end-node $(3, 1)$ of Fig. 34–4a is entered into cell $(e, 1)$, and only into this cell.



(b)

	(x_1, x_2)				
	0	1	3	2	
q	(0,0)	(0,1)	(1,1)	(1,0)	$y(q)$
a	(a)	c	—	(2) f	0
b	(b)	d	—	(2) f	1
c	a	(c)	(3) e	—	0
d	b	(d)	(3) e	—	1
e	—	(3 1) d	(e)	(2) f	1
f	(2 0) a	—	(3) e	(f)	0

Fig. 34–4 Beginning the development of the verification tree (a) and the verification flow table (b)

In **step four** we continue the development of all word-recognition tables of the next level, see Fig. 34–5.

(a) (x_j, x_k)

	0	1	3	2		
q	0,0	0,1	1,1	1,0	y_{10}	y_{01}
a	(a)	c	e	f	0	0
b	(b)	d	e	f	1	1
c	a	(c)	e	f	0	0
d	b	(d)	e	f	1	1
e	b	d	(e)	f	1	1
f	a	c	e	(f)	0	0

$FT\text{-}y_{10}$, $FT\text{-}y_{01}$

(b) $WRT\text{-}y_{10}$

0	1	3	2
0,0	0,1	1,1	1,0
0	—	1	—
1	—	1	—
0	—	1	—
1	—	1	—
1	—	1	—
0	—	1	—
y_{010}	—	$y_{310} = 1$	—

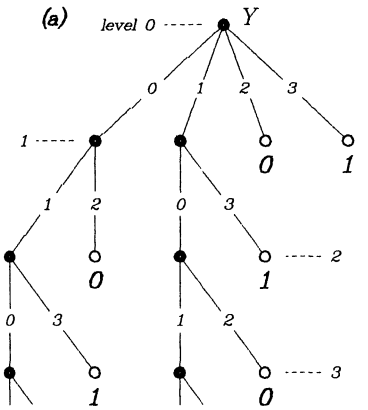
(c) $WRT\text{-}y_{01}$

0	1	3	2
0,0	0,1	1,1	1,0
—	0	—	0
—	1	—	0
—	0	—	0
—	1	—	0
—	1	—	0
—	0	—	0
—	y_{101}	—	$y_{201} = 0$

Fig. 34–5 Developing all ternary word-recognition tables

The word-recognition tables of Fig. 34–5bc enable us, in the next or **fifth step**, to extend the word-recognition tree of Fig. 34–4a to that of Fig. 34–6a. Both new leaves of this tree, $(3,1,0)$ and $(2,0,1)$, are *double* transitions. As in Section 29.2 (when describing Fig. 29–3b), we use brackets, $[\ldots]$, to mark a direct transition within a multiple transition. Because each cell of a (verification) flow table can only represent a *direct* transition, we need to split the double transition $(3,1,0)$ into two successive direct transitions,

$([3,1],0)$ and $(3,[1,0])$. In the flow table of Fig. 34–6b the direct transition $([3,1],0)$ is represented by cell $(e,1)$, while the successor transition $(3,[1,0])$ is represented by $(d,0)$. Similarly, the double transition $(2,0,1)$ stands for the successive direct transitions $([2,0],1)$ and $(2,[0,1])$, these being represented by the cells $(f,0)$ and $(a,1)$.



(b)

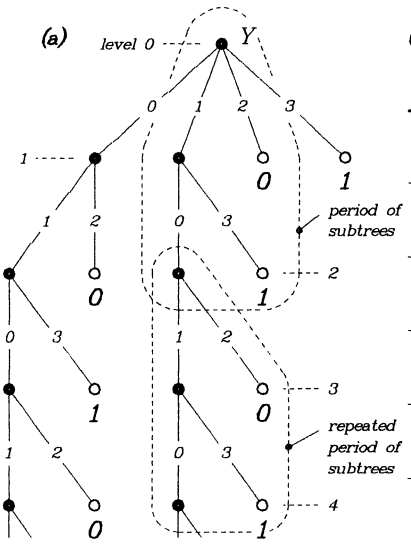
	(X_1, X_2) 0 (0,0)	1 (0,1)	3 (1,1)	2 (1,0)	y(q)
q					
a	(a)	(2 [0 1]) c	—	(2) f	0
b	(b)	d	—	(2) f	1
c	a	(c)	(3) e	—	0
d	(3 [1 0]) b	(d)	(3) e	—	1
e	—	(3 1) ([3 1] 0) d	(e)	(2) f	1
f	(2 0) ([2 0] 1) a	—	(3) e	(f)	0

Fig. 34–6 Further development of the verification tree (a) and the verification flow table (b)

The question arises, when to terminate the further development of the verification tree? **Further development of the verification tree is terminated as soon as the leaves of the tree account for all cells in the verification table**. In Fig. 34–6b the cells $(b,1)$ and $(c,0)$ are not accounted for making it necessary to develop yet another level of the verification tree, see Fig. 34–7a.

The next word-recognition tables you need to develop would be named WRT-$y_{0,1,0}$ and WRT-$y_{1,0,1}$. They are not shown here because the bodies of the tables are identical with those of Fig. 34–3 (but I would encourage you to verify this by developing these tables yourself). The word-recognition tables WRT-$y_{0,1,0}$ and WRT-$y_{1,0,1}$ allow us to add a further level to the verification tree of Fig. 34–6a thus obtaining the verification tree of Fig. 34–7a.

The triple transition of the new leaf $(2,0,1,0)$ of the verification tree of Fig. 34–7a incorporates the direct transitions $([2,0],1,0)$, $(2,[0,1],0)$, and $(2,0,[1,0])$ these being represented by the cells $(f,0)$, $(a,1)$, and the cell $(c,0)$ previously unaccounted for. Similarly, direct transition $(3,1,[0,1])$ of the new leaf $(3,1,0,1)$ represents the cell $(b,1)$ previously unaccounted for. We can now terminate the development of the verification tree at level 4 having accounted for all cells of the verification flow-table.



(b)

	0	1	3	2	
q	(0,0)	(0,1)	(1,1)	(1,0)	$y(q)$
a	(a)	(2 [0 1]) (2 [0 1] 0) c	—	(2) f	0
b	(b)	(3 1 [0 1]) d	—	(2) f	1
c	(2 0 [1 0]) a	(c)	(3) e	—	0
d	(3 [1 0]) (3 [1 0] 1) b	(d)	(3) e	—	1
e	—	(3 1) ([3 1] 0) ([3 1] 0 1) d	(e)	(2) f	1
f	(2 0) ([2 0] 1) ([2 0] 1 0) a	—	(3) e	(f)	0

Fig. 34–7 The final step in developing the verification tree (a) and the verification flow table (b)

The **last step** is to **develop the verification DSE** from the final verification table, or verification tree. For the present example this is atypically simple. As the two quadruple-input sequences $(2,0,1,0)$ and $(3,1,0,1)$, together with their sub-sequences, account for all cells of the flow table, it suffices to consider these quadruple sequences together with the output values they and their sub-sequences produce. The output value caused each time a new input event of a multiple-input sequence is entered can best be read from the (verification) flow table (although with a bit of practice the verification tree allows you to obtain the same information). Allow me, this once, to expound. Interpreting Fig. 34–7b, the input sequence $(2,0,1,0)$ corresponds to the sequence (f,a,c,a) of stable states, and these to the output sequence $(0,0,0,0)$. Now let us read the same information from the verification tree Fig. 34–7a. The multiple-input sequence $(2,0,1,0)$ requires us to first look at the single-event sequence (2), then the dual-event sequence $(2,0)$, then the triple-event sequence $(2,0,1)$, and finally the quadruple-event sequence $(2,0,1,0)$ providing us with the same sequence of output values as the verification table, i.e., with $(0,0,0,0)$. The same reasoning shows that the input sequence $(3,1,0,1)$ causes an output sequence of $(1,1,1,1)$.

The input sequence of the verification DSE must, in principle, consist of the catenation of the input sequences $(2,0,1,0)$ and $(3,1,0,1)$, i.e., of either of the sequences

$$(2,0,1,0),(3,1,0,1) = (2,0,1,0,3,1,0,1),$$

or

$$(3,1,0,1),(2,0,1,0) = (3,1,0,1,2,0,1,0).$$

But, in the first case, 0 and 3 are not adjacent input events, while in the second case 1 and 2 are not adjacent. Thus, we augment the catenated sequences of input and output events to be

$$\begin{pmatrix} 2,0,1,0,2,3,1,0,1 \\ 0,0,0,0,0,1,1,1,1 \end{pmatrix}, \quad \text{and} \quad \begin{pmatrix} 3,1,0,1,3,2,0,1,0 \\ 1,1,1,1,1,0,0,0,0 \end{pmatrix},$$

writing the output sequence below the input sequence. In the first case the inserted input event 2 is adjacent to the input events 0 and 3. In the second case the input event 3 is adjacent to the input events 1 and 2. The first of these cases is pictured in Fig. 34–1d.

Allow two last comments without going into details. **(a)** With practice, you can develop the verification tree of Fig. 34–7a directly from the word-recognition tree of Fig. 34–1b without referring to the development of the various word-recognition tables. **(b)** The branches of the verification tree can be **'pruned'** independently of a verification table (see Fig. 34–7a) as soon as a **'period consisting of sub-trees'** has been repeated — note that a period is established as a period by its repetition. In effect, this pruning criterion makes the verification table superfluous.

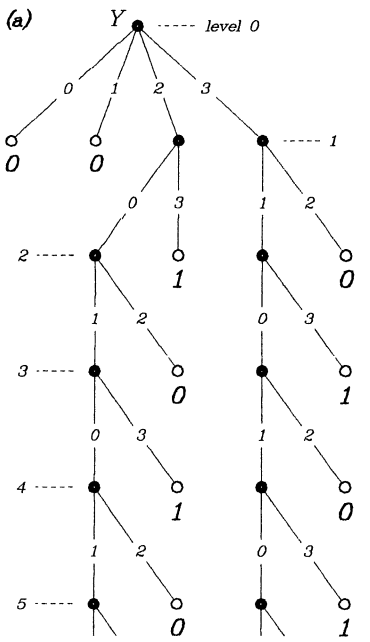
34.2 A Verification-DSE for a given Example

In this Section we develop a verification-DSE for the example specified verbally in Section 29.2 as:

> **Example** The single output, y, of a sequential circuit with two inputs, x_1 and x_2, is 0 when x_1 is 0, and when x_1 is 1 takes on the value x_2 had the last time x_1 was 1.

This example was examined thoroughly in Chapters 29 to 31. The primitive unmerged flow table is shown in Fig. 29–4b, its non-primitive (merged) form is shown in Fig. 30–1a, but it was only developed later, in Fig. 31–4. The word-recognition tree is pictured in Fig. 29–6a, but was developed in the successive word-recognition tables of Section 30.2.

In this Section I omit presenting the word-recognition tables that led to the verification tree of Fig. 34–8a. That is, the space consuming development of the previous Section, documented in the Figs. 34–2 to 34–7, is not laid out before you. On the contrary, I would like to encourage and ask you to do this work yourself along the lines demonstrated in the previous Section. Let me mention that the paper work is reduced if you use the non-primitive (merged) flow table of Fig. 30–1a instead of its primitive unmerged form of Fig. 29–4b. Whichever you use, the result will be the verification tree of Fig. 34–8a. When co-developing the verification flow table (Fig. 34–8b) together with the verification tree, it is imperative to use the **primitive** flow table. Why? A question I leave you with.



(b)

	0	1	3	2	
q	$(0,0)$	$(0,1)$	$(1,1)$	$(1,0)$	y
a	—	([3 1] 3) ([3 1] 0 2) ([3 1] 0 1 3) g	(a)	(3 2) d	0
b	—	([3 1] 3) ([3 1] 0 2) ([3 1] 0 1 3) g	(b)	(3 2) d	1
c	([2 0] 2) ([2 0] 1 3) ([2 0] 1 0 2) f	—	(2 3) a	(c)	0
d	([2 0] 2) ([2 0] 1 3) ([2 0] 1 0 2) f	—	(2 3) a	(d)	1
e	(2 0 [1 0] 2) f	(e)	(2 0 [1 3]) a	—	0
f	(f)	(2 [0 1] 3) (2 [0 1] 0 2) e	—	(2 [0 2]) (2 0 1 [0 2]) c	0
g	(3 [1 0] 2) (3 [1 0] 1 3) h	(g)	(3 [1 3]) (3 1 0 [1 3]) b	—	0
h	(h)	(3 1 [0 1] 3) g	—	(3 1 [0 2]) d	0

Fig. 34–8 Final verification tree (a) and verification flow table (b)

Of special importance are those input words — $(2,0,1,0,2)$, $(3,1,0,1,3)$, $(2,0,1,3)$, $(3,1,0,2)$, and $(2,3)$, $(3,2)$ — that are unique in at least one cell. For instance, $(2,0,1,3)$ is the only input sequence contained in cell $(e,3)$, the meaning of this cell for our example is thus solely determined by the input word $(2,0,1,3)$. This input word *must* therefore be contained as part of a verification-DSE as it is an **essential input-word**.

On the other hand, if an input word such as $(2,0,2)$ always occurs in a cell together with a certain other and longer input word — in our case, with $(2,0,1,0,2)$ — then $(2,0,2)$ is **redundant**. Redundant input words can nevertheless be put to good use as **'glue'** between other input words of a verification-DSE. For instance, the redundant input word $(2,0,2)$ can be used to combine an input word whose last input event is 2 with another input word whose first input event is also 2. The verification flow-table of Fig. 34–8b has two redundant input words: $(2,0,2)$ and $(3,1,3)$.

A verification-DSE consists of the sub-sequences contained in the verification flow table. It is good practice to list all sub-sequences as shown in (34.1) to (34.4).

$$\begin{pmatrix} 2, & 3 \\ 0_c, & 0 \\ 1_d, & 0 \end{pmatrix}, \qquad \begin{pmatrix} 3, & 2 \\ 0_a, & 1 \\ 1_b, & 1 \end{pmatrix}. \qquad (34.1a,b)$$

$$\begin{pmatrix} 2, & 0,2 \\ 0_c,0,0 \\ 1_d,0,0 \end{pmatrix}, \quad \begin{pmatrix} 3, & 1,3 \\ 0_a,0,1 \\ 1_b,0,1 \end{pmatrix}. \qquad (34.2a,b)$$

$$\begin{pmatrix} 2, & 0,1,3 \\ 0_c,0,0,0 \\ 1_d,0,0,0 \end{pmatrix}, \quad \begin{pmatrix} 3, & 1,0,2 \\ 0_a,0,0,1 \\ 1_b,0,0,1 \end{pmatrix}. \qquad (34.3a,b)$$

$$\begin{pmatrix} 2, & 0,1,0,2 \\ 0_c,0,0,0,0 \\ 1_d,0,0,0,0 \end{pmatrix}, \quad \begin{pmatrix} 3, & 1,0,1,3 \\ 0_a,0,0,0,1 \\ 1_b,0,0,0,1 \end{pmatrix}. \qquad (34.4a,b)$$

The first or top row of such a **sub-sequence notation** contains the input sequences, while each further row contains a possible sequence of implied output values with the starting stable-state being used as index. For instance, the sub-sequence notation (34.1a) states that the input sub-sequence $(2,3)$ causes the output sequence $(0,0)$ when the starting stable-state is c — we then write the output sequence as $(0_c,0)$ —, but causes the output sequence $(1,0)$ when the starting stable-state is d — we denote this output sequence as $(1_d,0)$.

A verification-DSE *must* contain all output sequences of all the non-redundant sub-sequences. A proposal for a verification-DSE is put forth in Fig. 34–9. It was obtained by exploration of possibilities to combine the sub-sequences of (34.1) to (34.4) rather than by following set rules — i.e., it was developed heuristically (simply because I know of no other way).

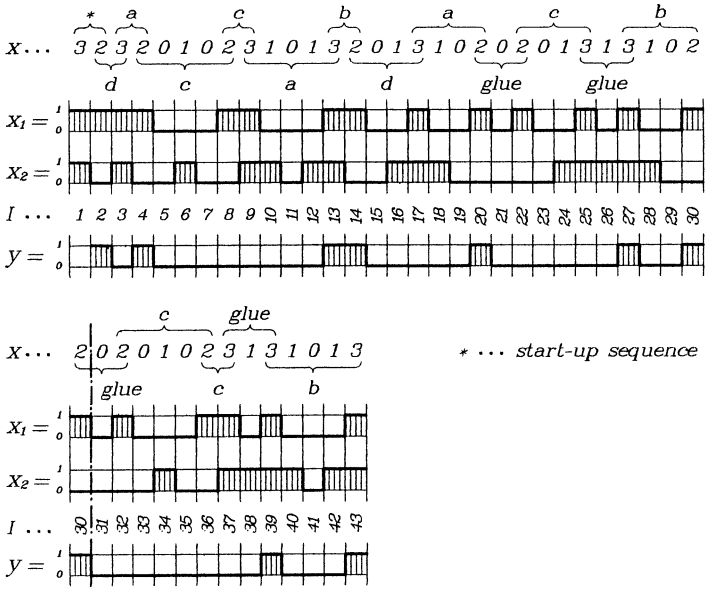


Fig. 34–9 A verification-DSE for the example

34.3 Grafting a Verification Tree

As shown in the examples of the previous two sections we need to co-develop the examples' respective verification trees and verification flow tables to be able to deduce verification-DSEs. In this section our first goal is to **develop the verification tree directly from the word-recognition tree** without developing and referring to word-recognition tables. With practice, this can reduce the work load considerably as the example of the D-flipflop, discussed here, will hopefully show.

We start with the slightly modified word-recognition tree of the D-flipflop shown in Fig. 34–10a, its original version presented in Fig. 33–11a. The prime difference is that the time-delayed variables Y' and Y_3' have been replaced by their non-delayed versions Y and Y_3 which is always permissible as the concept of a time-delay is not contained in the definition of a sequential circuit stated in (31.4)[1]. As minor as this difference seems, it points to the possibility, even necessity, of using an elementary sub-tree, or a whole sub-tree, whose root is $Y_{w(\ell)}$ as a scion **grafting** it onto those preliminary end-nodes also marked $Y_{w(\ell)}$.

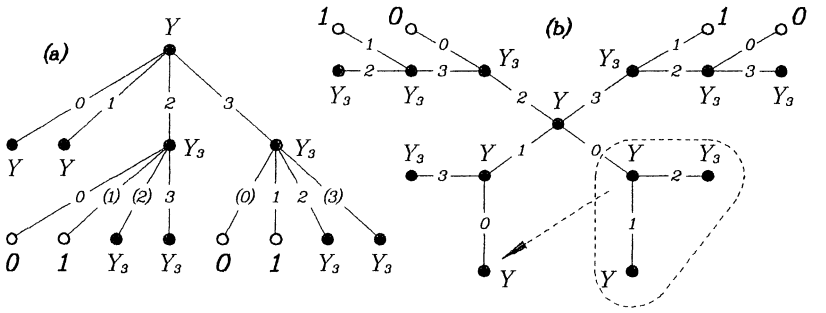


Fig. 34–10 From word-recognition tree (a) to verification tree (b)

We always graft a *complete* sub-tree, but once it has been grafted we **(a)** prune those edges non-adjacent to the input edge of the node to which we have grafted, and **(b)** prune the (single) edge whose input event is identical with that of the input edge to the node to which we have grafted. This is equivalent to the 'second step' in Section 34.1. As a first illustration of the grafting process, consider developing the tree of Fig. 34–10a from the word-recognition tree of Fig. 33–11a. The preliminary end-node (2) in Fig. 33–11a is marked by Y_3, which is at the same time the sub-tree growing from node (3): This sub-tree is grafted onto node (2) providing us with the tree pictured in Fig. 34–10a. In each version of the sub-tree Y_3 we then prune those edges whose input events are written in parentheses.

[1] The question of time delays not occurring in a sequential circuit was discussed thoroughly when developing the model of Fig. 31–3

For the further development of the verification tree it is advisable to arrange the edges stellately as in Fig. 34–10b. In this figure, an appropriate sub-tree is grafted to every preliminary end-node of Fig. 34–10a — to Y of edges (0) and (1), and to Y_3 of the input sequences (3, 2) and (2, 3) — omitting all pruneable edges. The next stage of grafting is indicated for Y of input sequence (0, 1) by the emphasized sub-tree. Please do take the time to ponder the transition and expansion of the tree of Fig. 34–10a to the verification tree shown in Fig. 34–10b. Further expansion of this tree by grafting leads to the verification tree for the D-flipflop of Fig. 34–11. Again, please take the time to understand this expansion, and to visualise before your mind's eye how this tree would continue growing (if this growth were necessary). But, as we shall see and discuss next, further growth of the verification tree is not required.

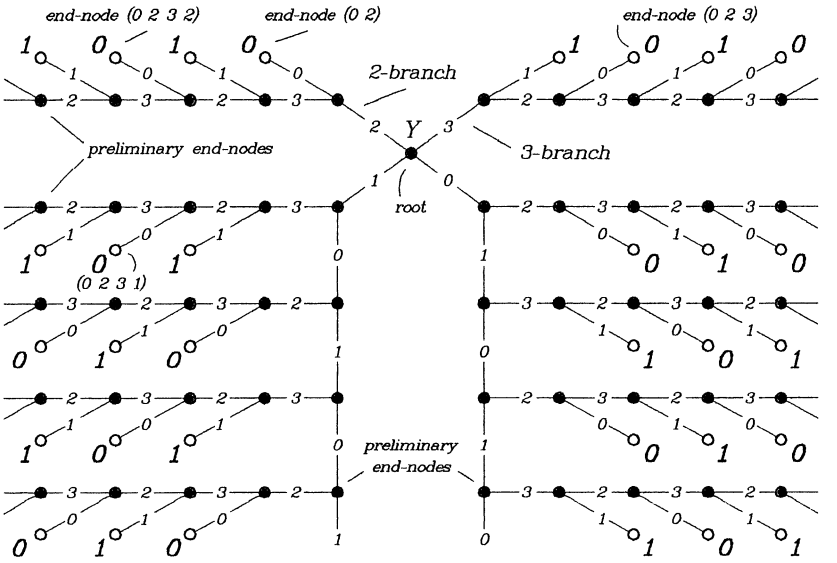


Fig. 34–11 The verification tree for the D-flipflop

For more elaborate problems than the ones discussed in Sections 34.1 and 34.2 the verification flow table (see for instance Fig. 34–8b) becomes a cumbersome instrument due to the many direct transitions

$$(x^{(1)}, \ldots, [x^{(j-1)}, x^{(j)}], \ldots, x^{(\ell)})$$

of input sub-sequences $(x^{(1)}, x^{(2)}, \ldots, x^{(\ell)})$ of various length ℓ entered into one and the same cell. You will find it much easier to develop and work with a **table of verification sub-sequences** as shown in Fig. 34–13. The table is developed from a flow table (in our case from that of Fig. 34–12 which represents the flow table of the D-flipflop, and is taken from Fig. 33–9)

for the *end-nodes* $(x^{(1)}, x^{(2)}, \ldots, x^{(\ell)})$ of the verification tree (in our example, for the verification tree of Fig. 34–11). The **rows** of a table of verification sub-sequences contain all cells of the flow table in which *unstable states* are inscribed. The **columns** mark the end-nodes — the sub-sequences $(x^{(1)}, \ldots, x^{(\ell)})$ — of the verification tree grouped according to their levels ℓ (the length of the sub-sequences).

(a)

	0	1	3	2	
s	$(0,0)$	$(0,1)$	$(1,1)$	$(1,0)$	y
a	(a)	c	—	g	0
b	(b)	d	—	g	1
c	a	(c)	f	—	0
d	b	(d)	f	—	1
e	—	c	(e)	g	0
f	—	d	(f)	h	1
g	a	—	e	(g)	0
h	b	—	f	(h)	1

(b)

	0	1	3	2	
q	$(0,0)$	$(0,1)$	$(1,1)$	$(1,0)$	y
a	(a)	c	—	([0 2] 3 1) g	0
b	(b)	d	—	([0 2] 3 1) g	1
c	a	(c)	f	—	0
d	b	(d)	f	—	1
e	—	(0 2 [3 1]) c	(e)	g	0
f	—	d	(f)	h	1
g	a	—	(0 [2 3] 1) e	(g)	0
h	b	—	f	(h)	1

Fig. 34–12 Successive versions of a verification flow table for the D-flipflop

Each direct transition $(x^{(1)}, \ldots, [x^{(j-1)}, x^{(j)}], \ldots, x^{(\ell)})$ of a sub-sequence $(x^{(1)}, x^{(2)}, \ldots, x^{(\ell)})$ marks a cell in the verification flow table. For instance, the sub-sequence $(0, 2, 3, 1)$ is a succession of direct transitions $([0,2], 3, 1)$, $(0, [2,3], 1)$, $(0, 2, [3,1])$. As can be seen from Fig. 34–12b, the direct transition $([0,2], 3, 1)$ characterises either of the cells $(a, 2)$ or $(b, 2)$, depending on the initial state a or b. Because both cells $(a, 2)$ or $(b, 2)$ have a GOTO-g command inscribed, the direct transition $([0,2], 3, 1)$ leads to cell $(g, 2)$. The following direct transition $(0, [2,3], 1)$ accounts for cell $(g, 3)$ while its successor transition $(0, 2, [3,1])$ accounts for cell $(e, 1)$. Summarising, the sub-sequence $(0, 2, 3, 1)$ accounts for the cells $(a, 2)$, $(b, 2)$, $(g, 3)$, and $(e, 1)$ as was deduced by following the GOTO-commands in the flow table or verification table of Fig. 34–12. All the cells thus accounted for — the cells $(a, 2)$, $(b, 2)$, $(g, 3)$, and $(e, 1)$ — are marked by **asterisks** (*) in the column $(0, 2, 3, 1)$ of the table of verification sub-sequences Fig. 34–13. As tedious as it is (when

not using a computer), you fill in the asterisks for one column after another of the table of verification sub-sequences according to the GOTO-commands of the flow table.

	(0 2 3 1)	(0 2 0 1)	(1 3 2 0)	(1 3 1 0)	(0 2 3 2 0)	(0 2 3 1 0)	(0 2 0 1 0)	(1 3 2 3 1)	(1 3 2 0 1)	(1 3 1 0 1)	(0 2 3 2 3 1)	(0 2 3 2 0 1)	(0 2 3 1 0 1)	(0 2 0 1 0 1)	(1 3 2 3 2 0)	(1 3 2 3 1 0)	(1 3 2 0 1 0)	(1 3 1 0 1 0)
(a, 1)		*					*					*	*	*				
(a, 2)	*	*			*	*	*				*	*	*	*				
(b, 1)									*	*							*	*
(b, 2)	*	*			*	*	*				*	*	*	*				
(c, 0)						*	*						*	*				
(c, 3)			*	*				*	*	*					*	*	*	*
(d, 0)				*						*						*	*	*
(d, 3)			*	*				*	*	*					*	*	*	*
(e, 1)	*					*					*		*					
(e, 2)					*						*	*						
(f, 1)				*				*		*						*		*
(f, 2)			*					*	*						*	*	*	
(g, 0)		*			*		*					*		*				
(g, 3)	*				*	*					*	*	*					
(h, 0)			*						*						*		*	
(h, 3)								*							*	*		

Fig. 34–13 Table of verification sub-sequences for the D-flipflop

With the help of the table of verification sub-sequences, Fig. 34–13, we are able to find sets of sub-sequences that can account for the unstable GOTO-commands of the flow table. Let us start with sub-sequences of **length 4**. Even taken together, the sub-sequences of length 4 cannot account for the cells $(b, 1)$, $(c, 0)$, $(e, 2)$, or $(h, 3)$ none of these rows containing an asterisk for a length-4 input sub-sequence. Next, considering sub-sequences of **length 5**, you will notice that every row of Fig. 34–13 contains at least one asterisk. But, do we need *all* sub-sequences of length 5 to account for all of the rows? Yes, because in any given length-5 column you will find one row accounted for only by that column making the input word an **essential** one.

Not shown in Fig. 34–13 are columns of length 2 and 3. Those would lead to the following input sub-sequences (given in their *sub-sequence notation*):

$$\begin{pmatrix} 0, & 2 \\ 0_a, & 0 \\ 1_b, & 0 \end{pmatrix} \quad \begin{pmatrix} 1, & 3 \\ 0_c, & 1 \\ 1_d, & 1 \end{pmatrix} \quad \begin{pmatrix} 0, & 2, & 3 \\ 0_a, & 0, & 0 \\ 1_b, & 0, & 0 \end{pmatrix} \quad \begin{pmatrix} 1, & 3, & 2 \\ 0_c, & 1, & 1 \\ 1_d, & 1, & 1 \end{pmatrix} \quad \underbrace{\begin{pmatrix} 0, & 2, & 0 \\ 0_a, & 0, & 0 \\ 1_b, & 0, & 0 \end{pmatrix} \quad \begin{pmatrix} 1, & 3, & 1 \\ 0_c, & 1, & 1 \\ 1_d, & 1, & 1 \end{pmatrix}}_{\text{possible glue}}$$

In this example it is advisable not to use sub-sequences (whatever their length) that end with an input event 2 of 3 because you cannot glue them to any other sub-sequences as no other sub-sequences *start* with an input event 2 or 3. By this token, we ignore the '2-branch' and the '3-branch' of the verification tree of Fig. 34–11.

A more detailed analysis shows that some length-5 sub-sequences can be replaced by length-4 sub-sequences. That is, you can realise a verification-DSE using the six input sub-sequences

$$\begin{array}{lll} (0,2,3,1), & (0,2,3,2,0), & (0,2,0,1,0), \\ (1,3,2,0), & (1,3,2,3,1), & (1,3,1,0,1). \end{array}$$

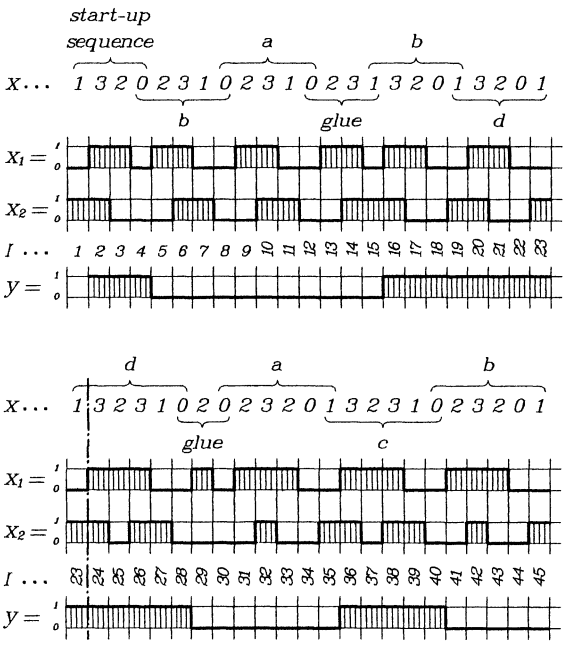


Fig. 34–14 A verification-DSE for the D-flipflop

You will be able to realise markedly shorter verification-DSEs if you can find a smaller number of necessary sub-sequences. After careful visual inspection of the table of verification sub-sequences you will find that both following sets of four sub-sequences (written here in their sub-sequence notation, i.e., together with the outputs they invoke)

$$\begin{pmatrix} 0,\ 2,0,1,0 \\ 0_a,0,0,0,0 \\ 1_b,0,0,0,0 \end{pmatrix} \quad \begin{pmatrix} 1,\ 3,1,0,1 \\ 0_c,1,1,1,1 \\ 1_d,1,1,1,1 \end{pmatrix} \quad \begin{pmatrix} 0,\ 2,3,2,3,1 \\ 0_a,0,0,0,0,0 \\ 1_b,0,0,0,0,0 \end{pmatrix} \quad \begin{pmatrix} 1,\ 3,2,3,2,0 \\ 0_c,1,1,1,1,1 \\ 1_d,1,1,1,1,1 \end{pmatrix}$$

and

$$\begin{pmatrix} 0,\ 2,3,1,0 \\ 0_a,0,0,0,0 \\ 1_b,0,0,0,0 \end{pmatrix} \quad \begin{pmatrix} 1,\ 3,2,0,1 \\ 0_c,1,1,1,1 \\ 1_d,1,1,1,1 \end{pmatrix} \quad \begin{pmatrix} 0,\ 2,3,2,1,0 \\ 0_a,0,0,0,0,0 \\ 1_b,0,0,0,0,0 \end{pmatrix} \quad \begin{pmatrix} 1,\ 3,2,3,1,0 \\ 0_c,1,1,1,1,1 \\ 1_d,1,1,1,1,1 \end{pmatrix}$$

will suffice. The latter is the basis for the verification-DSE of Fig. 34–14. Contrary to the possible impression given in Section 33.3, Fig. 33–8, the D-latch and the D-flipflop differ fundamentally in their input-output behaviour as a comparison of Fig. 34–1d and Fig. 34–14 shows.

Chapter 35

Discussing Huffman's Theory

In his doctor's thesis **Huffman** [1954] developed and presented almost all ideas basic to the analysis and synthesis of sequential circuits. Not only did he found this field of research, but his approach was to dominate all further work on sequential circuits and even strongly influence automata theory. To emphasize Huffman's achievement, let me point out that he invented the **flow table**, its **merging**, the **binary encoding of states**, he postulated the necessity to avoid **critical races**, further postulated that **every binary state variable is associated with a feedback loop** (i.e., a latch), introduced **delays** into feedback loops, and, finally, pointed to the existence of **hazards** and how to avoid them.

Huffman's theory is based on experience — experience he gathered when *analysing* sequential circuits. I have attempted to summarise this experience in Section 1 in what I have called *'Huffman's conjectures'*. In effect, these conjectures pose the basic hypotheses of Huffman's theory. Drawing on **Popper** [1972], **a scientific theory can never be verified.** The converse is postulated: **A scientific theory can only be falsified.** The logic principle used when falsifying is **modus tollens**, (35.1): Let P be the conjunction of basic hypotheses and/or already derived conclusions from which a new conclusion Q is drawn — then if the new conclusion Q can be shown to be false the conjunction P is false,

$$(P \rightarrow Q) \wedge \neg Q \Rightarrow \neg P. \qquad \textbf{(35.1)}$$

Whether you discard the theory totally or only partially or introduce *ad hoc* remedies after a certain conclusion has been shown to be false is another matter.

Huffman's theory poses a number of problems that I would like to summarise in the following two points. **(a)** Its basic hypotheses or postulates have never been formulated so stringently that they or conclusions drawn from them could be falsified (in effect making the theory unchallengeable). Formulating Huffman's conjectures (Section 1) is at least an attempt at stating the basic hypotheses. **(b)** Some of the conjectures contradict each other, and — most importantly — examples (the conclusions) can be given that contradict (falsify) each and every conjecture. Much of this is indicated in the body of this chapter.

35.1 Basics of Huffman's Synthesis Procedure *

Looking at the steps comprising Huffman's synthesis procedure you will find that we are already acquainted with each single step.

(35.1) Huffman's synthesis procedure
(a) Develop the **primitive flow table** of the problem.
(b) **Merge** the primitive flow table.
(c) **Encode** the internal states of the merged flow table **binarily**.
(d) Write the **logic equations** for the sequential circuit.

The *flow table* was introduced in Section 29.1 as a program governing the behaviour of an *automaton*. This admittedly differs from Huffman's approach (who developed the flow table as a means of analysing a sequential circuit), but the result is practically the same. The only difference is that Huffman imposed no restrictions on the design of the flow table (compare with (29.1) and (31.4)). Expressing a circuit's specification in a flow table, as required in (35.1a) above, was discussed in Section 29.2 for a verbally stated problem. Here too, the approach differs from the conventional one by explicitly employing input words of *finite length* to define the GOTO-command of each cell. *Merging*, as requested in (35.1b), has already been discussed in Section 31.4, the approach taken — using surrogate names for equivalent rows of the flow table — being merely a slight deviation from convention. *Binary encoding* (the third step in Huffman's procedure) was no more than mentioned in connection with the composite word-recognition table, but is considered in more detail in this and the next chapter. As shown in Section 31.6 the binary variables used to encode the rows are the output variables of internal latches — this is also one of Huffman's basic assumptions. Some of the ramifications of this assumption are considered in this chapter (especially in connection with delays and their placement in a feedback loop). The last step — writing the logic equations, (35.1d) — simply requires reading a K-map.

When analysing sequential circuits Huffman formed a number of conclusions on how to specify, analyse, and synthesize sequential circuits. These conclusions were drawn on the basis of incomplete information — information taken from examples, and a finite number of them *per se* — and may thus be referred to as *conjectures*. Allow me to refer to them as **Huffman's conjectures** although he in the main part did not formulate them explicitly.

(35.2) Conjecture on the succession of states. The value of every *next-state* variable q^+ **succeeds** the value of its *present-state* q after a short delay, say, of Δt with $\Delta t > 0$.

(35.3) Conjecture on fundamental mode. A present input event x must remain invariable while the circuit settles down to a stable state q. This is referred to as operating the circuit in **fundamental mode**.

(35.4) Conjecture on concurrent input values for g and h. Both functions, the transition function $g : \mathcal{Q} \times \mathcal{E} \mapsto \mathcal{Q}$ and the output function $h_{mr} : \mathcal{Q} \mapsto \{0,1\}$ or $h_{ml} : \mathcal{Q} \times \mathcal{E} \mapsto \{0,1\}$, are supplied with *simultaneously identical* (i.e., concurrent) input values q and x, which we express algebraically as $q^+ = g(q,x)$ and $y = h_{mr}(q)$ or $y = h_{mr}(q,x)$, or technically as $q^+(t) = g\big(q(t),x(t)\big)$, and $y(t) = h_{mr}\big(q(t)\big)$ or $y(t) = h_{mr}\big(q(t),x(t)\big)$.

(35.5) Conjecture on encoding only merged flow tables. A binary state assignment for a flow table is only chosen once the flow table has been completely merged.

(35.6) Conjecture on arbitrary binary encoding of the present states. Each name of a row or present state q of a flow table is assigned an *unique* p-tuple $(s_1, s_2, \ldots, s_p)$ of binary values[1] — uniqueness being the only requirement for the choice of the assignment. Each binary variable $s_i \in \{0,1\}$ of the p-tuple $(s_1, s_2, \ldots, s_p)$ is called a **binary state-variable.**

(35.7) Conjecture on the binary encoding of the next states. The unique binary code assigned to a row name is also given to all occurrences of that name within the *body* of the transition table.

(35.8) Conjecture on avoiding 'critical races'. Any binary encoding (i.e., state assignment) of the rows of a transition table must be chosen so as to avoid *'critical races'*.

(35.9) Conjecture on the correspondence between state variables and latches. There is a 1-to-1 correspondence between the binary state variables s_i and the internal latches of a sequential circuit realising the transition table, i.e., the binary state variables s_i are the outputs of the internal latches.

In the following sections we take a critical look at these conjectures. We shall see that some of them are mutually contradictory, and that none of them are generally valid — this latter aspect refers to the fact that it is sometimes advisable and sometimes necessary to violate them.

35.2 Delay and Feedback

Huffman's model takes a completely different approach to feedback than the catenation model does. In the catenation model feedback is the formal result drawn from the postulate of stability, (31.4c), as discussed in the latter half of Section 31.2. Huffman's reasoning is less formal. Prompted by the analysis of sequential circuits, he in effect postulated conjecture (35.2) on the succession

[1] Such a binary assignment of a row is said to be *unique* if no two rows are given the same assignment

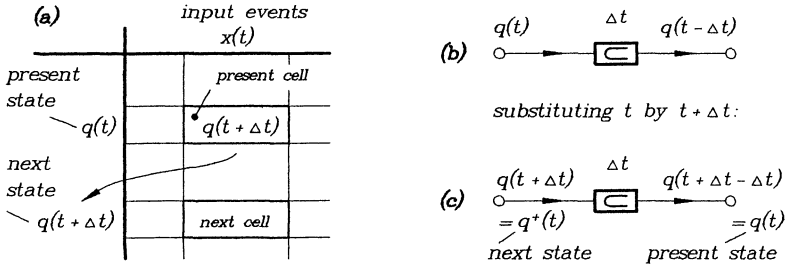


Fig. 35–1 Intended meaning of conjecture (35.2)

of states: *The value of every next-state variable* q^+ succeeds *the value of its present-state* q *after a short delay, say, of* Δt. The intended interpretation of this conjecture is pictured in the transition table of Fig. 35–1a. This picture is an attempt to persuade you that any output, $q^+(t) \stackrel{df}{=} q(t + \Delta t)$, generated by the transition function g *must* be used as the *'next'* input to the function. This usage of g's output as its next input is not contained in either Moore's or Mealy's formulation of Huffman's model, (31.1) and (31.2). The definition of the transition function $g : \mathcal{Q} \times \mathcal{E} \mapsto \mathcal{Q}$ with $g(q, x) = q^+$ given in these models merely states what output value to expect, namely q^+, but not how to use it (i.e., as the next input value). As explicated in Figs. 35–1bc, using the next-state variable $q^+(t)$ as input to a delay element allows us to describe its output as $q(t)$ which is the present-state variable.

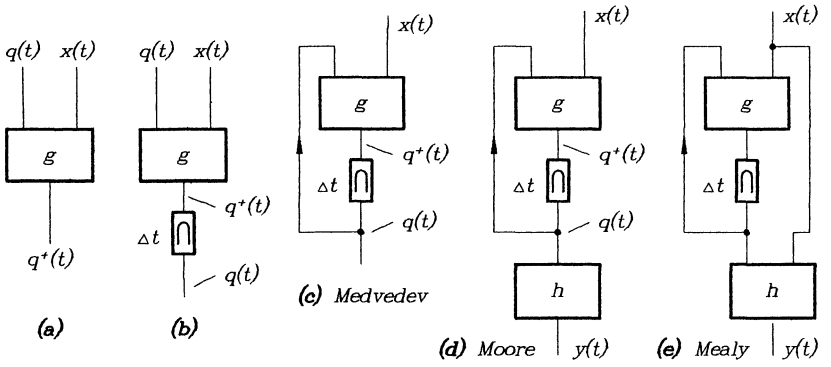


Fig. 35–2 Developing the structure of a sequential circuit

Now let us see how the above considerations influence the **structure of a sequential circuit.** The input-output behaviour of the *'black box'* of Fig. 35–2a — whose output $q^+(t)$ is an abbreviation for $q(t + \Delta t)$ — is explained as realising the transition table of Fig. 35–1a. We can realise the requirements of conjecture (35.2) after **placing a delay element at the**

output of g, see Fig. 35–2b, thereby delaying $q^+(t)$ to $q(t)$. The output of the delay being identical with the input $q(t)$ of the transition function g allows us to connect this input to the delay's output thereby **creating a feedback**, see Fig. 35–2c. This feedback realises the requirement of conjecture (35.2). It ensures that the value of the next state $q^+(t)$ succeeds that of the present state $q(t)$ after a delay time of Δt.

The final step is to employ conjecture (35.4) on concurrent input values to the functions g and h to allow us to append a *'black box'* for the output function h hereby obtaining the circuit structure of Figs. 35–2de.

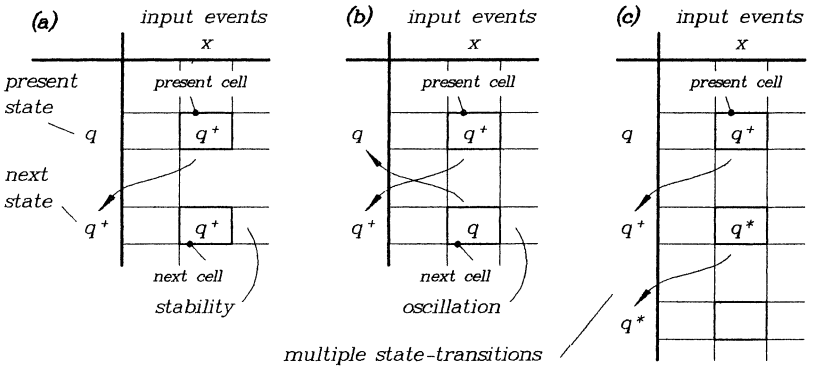


Fig. 35–3 The three basic modes of transition in a transition table

A noteworthy aspect of Huffman's theory is the way the transition table is used: No restriction is placed on the GOTO-command entered into the **'next cell'**, see Fig. 35–1a. The three basic possibilities to choose a next GOTO-command are pictured in Fig. 35–3. They can lead to a **stable condition** as in Fig. 35–3a, to a permanent **oscillation** as in Fig. 35–3b, or to **multiple state-transitions** as in Fig. 35–3c. These three patterns may and do occur intertwined.

35.3 State Encoding and Circuit Realisation

The s rows of a flow table are always referred to by name, say, by the elements of $\mathcal{Q} = \{q_1, \ldots, q_s\}$. These names (as the rows they stand for) are called the *present states* of the flow table. Each present state $q \in \mathcal{Q}$ together with an input event $x \in \mathcal{E}$ determines the *next state* $q^+ \in \mathcal{Q}$ as pictured in Fig. 35–4a. **Encoding the states** of a flow table refers to assigning an unique binary code or number, say $(s_1, \ldots, s_p)$ where any $s_i \in \{0, 1\}$, to each present state or row of a flow table as indicated in Fig. 35–4b. But, how many binary digits do we need when encoding s states? Any number p larger than or equal to

$$p_{min} = \lceil \log_2 s \qquad (35.10)$$

allows us to encode s states[1]). Design engineers have traditionally wanted to use the smallest viable number p of binary variables $(s_1, \ldots, s_p)$, as each binary variable (according to conjecture (35.9)) represents a latch. This entails working with *merged* flow tables as formulated in conjecture (35.5).

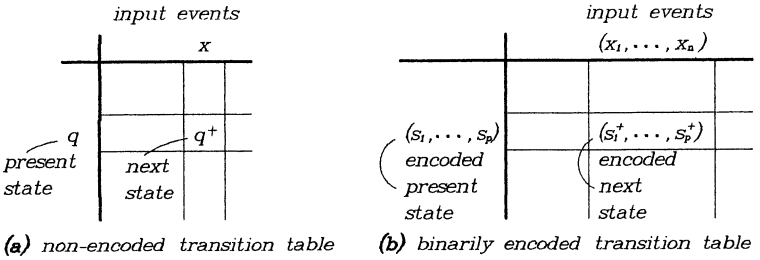


Fig. 35–4 Binary state encoding

The simplest way to introduce Huffman's synthesis procedure is by example. To this end, let us reconsider the T-flipflop (already discussed in Section 32.1). The first step of the procedure, (35.1a), is to specify the primitive flow table. This is done in Fig. 35–5a, and is a repetition of Fig. 32–1c. The second step, (35.1b), on merging the primitive flow table, requires no specific action in our case as the flow table of Fig. 35–5a is already in its merged form (which you are invited to argue).

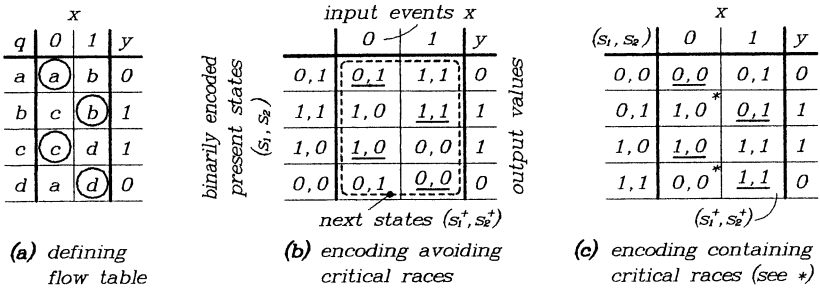


Fig. 35–5 Two typical binary state encodings for a given flow table

The heart of Huffman's synthesis procedure is the unique binary encoding of the present states — i.e., the row names — of the flow table as required in (35.1c). In an ideal system we can choose any encoding as long as it is *unique* (as expressed in conjecture (35.6)). Two such encodings are shown in the left-most columns of Figs. 35–5bc[2]). Each next-state name a, b, c, or d contained in the body of the flow table is given the binary encoding of

[1]) For any real number x the 'ceiling of x', written $\lceil x$, is the smallest integer greater than or equal to x.

[2]) Of course it would be easy to claim that the encodings shown in Figs. 35–5bc

that name's present state (as conjecture (35.7) requires). Thus we obtain the *completely* encoded flow tables Figs. 35–5bc.

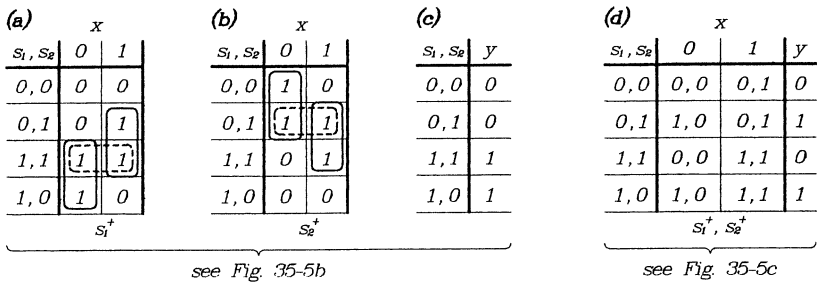


Fig. 35–6 Evaluating the flow tables of Figs. 35–5bc

To evaluate the flow tables of Figs. 35–5bc it is good practice to write them as K-maps. This only requires us to rearrange (if necessary) their rows and columns according to a Gray code (see Section 6.3). For Fig. 35–5c rearranging its rows leads to Fig. 35–6d, a table that actually represents 3 different K-maps: One for each of the variables s_1^+, s_2^+, and y. These three K-maps are shown separately in Figs. 35–6abc for the flow table of Fig. 35–5b. Evaluating the K-maps of Figs. 35–6abc leads to

$$S_1^+ \Leftrightarrow XS_2 \vee \overline{X}S_1 \vee S_1S_2 \Leftrightarrow XS_2 \vee (\overline{X} \vee S_2)S_1, \qquad (35.11a)$$

$$S_2^+ \Leftrightarrow \overline{X}\,\overline{S}_1 \vee XS_2 \vee \overline{S}_1S_2 \Leftrightarrow \overline{X}\,\overline{S}_1 \vee (X \vee \overline{S}_1)S_2, \qquad (35.11b)$$

$$Y \Leftrightarrow S_1. \qquad (35.11c)$$

These formulas can be expressed by the circuit of Fig. 35–7a. As Huffman's model implies no structure, the internal states S_1 and S_2 of the equations (35.11ab) are taken to be of like standing. This is often expressed by drawing the circuits they imply in parallel and with cross-coupled feedbacks as shown in Fig. 35–7a. But in principle (that is, considering only the logic, but not the placement of the delays) the circuits of Figs. 35–7a and 32–6 are identical.

Splitting the flow table of Fig.35–6d into three K-maps for S_1^+, S_2^+, and Y which conform to those of Figs. 35–6abc allow us to obtain the formulas

$$S_1^+ \Leftrightarrow \overline{X}\,\overline{S}_1S_2 \vee XS_1 \vee S_1\overline{S}_2, \qquad (35.12a)$$

$$S_2^+ \Leftrightarrow X, \qquad (35.12b)$$

$$Y \Leftrightarrow \overline{S}_1S_2 \vee S_1\overline{S}_2. \qquad (35.12c)$$

With practice you can read these formulas directly from the flow table of Fig. 35–6d. These formulas are in obvious contrast to conjecture (35.9):

---

are arbitrary. But, actually, each was chosen to represent a specific property with respect to so-called critical races. We discuss these properties and critical races in the next section.

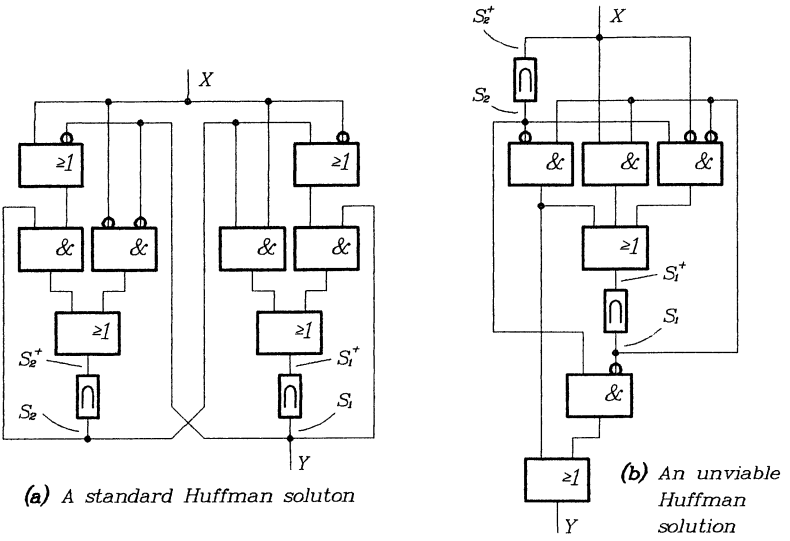


Fig. 35–7 Two solutions for the T-flipflop of Fig. 35–5a

S_2^+ cannot be interpreted as the output of a latch, meaning, *we have lost one latch.* The circuit obtained — see Fig. 35–7b — is hardly a viable solution. Clearly this is a result so disturbing that it warrants further investigation — an investigation undertaken in the next section.

35.4 Problems with Arbitrary Binary Encodings

The first thing to note about the circuit of Fig. 35–7b is that it has only *one* internal latch when it should have *two* according to the *conjecture on the correspondence between state variables and latches*, (35.9). To remedy the situation, we might consider discarding the *conjecture on the correspondence between state variables and latches*, (35.9), or possibly the *conjecture on arbitrary binary encoding of the present states*, (35.6) — the latter anyway contradicting the *conjecture on avoiding critical races*, (36.8)[1].

Interestingly, few seem to take offence when the calculation of a circuit described by a flow table of s states produces less latches than the minimum number $p_{min} = \lceil \log_2 s$, (35.10), of binary state variable, i.e., that conjecture (35.9) is violated. But if we do discard (35.9), we must answer the question

[1] The actual problem, though, is not which conjecture to retain or discard. The problem is that in Huffman's theory there are no tangible prerequisites, i.e., postulates or axioms, from which to develop theorems and on which to base proofs.

'What then is the smallest number of latches needed to encode a given transition table?', or *'Is a latch (nothing else than) a delay?'*, or possibly *'What is a latch in the context of the Huffman theory?'*.

The second thing to note about the circuit of Fig. 35–7b is that its correct functioning depends on the relative magnitudes of the delays. The output value you get can be anything from a correct output value, a wrong output value, or an oscillating output[1]. There seems to be general consent[2] that this deficiency is due to the phenomenon of *races*. Because Huffman's model of a binarily encoded system describes the system in transition, not in its stable states, this can lead to races between states before a stable state is reached. Let us look into this in detail with the help of Fig. 35–8.

Consider switching from a stable state $(s_1, \ldots, s_p)$ in column $(x_1, \ldots, x_n)$ of a transition table to the next state $(s_1^*, \ldots, s_p^*)$ in column $(x_1^*, \ldots, x_n^*)$. Assume that the binary p-tuples of the present state $(s_1, \ldots, s_p)$ and the next state $(s_1^*, \ldots, s_p^*)$ differ in two or more values. If those state variables required to change their values when the present state $(s_1, \ldots, s_p)$ changes to the next state $(s_1^*, \ldots, s_p^*)$ do so simultaneously, then you correctly GOTO row $(s_1^*, \ldots, s_p^*)$. But due to various and sometimes varying delay times in gates, latches and leads, variables changing their values will do so at different moments. With the first next-state variable that changes, a new present state is created, the GOTO-command in its row sending us to an unwanted destination. This process continues anew every time another next-state variable changes its value. The phenomenon is called a **race** because the sequence of cells in column $(x_1^*, \ldots, x_n^*)$ passed through depends on which next-state variables change their values quickest.

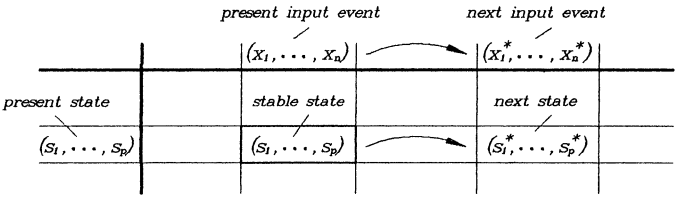


Fig. 35–8 Discussing the prerequisite for a *race*

Races can have very negative effects as Fig. 35–3 can help us see. We can be sent to a wrong stable state, into an oscillation loop, or we can be forced through multiple transition changes (before coming to rest in a stable state, or entering an oscillation loop). If all possible state transitions lead to the originally specified next state $(s_1^*, \ldots, s_p^*)$, the race is called **non-critical**, otherwise **critical**.

---

[1] A circuit of this kind is analysed in great detail by Dietmeyer [1971] in Section 5.3, Figures 5.11 and 5.12.

[2] You can interpret this to mean that I, for one, do not agree.

One possible cause for the misbehaviour of a circuit follows from so-called **race analysis**, analysis always undertaken on the transition table from which the internal states were developed. The possible misbehaviour of the circuit of Fig. 35–7b can be deduced from the flow table, Fig. 35–6d, from which the circuit was developed. The transition table contained in that flow table is repeated in Fig. 35–9. Assume we begin in cell $(s_1, s_2; x) = (0, 1; 1)$ marked 'start' in Fig. 35–9a. Changing the input from 1 to 0 we switch from $(0, 1; 1)$ to $(0, 1; 0)$, this cell containing the GOTO-command $(s_1^*, s_2^*) = (1, 0)$ requiring *both* state variables s_1 and s_2 to change their values. The three possible outcomes of the incurring *race* are represented by the arrows a, b, and c of Fig. 35–9a.

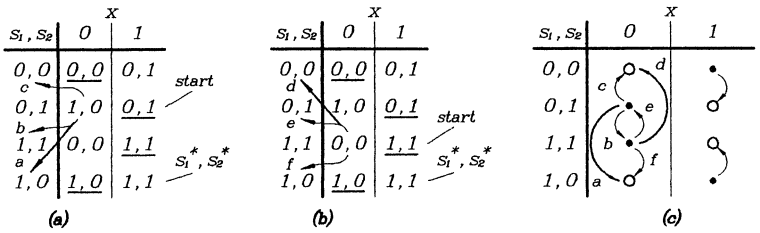


Fig. 35–9 Race analysis in the flow table that led to circuit Fig. 35–7b

Had we started in cell $(1, 1; 1)$, as shown in Fig. 35–9b, and changed the input from 1 to 0, each of the arrows d, e, and f would picture a possible outcome of an incurring race. The race conditions pictured in Figs. 35–9ab are depicted together in a single column in the abbreviated form shown in the column for $x = 0$ of Fig. 35–9c. This figure also shows that no races occur when switching from $x = 0$ or $x = 1$. The arrows a and d mark the intended direct transitions (to a stable state). Following arrows b and e shows that an oscillation can occur. Arrows c and f mark the outcomes of *critical* races — in both cases we end up in a stable state, but in the wrong one. The arrow e followed by arrow c documents a non-critical race (as does b followed by f).

35.5 Race Analysis in the Catenation Model *

In Section 29.4 we developed the circuit repeated in Fig. 35–10. This circuit was calculated from the word-recognition tree of Fig. 29–6 on the basis of the catenation model introduced in Section 31.2. The circuit differs from a Huffman solution — which is the basis for race analysis — in that the only time-delayed input-variable to an internal latch is the latch's own output variable. To carry out race analysis, we need to work with a transition table as the one shown in Fig. 35–4b the rows of which are marked by the present state-variables, in our case by Y_{03}, Y_2, Y_3, and whose cells contain the next

state-variables Y_{03}^+, Y_2^+, Y_3^+ — its columns are labelled by the input events. As discussed in connection with Figs. 35–1bc we can equally well refer to the row labels as the *previous* states Y_{03}', Y_2', Y_3', and subsequently to the contents of the cells as the present states Y_{03}, Y_2, Y_3. As we shall see, this convention is better adapted to circuits developed according to the catenation model, and is thus used here.

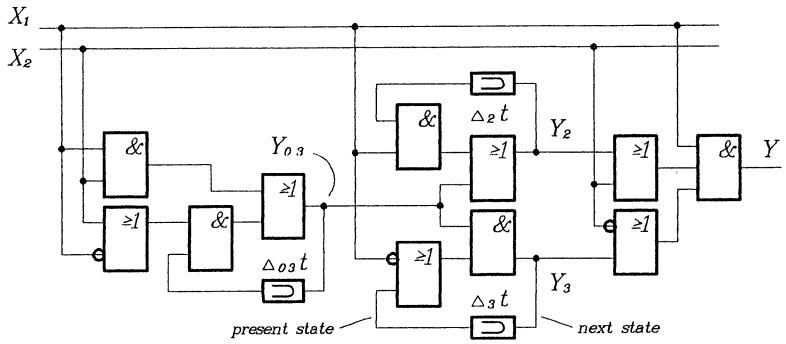


Fig. 35–10 The sequential circuit to be checked for races

We start the development of the circuit's transition table by reading the formulas for the internal latches Y_{03}, Y_2, Y_3 from the circuit of Fig. 35–10. For subsequent use of Y_{03} we also express this variable in its canonical conjunctive form of (35.13b). You might like to compare the formulas of (35.13) with those of (29.6).

$$Y_{03} \Leftrightarrow G_{03}(X_1, X_2; Y_{03}') \Leftrightarrow X_1X_2 \vee \overline{X}_1Y_{0,3}' \vee X_2Y_{0,3}', \tag{35.13a}$$

$$Y_{03} \Leftrightarrow G_{03}(X_1, X_2; Y_{03}') \Leftrightarrow (\overline{X}_1 \vee X_2)(X_1 \vee Y_{03}')(X_2 \vee Y_{03}'), \tag{35.13b}$$

$$Y_2 \Leftrightarrow G_2(X_1, X_2; Y_{03}', Y_2') \Leftrightarrow Y_{03} \vee X_1Y_2', \tag{35.13c}$$

$$Y_3 \Leftrightarrow G_3(X_1, X_2; Y_{03}', Y_3') \Leftrightarrow Y_{03}(\overline{X}_1 \vee Y_3'). \tag{35.13d}$$

These formulas must be transformed so that *all* their *independant* state variables are delayed in which case they describe a transition table whose rows are delayed state variables as required for Huffman's race analysis. This is achieved by substituting Y_{03} of (35.13a) in

$$\begin{aligned} Y_2 &\overset{(35.13c)}{\Longleftrightarrow} Y_{0,3} \vee X_1Y_2' \\ &\Longleftrightarrow X_1X_2 \vee X_2Y_{0,3}' \vee \overline{X}_1Y_{0,3}' \vee X_1Y_2'. \end{aligned} \tag{35.14a}$$

Next, substitute Y_{03} of (35.13d) by Y_{03} of (35.13b)

$$\begin{aligned} Y_3 &\overset{(35.13d)}{\Longleftrightarrow} Y_{0,3}(\overline{X}_1 \vee Y_3') \\ &\Longleftrightarrow (\overline{X}_1 \vee X_2)(X_1 \vee Y_{0,3}')(X_2 \vee Y_{0,3}')(\overline{X}_1 \vee Y_3'). \end{aligned} \tag{35.14b}$$

The easiest way to obtain the circuit's transition table is to first develop the K-maps for each of the formulas (35.13a) and (35.14), see Fig. 35–11. Each loop in the K-map of Fig. 35–11a corresponds to a conjunctive term in (35.13a). Each loop in the K-map of Fig. 35–11b corresponds to a conjunctive term in (35.14a), and each loop in the K-map of Fig. 35–11c corresponds to a disjunctive term in (35.14b).

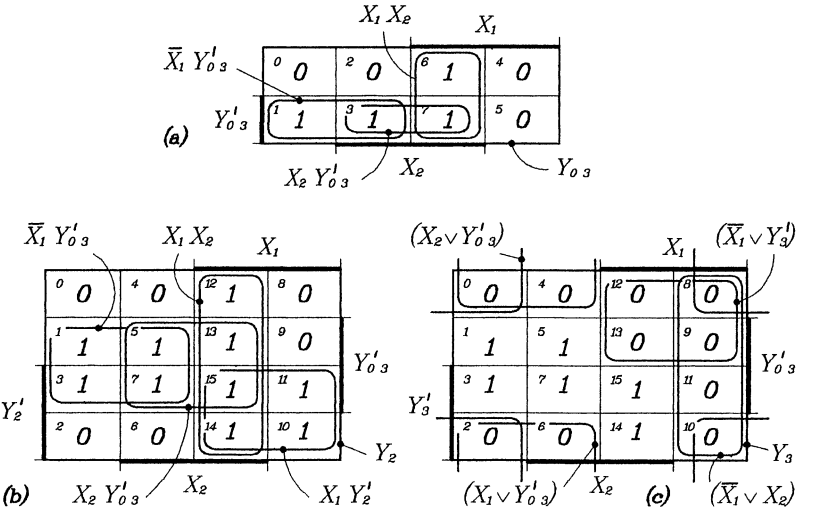


Fig. 35–11 The K-maps of the formulas (35.13a) and (35.14)

You can now combine the two K-maps of Figs. 35–11ab to obtain the dual-output K-map of Fig. 35–12a. This K-map can be interpreted as a transition table for Y_2 and $Y_{0,3}$ (this is a *partial* transition table as in it only two internal states (Y_2 and $Y_{0,3}$) of all three (Y_3, Y_2, and $Y_{0,3}$) are considered) — allowing us to gauge any races between these internal latches. As even a cursory inspection shows, no critical races occur between Y_2 and $Y_{0,3}$.

Combining the K-maps for Y_3 and $Y_{0,3}$, Figs. 35–11ac, we obtain the partial transition table Fig. 35–12b which allows us to recognise possible races between Y_3 and $Y_{0,3}$. Again, as you can see, no critical races occur.

(a)

$Y_2', Y_{0\,3}'$ \ X_1, X_2	0,0	0,1	1,1	1,0
0,0	0,0	0,0	1,1	0,0
*0,1	1,1	1,1	1,1	0,0
1,1	1,1	1,1	1,1	1,0
1,0	0,0	0,0	1,1	1,0

$Y_2, Y_{0\,3}$

(b)

$Y_3', Y_{0\,3}'$ \ X_1, X_2	0,0	0,1	1,1	1,0
0,0	0,0	0,0	0,1	0,0
0,1	1,1	1,1	0,1	0,0
1,1	1,1	1,1	1,1	0,0
*1,0	0,0	0,0	1,1	0,0

$Y_3, Y_{0\,3}$

Fig. 35–12 Partial transition tables for (Y_2Y_{03}) and (Y_3Y_{03})

Please forgive me for not discussing in detail those rows of Fig. 35–12 marked by an asterisk (*). Ideally, the input events $(Y'_2, Y'_{0,3}) \Leftrightarrow (\mathbf{0}, \mathbf{1})$ and $(Y'_3, Y'_{0,3}) \Leftrightarrow (\mathbf{1}, \mathbf{0})$ will not occur. But if they do occur briefly (due to races) the entries in these marked rows (none of which represent stable states) ensure *non-critical* races.

	y'_3	y'_2	$y'_{0\,3}$	$(x_1, x_2) =$ (0,0)			(0,1)			(1,1)			(1,0)		
c	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0
*	0	0	1	1	1	1	1	1	1	0	1	1	0	0	0
d	0	1	0	0	0	0	0	0	0	0	1	1	0	1	0
a	0	1	1	1	1	1	1	1	1	0	1	1	0	1	0
*	1	0	0	0	0	0	0	0	0	1	1	1	0	0	0
*	1	0	1	1	1	1	1	1	1	1	1	1	0	0	0
*	1	1	0	0	0	0	0	0	0	1	1	1	0	1	0
b	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0

$y_3, y_2, y_{0\,3}$

Fig. 35–13 Single overall transition table in which the partial transition tables of Figs. 35–12 are combined

Although it is not any more necessary (as no further insight is to be expected), we can combine the partial transition tables of Figs. 35–12 to obtain the single overall transition table shown in Fig. 35–13. The rows marked by an asterisk (*) coincide with the equivalent rows of Figs. 35–12. The rows labelled *a*, *b*, *c*, *d* coincide with the equally labelled rows of the transition table of Fig. 31–15a. This closes the circle to the *composite* word-recognition tables of the catenation models.

Two cells in Fig. 35–13 — $(c; 1, 1)$ and $(b; 1, 0)$ — indicate the existence of races. But, as one can see almost at a glance, none of the races that might occur are critical — as was to be expected.

Chapter 36

State Encoding Techniques

Huffman's method for synthesizing a sequential circuit consists of only three steps before you can calculate the circuit's formulas: (a) formulating the flow table, (b) merging the flow table, and (c) encoding its transition table binarily. Merging is often viewed as desirable to reduce the number of rows of a transition table and thereby the number of binary variables needed to encode them. Alas, it is often necessary to *append* rows to a flow table (all the more so if the flow table was merged) only to ensure that no critical races occur. And precisely this is the point — to avoid switching to an incorrect stable state due to races, it is categorically required that *'any binary encoding of the rows of a transition table is chosen so as to avoid critical races'*. We thus choose to abide by the *conjecture on avoiding critical races*, (35.8), tacitly dropping (35.6) which would allow us to use any unique encoding of rows.

In the first two sections we discuss two *trial and error* state-encoding techniques. Sections 4, 5, and 6 present methodical techniques developed by **Huffman** [1955] and extensively discussed by **Caldwell** [1958] whose representation of the subject served as the basis for this chapter. The methodical techniques are relatively easy to apply. Happily, the number of states they require must be viewed as an upper bound. This means, using *trial and error* techniques may sometimes still allow a reduction of the number of states — an eldorado for the creative designer.

The methodical state-assignment methods lead to a large number of state variables. This means you have to work with very large, expanded transition tables. When creating these expanded transition tables the contents of most of their cells remain undefined meaning that you can, in fact must, enter GOTO-commands at your discretion. The number of options open to you is legend. Each option will lead to a different circuit. But to obtain a circuit, you must first evaluate its large expanded transition table which, due to its size, can in itself be quite a feat. If you want to find an *optimal* circuit there is nothing to it but to calculate and draw the circuits for all possible encodings, and to choose a circuit you think *optimal*.

The workload required by these methods can be formidable. Furthermore, they do not lend themselves well to computerisation. In this situation, you might want to take another look at the catenation method discussed in

Chapters 29 to 31. It copes well with large flow tables, is to a large extent open to computerisation, and gives good to reasonable results.

36.1 Introducing Multiple State-Transitions

Suppose we want to design the internal latches of a sequential circuit whose transition table is given as that of Fig. 36–1a. To do this, we need to encode the rows of the transition table so that no critical races will occur. The *transition diagram*, Fig. 36–1b, is a useful tool to see if the given number of rows can be encoded in a race-free manner. A **transition diagram** has one node for each row of the transition table. Any *unstable* GOTO-command, say q_s, in a row q_r lets us draw a directed edge (an arrow) from node q_r to q_s. For instance, row a holds the unstable GOTO-commands b and d. In the transition diagram these unstable GOTO-commands are represented by arrows from node a to nodes b and d. With hardly more than a glance needed, you can verify the transition diagram of Fig. 36–1b.

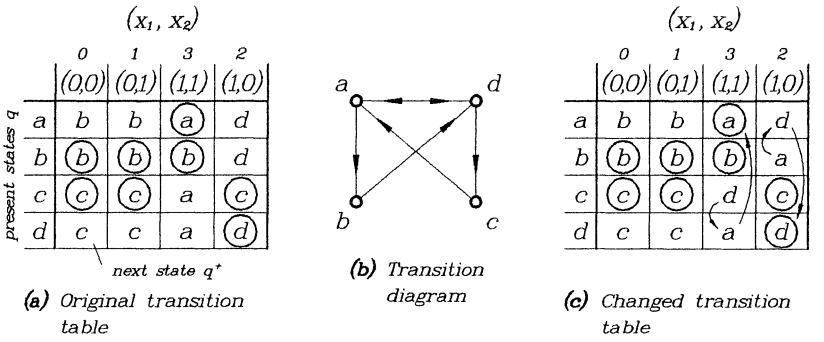


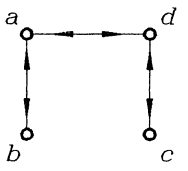
Fig. 36–1 Example on the introduction of multiple state-transitions

Let us attempt to assign to each node of the transition diagram a two element binary code so that all *connected nodes* have adjacent code words, i.e., differ in value in only one position. It is easy to see that no such code can be assigned to either of the triangles (a, b, d) or (a, c, d). For instance, choosing $(0, 0)$ for a, it is of course possible to give b and d adjacent code words: $(0, 1)$ and $(1, 0)$. But b and d, also being connected by an edge, must be assigned code words that are adjacent — which $(0, 1)$ and $(1, 0)$ are not. Considering transition tables of more than 4, 8, 16, etc., rows, we can encounter transition diagrams which incorporate **odd-sided polygons**. No odd-sided polygon can be encoded so that each of its nodes carries a code word that is adjacent to one of those of its two neighbouring nodes.

The so-called **encoding problem** occurs when the transition diagram of a transition table contains one or more odd-sided polygons. The **solution**

of the encoding problem inevitably consists in altering the transition *table* so that its associated transition *diagram* does not any more contain an odd-sided polygon. The changes to the transition table must be so that the *stable* input-output behaviour for the original and changed transition table are identical for any given sequence of input words.

Sometimes it is possible to break open an odd-sided polygon by introducing multiple state-transitions. In our example of Fig. 36–1 we shall attempt to break the connections from b to d, and from c to a. The transition diagram, Fig. 36–1b, indicates that it might be possible, instead of going from b to d directly, to reach d via a. The arrow from b to d corresponds to the GOTO-d command entered in cell $(b, 2)$ of Fig. 36–1a. Changing this entry to a GOTO-a command, see cell $(b, 2)$ of Fig. 36–1c, allows us to switch from b to d via a as desired. Interrupting the connection from c to a, Fig. 36–1b, is achieved by changing the GOTO-command of cell $(c, 3)$, Fig. 36–1a, to a GOTO-command, Fig. 36–1c.



(a) Transition diagram

code number

	1	2	3	4	5	6	7	8
a	0,0	0,0	0,1	0,1	1,0	1,0	1,1	1,1
b	0,1	1,0	0,0	1,1	0,0	1,1	0,1	1,0
c	1,1	1,1	1,0	1,0	0,1	0,1	0,0	0,0
d	1,0	0,1	1,1	0,0	1,1	0,0	1,0	0,1

S_1, S_2

(b) Possible adjacent encodings

Fig. 36–2 Listing all adjacent encodings for the given transition diagram

It is good practice to develop a new transition diagram, see Fig. 36–2a, from the changed transition table, Fig. 36–1c, to make sure that no odd-sided polygons have been overlooked. The question still remains, which binary code word, (s_1, s_2) with $s_1, s_2 \in \{0, 1\}$, to assign to each of the nodes of the new transition diagram of Fig. 36–2a. To ensure race-free encodings, connected nodes must have adjacent code words — a code with this property is called an **adjacent encoding**. All possible adjacent encodings for the transition diagram of Fig. 36–2a are listed (and numbered) in the table of Fig.36–2b. The circuits calculated from these codes are mutually distinct. Which circuit is the *'best'* (according to some personal criterion you may wish to choose) you will know after calculating and comparing them. Admittedly, this is not very satisfactory.

The encoding method discussed above is characterised by *not adding* new rows to the transition table. This restriction is not often realisable, which means, you will not often be able to use the method as shown here. In a more general approach, you will add surplus rows to the original transition table as we start discussing in the following section.

36.2 Adding Unstable States Intuitively

The transition table of Fig. 36–3a has (intentionally) been constructed so as not to be able to apply the encoding method of the previous section. As the transition diagram of Fig. 36–3b shows, it is not possible to impress on it an adjacent encoding. On the way to such an encoding, suppose we want to break open the connection from node b to node c.

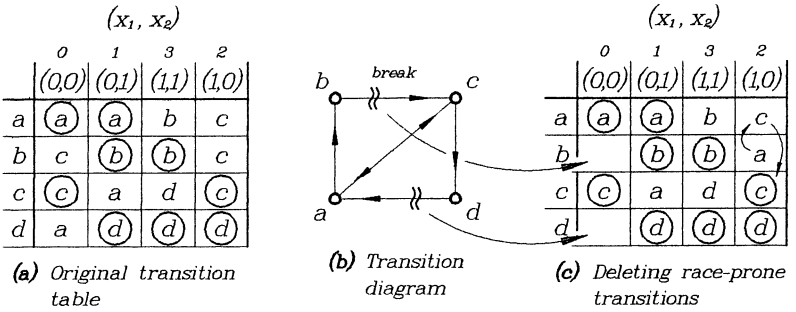


Fig. 36–3 A specially constructed example

Of course we can alter the GOTO-c command of cell $(b, 2)$, Fig. 36–3a, to a GOTO-a command, as shown in Fig. 36–3c, but the incurred multiple state-transition does not suffice to break the connection from node b to node c due to the GOTO-c command in cell $(b, 0)$. To finally break the connection, we must erase the GOTO-c command in cell $(b, 0)$ as indicated in Fig. 36–3c. If we next introduce a new row, state E in Fig. 36–4a, with a GOTO-c command in cell $(E, 0)$, we can enter a GOTO-E command into cell $(b, 0)$ to go from b to c via E. Similarly, after breaking the connection from d to a, Figs. 36–3bc, we can introduce a second row, state F in Fig. 36–4a, so as to go from d to a via F.

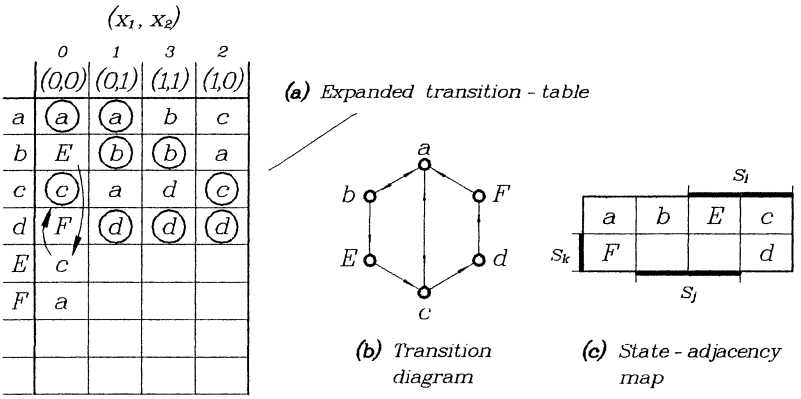


Fig. 36–4 Multiple state transitions in an expanded transition table

It is strongly recommended to draw the transition diagram, Fig. 36–4b, for the expanded transition table of Fig. 36–4a to verify that no odd-sided polygons occur. To assign appropriate codes to the transition diagram, you can enter each node name into a K-map that I refer to as a **state-adjacency map**, Fig. 36–4c. The variables assigned to the perimeter of the map are the state variables s_1, s_2, s_3 in any permutation (hence writing them as s_i, s_j, s_k). The only requirement for the placement of the node names is that their adjacency relationship in the map is the same as in the transition diagram.

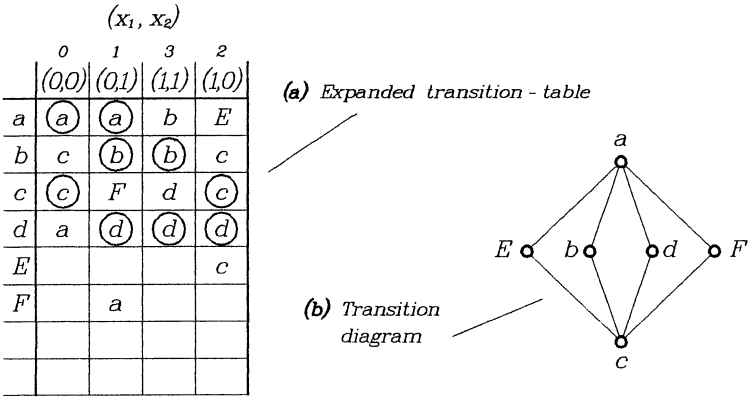


Fig. 36–5 An inappropriate expanded transition table

Another seemingly obvious way to eliminate triangles in the transition diagram of Fig. 36–3b is to break open the diagonal between nodes a and c. This too can be achieved by adding two further rows to the transition table, and altering its entries as shown Fig. 36–5a. The result seems slightly simpler and more symmetric than our first attempt shown in Fig. 36–4a. To verify that we did not inadvertently create new odd-sided polygons, we draw the associated transition diagram, Fig. 36–5b, noting that every polygon contained is four-sided. But, unhappily, this transition diagram cannot be encoded adjacently using only 3 state variables — we would need four. This is usually considered unwarranted which is why the transition table of Fig. 36–5a is rejected.

36.3 On the Number of Adjacent Encodings

An **adjacent encoding** is a binary encoding of a transition diagram such that *the codes of any connected nodes* differ in only one position, i.e., have a hamming distance of 1 ($h_d = 1$). The possible adjacent encodings for the transition diagram of Fig. 36–2a are listed in Fig. 36–2b. To find the possible adjacent encodings for the three binary variables s_1, s_2, s_3 needed

to encode the nodes of the transition diagram of Fig. 36–4b, it is advisable to transform the transition diagram into *state-adjacency map*, Fig. 36–4c. The **state-adjacency map** is a K-map whose input variables (on the perimeter of the map) are the state variables written in any order, i.e., $\{s_i, s_j, s_k\} = \{s_1, s_2, s_3\}$. The state names — in our example, the symbols a, b, c, d, E, F — are entered into the cells so that the adjacency conditions satisfy the transition diagram. By choosing any assignment (s_1, s_2, s_3) for (s_i, s_j, s_k) you can read a valid adjacent encoding from the state-adjacency map. For instance, if $(s_i, s_j, s_k) = (s_1, s_2, s_3)$, then a is assigned $(0, 0, 0)$, b is encoded as $(0, 1, 0)$, E as $(1, 1, 0)$, etc.

How many adjacent encodings can be developed for the three binary state variables s_1, s_2, s_3? First of all, the elements that comprise the range of a variable can be written in any permutation or order. The number of permutations for a binary variable is $2! = 2$. The number of the combined permutations of all three binary state variables is the product of the individual permutations, i.e., $(2!) \times (2!) \times (2!) = 2 \times 2 \times 2 = 2^3$. Finally, the ordering of the 3 state variables around the perimeter of the state-adjacency map is arbitrary. The number of these orderings or permutations of the 3 state variables is $1 \times 2 \times 3 = 3!$. The total number of adjacent encodings is thus $2^3 \times 3! = 8 \times 6 = 48$.

Generally, the number of adjacent binary encodings of a transition diagram of s nodes is

$$2^p \times (p!) \tag{36.1}$$

where $p = \lceil \log_2 s$ is the (smallest) number of binary state-variables $s_1, \ldots, s_p$ needed.

36.4 One-Hot Code

The encoding techniques of Sections 36.1 and 36.2 are neither systematic nor can they be systematised. But they are useful in demonstrating the basic properties of all encoding techniques — altering unstable states and/or appending surplus states to the transition table. More often than not, the result is a transition table in which the number of transitions differ considerably when switching from one stable state to another. As each state transition takes a certain unit of time, the speed with which the circuit reacts to each new input event depends on the number of state transitions needed to reach the next stable state.

A one-hot encoding ensures that two, but only two, state transitions are needed to go from one stable state to another so that the reaction time of the circuit is uniformly *'two units of time'* whenever a new input event is applied. You could call this encoding a **dual transition time (DTT)** encoding in contrast to the *single transition time* (STT) to be touched on later. Most importantly, the one-hot DTT-encoding needs no ingenuity, it

is systematic, and can thus be programmed. One disadvantage is the large number of internal latches needed — one for each row of the transition table (hence making it important to use a completely merged transition table).

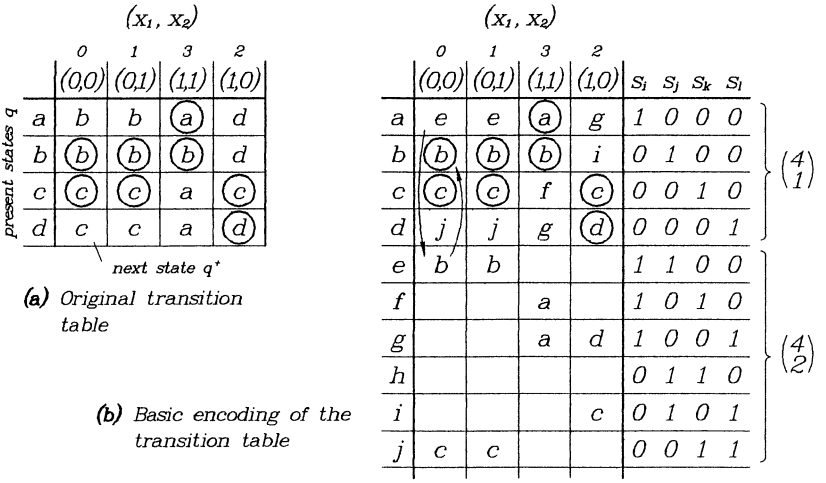


Fig. 36–6 A one-hot DTT-encoding for a given transition table

We prepare the encoding of our example transition-table, Fig. 36–6a, by repeating its s (in our case 4) rows *without* the unstable GOTO-commands, and adding an unique one-hot code-word of length s to each row (a one-hot code-word is comprised of 0s and a single 1 — the *'hot'* digit). The next step is to append $\binom{s}{2}$ rows to the transition table to accommodate all possible two-hot codes — see Fig. 36–6b.

Let us now see how to fill in the transition table as it was prepared in the paragraph above. We first concentrate our attention on cell $(a, 0)$ which contains an unstable b in the original transition table, Fig. 36–6a, but is still empty in the newly prepared table. Going from stable a, cell $(a, 3)$, to the stable b of $(b, 0)$ in the newly prepared table is achieved by entering the name of the row that contains the **scalar disjunction** (see Section 19.4) or the *scalar addition*, if you prefer, of the one-hot codes of rows a and b — $(1,0,0,0) \vee (0,1,0,0) = (1,1,0,0)$. This code you will find in row e of Fig. 36–6b — so e is entered into the cell of $(a, 0)$ while b is entered into cell $(e, 0)$. This explanation should enable you complete the remaining unstable GOTO-commands to obtain Fig. 36–6b.

36.5 Safe but Slow Standardised Encodings

The endcoding discussed here employs the smallest possible number of state variables needed to guarantee that no critical races occur. But, most importantly, it allows us to encode *any* transition table of a given size *quite*

irrespective of the placement of the GOTO-commands. That is, you can apply one and the same encoding scheme to any transition table for whose size the scheme was developed — in this sense it is a *standardised encoding scheme.* But as you can no doubt imagine, this considerable advantage comes at a price.

To introduce the method, consider encoding the fully merged transition table of Fig. 36–7a. We can encode any transition table of four rows so that no critical races occur by using three state variables. Instead of encoding the state variables intuitively (as in Section 36.2), it is possible to do this systematically. You **first** duplicate the states of the original transition table taking duplicate rows to be equivalent[1] — the duplicate rows are renamed, in our example (Fig. 36–7b) from lower-case to like upper-case.

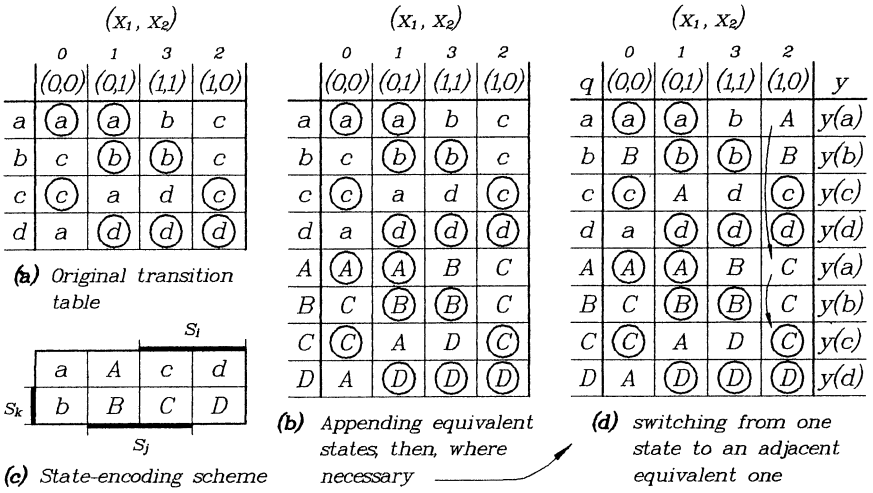


Fig. 36–7 Example employing Huffman's encoding scheme

The **second** step is to introduce the adjacency of equivalent rows by encoding them according to a special scheme developed by **Huffman** [1955]. For our example this encoding scheme is shown in the K-map of Fig. 36–7c. Each cell of the K-map is characterised by a triple (s_i, s_j, s_k) of binary state variables. A row name entered into cell (s_i, s_j, s_k) is assigned (s_i, s_j, s_k) as state encoding. For instance, in Fig. 36–7c cell $(s_i, s_j, s_k) = (0, 0, 0)$ contains row name a, therefore, state a in Fig. 36–7b (as well as in Fig. 36–7c) would be assigned the code $(0, 0, 0)$ — this numeric encoding I leave to you. State A, equivalent to state a, is placed in cell $(0, 1, 0)$ (of Fig. 36–7c) adjacent to the cell for a.

[1] In this context let us understand states to be equivalent if they cause identical outputs for any sequence of input events.

We make use of the adjacent encoding of states pictured in Fig. 36–7c in the **third** step by ensuring that all transitions occur between adjacent states. Consider, for instance, the GOTO-c command in row a of Fig. 36–7b. Row c, to which we are required to go, is not adjacent to row a (see the state-encoding scheme of Fig. 36–7c), but row A (equivalent, and thus indistinguishable in its behaviour from row a) *is* adjacent to row c. Thus, the GOTO-c command of cell $(a, 2)$ is replaced by the GOTO-A command — as shown in Fig. 36–7d. Following the GOTO-A command in cell $(a, 2)$, we are sent to cell $(A, 2)$ in which we encounter a GOTO-C command — a command we accept to be indistinguishable to a GOTO-c.

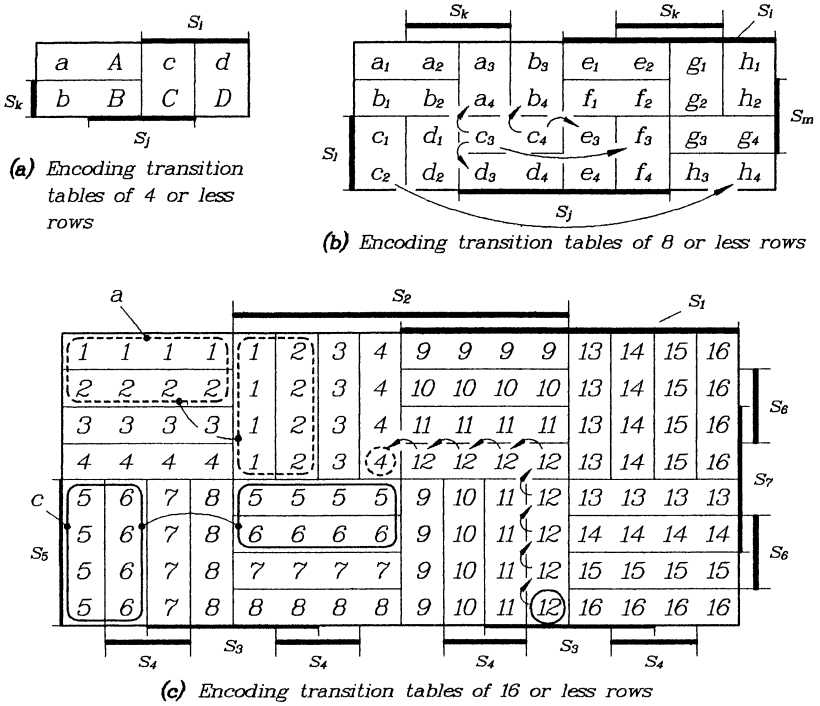


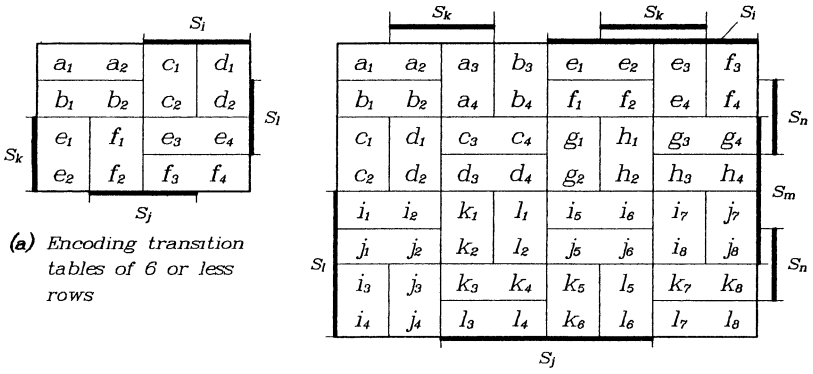
Fig. 36–8 Huffman's standardised encoding-schemes

The encoding scheme used in the above example is repeated in the K-map in Fig. 36–8a in a way that emphasizes the adjacency of equivalent states. This makes it easier to generalise the scheme for the *five* state variables needed to encode a transition table of up to 8 rows in a standardised way free of critical races shown in Fig. 36–8b. Let us briefly ponder what this encoding scheme implies and how it works. Given an original transition table of 8 rows, you may have to duplicate the table up to 3 times obtaining 4 identical sub-tables with an overall number of 32 rows. According to the encoding scheme of Fig. 36–8b, think of the rows of the first sub-table being

named a_1 to h_1, those of the second sub-table as a_2 to h_2, with a_3 to h_3, and a_4 to h_4 for the third and fourth sub-tables. We must realise that all rows marked with the same letter (irrespective of its index) are equivalent, i.e., no experiment would enable us to say in which of the equivalent states we are. For instance, saying we are in state c would mean that we are in one of the rows c_1, c_2, c_3, or c_4.

In a *general* encoding scheme (and that of Fig. 36–8b is such a scheme) we would want to go from a given state to *any* other adjacent state, i.e., by changing the value of only a single state variable. For instance, going from state c to state a is achieved if we can go from any of the rows c_1, c_2, c_3, c_4 to any of the rows a_1, a_2, a_3, a_4 by changing the value of only a single state variable. In the encoding scheme of Fig. 36–8b this is possible for the transition from c_3 to a_4 (see the appropriate arrow in the figure). The arrows in Fig. 36–8b show most possible adjacent transitions from state c to some other state.

The **general rule** to apply the encoding scheme of Fig. 36–8b is: In any current row of an extended transition table (consisting of identical Sub-tables) you change any unstable GOTO-command to a re-indexed version of the current row's name that (according to the encoding scheme) is adjacent to the GOTO-command to be changed.



(a) *Encoding transition tables of 6 or less rows*

(b) *Encoding transition tables of 12 or less rows*

Fig. 36–9 Huffman's standardised $\frac{3}{4}2^{p_{min}}$-encoding-schemes

The price we pay for keeping the upper bounds for the number of state variables as small as possible is a circuit that may react excessively slowly in reaching a new stable state. To emphasize this aspect, consider the encoding scheme of Fig. 36–8c in which a worst case scenario is depicted. If we were in the row of one of the *eight* sub-tables of the extended transition table, characterised by the encircled 12 of Fig. 36–8c, and had to GOTO-4, we would have to pass from one 12-entry to an adjacent one until we came to the 12-entry adjacent to a 4-entry. In the example shown, 8 successive

changes of the value of some secondary variables must occur before we reach the stable state desired (a row number 4). In cases like this, the reaction of the circuit may be undesireably or unacceptably slow.

Clearly any reduction in the number of state variables needed is welcome. Again it was **Huffman** [1955] who pointed out when such a reduction was possible while still retaining a standardised encoding scheme. To encode an original transition table of 4 rows in a standardised manner so that no critical races occur, 3 state variables are needed, while to achieve such an encoding for an original transition table of 8 rows 5 state variables are needed. Huffman found that for an original transition table with a maximum number of rows exactly between a 4-row and an 8-row table — that is, a transition table with 5 or 6 rows — a standard scheme employing only 4 state variables suffices. An original transition table with 6 rows has $\frac{3}{4}$ the number of rows you can encode (not necessarily free of critical races) with 3 state variables ($6 = \frac{3}{4}2^3$). This is used to characterise this standard encoding scheme as a $\frac{3}{4}2^{p_{min}}$-scheme. Using this scheme, an original transition table that has 9, 10, 11, or 12 rows will never need more than 6 state variables to be encoded free of critical races.

36.6 Safe and Fast Standardised Encodings

The advantages and drawbacks of the standardised encoding schemes discussed in this section both come in extremes. On the one hand, this encoding scheme enables the design of the fastest circuits possible: Every GOTO-command in a transition table leads directly to a stable state, only a single state variable changing only one of its binary values. These encodings are thus called **STT** or **single transition time** encodings.

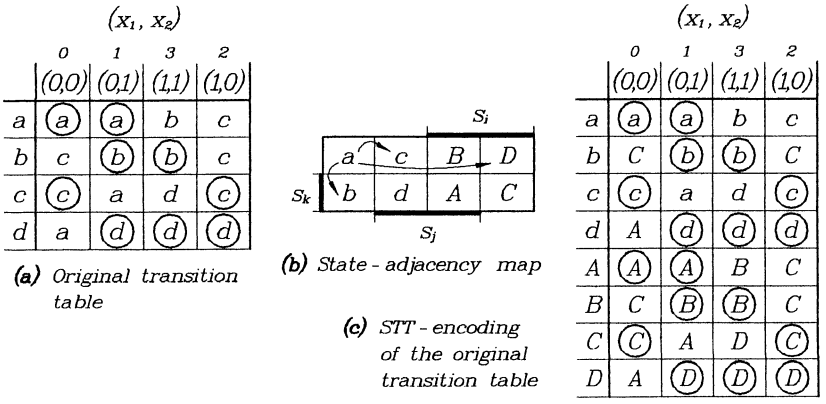


Fig. 36–10 An example for a STT-encoding

On the other hand, the formidable drawback is a truly exponential growth of the number of state variables.

Huffman's [1955] encoding scheme for an original transition table of 3 or 4 rows is shown in Fig. 36–10b. Note that any given state (say, state a) is either adjacent to any other state (in our examples, to states b and c) or to an equivalent of another state (in the example shown, to state D — which is equivalent to state d). The application of the STT-encoding scheme is the same as discussed in the previous section.

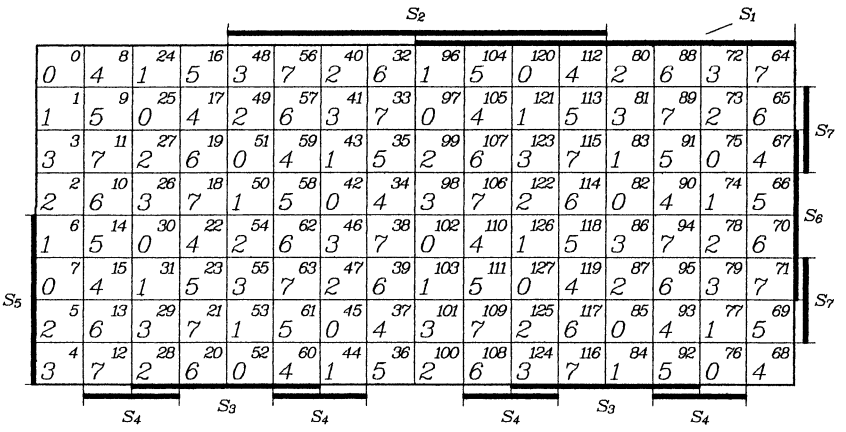


Fig. 36–11 STT-encoding for an original transition table of 8 or less rows

The numbers p (from 0 to 7) entered into the cells abide by the following remarkable property.

$$p_1 = s_1 \oplus s_2 \oplus s_3 \oplus s_4,$$
$$p_2 = s_1 \oplus s_2 \oplus s_5 \oplus s_6,$$
$$p_3 = s_1 \oplus s_3 \oplus s_5 \oplus s_7,$$
$$p = 2^2 \times p_1 + 2^1 \times p_2 + 2^0 \times p_3.$$

STT-encodings for small transition tables (of 3 or 4 rows) are easy and economical to achieve. But, as soon as you want to encode a transition table of 5 to 8 rows, you need to employ 7 (in words: seven) state variables. The encoding scheme you need to follow is shown in Fig. 36–11. Note that this K-map has 128 cells corresponding to an expanded transition table of 128 rows. Working with K-maps of this size is not a happy prospect. It was **Tracey** [1966] who analysed the possibility of obtaining STT-encodings when allowing non-critical races hereby enabling a certain reduction in the number of state variables needed. The three methods he developed are basic if you want to follow up on STT-encoding.

36.7 Number of Required State Variables

Let p denote the number of binary state variables used to encode the s states of a transition table. The smallest number of binary variables, p_{min}, needed to encode s states must abide by $s < 2^{p_{min}}$ so that

$$p_{min} = \lceil \log_2 s. \tag{35.10}$$

The other codes (as are: one-hot, safe-but-slow, and safe-and-fast) need more or even considerably more state variables as shown in Fig. 36–12.

number of states s	p_{min}	one-hot	safe-but-slow	(STT) safe-and-fast
2	1	1	1	1
3 4	2	3 4	2 3	3
5 or 6 7 or 8	3	5 or 6 7 or 8	4 5	7
9 to 12 13 to 16	4	9 to 12 13 to 16	6 7	15
17 to 24 25 to 32	5	17 to 24 25 to 32	8 9	31
	$\lceil \log_2 s$	one variable per row	$2p_{min} - 2$ $2p_{min} - 1$	$2^{p_{min}} - 1$

Fig. 36–12 Number of state variables needed per code

For instance, if you want to encode a transition table of 8 rows (i.e., states), you will need at least 3 state variables according to the p_{min}-column of Fig. 36–12. The one-hot code requires 8 state variables which means you would be working with a prohibitively large expanded transition table of $2^8 = 256$ rows. A safe-but-slow code would require 5 state variables leading to an expanded transition table of $2^5 = 32$ rows. The safe-and-fast code (the STT-code) would employ 7 variables and a still excessively large expanded transition table of 128 rows. These considerations speak strongly for the catenation method of Chapters 29 and 30.

Bibliography

A. Logic and more

Asser, G.: *'Einführung in die mathematische Logik'.* B. G. Teubner Verlagsgesellschaft. Volume I, *'Aussagenkalkül'*, 1982. Volume II, *'Prädikatenkalkül der ersten Stufe'*, 1975.

Frege, G.: 'Begriffsschrift, *a formula language, modelled upon that of arithmatic, for pure thought*'. In: van Heijenoort, J. (Editor): *'From Frege to Gödel — A Source Book in Mathematical Logic, 1879-1931.* Harvard University Press, 1977.

Hofstadter, D. R.: *'Gödel, Escher, Bach — An Eternal Golden Braid'.* Penguin Books, 1979.

Popper, K. R.: *'The Logic of Scientific Discovery'.* Hutchinson & Co (Publishers), 1977.

Shoenfield, J.: *'Mathematical Logic'.* Addison-Wesley Publishing Company, 1967.

B. Mathematics

Hilbert, D., P. Bernays: *'Grundlagen der Mathematik'.* Springer-Verlag. Volume I, 1968. Volume II, 1970.

Iverson, K. E.: *'Algebra — An Algorithmic Treatment'.* APL Press, 1977.

Mac Lane, S.: *'Mathematics — Form and Function'.* Springer-Verlag, 1986.

Kleene, S. C.: *'Introduction to Metamathematics'.* North-Holland Publishing Company, 1971.

C. Switching Theory and Logic Design

Almaini, A. E. A.: *'Electronic Logic Systems'.* Prentice Hall, 1986.

Brzozowski, J. A., M. Yoeli: *'Digital Networks'.* Prentice Hall, 1976.

Brzozowski, J. A., C.-J. H. Seger: *'Asynchronous Circuits'.* Springer-Verlag, 1995.

Caldwell, S. H.: *'Switching Circuits and Logical Design'.* John Wiley & Sons, 1958.

Clare, C. R.: *'Designing Logic Systems using State Machines'.* McGraw-Hill Book Company, 1973.

Dietmeyer, D. L.: *'Logic Design of Digital Systems'.* Allyn and Bacon, 1971.

Harrison, M. A.: *'Introduction to Switching and Automata Theory'.* McGraw-Hill Book Company, 1965.

Katz, R. H.: *'Contemporary Logic Design'.* Benjamin/Cummings Publishing Company, 1994.

Kohavi, Z.: *'Switching and Finite Automata Theory'.* McGraw-Hill Book Company, 1970.

Krieger, M.: *'Basic Switching Circuit Theory'.* MacMillan Company, 1969.

McCluskey, E. J.: *'Logic Design Principles'.* Prentice-Hall, 1986.

Muroga, S.: *'Logic Design and Switching Theory'.* John Wiley & Sons, 1979.

D. Selected Papers

Ashenhurst, R. A.: *'The Decomposition of Switching Functions'.* Proc. of an International Symposium on the Theory of Switching, April 2–5, 1957. Annals of the Computation Lab. of Harard University, vol. 29, pp. 74–116, 1959.

Huffman, D. A.: *'The Synthesis of Sequential Switching Circuits'.* Jour. of the Franklin Institute, vol. 257, pp. 161–190, March 1954 and pp. 275–303, April 1954.

Huffman, D. A.: *'A Study of Memory Requirements of Sequential Switching Circuits'.* Massachusetts Institute of Technology, Research Laboratory of Electronics, Technical Report No. 293, April 1955.

Huffman, D. A.: *'The Design and Use of Hazard-Free Switching Networks'.* Journal of the Association of Computing Machinery, vol. 4, pp. 47–62, January 1957.

Karnaugh, M.: *'The Map Method for Synthesis of Combinational Logic Circuits'.* Trans. AIEE, part I, vol. 72, no. 9. pp. 593–599, 1953.

McCluskey, E. J.: *'Minimization of Boolean Functions'.* Bell System, Technical Journal, vol. 35, no. 6, pp. 1417–1445, 1956.

Mealy, G. H.: *'A Method for Synthesizing Sequential Circuits'.* Bell System, Technical Journal, vol. 34, no. 5, pp. 1045–1079, 1955.

Medvedev, I. T.: *'On a Class of Events Representable in a Finite Automaton'.* Avtomaty, Moscow, pp. 385–401, 1956. English translation in MIT Lincoln Laboratory Group Report **34**–73, June, 1958.

Moore, E. F.: *'Gedankenexperiments on Sequential Machines'.* In *'Automata Studies'.* Princeton Univerity Press, 1956.

Quine, W. V.: *'The Problem of Simplifying Truth Functions'.* American Mathematical Monthly, vol. 59, pp. 521–531, 1952.

Quine, W. V.: *'A Way to Simplify Truth Functions'.* American Mathematical Monthly, vol. 62, pp. 627–631, 1955.

Shannon C. E.: *'A Symbolic Analysis of Relays and Switching Circuits'.* Trans. AIEE, vol. 57, pp. 713–723, 1938.

Shegalkin I. I.: The French title of the Russian original is **Gégalkine I. I.:** *'Sur le calcul des propositions dans la logique symbolique'.* Mat. Sbornik, vol. 34, pp. 9–28, 1927.

Tracey J. H.: *'Internal State Assignment for Asynchronous Sequential Circuits'.* IEEE Trans. on Electronic Computers, vol. EC-15, pp. 551–560, 1966.

Unger S. H.: *'Hazards and Delays in Asynchronous Sequential Circuits'.* IRE Trans. on Circuit Theory, vol. CT-6, pp. 12–25, 1959.

Veitch E. W.: *'A Chart Method for Simplifying Truth Functions'.* Proc. Pittsburgh Assoc. Comp. Mach., May 1952.

Vingron P.: *'Coherent Design of Asynchronous Circuits'.* IEE Proc., vol. 130E, pp. 190–201, 1983.

Index

Keywords in this index are referenced by page (e.g., 370), by a group of pages (e.g., 370–374), by a table or figure (e.g., Fig. 30–3), or by a definition, theorem, or formula number (e.g., (30.1)). The chapter numbers (as the '30' in Fig. 30–3 and in (30.1)) can be found in the header of every page.

0-nucleus, 186
1-nucleus, 186

A

adjacency vector, (19.3)
adjacent encoding
    number of adjacent encodings, (36.1)
analysis, 153, 159
argument, 86
    invalid argument, 88
    see conclusion, premise
    sound argument, 88
    unsound argument, 87
    valid argument, 86
Aristotle, 129
Ashenhurst, 231
Asser, 78
associativitiy, see logic connectives
automaton, 311–312

B

BCD code, 163, 184
Bernays, 78
binary encoding of states, 347
    see state encoding
Boole, 129
Brzozowsky, 245, 248, 249

C

Caldwell, 343, 398
Cantor, 57
cardinal number, 28
Cartesian product, 29

catenation
... diagram, 234, 300
... of functions, 233–234
... model, see sequential circuit
causality, 326, 327
cluster
... of flow tables, 326
... of word-recognition tables, 326
co-domain, 32
column vectors, 326
combinational circuit, 3, 6
combinations, 156–157
commutativity, see logic connectives
complement, 184
completely specified circuit, 7
composing a combinational circuit, 231
composing a latch, 300–302
conclusion, 86
see argument, necessary, premise
condition, 313
input condition, 259
stable condition, 313
stationary condition, 262
transient condition, 262
connectives, see logic connectives
consensus, 191, 193
... theorem, (18.12b)
definition, (19.1), (19.7)
covers, (17.21), (18.3)
0-cover, 178
1-cover, 178
full cover, (17.21a), (18.3a), (20.3a)
irredundant cover, (17.21b), (18.3b), (20.3b)
minimal cover, (17.21c), (18.3c), (20.3c)
see table of possible covers

D

delay
... and the latch, 258–263, 264–266
... and the sequential circuit, 339, 388
inertial delay, 245, 258
DeMorgan theorems, (9.26), (9.27)
all dyadic DeMorgan theorems, Fig. 10–10
generalised DeMorgan theorems, Fig. 10–11, 118

demultiplexer, 162
determinateness, 327
general determinateness, (30.5)
non-determinate, (30.7)
primitive determinateness, (30.4)
diagram of successive events (DSE), 4, 5
see verification-DSE
distributivity, see logic connectives
domain, 32
don't care event, 7
see incompletely specified circuits
DSE, see diagram of successive events
duality, 53–56
... of elementary logic functions, Fig. 5–11
... of logic formulas, (5.17)
... of logic functions, (5.15), (5.16)
... theorem, (5.18)

E

edge, 319
directed edge, 319
input edge, 319
output edge, 319
elementary sub-tree, 321
encoding problem, 399
equivalence set, see equivalent
equivalent, 176
0-equivalent, (17.16)
1-equivalent, (17.16)
memory equivalents, (24.5c), (25.1b)
reset equivalents, (24.5b), (25.1b)
set equivalents, (24.5a), (25.1b)
evaluation formulas
... for don't cares, (18.4)–(18.7)
... for NOT-AND, (21.14)
... for NOT-OR, (21.15)
... for reduced K-maps, (20.9)–(20.12)
exclusion evaluation formulas, (17.31)
inclusion evaluation formulas, (17.30)
event, see input event, output event
excess-3 code, 184
exclusion, (17.29), (20.6), (20.7)

F

factorial, 157
false, 6
falsifying a scientific theory, 385
falsity, 10, 89
feedback, 260
    ... within a sequential circuit, 338–340, 388–389
    closed feedback loop, 262
filtered output variable 319, 320
flipflop
    D-flipflop, 369
    JK-flipflop, 355–359
    T-flipflop, 349–354
flow table (FT), 311
    definition, (29.1)
    Mealy flow table, 340–342
    Mealy type flow table, 341
    Moore flow table, 340
    organisation of a flow table, Fig. 29–1
    primitive flow table, 314
    see Huffman's use of the flow table
formal systems, 89
formation rules, 79
    see logic expressions
Frege, 129
FT, see flow table
function
    ... in predicate logic, 122, 123
    elementary logic function, (4.4), Fig. 4–3
    elementary switching function, (3.17), Fig. 4–1
    general function, 31–33, 32
    hinge function, 231, 236
    logic function, (4.3)
    mapping of a function, 32
    output function, 336–341
    partial function, 235
    see generic function
    switching function, 33, (3.15)
    transition function, 336–341

G

gate-defining set, (21.19a)
gates, 9, 40–45
    ... with an inverted input, 9

... with an inverted output, 9
... with multiple inputs, 42
basic gates, 40
behaviour of gates, Fig. 1–6
connecting gates, 9
derived gates, 43
dyadic gates, 40, Fig. 4–5
monadic gates, 40, Fig. 4–4
generalisation, 125
generic function, 231
see prelogic
generic latch, 271
glue, 377
grafting, 379
Gray code, 63

H

Hamming
... distance, 35
... weight, 35, (19.2)
Harrison, 336
hazard, 242
0-hazard, 243
1-hazard, 243
single-input hazard, 243
static hazard, 242
stationary hazard, 242
hazard-detecting formulas, (23.7)
hazard-equivalent formulas, (23.8)
Hilbert, 78
hinge latch, 300
hinge set, 236, 301
Huffman, 310, 311, 318, 335
Huffman's conjectures, (35.2)–(35.9)
Huffman's structure of a sequential circuit
... when using the Mealy model, Fig. 35–2e
... when using the Medvedev model, Fig. 35–2c
... when using the Moore model, Fig. 35–2d
Huffman's synthesis procedure, (35.1)
Huffman's use of the flow table, Fig. 35–3

I

identity theorems, (13.30), (13.31)
iff, 8

inclusion, (17.27), (20.4), (20.5)
incompletely specified circuit, 7, 183–185
induction
    mathematical induction, 119
    perfect induction, 81
input event, 5
    asserted input event, 6
    memorising input event, 11, 256
    multiple input event, (17.2)
    next input event, 312, 313
    non-adjacent successive input events, 317
    present input event, (24.1b), 312
    previous input event, (24.1b)
    resetting input event, 255
    see (input) condition, output event
    setting input event, 255
    successive input events, (24.1)
input word, 319
    empty word, 319
    essential input word, 377
    length of an input word, 319
    reference input word, 319
instantiation, 125

K

K-map, see Karnaugh map
K-set, see Karnaugh set
Karnaugh map (K-map), 3, 12
    ... with multiple inputs, Fig. 6–4
    definition, (6.9)
    dimension of a Karnaugh map, 62
    see reduced Karnaugh maps
Karnaugh map, its usage
    circuit specification, 66
    proving a logic equivalence, 70–72, 84
    proving a logic implication, 72–74, 84
    reading conjunctive formulas, 68–70
    reading disjunctive formulas, 67, 68
    representing dual functions, 74–76
Karnaugh set, (K-set)
    adjacent Karnaugh sets, 193, (19.7)
    anchored Karnaugh sets, (21.18), (21.20)
    definition, (6.12)
    general definition, (17.5)

maximum Karnaugh set, (6.9a)
theorem on disjoint Karnaugh sets, (19.6)
theorem on non-disjoint Karnaugh sets, (19.4)
using maximum Karnaugh sets, (6.10), (6.11)

L

latch symbols, see symbolic latches
latches without feedback
spool valve, 264
thyristor, 275
latch, 255
definition, (25.1)
D-latch, Fig. 27–12, 302–305, 353, 358, 363, 369, 370–376
evaluation, (25.3)–(25.6)
hazard-free latches, 282–285
identity of latches, (25.7)
inverting a latch, 275–278
isomorphic latches, 274
minimised latches, 279–280
see generic latch, specifying a latch
SR-latch, Fig. 27–13, Fig. 28–7
state variable of a latch, 260
leaf, see (end) node
logic
negative logic, 38
positive logic, 38
see predicate logic
see propositional logic
logic connectives
associative logic connectives, (10.2), Fig. 10–2
commutative logic connectives, (10.1), Fig. 10–1
convention of adhesion, 80
convention of separation, 80
definitions, Fig. 4–3, Fig. 8–1
distributive logic connectives, (10.3), (10.4), Fig. 10-8, Fig. 10–9
main logic connective, 80
sets of necessary and sufficient connectives, Fig. 15–5
logic equivalence, 8, 83
logic expression, 79–81
atomic logic expression, 122, 123
compound logic expression, 124
formation rules for logic expressions, 79
logic formula, 3, 8
single-variable formulas, 103–104

logic implication, 83
logic reasoning, 86
see modus ponens
logic variable, 3, 6, 8, 12
definition, (1.2)
negative logic variable, 38
positive logic variable, 38
primed logic variable, 246

M
maxterm function, 49
generalised definition, (17.10)
maxterm, 49–50
... formula, (5.10)
definition, (5.6)
general maxterm formula, (17.13)
McCluskey, 192, 201
Mealy, 335
... model, see sequential circuit
Medvedev, 335, 336
... model, see sequential circuit
memorisation and stationarity, (24.2)
memorising
... by feedback, 258
... without feedback, 263, 275
memory function, 255
merger diagram, 342
merging, 342–344
metatheorem, 88
minimisation
... by Karnaugh map, (18.8)
... by the iterative consensus algorithm, (19.9)
... when considering don't cares, 204–206
aspects of minimisation, 182–183
Quine's methods, (18.9), (18.12)
minterm function, 47
generalised definition, (17.6)
minterm, 46–48,
... formula, (5.4)
definition, (5.2)
general minterm formula, (17.8)
module, 167
modus ponens, (8.16)
modus tollens, (35.1)

Moore, 335
... model, see sequential circuit
multiplexer, 142, 162

N
NC, see relay (normally closed)
n-cube, 34
necessary, 86, 160, 370
see conclusion
negation, rule of, 107
NO, see relay (normally open)
node, 319
end node, 319
internal node, 319
root, 319
normal forms
canonical normal forms, (14.1), (14.2), 181
conjunctive normal forms, 53, (14.2)
disjunctive normal forms, 51, (14.1)
expansion to normal forms, 191
Shegalkin normal forms, (15.1), (15.2)
NOT-AND
designing multilevel NOT-AND circuits, (21.19)
designing two-level NOT-AND circuits, (21.14)
NOT-OR
designing multilevel NOT-OR circuits, 228
designing two-level NOT-OR circuits, (21.15)

O
one-hot code, 403
ordered
... n-tuple, 28
... pair, 28
oscillation, 389
output event, 5
output vectors, see column vectors

P
partition, 176
Petrick, 193
Popper, 385
predicate logic, 121
... with identity, 123
axioms of predicate logic, 130
rules of deduction for predicate logic, 130–132

predicate, 121, 124
degree of a predicate, 121
dyadic predicate, 121
identity predicate, 123
monadic predicate, 121
prelogic, 271, 301
premise, 86
see argument, conclusion, sufficient
prime implicant, 177
prime set, (17.20)
essential prime set, (18.1)
redundant prime set, (18.2)
see table of prime sets
proposition, 6
atomic proposition, 122
compound proposition, 122
false proposition, 12
true proposition, 12
propositional form, 6
see logic variable
propositional logic
axiomatic approach to propositional logic, 89–91
axioms of propositional logic, 90
pulse, 243

Q

quantification, 126
... theorems, (13.44)–(13.68)
quantifier, 124–125
existential quantifier, 125
scope of a quantifier, 127
universal quantifier, 125
Quine, 190–192

R

race, 393
... analysis, 394
critical race, 393
non-critical race, 393
range of a function, 32
reduced Karnaugh maps, 207–218
... for specifying latches, 269
evaluation, (20.9)–(20.12)
relation, 32

relay, 18
normally closed (NC) relay, 19
normally open (NO) relay, 19
replacement, rule of 85
replication, 126
root, see node
runt pulse, 243

S
salient, 243
sampling circuit, 363–366
scalar disjunction, 202, 404
Seger, 245
sequential circuit, 3, 6
asynchronous sequential circuits, 310
catenation model of a sequential circuit, (31.4), 340, Fig. 31–3
Mealy model of a sequential circuit, (31.2)
Medvedev model of a sequential circuit, (31.3)
Moore model of a sequential circuit, (31.1)
Moore model versus catenation model, 342
synchronous sequential circuits, 310
set, 27
... of unassigned input events, 185
complement of a set, (6.5)
element of a set, 57, (6.1)
identity of sets, (6.4)
intersection of sets, (6.6)
null set, 59
power set, 59
prime set, see prime set
subset of a set, (6.3)
union of sets, (6.7), (11.20)
universal set, 58
Shannon's expansion theorems, (14.17), (14,18)
Shegalkin polynomial, (15.1)
dual Shegalkin polynomial, (15.8)
signal, 14
binary signal, 4, 14
input signal, 4
output signal, 4
see runt pulse
skew signal, 243
single transition time (STT) encoding, 408

specifying a circuit
... by diagram of successive events, 4–5
... by flow table, 313–317
... by K-map, 66, 269, Fig. 29–6
... by table of asserted events, 5–7
... by word-recognition tree, Fig. 29–6
... in plain prose, 3–4
specifying a latch, 269
stability, (31.4c)
and feedback, 338–340
state
... encoding, 347, 398–410
... minimisation, see merging
... variable, 260, 311
internal state, 260, 268, 311
next state, 312
present state, 312
stable state, 314
STT, see single transition time
subject, 121
sub-sequence notation, 378
substitution, 12
rule of substitution, 85
see replacement
sufficient, 86, 160, 370
see premise
switching algebra, 124
switching circuit
drawing a switching circuit, 9
well behaved switching circuit, 5, 255
symbolic latches, 291–293
... with a common event, Fig. 27–10, Fig. 27–11
... with complementary outputs, Fig. 27–8, Fig. 27–9
standard symbols, 297–299
synchronising circuit, 366–369
synthesis, 3, 153, 159

T

table of asserted events (TAE), 3, 5
brief form of a TAE, 6
explicit form of a TAE, 6
table of output vectors (TOV), 330
table of possible covers, 202
table of prime sets, 201

table of verification sub-sequences, 380
TAE, see table of asserted events
tautology, (8.1)
    proving a tautology by Karnaugh map, 84
    proving a tautology by truth table, 83
term, 123
theorems, 89
thyristor as latch, 275
TOV, see table of output vectors
Tracey, 409
transition diagram, 399
transition table, 338
    partial transition table, 395
    see (transition) function
transistor, 23
    CMOS, 23
    IG-FET, 23
transmission, 19
triggering circuit, 360
    adaptive triggering circuit, 361
trough, 243
true, 6
truth table, 81–82
truth, 10, 89

U

universe of subjects, 122

V

valve
    active valve, 16
    normally closed (NC) valve, 15
    normally open (NO) valve, 15
    passive valve, 17
    shuttle valve, 18
    spool valve as latch, 264
    spool-and-sleeve valve, 15
variable
    bound variable, 127
    free variable, 126
    numeric variable, 14
    see logic variable
Veitch, 57
Venn diagram, 58, Fig. 6–1

verification-DSE, 370
... for a D-latch, 370-376, Fig. 34–1d
pruning a verification-DSE, 376
verifying a scientific theory, 385
Vingron, 310

W

Well behaved circuit, see switching circuit
well formed strings, see logic expressions
word-recognition table (WRT), 325
composite word-recognition table, 345–348
primary word-recognition table, 326, 338
word-recognition tree, 3, 11, 318
WRT, see word-recognition table

Y

Yoeli, 248, 249