

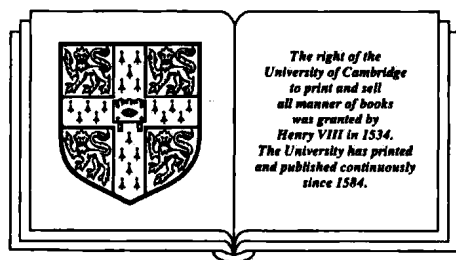
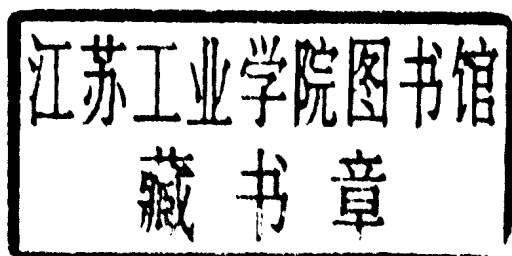
Theoretical foundations of VLSI design

Edited by
K. McEvoy and
J.V. Tucker

Cambridge Tracts
in Theoretical
Computer Science 10

THEORETICAL FOUNDATIONS OF VLSI DESIGN

Edited by
K. MCEVOY
University of Leeds
and J. V. TUCKER
University College of Swansea



CAMBRIDGE UNIVERSITY PRESS

Cambridge

New York Port Chester Melbourne Sydney

PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

CAMBRIDGE UNIVERSITY PRESS

The Edinburgh Building, Cambridge CB2 2RU, UK
40 West 20th Street, New York NY 10011-4211, USA
477 Williamstown Road, Port Melbourne, VIC 3207, Australia
Ruiz de Alarcón 13, 28014 Madrid, Spain
Dock House, The Waterfront, Cape Town 8001, South Africa

<http://www.cambridge.org>

© Cambridge University Press 1990

This book is in copyright. Subject to statutory exception
and to the provisions of relevant collective licensing agreements,
no reproduction of any part may take place without
the written permission of Cambridge University Press.

First published 1990

First paperback edition 2003

A catalogue record for this book is available from the British Library

ISBN 0 521 36631 3 hardback

ISBN 0 521 54565 X paperback

List of contributors

A. Cohn

**Computer Laboratory, University of Cambridge, Pembroke Street, Cambridge,
CB2 3QG.**

P. M. Dew

**Centre for Theoretical Computer Science and School of Computer Studies,
University of Leeds, Leeds, LS2 9JT.**

P. E. Dunne

Department of Computer Science, University of Liverpool, Liverpool, L69 3BX.

M. Gordon

**Computer Laboratory, University of Cambridge, Pembroke Street, Cambridge,
CB2 3QG.**

N. A. Harman

**Department of Mathematics and Computer Science, University College,
Swansea, Singleton Park, Swansea, SA2 8PP.**

E. S. King

School of Computer Studies, University of Leeds, Leeds, LS2 9JT.

K. McEvoy

**Centre for Theoretical Computer Science and School of Computer Studies,
University of Leeds, Leeds, LS2 9JT.**

M. Sheeran

Department of Computing Science, Glasgow University, Glasgow, G12 8QQ.

List of contributors

J. V. Tucker

Department of Mathematics and Computer Science, University College,
Swansea, Singleton Park, Swansea, SA2 8PP.

W. P. Weijland

Department of Computer Science, University of Amsterdam, P. O. Box 41882,
1009 DB Amsterdam, The Netherlands.

A. Williams

Centre for Theoretical Computer Science and School of Computer Studies,
University of Leeds, Leeds, LS2 9JT.

H. C. Yung

Department of Electrical and Electronic Engineering, The Merz Laboratories,
University of Newcastle Upon Tyne, Newcastle Upon Tyne, NE1 7RU.

PREFACE

The development of VLSI fabrication technology has resulted in a wide range of new ideas for application specific hardware and computer architectures, and in an extensive set of significant new theoretical problems for the design of hardware. The design of hardware is a process of creating a device that realises an algorithm, and many of the problems are concerned with the nature of algorithms that may be realised. Thus fundamental research on the design of algorithms, programming and programming languages is directly relevant to research on the design of hardware. And conversely, research on hardware raises many new questions for research on software. These points are discussed at some length in the introductory chapter.

The papers that make up this volume are concerned with the theoretical foundations of the design of hardware, as viewed from computer science. The topics addressed are the complexity of computation; the methodology of design; and the specification, derivation and verification of designs. Most of the papers are based on lectures delivered at our workshop on *Theoretical aspects of VLSI design* held at the Centre for Theoretical Computer Science, University of Leeds in September 1986. We wish to express our thanks to the contributors and referees for their cooperation in producing this work.

K. McEvoy
J. V. Tucker

Leeds, September 1988

Contents

List of contributors	vii
Preface	ix
Introduction	
1 Theoretical foundations of hardware design <i>K. McEvoy and J.V. Tucker</i>	1
Part 1 Formal methods and verification	
2 A mechanized proof of correctness of a simple counter <i>A. Cohn and M. Gordon</i>	65
3 A formal model for the hierarchical design of synchronous and systolic algorithms <i>K. McEvoy</i>	97
4 Verification of a systolic algorithm in process algebra <i>W. P. Weijland</i>	139
Part 2 Theory and methodology of design	
5 Formal specification of a digital correlator <i>N.A. Harman and J.V. Tucker</i>	161
6 Describing and reasoning about circuits using relations <i>M. Sheeran</i>	263
7 The synthesis of VLSI signal processors: theory and example <i>H.C. Yung</i>	299
Part 3 Methods of circuits and complexity theory.	
8 The prioritiser experiment: estimation and measurement of computation time in VLSI <i>P.M. Dew, E. King, J.V. Tucker and A. Williams</i>	347
9 Superpolynomial lower bounds on monotone network complexity <i>P.E. Dunne</i>	403

1 Theoretical foundations of hardware design

K. McEVOY AND J. V. TUCKER

1 INTRODUCTION

The specification, design, construction, evaluation and maintenance of computing systems involve significant theoretical problems that are common to hardware and software. Some of these problems are long standing, although they change in their form, difficulty and importance as technologies for the manufacture of digital systems change. For example, theoretical areas addressed in this volume about hardware include

- models of computation and semantics,
- computational complexity,
- methodology of design,
- specification methods,
- design and synthesis, and
- verification methods and tools;

and the material presented is intimately related to material about software. It is interesting to attempt a comparison of theoretical problems of interest in these areas in the decades 1960–69 and 1980–89. *Plus ça change, plus c'est la même chose?*

Of course, the latest technologies permit the manufacture of larger digital systems at smaller cost. To enlarge the scope of digital computation in the world's work it is necessary to enlarge the scope of the design process. This involves the development of the areas listed above, and the related development of tools for CAD and CIM.

Most importantly, it involves the unification of the study of hardware and software. For example, a fundamental problem in hardware design is to make hardware that is independent of specific fabricating technologies. This complements a fundamental problem in software design – to make software that is independent of specific hardware (i.e., machines and peripherals). Such common problems are tackled by modelling and abstraction, using common design concepts and formal tools which are characteristic of computer science.

This volume joins a series of many-authored compendia and proceedings that record progress in the development of a comprehensive theory of VLSI computation and design. For instance, we have in mind Kung, Sproull & Steel [1981], Milne & Subrahmanyam [1986], Moore, McCabe & Urquhart [1987], Fogelman Soulie, Robert & Tchente [1987], Makedon *et al.* [1986], Birtwhistle & Subrahmanyam [1988], Milne [1988] and Reif [1988]. In addition, there is the textbook Ullman [1984].

In this introduction to the volume we will examine some general ideas underlying a comprehensive theory of VLSI computation and design, and we will perform the useful task of attempting a literature survey.

At this stage in its development it is possible to see the subject only as a broad collection of interrelated problems and techniques. The theory of hardware is still limited by the theory of software. Perhaps the theoretical foundations are shallow, but they are also broad, and in each of the areas listed above there are significant and promising achievements. We hope that this volume encourages the reader to help contribute to improving the situation.

2 COMPUTER SCIENCE AND HARDWARE

We will make explicit certain ideas and problems that underlie the development of a comprehensive theory of computation and design for hardware within computer science.

2.1 What is computer science?

Computer science is about computations and computers. It concerns

- (i) the invention and analysis of algorithms,
- (ii) the design and development of programs and programming languages, and
- (iii) the design and construction of computer systems, including hardware, for implementing programs and programming languages.

It is with (iii) that we are concerned in this volume, though the subjects are intimately related. A fundamental point is that computer science aims to abstract from physical devices and view its machines through the formalisms used for their operation and programming. This emphasis on formalisms – software – gives computer science some coherence and continuity in the face of changes in the physical technologies of hardware construction.

2.2 What is hardware?

We may use the term *hardware* to mean a physical system that implements a program or an *algorithm*. Other synonyms in use are *machine*, *device*, *computer*, and *digital*

system. (Notice that only the last two terms are specific to computation.) Examples of hardware include the primitive calculators of Leibnitz and Pascal, the difference and analytical engines of Babbage, the devices made by electromechanical systems, and the application-specific devices and general computers manufactured by silicon technologies.

Whether old or new, specific or general, the essential feature of hardware is that, in realising or embodying the algorithm, its purpose is to process symbols and hence information. It is difficult to formulate definitions that correctly distinguish the many types of machines that process physical entities (e.g., a loom) from those that process information (e.g., a gauge). All physical systems process physical quantities: we impose on the physical system our framework of abstract information to obtain a digital or analogue computer. It is not easy to resolve but clearly the discussion underlies the distinct points of view of hardware possessed by the computer scientist and, *in our time*, the electronic engineer.

To design a computer we must design to physical and algorithmic specifications. In practice, the physical characteristics of technologies influence considerably our thinking about algorithms. For example, the von Neumann and systolic architectures are successful because of their suitability for implementation. Although in designing algorithms we are concerned with physical characteristics such as *time*, *space* or *area*, these quantities are actually basic properties of the symbolism that model in an abstract way the physical properties after which they are named. Rarely do algorithm designers model these quantities more exactly, or attempt to model other physical measures of efficiency or reliability such as *energy*, *power consumption*, *communication costs*, *operating temperatures*, *stress*, and so on. It is remarkable that simplifications used in work on algorithms are as useful to machine design as they are; or, to put it another way, result in designs that can be implemented and used at all. However, models of physical quantities such as energy, communication costs and so on are relevant to algorithm design and will therefore join the list of basic algorithmic concepts in due course.

2.3 The gap between hardware and software

A computing system is composed of hardware and software. The design of its hardware results in a physical system that realises a set of programs of programming languages. The design of its software results in symbolic systems that represent data and algorithms. There is a significant gap between our physical conception of devices and our logical conception of notations. There is a discontinuity at the bottom of well used images of the hierarchical nature of computation, as described in Bell & Newell [1971], for example.

In software the gap is seen in the comparison of the theoretical complexity of a program or computation and the empirical performance. Calculations based on models at different levels of abstraction can be refined to make estimates of the number of clock cycles required, *en route* to estimates of run-times in seconds that may be tested. This distinction is seen in the specification and verification of real-time computations (often controlling physical systems, for instance).

In hardware the gap is seen in the essential role that timing and performance play in the many notions of specification and correctness criteria for devices. In a sense, each model of computation for hardware design attempts to bridge, or more accurately, hide this gap.

The distinction between physical and logical concepts in computing is intimately related to the distinction between analogue and digital notions of computation. The notion of analogue computation is present in technologies such as neuro-computing (see Anderson & Rosenfield [1988]), and in any new chemical technology for image processing. It is present in discussions about new discrete space and discrete time models of physical and chemical systems (see Crutchfield & Kaneko [1987]). It is fundamental to long standing discussions about the nature of simulations in physics (see Feynman [1982]).

The gap in understanding is intimately related to the gap between mathematical models and their application in nature, which is one of enormous philosophical complexity. It is pleasing to think that the practical motivations of computer science lead us to technologies for hardware and software that require us to postulate borders between physics and logic, and hence raise fundamental scientific questions immediately.

3 THEORETICAL FOUNDATIONS

In the study of algorithms, and their realisation in software and hardware, there are certain fundamental concerns, including

- (1) models of computation,
- (2) specification,
- (3) derivation and synthesis,
- (4) verification,
- (5) testing,
- (6) maintenance.

Each specialised area of computer science – databases, theorem proving, architectures, computational geometry, VLSI, and so on – is characterised by its models of

computation, and their associated methods classified under (2)–(6). In particular, each specialised area can be surveyed under these headings.

We will discuss the literature on the theoretical foundations of VLSI computations under the headings (1)–(4), combining (2) and (4); and we will neglect (5) and (6). This arrangement of three subjects – models of computation, derivation and synthesis, and specification and verification – reflects the situation as we have found it. In preparation, we will discuss the general concepts we associate with these four topics, independently of their relevance to hardware algorithms. The primary topic is models of computation.

3.1 Models of computation

A *model of computation* codifies a means of defining and composing algorithms: it defines *data*, primitive *actions* on data such as operations and tests, and methods of defining families of actions that constitute *computations*. An algorithm specifies families of computations. For example, the natural numbers together with their usual operations (e.g., successor, addition, multiplication) and the methods of composition and primitive recursion constitute a model of computation; an algorithm in this model is a *definition* of a primitive recursive function.

A model should also give *performance criteria* to evaluate the complexity of different algorithms. One fundamental idea is to count the number of primitive actions of the model involved in a computation, measuring this number as a function of input data. This method is called the *unit cost criterion* because each basic action is implicitly charged a unit. Clearly this criterion is related to time taken to compute.

To define formally a model of computation we often define a language in terms of a syntax and semantics. Conversely, a language definition incorporates a model of computation. Practical languages are often made from several disparate (even inconsistent) models of computation.

The theoretical purpose of a model of computation is

- (i) to clarify the basic principles of a computing method,
- (ii) to classify the algorithms based upon the method, and
- (iii) to establish the scope and limits of computation using the method.

Thus a model allows us to determine whether or not a specification can be met by an algorithm; and, if performance is involved, what costs are necessarily incurred. An example of a specification that cannot be implemented is the universal function for

the primitive recursive functions, which is recursive but not primitive recursive. The performance property is recorded by *lower bound theorems* for performance criteria. For example, when C.D. Thompson first devised a model for circuits (in Thompson [1980]), he was able to prove that $AT^2 = \Omega(n^2 \log^2 n)$ for sorting n elements; in Thompson [1983] some of the conditions on the models are relaxed, and consequently the lower bound on sorting must be weakened to $AT^2 = \Omega(n^2 \log n)$.

In its origins, however, a model may have been devised for one or more of the following purposes:

- (i) to systematise algorithm development for an applications area;
- (ii) to systematise algorithm development for an implementing technology;
- (iii) to allow the analysis of computational properties.

For example, the systolic algorithm model satisfies (i) and (ii), supporting signal processing and VLSI technology; and the arbitrary interleaving model of concurrency satisfies all of (i), (ii) and (iii), supporting multiprocessing on a von Neumann computer, and the analysis of non-determinism and the independence of parallel actions.

3.2 Specification

A *specification* of an algorithm is an independent and abstract description of properties of the algorithm, or of the problem it is intended to solve. The statements making up a specification concern its inputs, outputs, and efficiency, for example. A specification is intended as a record of

- (i) information relevant for users of an algorithm, and
- (ii) requirements relevant for designers of an algorithm.

These two uses ensure that a specification is as fundamental as the algorithm itself. In connection with (i), specifications are used in the modelling of the task to be accomplished by the algorithm. In connection with (ii) specifications are used in confirming the correctness of the algorithm by both empirical testing and mathematical verification.

The precision in the description of a specification should be comparable with the precision in the description of an algorithm. Precision and rigour are indispensable in the process of algorithm design, which involves classifying properties, and their ramifications concerning the user's task and the designer's resources. Formally defined specifications complement formally defined algorithms. If specifications are machine processable then they can be animated, transformed, tested and verified with the assistance of computers.

A theoretical distinction between the general concept of a specification and the general concept of an algorithm is hard to draw. A specification can be very detailed, and indistinguishable from a coding of an algorithm. This attention to detail is common in practical work with specifications, and is an insidious problem. The point is that a specification is of use when it is an abstract description of some properties of an algorithm, or set of algorithms.

Given a specification method for a model of computation, important theoretical questions arise about its *expressiveness*:

Soundness or consistency problem Can every specification be realised by a set of algorithms based on the model?

Adequacy problem Can every set of algorithms based on the model be defined by a specification?

To formulate precisely and answer these questions for any specific model of computation and specification method involves considerable theoretical research. For example, the scope and limits of the algebraic specification methods for computable data types are surveyed in Meseguer & Goguen [1985].

An important theoretical use of specifications is in defining notions of equivalence for algorithms. Notice that if algorithms A and A' satisfy specification S then they are equivalent as far as S is concerned. More generally, given a specification method M , it is important to study the following equivalence relation on algorithms: given algorithms A and A' define that A is *equivalent* to A' under M if, and only if, for every specification S based on M , A satisfies S if and only if A' satisfies S .

Nevertheless, it is useful to disconnect the study of specifications and their use from that of algorithms. This attention to the theory of specifications is an original and important contribution to computer science from the field of programming methodology. An extensive study in the context of hardware is the chapter by Harman and Tucker contained in this volume.

3.3 Derivation

A *derivation* of an algorithm A from a specification S is a process of defining a sequence

$$A_0, A_1, \dots, A_n$$

of algorithms in which $A = A_n$, A_0 satisfies the specification, and the *transformation* or *refinement* of A_i to A_{i+1} for $i = 0, \dots, n - 1$ preserves the specification S . The

sequence is called a *derivation*; the transformations or refinements are said to be *specification preserving* or *correctness preserving*, and the process is also called the *stepwise refinement* or *synthesis* of the algorithm.

The notion of derivation is *very* general. Typically, a derivation arises in the solution of a problem, represented by the specification S . Here A_0 is some simple first algorithm that meets S but is not satisfactory; perhaps it is inefficient, sequential and unsuited to implementation in hardware. The transformations result in a complicated last algorithm A_n that is satisfactory; perhaps it is efficient, concurrent, and readily implemented. Many notions are involved in derivations, such as top-down design, automatic synthesis and compilation.

Among the basic concerns are

- (i) transformation methods for developing algorithms for a given model of computation,
- (ii) logical systems for formulating and proving that transformations preserve correctness,
- (iii) automatic tools for processing derivations.

These concerns guide much theoretical research on models of computation, and specification and programming languages, throughout computer science. For example, in programming methodology, the concern (ii) for correctness is analysed by refinement calculi, such as the weakest precondition calculus described in Dijkstra [1976]; this has led to significant theoretical understanding of the process of derivation (see Back [1980] and Back [1981]) and its practical extension (see Back & Sere [1989], Back & Kurki-Suonio [1988] and Chandy & Misra [1988]).

Of course, the original example of a theoretically well-founded and practically well-developed formal derivation process is the theory of boolean algebra and its applications to circuit design.

3.4 Verification

A *verification* that an algorithm A meets a specification S is a process of defining a sequence

$$P_0, P_1, \dots, P_n$$

of statements in which P_n asserts that A satisfies S , and each statement P_i is either an assumption about the model of computation and specification, or the result of deduction from statements preceding P_i in the sequence. The sequence is called a *proof of correctness*. A verification can be performed independently of a derivation, although a derivation ought to determine a verification.

Verification techniques must be founded upon mathematical theories which include mathematical models of computation and specification, but they can be divided according to the nature of the method of proof:

- informal methods, which are based on standard mathematical concepts, techniques and reasoning; or
- formal methods, which are based on formally defined languages with associated proof rules, and are often standard logical systems of mathematical logic.

The advantages of informal mathematical methods are that they are focussed on the human understanding of the essential technical points in the proof, they are understandable by a wide audience of people with mathematical training, and they are independent of specific logical and computer systems. The advantage of formal methods is that formal specifications and proofs are machine processable, and so formal proofs can be constructed, or at least checked, by computers. This is significant for raising the standard of rigour in a verification, and for solving the large problems that arise in specifications and verifications of algorithms of practical interest. The informal and formal methods are quite distinct but complement one another, of course; this distinction is true of both the nature of proofs and the talents necessary to construct them.

Currently there is renewed interest in the use of automatic theorem provers and proof checkers in the verification of software and hardware. It is essential that these computer systems should be based upon formally defined frameworks for doing proofs. The formal proof framework is usually some established formal logic. Examples of logics used in theorem proving software are

- higher order logic,
- first order logic,
- equational logic,
- temporal logic,
- Church's type theory,
- Martin-Löf's type theory.

Thus the algorithms and their specifications must be described directly, or compiled into such a logical language. Automatic theorem proving originates in attempts to prove theorems of mathematics and logic, and each of the above logics was first implemented for this purpose. The first implementation of a program verifier is reported in King [1969]. For basic historical and contemporary surveys, and source material, see Siekmann & Wrightson [1983] and Bledsoe & Loveland [1984].

Let us consider the terms *theorem prover* and *proof checker*. Strictly speaking we imagine a *theorem prover* to be a system that inputs a statement and, if the statement is true, returns a proof of the statement; if the statement is false, or cannot be proved, the theorem prover could react in several disciplined ways – it may give a proof of its negation, or simply reply that it cannot find a proof (it should not, of course, search *ad infinitum* for a proof that does not exist). A *proof checker*, however, inputs a statement and a proof of that statement, and returns information concerning the validity of the proof.

This distinction needs further analysis. First there is the distinction between the truth and falsity of a statement, and its provability or non-provability in a formalised logical theory. This distinction is fundamental in mathematical logic, and is analysed in terms of various notions of *completeness* and *incompleteness* of formal theories.

A formal theory T arises from the codification of certain properties of a model of computation M_0 about which one wants to reason. The statements and proof rules of T are true of M_0 . However, the theory T is more abstract and possesses a semantics M defined by the following so-called completeness condition:

P is provable in T if, and only if, P is true of M .

Since T is true of M_0 by design, we expect that

P is provable in T implies P is true of M_0 ,

but that the converse can fail, namely

P is true of M_0 does not imply P is provable in T .

Thus it is essential to distinguish carefully between the notions of true statements and provable statements. For example, it can be proved using the theorems of K. Gödel, that given any formal theory designed to reason about specifications concerning algorithms on the natural numbers $\{0, 1, 2, \dots\}$ there are specifications and algorithms that are true but which cannot be proved in T .

The algorithmic notions of theorem prover and proof checker described above are better described in terms of *decision procedures*. A decision procedure for *provability* of statements in a theory is an algorithm that given any statement decides whether or not the statement has a proof in the theory. For some simple theories, including propositional calculi, there are decision procedures; however, for most basic theories