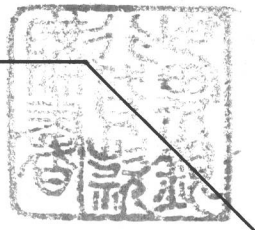
VLSI

SILICON COMPILATION AND THE ART OF AUTOMATIC MICROCHIP DESIGN

RONALD F. AYRES



VLSI

silicon compilation and the art of automatic microchip design

RONALD F. AYRES

California Institute of Technology
Information Sciences Institute

Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging in Publication Data

Ayres, Ronald F.
VLSI: silicon compilation and the art of automatic microchip design.

Bibliography: p.
Includes index.
1. Integrated circuits—Very large scale integration —Design and construction. I. Title. II. Title: V.L.S.I.
TK7874.A97 1983 621.381'73 82-23177
ISBN 0-13-942680-9

Editorial/production supervision and interior design: *Mary Carnis*
Cover design: *Diane Saxe*
Manufacturing buyer: *Gordon Osbourne*

Dedicated to

Giorgio Ingargiola and programs without roses, Judy and cats, Dick Suchter, Dave Plumer, JNG, DQP, WLL, the "KL", memory location 80 on the IBM/360 Mod 44, computer memory references both legal and "illegal", and all the op-codes that will ever be "reserved to DEC."

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-942680-9

Prentice-Hall International, Inc., *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*
Whitehall Books Limited, *Wellington, New Zealand*

Preface

This text provides an introduction to the emerging field of silicon compilation and the art of automatic chip design generation, perhaps the most exciting area in VLSI chip design today and for some time to come.

This text is intended to give the reader a foundation in techniques and concepts with which to create, understand, and explore silicon compilation. The text can serve as a university textbook for the majority of a one-year graduate course in VLSI computer-aided design, as a handbook for designers of computer-aided design systems, and as a case study of a large software application.

Silicon compilation helps resolve today's acknowledged "hardware crisis": the great difficulty encountered in the generation of new, custom integrated circuit micro-chips. This difficulty has become a "crisis" only because the physical technology readily exists to produce wide varieties of very powerful and highly desired computational engines. However, the greatest bottleneck involves the translation from the human intent, a behavioral description of what the new chip is supposed to do, into the chip "layout," a complete geometric representation from which these marvelous physical technologies can readily produce reliable, working chips.

Any step toward the resolution of this hardware design crisis brings with it two very important results. First, it brings a vast reduction in cost, and hence an increase in the availability of products that make everybody's life more pleasant. It widens profoundly the applicability of these micro-chip technologies to facilitate even greater and previously unimagined benefits to all of us. Second, it widens the scientific view of information processing and the very character of information itself, by removing some of the barriers that have provided artificial divisions in our ways of thinking, for example, the great division between hardware and software. It brings us closer to a unified feeling for information and its evolution.

The relative costs between the chip design process and the subsequent chip fabrication dramatically expose the design bottleneck. The design process produces a chip layout and the fabrication process turns this layout into real chips. The design process can

cost, for state-of-the-art designs, a few million dollars, and a year or more elapsed time from the initial characterization of the chip (usually a short document specifying the intended behavior for the chip) until a working design emerges. Once the design is completed (at which time it is a year behind the state of the art), the fabrication process even for small-volume prototypes can cost well under $10,000 and take under a month.

How is it that we have become accustomed to such immense design costs? The product of this process, a layout, is a picture consisting of about a million individual shapes, all of which together must satisfy the constraints or "syntax" imposed by the fabrication process. A single error in this maze of details often renders the entire chip inoperative. Not only must this giant set of shapes satisfy the silicon "syntax," but it also must make the appropriate statement in this silicon language so that the chip will perform as intended. In addition, the design must satisfy electrical constraints simply because the whole technology is based on properties of electrical phenomena.

This text exposes "silicon compilation," the art of automatic layout design generation, as a substantial solution to our hardware crisis. We who created silicon compilation and introduced the term into computer science use the term "silicon compiler" to denote totally automatic translations from behavioral specifications to correct chip layouts, just as the term "compiler" in software implies totally automatic translation from high-level languages into machine languages. Human interaction is entirely optional, and is limited so to affect in no way the behavioral correctness of the resulting layout.

Working silicon compilers have produced chip layouts in under one hour, from behavioral specifications developed in under one week. These chips range from complete microprocessors to special-purpose, application-dependent microprocessors. These silicon compilers are only the first such tools—design aids that implement the entire design process automatically from behavioral specifications to correct layouts. They exploit silicon's native potentials without adopting the restrictions, inefficiencies, and ways of thinking inherent with older technologies, like the PC-board place-and-route paradigm adopted in "gate array" and "polycell" technologies.

A variety of semiautomatic techniques have emerged from industry and academia which have tended toward isolated and/or partial solutions to limited, well-defined problems. Even complete solutions to subproblems, developed without considering the overall compilation problem, very often render few, none, or negative improvements for the overall chip, particularly when loose ends are left to manual, error-prone intervention. This text, in contrast, emphasizes the overall problem and detailed solutions, sometimes even by using less-than-optimal solutions to subproblems, again, to show a great flexibility and potential.

It is surprising perhaps that the effort required to provide complete automation is actually considerably less than solving in a practical way a majority of subproblems independently, mainly because the compiler writer has complete control over the entire automation. This of course facilitates more efficient solutions whose overall effect surpasses concentrated optimized local solutions.

The possibility of such complex uniformities also makes possible physical technologies as yet unimagined, geared particularly toward use with silicon compilation; the great degree of complexity supported by compilers facilitates new technologies which might otherwise be out of reach to manual design. (We have seen recently that technology has adapted to software compilers with the emergence of Pascal and LISP machines.)

The field of silicon compilation bridges the fields of electrical engineering and software engineering. We find today in the domain of chip design the same kinds of problems and attitudes that existed one or two decades ago in the software domain. In both cases, chip and software designers have been faced with a rapidly growing base technology, which has changed and is changing the relative costs of alternative ways of doing business. The bottleneck ceases to be the limitations of the base technology, but becomes instead the human limitations of formulating the design (layout or program).

Software compiler engineers have been and are faced with the question of how to

make most effective use of the growing capacity of memory and speed. When software compilers came to be, people began reluctantly to use them amid fears of losing efficiency and control. Nonetheless, today these fears have nearly vanished, and large software designers now ask not whether to use a compiler at all, but instead which compiler (language) to use. Now, with the capacity of silicon chips continuing to grow very rapidly, chip designers are faced with the same problem of how best to map human skills into this generous silicon terrain.

We explore one silicon compiler in detail to provide continuity among the various techniques and concepts shown. This silicon compiler, RELAY (REcursive Logic ArraYs), was chosen because it is general purpose and illustrates many of the techniques and concepts that apply to all kinds of silicon compilers. RELAY involves the capture of behavior specified in terms of synchronous logic, its translation into an optimized disjunctive form, its simulation and translation into clocked PLA layout structures, its placement and routing, and its electrical or performance modeling.

RELAY is also especially well suited to illustrate the wide and subtle variations and optimizations afforded by the redistribution or reorganization of logic over the chip area. The latter effect clearly exposes the novel flexibility and use of silicon compilers, which renders them indispensible for all high-level design phases, including floor planning—those phases where the majority of major headaches tend to focus in practice because their resolutions are expensive.

The most valuable information derived from these efforts has been a collection of techniques and new ways of thinking which facilitate the creation of these and other silicon compilers. These techniques and concepts form the bulk of this text.

In fact, these very techniques are applicable not only to silicon chips, but also to technologies, some yet to be invented, which derive their function from specifications rendered in terms of two (or three) dimensional layouts (pictures).

We do not cover in detail layout techniques which tend to be dependent on individual fabricators and physical technologies. We also do not cover the underlying physics of microchips in general. Other texts cover these areas and provide the rigorous bases for our abstractions. We resort, for example, to electrical properties only when they must be taken into consideration. We focus instead on the software techniques used to generate layouts of any kind, and to generate complete layouts of a more particular character.

We include a multitude of related programming examples; this is essential in order to enhance confidence and a ''can do'' attitude, and to show all the essential details that must be considered inevitably in any programming relationship with the computer. We have chosen for this purpose the programming language ICL (Integrated Circuit Language) which was developed especially for exploration into VLSI. ICL is appropriate for use in this kind of text, primarily because many programming details (for many applications) are handled implicitly by ICL. ICL does not burden us with most of those programming details having nothing to do with VLSI (e.g., pointers, memory management, and various other awkwardnesses). In fact, ICL serves well as a clear formal notation, useful in the way that mathematical notations are useful in math texts.

ICL notations are explained as they become parts of examples, so the reader can make very practical use of the examples, at least in their understanding. In fact, all the programming examples have been extracted from this text and run on the computer to verify their correctness and to generate many of the illustrations. ICL is a complete working language which has been used exclusively in the formulation and implementation of two complete silicon compilers, RELAY and BRISTLE BLOCKS, and has been also used to service chip designers nationwide in providing exceptionally economical chip fabrication from layouts.

The seasoned programmer may wonder if the simplicity of expression afforded by this language carries with it a significant loss of efficiency. Surprisingly, empirical evidence to date shows the opposite to be true, perhaps due in large part to the very direct communication afforded by its human orientation.

Acknowledgments

This book was made possible in large part by a wide variety of people engaged in many disciplines, both in the academic and industrial worlds.

For the courageous long-term and persistent presentation of fundamental principles relevant to this subject matter, special thanks go to Dr. Frederick B. Thompson for emphasizing the flexibility of computational linguistics, to Dr. Per Brinch-Hansen for exposing the importance and pleasure of particularly clear program specification, to Dr. Carver A. Mead for discovering a very straightforward and practical view of VLSI systems, to Dr. Tom Apostol for illustrating the ease and practicality of applying abstract mathematical techniques to the widest variety of problems, and to Dr. Herbert Benjamin for lending a boldly realistic view to the very nature of information itself.

More specific to this subject matter, the author is particularly grateful to David Johannsen, Carver Mead, John Wawrzynek, and Charle Rupp for participating in the growth of this field. David Johannsen in particular embraced silicon compilation the very same day the author introduced this subject to an audience at Caltech. He has since expanded and applied many of the early concepts directly to datapath chip architectures, which had previously been proposed and manually implemented by Mead and Johannsen. Carver Mead contributed, among other things, encouragement, excitement and effective generalizations, so as to expand participation in this new field. John Wawrzynek singlehandedly lashed together two early silicon compilers, RELAY and BRISTLE BLOCKS, so as to render chip layouts that utilize some of the best abilities of each compiler. Charle Rupp pursued the field by attempting to generate layouts from pieces as small as individual transistors. These and other points of view together have given rise to this very fertile field.

Also instrumental in developing the field were many fruitful discussions with Ivan Sutherland, Jonathan Allen, George Mager, Walter Klein, Chuck Seitz, George Lewicki, Ricky Mosteller, Telle Whitney, Juda Afek, Danny Cohen, Bernard Goodwin, John Newkirk, Bob Matthews, Ed Cheng, and Paul Tyner.

Of course, the importance of a working environment cannot be overemphasized. The spirit and scientific resources of the California Institute of Technology, together with the industrial participation afforded by the Silicon Structures Project, has provided perhaps the most fertile of possible environments for this and other kinds of scientific and industrial progress.

The guests from industry provided a secure sense of what people really do under the stress of trying to get a chip out the door under a tight schedule, exposing the kinds of computer tools embraced most fervently and the chronic bottlenecks experienced in that endeavor. The academic spirit provided a timeless security entirely unencumbered by hurried short-term business goals. Ivan Sutherland personally exemplified the best of these two worlds simultaneously, providing truly down-to-earth information and resources while maintaining the best of scientific integrity by providing an extraordinarily open mind to ideas from all disciplines.

Another very important working environment is the USC/Information Sciences Institute, providing among other things superb computing resources and very direct introductions to the VLSI activities occurring nationwide within the research community funded by the Advanced Research Projects Agency. It is remarkable and encouraging that major industrial-grade services can be developed quickly and maintained securely within such a delightful and free environment.

The author is indebted to Larry Seiler and Alan Paeth for providing a PLA layout generation program and for providing the character fonts used in the illustrations, all in their independent effort to produce a key debouncing chip for a VLSI design course. Another person, whose name cannot be mentioned for obvious reasons, ''magically'' provided an abundance of computing time on one critical occasion. Also instrumental have been the superb computers and software provided commercially by Digital Equipment Corporation.

The dedication and illumination provided by the people at Prentice-Hall, the anonymous reviewers of the manuscript, John Wawrzynek, Bernard Goodwin, and George Lewicki have made it possible to accommodate the varied audiences whose distinct fields of interest are necessarily brought together in a text of this kind.

Special thanks go to Dr. Herbert Benjamin for encouraging the conception of this book, and to my brother and sister Gary and Denise, and my parents Anita and Marx, for providing the greatest support, dedication, and challenge over nearly an infinite number of years.

Contents

Introduction

The explosive progress occurring in integrated circuit fabrication has resulted in a doubling of circuit density about every two years. Increased circuit densities attract circuits of increased complexity. This progress makes possible today's single-chip microprocessors and large memories. This progress also brings great increases in the sheer complexity involved in the design process. This text introduces compilation as a means of dealing with this increased complexity, a means long standing in the software domain, where enormous complexity first emerged as a serious problem.

The creation of an integrated circuit proceeds in five basic steps. The creators begin by drawing up a functional specification. They then formalize that specification and perform simulations to build their confidence in its correctness. The major task follows as they translate the desired behavior into a two-dimensional language of silicon where implementation appears as very complex sets of intricately overlapping shapes. They send these sets of shapes, together called a layout, to a silicon foundry and receive a large number of chips in return. The creation is completed as they test and package each chip.

The major cost arises in the translation from the formal functional specification into a two-dimensional layout. Typically, a human translator creates pieces of layout with either paper and pencil or computerized graphical editors. Extreme care must be taken in this process, much greater than the care required for software development because the turnaround time and the cost for a run (chip fabrication) is very expensive. The tedium involved in manual layout makes this process very error prone. Even a computerized graphical editor admits anything the designer draws and thus ignores the inherent syntax or constraints imposed by the two-dimensional language of silicon.

To verify adherence to silicon constraints, software has been developed which checks layouts prior to their fabrication on silicon. Some programs look at layouts and flag certain kinds of geometric violations. Other programs try to recreate the schematic representation directly from the layout. These techniques provide for early error detection and thus cut down on the number of silicon fabrications required to debug a layout.

However, such programs cannot catch all kinds of errors. Somebody still has to produce the layout in the first place.

This hardware development technique contrasts current software development techniques. On the one hand, programmers have compilers that translate high-level specifications into machine language. On the other hand, layout designers have decompilers, programs that read the silicon language and try to reconstruct higher-level descriptions.

Decompilers are difficult to create because they must accept all possible expressions in the silicon language, including all tricks of the trade. All design rule checkers, for example, either do not flag all errors, or actually flag many perfectly correct situations as though they were errors.

A compiler, on the other hand, is much easier to create; it has only to accept a well-defined formal language. The only requirement of such a formal language is that it capture the domain of digital behavior.

The layout generated by a working silicon compiler is guaranteed to be correct with respect to both the functional specification and the constraints of the silicon language. Thus, the compiler users (e.g., layout planners) are freed to concentrate on more productive tasks. They can try various architectures and quickly weigh the tradeoffs among various layouts and timing proposals. They can detect bottlenecks and choose to lay these out manually. In analogy, software designers hand-code those routines that consume much of the computer resources.

Even though the layout generated by a silicon compiler may consume more area than a layout generated manually, there are two factors that encourage the use of silicon compilers. First, in tomorrow's technology, we may ultimately have 100 times the density available today. This means effectively 100 times today's area and perhaps 100 times today's speed. Software compilers gained acceptance when available memory and speed rendered unprofitable the human effort required for machine language coding. The same applies to the silicon design effort.

Second, in today's technology, one can buy a full microprocessor on a chip; however, the external logic required to employ such a chip in a system can require many more chips than the microprocessor chip, particularly if the external logic is implemented in commercially available SSI or MSI chips. The minimal design time associated with a silicon compiler makes profitable the creation of a single chip to implement the external logic. Implementing many SSI or MSI chips in one custom chip reduces the cost of PC-board manufacturing, a particularly attractive cost reduction for high-volume operations.

ANATOMY OF A SILICON COMPILER

There are two focal points of concern in a silicon compiler. These are readily apparent to users of such a compiler:

1. The *source language*: the language in which the user specifies the desired function, or behavior, to be performed by the new integrated circuit chip.
2. The *target language*: the capabilities of silicon, utilized in a very complex two-dimensional color picture called a layout.

The quality or usefulness of a silicon compiler is measured by:

1. The ease of behavior specification afforded by the source language
2. The efficiency in use of silicon area and the operating speed of the finished chip

The first of these two considerations is perhaps more important and overlooked more. Since the birth of integrated circuit technologies, designers have had direct access to the target language, and have been most concerned with efficient use of the silicon resource.

However, the ultimate value of an integrated circuit chip is measured by how well it solves a particular problem, usually within a larger system. Even the most efficient chip loses substantial value if it performs a service that mismatches the need within the system. The most efficient chip is utilized inefficiently as the system designer has to remold the system and introduce new chips around the given chip, so to accept the services provided by the given chip.

Thus, integrated circuit chips designed with tunnel vision focused only on silicon efficiency serve merely to pass the buck, or inefficiency, off into the overall system within which it ultimately resides.

In contrast, a chip designed with the particular application foremost in mind can afford some silicon inefficiencies if its role within an overall system matches the system's needs very closely. The design of such functionally optimal chips occurs primarily not in the domain of silicon, but in the domain of possible behaviors or functions. The ''source language'' of a silicon compiler covers precisely this high-level domain.

Perhaps the most innovative progress in the field of silicon compilation revolves around the design of the source language. While there exist many languages capable of representing behavior, for example, nearly all software programming languages, a source language for silicon compilation must:

1. Provide for direct specification of behaviors that are supported most profitably in the native silicon (e.g., parallelism)
2. Simultaneously provide an overall integrity so that the language persists in being a language of behavior, as opposed to merely a language of layout geometry

The overriding importance of the source language is clearly evident today in the software domain. For some time now people have based decisions concerning the choice of a software compiler on the source language supported by the compiler, as opposed to the efficiency of the code it ultimately produces in the target (machine) language. Concern about the efficient use of inanimate resources takes a backseat to human convenience when the entire effort is considered realistically as a whole.

COMPARING SOFTWARE AND SILICON CONSTRAINTS

Software and hardware revolve around the same basic concern: The software designer lays out a one-dimensoinal array of memory, whereas the integrated circuit designer lays out a two-dimensional area of silicon. In each case, various constraints must be satisfied to obtain a working product. In addition, both efforts involve lots of modification.

There are some fundamental differences between the constraints of software and hardware. For example, GOTOs in software implementations cost only the memory required for the GOTO instruction itself. The distance between the location of the instruction and the target address plays no role in either execution time or memory consumption. In contrast, a silicon GOTO requires area for the ''wire'' connecting the two locations. A longer wire consumes more area and more execution time. Furthermore, a wire in silicon has to avoid obstacles in order to avoid unwanted short circuits. A software GOTO has no obstacles; it does not have to dodge a certain set of words in memory.

Another difference between software and silicon concerns the different layers required in silicon. A signal in silicon may be represented on any one of several layers.