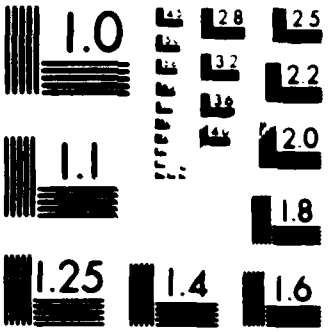
AD-A180 136 A METHOD FOR THE AUTOMATIC TRANSLATION OF ALGORITHMS 1/1
FROM A HIGH-LEVEL LA..(U) AEROSPACE CORP EL SEGUNDO CA
COMPUTER SCIENCE LAB S H KELEM 30 SEP 86
UNCLASSIFIED TR-0084A(5920-03)-1 SD-TR-86-60 F/G 12/5 NL



REPORT SD-TR-86-60

(12)

A Method for the Automatic Translation of Algorithms from a High-Level Language into Self-Timed Integrated Circuits

S. H. KELEM
Computer Science Laboratory
The Aerospace Corporation
El Segundo, CA 90245

30 September 1986

Prepared for

SPACE DIVISION
AIR FORCE SYSTEMS COMMAND
Los Angeles Air Force Station
P.O. Box 92960, Worldway Postal Center
Los Angeles, CA 90009-2960

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER SD-TR-86-60	2. GOVT ACCESSION NO. AD-A180	3. RECIPIENT'S CATALOG NUMBER 136
4. TITLE (and Subtitle) A METHOD FOR THE AUTOMATIC TRANSLATION OF ALGORITHMS FROM A HIGH-LEVEL LANGUAGE INTO SELF-TIMED INTEGRATED CIRCUITS		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER TR-0084A(5920-03)-1
7. AUTHOR(s) Steven H. Kelem		8. CONTRACT OR GRANT NUMBER(s) F04701-85-C-0086
9. PERFORMING ORGANIZATION NAME AND ADDRESS The Aerospace Corporation El Segundo, CA 90245		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Space Division Los Angeles Air Force Station Los Angeles, CA 90009-2960		12. REPORT DATE 30 September 1986 13. NUMBER OF PAGES 10
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Algorithms	Compilers
Integrated circuit masks	Translator
Self-timed ICs	Algol 68
High-level language	Templates

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A method for generating custom self-timed integrated circuits (ICs) from algorithmic descriptions of the desired circuits. The goal is to quickly produce prototype integrated circuit masks that implement various algorithms and data types in order to evaluate the IC power, delay, and area characteristics. A topology and behavior-preserving mapping is used to perform the translation from constructs in the functional language to mask primitives.

DD FORM 1473 (FACSIMILE)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

SUMMARY

A method for generating custom self-timed integrated circuits (ICs) from an algorithmic description of the behavior of the desired circuit is described in this paper. The goal is to quickly produce prototype integrated circuit masks that implement various algorithms and datatypes to evaluate the IC power, delay, and area characteristics. The behavior of the circuits is described in a functional subset of Algol 68 and is given a dataflow interpretation. To translate constructs in the functional language to mask primitives (CIF code) a topology- and behavior-preserving mapping is used. The mask primitives have been validated by simulation and testing so that the syntax-directed translation is assured of generating only working circuits. This mapping requires execution time proportional to the length of the algorithmic description. Therefore the execution time will be fast and allow for algorithm and data type experimentation.

CONTENTS

SECTION I
INTRODUCTION

A. GENERAL

Designing integrated circuits is a lengthy and costly task. The steps required can be organized into levels of abstraction ordered roughly as follows[3, 4]:

Requirements:	The specification of the overall performance, area, and I/O for the circuit.
Abstract Algorithms:	The behavior of the circuit without a binding for the actual operations and data types.
Concrete Algorithms:	The behavior of the circuit expressed in a machine-independent programming language. The operators and data types are specified.
Programming:	The machine language for the circuit if it is programmable.
Register Transfer:	The behavior described in terms of states during which data is transferred between registers in the circuit.
Logic:	The circuit description in terms of logic components and their interconnections.
Discrete Circuit:	Logic functions in terms of transistors, resistors, capacitors, etc.
Topology:	A circuit in which physical dimensions are absent, but in which relative positioning is expressed.
Masks:	Transistors are defined by the intersections of polygonal areas on masks that are used in the fabrication process for integrated circuits.

Each of the above levels represents a class of decisions made in the design process. In the higher levels the decisions have a larger scope than those at the lower levels and may hide details of the lower levels. Decisions at the lower levels can be made without much information or guidance from the higher levels. To evaluate a design, the non-behavioral characteristics[2] such as power, area, and delay must be determined. The effects of decisions made at the higher levels on these non-behavioral characteristics often cannot be determined until a design is refined all the way to the bottom level.

The concrete algorithm level is the highest level considered here. Many design alternatives can be examined at this level. (Examining many alternatives is impractical at the lower levels because of the amount of work that must be done.) However, this approach allows for rapid descent through the levels of abstraction so that an estimate of the power, delay, and area of an implementation can be quickly determined. For example, a decision may be made to change the precision or representation of numbers used in an algorithm. This kind of decision changes the delay by an order of magnitude[1] and also affects the power and area. Without refining the design to the level of IC masks, the effect of these changes cannot be easily determined. It may be possible to change the logic equations or transistor schematics or possibly the mask specifications, but it would be a difficult, tedious, and error-prone job for a human to perform. The task of converting from a two's complement number system to a another number system is even more difficult to perform. All the logic equations would have to be re-derived. Everything at the lower levels would also have to be discarded.

B. DOMAIN

The algorithmic language used in this approach is a functional language that allows for description of tree-structured algorithms and computations. The compiler utilizes templates designed with self-timed logic[13, 15], thus assuring proper function, no matter how the cells are placed or stretched on a chip. This simplifies the automatic synthesis of the circuits. The circuit elements need only be connected to work. With self-timed logic, a clock signal does not have to be routed to the synchronous elements since there are none. Determination of worst-case timing for parallel computational paths does not have to be performed to determine a clock period. Unfortunately, self-timed circuits are larger than their clocked counterparts. They require more circuitry, more power, more wires, and more pins on the chips. The interface to a self-timed chip must either convert synchronous signals to self-timed signals or pay a penalty in the number of pins used on the chip. I feel that the advantages offset these liabilities for IC protyping.

SECTION II
APPROACH

Work is continuing on a compiler that reads a description at the concrete algorithm level and translates it into mask specifications for an IC. When the IC is built, it will have the desired behavior. No placement information is given by the designer. It is instead derived from the structure of the algorithm. The behavior is specified in a strongly-typed, functional subset of Algol 68. As in other functional languages, variables are bound in functions, but cannot be re-assigned.

An algorithm is viewed as having two dimensions. These are mapped directly to the two dimensions available on planar chips. One dimension is the flow of the data through the algorithm. The other dimension is the concurrency of tasks and the width of the data and operators. This straightforward mapping greatly reduces the number of options at the lower levels of abstraction that the compiler must consider. The execution time of the compiler methodology being employed is linear, and so many algorithmic possibilities can be explored quickly by a designer. A designer must be able to experiment with these to find a practical implementation of special-purpose architectures and special-purpose arithmetic chips.

Experimentation may consist of varying the amount of parallelism in different parts of the algorithm. It is important to try various number representations for the data and operators used in the algorithms. Choosing between representations like one's complement, two's complement, unsigned, residue, signed digit, and other redundant number systems has a much bigger impact on algorithm performance, and therefore, chip performance, than is attainable when considering changes at the logic, transistor, or mask levels.

Because the mask layouts can be rapidly produced, it is possible to produce metrics for design comparisons quickly. The area and power consumed by the circuit can be calculated as the circuit is laid out. Timing information can be obtained by running conventional timing simulators[16, 14] after the mask description is produced.

A. TARGET CIRCUITRY

The compiler will translate the Algol language expressions into Chisel[8], an algorithmic geometry language. Chisel translates geometry information directly into CIF[5, 12] descriptions of IC masks. The compiler uses pre-defined mappings from the programming language constructs to circuits. Each Algol construct has a corresponding circuit template that implements the high-level behavior. There are templates for IF, CASE, function calls, and the Boolean operators. The term *template* is used, rather than *circuit*, because the circuitry must be stretchable and parametric to accommodate functions of any size. The only

details of the interior of the templates needed by the compiler are the number and position of wires in the template. In the case of the IF function, the IF template contains three holes, one for the conditional, one for the circuitry that comprises the THEN part, and one for the circuitry that comprises the ELSE part. Since the current target domain is self-timed logic, the IF template consists of multiplexors and acknowledgement circuitry for the self-timed signals.

Automatic placement and routing is performed by mapping the tree-structured algorithm into a planar tree structure[11, 10]. The first version of the compiler being built uses a simple placement method. Tree-structured layout is inefficient in terms of area and wire length, but placement and routing may be performed easily and quickly. After initial layout, the area is improved by using the Bristle Blocks[6] approach of stretching the cells so that connection points align. When the connection points align, the cells can be moved closer together for abutment.

B. CONCURRENT PROCESSING

The number of concurrent operations is maximized with respect to the specified algorithm to simplify the placement and routing, and to improve the speed of the algorithm. One copy of each function is provided for each invocation in the algorithmic description, thus there is no sharing of "code" between processing functions. Because the various functions on a chip can execute at the same time, the amount of parallelism is *maximal. Sharing of resources is not being considered at this time.* If it were, then the various functions could not execute at the same time. Contention arbiters would be required for synchronization between call instances. Busses would be required to connect the function circuitry to the places where the inputs are read and the outputs are to be delivered.

C. OPTIMIZATIONS

Many types of optimizations are possible with a compiler methodology. They can be grouped into two categories – language-level and circuit-level. All optimizations require searching some space of possibilities. Depending on the characteristics of the optimization, the search of the space may dramatically affect the overall performance of the compiler.

It is useful to summarize the steps in a compilation to see where the optimizations are performed. The compiler is an LALR(1) translator written in yacc[7]. The sequence of compiler operations is essentially as follows.

1) A lexer reads the algorithm specification and groups the characters into syntactic units called tokens.
2) The tokens are then grouped structurally according to the grammar rules.
3) The parser uses the rules to generate a parse tree from the sequence of tokens.

4) The annotated tree can then be optimized according to some language-level criteria.
5) Each node in the parse tree is then annotated with a spatial and functional description of the associated circuitry.
6) The circuitry associated with each node in the annotated tree then is placed spatially and the wires routed. Circuit-level optimization are performed at this stage.
7) This circuitry must then be placed in and wired to a pad frame so that it can be fabricated.

The lexing, parsing, placement, and routing of tree structures require linear time. The optimizations that are being considered are ones that are deterministic and close to linear.

One form of area optimization has already been mentioned: stretching cells so that they align, thereby eliminating space for wiring channels in one of the axes. A standard language-level technique called *constant folding* is used to evaluate constant expressions at compile-time. This allows a designer to write general-purpose routines, knowing that the compiler will eliminate code that will not be executed in certain circumstances. For example, an *n*-bit signed-digit adder may be designed to test for special circumstances with an IF or CASE expression, perhaps using one method if fewer than eight bits are used, another for eight to fifteen, and so on. The conditionally executed logic will be eliminated if it is known that it will never be executed. An example of language-level optimizations is the use of parametric definitions so that the word length in a function may differ in the various instantiations of the same function. The cases where fewer bits are being used do not need to generate as many wires or as much circuitry as will be required for higher precision numbers.

SECTION III
CONCLUSIONS

A functional language was created by taking a subset of the strongly-typed language Algol 68. The function language is then mapped into the target self-timed dataflow hardware. This mapping exploits the structure of the functional language to produce a topological layout of integrated circuit masks directly from the behavior written in Algol. The entire process is linearly complex and, therefore, fast, allowing a designer to experiment with various algorithms and data types. Changing data types and the amount of parallelism is easy to do in Algol. Working at the algorithmic level provides good leverage for producing designs that have to meet non-behavioral limitations of power, delay, and area.

A. STATUS

The parser and lexer have been implemented. The self-timed circuits for the Boolean operators and conditional statements have been designed and the IC masks laid out manually. These circuits will be used as the primitive elements in the compiler. The compiler will translate the Algol language expressions into Chisel[8], an algorithmic geometry language. Chisel translates geometry information directly into CIF[5, 12] descriptions of IC masks.

B. FURTHER WORK

The goal of this project is to design prototype integrated circuits quickly in order to compare the effect of many algorithms and datatypes on non-behavioral chip parameters. The methodology allows for further optimizations and improvements[9]. The optimization phase could minimize logic, but this would require a fast minimization program in order for the translation process to remain near linear. Linear complexity is necessary so that comparisons of various designs can be made quickly.

At a different level, the self-timed circuitry has been designed using small equipotential domains[12]. Work needs to be done on techniques to collapse several equipotential domains together to reduce the amount of circuitry required.

The set of IC templates have so far been designed only in NMOS. The same templates have to be designed in other technologies to achieve fabrication technology independence.

Another set of templates can be created that incorporates some testing mechanisms to increase reliability of the generated circuits. Examples of techniques that can be incorporated are LSSD, self-testing templates, and fault tolerant templates.

REFERENCES

[1] A. Avizienis.
Signed-Digit Number Representations for Fast Parallel Arithmetic.
IRE Transactions on Electronic Computers, 1961.

[2] Ken Cline, Mel Cutler, Carl Kesselman, Gary York.
Automated Attribute Optimization for VLSI Systems.
In *Third Annual International Phoenix Conference on Computers and Communications*, pages 106-113. IEEE, Phoenix, Arizona, March, 1984.

[3] Robert Cuykendall, Anton Domic, William H. Joyner, Steve C. Johnson, Steve Kelem, Dennis McBride, Jack Mostow, John E. Savage, and Gabriele Saucier.
Design Synthesis and Measurement.
In *Workshop Report VLSI and Sortware Engineering*. IEEE, Port Chester, New York, October, 1982.

[4] Robert Cuykendall, Anton Domic, William H. Joyner, Steve C. Johnson, Steve Kelem, Dennis McBride, Jack Mostow, John E. Savage, and Gabriele Saucier.
Design Synthesis in VLSI and Software Engineering.
The Journal of Systems and Software 4(1):7-12, April, 1984.

[5] Robert W. Hon and Carlo H. Sequin.
A Guide to LSI Implementation, Second Edition.
Technical Report SSL-79-7, Xerox Palo Alto Research Center, January, 1980.

[6] D.L. Johannsen.
Bristle Blocks: A Silicon Compiler.
Display File 2587, Caltech Dept. of Computer Science, January, 1978.

[7] Stephen C. Johnson.
Yacc: Yet Another Compiler-Compiler.
In *Unix Programmer's Manual*. Bell Laboratories, 1978.

[8] Kevin Karplus.
CHISEL An Extension to the Programming Language C for VLSI Layout.
PhD thesis, Stanford University, January, 1983.

[9] Steven H. Kelem.
A Compiler for Silicon: An Automatic Method for the Translation of High-Level Algorithms into Integrated Circuit Masks.
Technical Report ATR-85(8497)-1, The Aerospace Corporation, September, 1982.

[10] Musaravakkam Samaram Krishnan.
A Structured Layout Design Methodology for VLSI Circuits.
PhD thesis, University of Southern California, May, 1984.

[11] Charles E. Leiserson.
Area-Efficient VLSI Computation.
The MIT Press, Cambridge, Massachusetts, 1983.

[12] Carver Mead and Lynn Conway.
Introduction to VLSI Systems.
Addison-Wesley, Reading, Massachusetts, 1980.

[13] Raymond E. Miller.
Switching Theory.
John Wiley & Sons, Inc., New York, 1965.

[14] John Ousterhout.
A Timing Analyzer for nMOS VLSI Circuits.
In Randal Bryant (editor), *Third Caltech Conference on Very Large Scale Integration*, pages 57-69. Computer Science Division, Electrical Engineering and Computer Sciences, University of California, Berkeley, 1983.

[15] Stephen H. Unger.
Asynchronous Sequential Switching Circuits.
Wiley-Interscience, New York, 1969.

[16] A. Vladimirescu, Kaihe Zhang, A.R. Newton, D.O. Pederson, A. Sangiovanni-Vincentelli.
SPICE Version 2G User's Guide
Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 1981.

END

5-87

DTIC