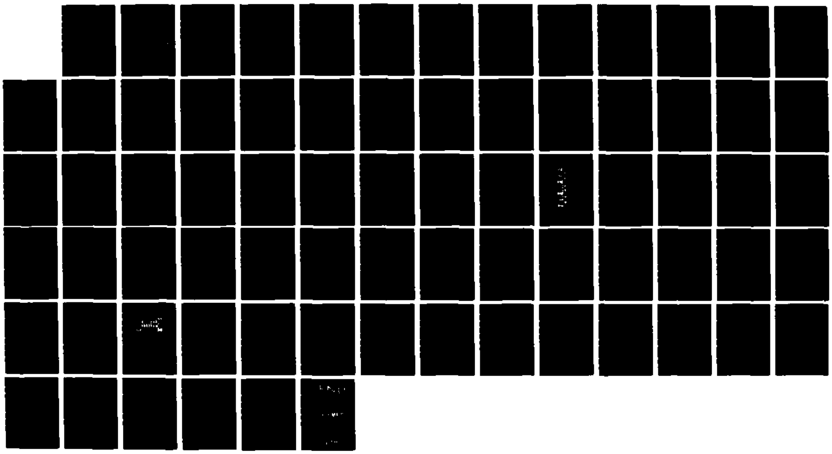
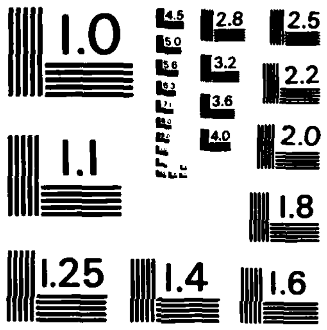
AD-A162 564   SILICON COMPILATION: A SOLUTION TO THE COMPLEXITY OF   1/1
VLSI (VERY LARGE-SCA..(U) ILLINOIS UNIV AT URBANA
COORDINATED SCIENCE LAB P GEE SEP 85 UILU-ENG-85-2230
UNCLASSIFIED   N00014-84-C-0149   F/G 9/5   NL




1.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

September 1985 UILU-ENG-85-2230

12

AD-A162 564

COORDINATED SCIENCE LABORATORY

College of Engineering

SILICON COMPILATION: A SOLUTION TO THE COMPLEXITY OF VLSI CIRCUIT DESIGN

Perry Gee

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

ADA 162564

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-85-2230	5. MONITORING ORGANIZATION REPORT NUMBER(S) N/A
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Laboratory University of Illinois; 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS (City, State and ZIP Code) University of Illinois at Urbana-Chamapign 1101 West Springfield Ave. Urbana, IL 61801	7b. ADDRESS (City, State and ZIP Code) 800 N. Quincy Street Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Joint Services Electronics Program; 8b. OFFICE SYMBOL (If applicable) N/A	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N000-14-84-C-0149
8c. ADDRESS (City, State and ZIP Code) 800 N. Quincy Street Arlington, VA 22217	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO. N/A; PROJECT NO. N/A; TASK NO. N/A; WORK UNIT NO. N/A

11. TITLE (Include Security Classification) SILICON COMPILATION: A SOLUTION TO THE COMPLEXITY OF VLSI CIRCUIT DESIGN.

12. PERSONAL AUTHOR(S) GEE, PERRY

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Interim Technical, final	FROM Sept. '84 TO Sept. '85	September 1985	72

16. SUPPLEMENTARY NOTATION N/A

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	Silicon compilation; VLSI circuit design; synthesis; computer-aid design; logic minimization; PLA generation; placement; routing; layout; integrated circuits.

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

Designing very large-scale integrated circuits requires several months or more and increases as the complexity of the design increases. This thesis proposes to reduce the complexity of the design problem and to reduce the design time by using a high-level language to specify only the behavioral aspects of a circuit. The circuit is then synthesized from this specification.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED [✓] SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code); 22c. OFFICE SYMBOL

DD FORM 1473, 83 APR   EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

SILICON COMPILATION: A SOLUTION TO THE COMPLEXITY OF VLSI CIRCUIT DESIGN

BY

PERRY GEE

B.S., University of California, Berkeley, 1983

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1985

Urbana, Illinois

ABSTRACT

Designing very large-scale integrated circuits requires several months or more and increases as the complexity of the design increases. This thesis proposes to reduce the complexity of the design problem and to reduce the design time by using a high-level language to specify only the behavioral aspects of a circuit. The circuit is then synthesized from this specification.

ACKNOWLEDGEMENTS

I would like to thank my research advisor, Professor I. N. Hajj, for his valuable guidance and suggestions. I would like to thank David Overhauser for his assistance in preparing this thesis. Also I would like to thank Danial Saab for his help with the CAD tools.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

With VLSI chips becoming more complex, it is harder to design and verify the correct operation of a chip. In order to reduce the design time, it is becoming necessary to have a design environment which reduces the complexity of the problem and allows the design process to be automated as much as possible.

There are three interrelated design approaches currently being pursued. They are

- Computer-Aided Design (CAD)
- Silicon compilers
- Expert systems

1.1. COMPUTER-AIDED DESIGN (CAD)

In the computer-aided design approach there are three major disciplines for designing chips: full custom, gate array and master slice. The approach used depends on the performance required, the number of chips that will be produced and the amount of design time available.

In the full custom approach the designer makes all the design decisions. The designer first decides on the data paths to be used. Then he lays out the major blocks in a floor plan. The designer then decides on how to implement each block and usually draws up a schematic of the circuit at the logic level. The CAD system provides an efficient method for the designer to draw his schematic and provides tools so that the schematic can be simulated. When the circuit is satisfactory the designer then decides on how to lay out the actual circuit on silicon. The CAD system provides the designer with several editors to accomplish this task. The designer may choose to lay out each transistor himself by using a graphical e 'itor and draw each of the rectangles, or he may choose to start in a sticks editor and have the CAD system generate the layout from this simpler format. The layout generated from the sticks editor is usually not of high quality, but is fast and can always be modified by a layout editor. While the circuit is being laid out, the CAD system checks to make sure that the circuit meets all design rules. After the circuit has been laid out in silicon, the designer runs a program to extract out the transistors, resistors and capacitors, which are then compared with the schematic to make sure the circuit is correctly laid out. This extracted circuit is also simulated to make sure that the circuit meets specifications.

This method of designing chips is very slow. One obvious way to speed up the design is to save cells from one design and use them in future designs. One method (gate array) has taken this idea to the extreme. In the gate array approach, the only cells that can be used are from a special library. So the designer only has to decide what cells to use and how to interconnect them. The cells in the library are made from transistors of the same size so that all the transistors on the chip can be manufactured up to the point of metalization. So the advantage of this method of designing chips is that very fast turn-around times can be achieved, cost of design is very low and the designer has a very high confidence that the chips will work; however, the disadvantage of this system is that the chips are poorly utilized in terms of area and are usually of low performance because all the transistors are of the same size.

A less restrictive approach is the master slice approach which is a compromise between full custom and gate arrays. Here the designer is restricted to use only PLAs and cells in the predefined library. But unlike the cells in the gate array library these cells have been optimized for speed or area (with the restriction that all the cells are of the same height so that they will abut nicely). This method of design produces denser chips with higher performance than in the gate array method, but the design time is longer, since the transistors in each cell are of different sizes and the the chips cannot be prefabricated as in the gate array approach, but the turn-around time is still rather fast.

1.2. SILICON COMPILERS

In this approach, the circuit is specified algorithmically using a high-level language. The advantages of this approach are that the circuit specifications are technology independent, the circuit functions are inherently easier to understand because the circuit is described in a hierarchal manner, and the development of the circuit would be less prone to errors because all translations that are performed between the levels of the hierarchy are done by the compiler.

The high-level language used by the silicon compiler is an algorithmic language which captures the architectural description of the circuit. This architecture can then be implemented in any technology such as bipolar, NMOS or CMOS. The language does not tie the circuit to the particular methodology used within a particular technology, so the subcircuits could be implemented as gate arrays, programmable logic arrays (PLA), domino logic, etc. So the circuit does not need to be redesigned when improvements or changes in technology occur. The circuits only need to be recompiled for the new technology.

By using a silicon compiler, design time would be reduced, and as a result more complicated circuits could be designed. The hierarchal nature of the language provides a structured way of describing the circuit functions, which makes it easier to understand these functions. Also specifying an electronic circuit in a language allows the designer to separate the logic design from the physical and geometrical aspects of VLSI circuits. Frequently used circuits could be saved in a library, further reducing the design time, but unlike the libraries used in the gate array or semi-custom approaches, this would be a parametrized library of subcircuits which allows for more design flexibility and better optimized circuits.

This method of designing circuits is less prone to errors because all the necessary information needed to use a particular tool would be derived from this description. For instance, if a designer wants to simulate his circuit at the register-transfer level and perhaps at the gate-level he must enter two different descriptions of his circuit. So if the designer made a mistake in translating his circuit he would be simulating two different circuits! In the silicon compiler approach, this mistake would be eliminated because all circuit descriptions would be derived directly or indirectly from the algorithmic description of the circuit.

The current approach to designing chips can be divided into the following tasks, which are shown in Figure 1.1:

1) Architectural Design

2) Logic Synthesis and Design

3) Gate/Circuit Design

4) Layout

5) Simulation and Verification

6) Reiterate above until specs are met

A high-level language allows the architecture to be described in a concise form from which the computer can generate Steps 2 - 5. Since the majority of the errors made during design are from translating between steps, we would like to automate this translation as much as possible.

At each level of the design, several tools are available to aid the designer, such as PLA generators, routers, compacters, etc. By using a high-level language we would be able to free the designer from having to convert his problem into a form that was usable by these programs. Instead, the compiler could call these programs as necessary and convert the data as needed by each of the programs.

1.3. EXPERT SYSTEMS

Expert-systems-based CAD tools are very similar to the silicon compiler design approach. Both methodologies specify the circuit behavior rather than the structural aspects of the design, so expert systems offer the same advantages as a silicon compiler. The difference occurs in the method used to generate the data paths. Here the computer system captures the knowledge of expert human designers. This knowledge is represented as a

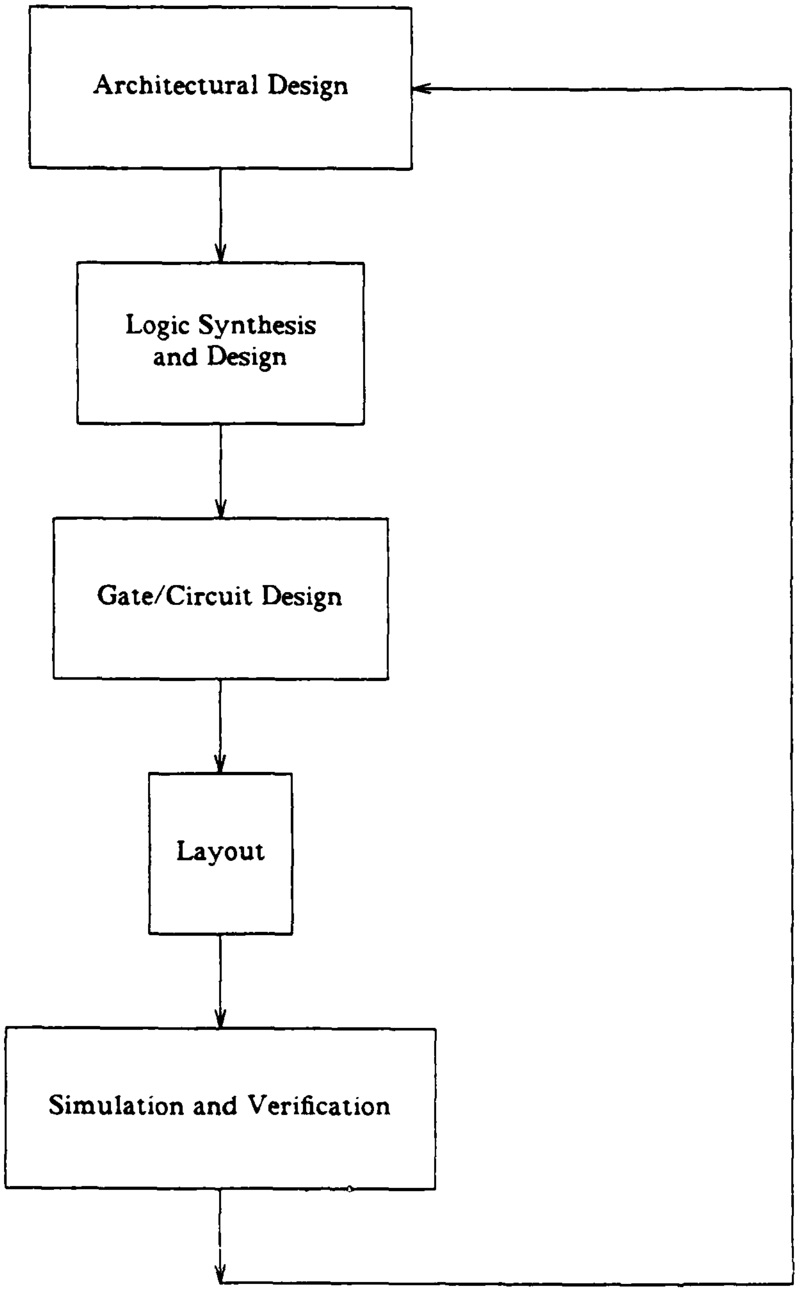


Figure 1.1
Progression of VLSI circuit design

collection of several hundred simple rules, unlike the silicon compiler which represents this same knowledge algorithmically with a few very complex rules and actions. An advantage to expert systems is that new design styles or changes in technology can be incorporated by adding more rules to the knowledge base.

1.4. OVERVIEW

The thesis is organized as follows. Chapter 2 describes in detail two known approaches to silicon compilation and how these approaches differ from the one proposed in this thesis. These approaches are selected as case studies, although other approaches have been proposed recently. Chapter 3 focuses on the tools necessary for automatic circuit synthesis. Most of these tools are used in the CAD design approach and are covered in the order they are used in the silicon compiler. Heuristic logic minimization is covered, followed by the methodology chosen in two successful programs MINI [HON 74] and ESPRESSO [BRA 84a]. Next, PLA folding is covered. First, a heuristic method developed by Hachtel et al. [HAC 82] is discussed. Then an optimal algorithm based on the work by Lewandowski et al. [LEW 84] is reviewed, which is followed by a short section on PLA generation. Automatic subcircuit placement is discussed next. The approaches covered in this section are min-cut partitioning, quadratic assignment, iterative improvement and simulated annealing. This is followed with a detailed discussion of hierarchal iterative improvement, which is a new approach developed in this thesis. Chapter 3 closes with a discussion on routing subcircuits. In Chapter 4, the compiler developed in this thesis is discussed, followed by a case study of a special purpose processor for hardware LU factorization designed using the compiler. Chapter 5 summarizes the problems encountered with the compiler and possible solutions to these problems. The Appendix describes the complete silicon program which is used to synthesize the processor discussed in Chapter 4.

CHAPTER 2

BACKGROUND

Silicon compilation is still in its infancy. Many people are currently working on developing silicon compilers. At Bell Laboratories a compiler called Xi is being developed [FEL 83]. Carnegie-Mellon has the Design Automation system [THO 83] and MIT has been working on the MacPitts silicon compiler [SOU 83] for some time. The University of California at Berkeley is developing the HAWK/SQUID system [NEW 85] and the University of Illinois has ARSENIC [GAJ 84]. Industry has also been actively pursuing this field. IBM is developing the Yorktown compiler [BRA 84b] which is based on APL [BRE 84]. Silicon Compiler's Inc. is developing the Genesil System [CHE 84], while the Concorde compiler is being developed at Seattle Silicon [YOU 85]. MetaLogic has also entered into the silicon compiler field with Metasyn, which is based on the MacPitts compiler [YOU 85]. HAWK/SQUID, Xi, and the Yorktown compiler implement the logic equations as PLAs. ARSENIC is also a PLA-based system, but considers routing to be an integral part of the cell [LUR 84]. Both Carnegie-Mellon and MIT use data path synthesis. The Concorde compiler is different from the other compilers. The Concorde compiler supports the development of both analog and digital circuits. Among the best compilers, based on speed and area, are the Carnegie-Mellon and the MIT compilers. These two compilers will be discussed in detail in the following sections.

2.1. MacPITTS

There are many approaches to silicon compilation. The approach used at MIT is similar to data path synthesis and the masterslice approach used in traditional CAD systems, except that the MacPitts compiler allows the designer to introduce controls such as conditional data flow or looping. Like other compilers, the designer specifies the behavior of the circuit rather than its structural aspects. Each language feature and concept is carefully considered on the basis of usefulness and the ability to synthesize it. The designer is able to make speed/area tradeoffs by specifying the degree of parallelism/serialism in the circuit. The designer can instruct the MacPitts compiler to construct multiple parallel finite state machines or to construct a pipelined machine.

To design a circuit using MacPitts, the designer must understand two concepts: state transitions and forms. Forms are the arithmetic and control expressions. They are composed of an operator and arguments. The syntax and semantics of the language are similar to LISP, but unlike LISP, the arguments to forms are executed in parallel. State transitions are implied by forms. For instance (setq b (+ b 1)) will increment b after the next state transition.

The MacPitts specification language restricts the user to two data types: integer and boolean. These two data types are implemented differently in silicon. Boolean type variables are implemented in the control section or in the flags sections. While integer type variables are implemented in the data path and are of the same length. All storage elements are master-slave flip-flops and are synchronized by the system clock, while non-storage variables can be modified asynchronously. To reduce the number of side-effects, all flip-flops are designed so that the new outputs cannot modify the inputs during a state transition. The MacPitts compiler constructs the circuit by first determining what specifications cannot be executed in parallel. These specifications can then share physical units in the data path. The MacPitts compiler automatically provides the controls necessary to switch the proper input signals into the appropriate hardware elements. Next, MacPitts determines which actions cannot be executed simultaneously, such as actions resulting from conditionals, and shares as many hardware elements between the two actions as possible. The physical units are selected from a standard library of cells consisting of adders, subtractors, shifters, comparators, etc. The MacPitts compiler is different from data path generators in many important ways. The MacPitts compiler provides a cond operator which allows control-flow. This powerful operator is parallelized and may be executed as a finite state machine or with multiplexers. The compiler automatically determines which units can be multiplexed, and the extra controls needed are automatically routed. The compiler also generates hardware to start finite state machines in a known state. The MacPitts introduces some side-effects not present in conventional programming languages, since the MacPitts tries to execute the arguments of forms in parallel instead of serially. The degree of parallelism is specified by declaring the integer variables as registers or ports and using the command *always*, which implements cond as a finite state machine with one state, so a fully pipelined machine could simply be designed by changing all variables to registers.

The MacPitts compiler also has an interpretor which runs off the circuit specification for simulating the circuit. Since no translation of the circuit specification is necessary to run the interpretor, errors are reduced and the designer does not have to worry about simulating a different circuit.

The compiler also is able to estimate the delay through the hardware. The performance analyzer is intended to assist the designer locate source statements that cause the compiler to generate critical paths. The performance analyzer processes each statement sequentially and flags all critical paths and any statement which exceeds the user specified clock width. The delay used for each module takes into account capacitance, transistor sizes and number of logic levels. But it does not consider the delay through the control circuitry when modules are shared or layout specific delays, such as wire length. The analysis is similar to conventional timing verifiers, but much simpler. The performance analyzer is intended only to be used for a rough estimate during program development. The final circuit must still be extracted and verified to make sure that it meets all timing specifications.

Current work on the MacPitts compiler is in partitioning designs, developing test generation patterns and designing circuits that are easily testable.

2.2. CMU-DA

Carnegie-Mellon University has developed a silicon compiler with two subcompilers:

one which includes an expert system and the other which uses a simple parameterized cost algorithm to bind cells. The CMU compiler is similar to the MIT MacPitts compiler: both are based on data path synthesis.

The CMU compiler is part of the Carnegie-Mellon University Design Automation system (CMU-DA). The input is an ISPS behavioral description language [BAR 81]. This description is compiled and an internal data flow representation (value trace) is generated. The value trace can then be used by other design programs, such as graphics programs, to display the value trace, circuit optimizers, partitioners, and control step allocators to determine the amount of parallelism/serialism. The value trace is a directed acyclic graph representation which simplifies the programs needed to recognize and implement design features. As in optimizing compilers used for programming languages, the nodes of the value trace represent operators, and the arcs represent the data flow from one operator to another. In addition, the compiler adds control constructs to represent conditionals and subroutines.

The value trace is divided into units called VT bodies. Each unit corresponds to a subroutine or a labeled block. Each body is further divided into three operators: arithmetic-logic, control, and value trace specific operators. The arithmetic-logic operators are those that are defined in ISPS. Control operators represent conditionals and subroutines. The value trace specific operators are operators with special needs in hardware synthesis like reading a subfield, or performing sign-extension.

The compiler also performs many classical optimizing transformations before generating the value trace, such as constant folding (evaluating a constant expression at compile time), common sub-expression elimination, VT body inline expansion and encapsulation, code floating (code which is invariant inside a loop body is moved outside the loop), and loop unrolling (a constant loop is replaced with a finite number of calls to the loop body).

When the value trace is first generated, it contains only necessary controls and data dependencies. The value trace in this form represents a machine with maximum parallelism. The control step allocator assigns value trace operators to states (control-steps), which can be executed serially. It also specifies which values need to be stored in the next state. The control-step allocator enables the designer to make speed/area tradeoffs by adjusting the amount of serialism.

To generate a circuit each value trace element must be bound to hardware. For instance, operators are bound to ALUs, stored values to registers, and the data flows as multiplexers and buses. The two subcompilers use different methods for data path synthesis. The first one, EMUCS, uses a simple parameterized cost algorithm to bind cells, while the second, DAA, uses an expert system to determine the best bindings. EMUCS attempts to implement the value trace with minimal *cost*, where cost is a function of the amount and the complexity of the hardware, although any quantitative parameter, such as power, or any combination of parameters can be used. The algorithm analyzes the existing data path and decides which element to bind and modifies the data path as necessary. It then reiterates this procedure until all elements are bound.

In the analysis phase, cost tables are generated to determine what should be bound. The cost tables are based on the need to use, create and/or modify hardware in order to make a binding. A powerful feature of EMUCS is that by changing the cost table database, a different design will be created from the same value trace. The algorithm chooses the least costly element to bind, which satisfies all the hardware design rules. For instance, two operators cannot be bound to the same ALU unless the operators occur in different states. EMUCS not only considers costs in the present step, but also the cost of binding the element in the future.

Currently, EMUCS can only process a single VT body. However, the inline expansion capability reduces most programs into a single VT body. As a test, the MCS6502 processor instruction set was implemented. Since EMUCS only uses multiplexers and point-to-point connections, a bus was manually inserted and a comparison of the two designs was made. Since the EMUCS has good design interactions, it is a simple matter to add/subtract extra buses, set break points, examine the value trace, and bind hardware elemen s. The two designs have very few differences. The major difference is that the added bus reduced the number of multiplexers that are needed. Both designs used the same number of states and state registers. The bus design required slightly more area, but the bus design is also easier to test. One drawback with EMUCS is that it maps VT operators to units of the same width, whereas it should map the operators to units of equal or greater width.

The second subcompiler, DAA, uses a knowledge-based expert system to determine the *best* bindings. DAA is written with the OPS5 [HAY 84] knowledge base expert system. The program consists of about 300 rules for designing VLSI circuits. DAA consists of three major components: working memory which describes the current problem, a rule memory which has the actions to be taken if the conditionals are satisfied, and a rule interpreter which decides which rule to apply. The rule interpreter scans the rule memory for rules in which all the conditionals are true. If there is more than one applicable rule, then the rule which deals with the most recently modified element in working memory is chosen first. If there is still more than one rule, then the rule with the most conditionals true is chosen because this corresponds to the most specific rule.

First DAA assigns registers to all variables declared in the ISPS description. Now a VT body is chosen and all operator outputs are assigned to registers. Next, the expert subsystem removes extraneous registers. For instance, registers after combinational logic where the inputs are stable are unnecessary. The expert subsystem also merges operators into ALUs and attempts to share temporary variables.

Both synthesis programs were used to implement the MCS6502 processor, and both produced good designs. EMUCS allocated registers more effectively than DAA, although the DAA design was more comprehensible because all the declared ISPS registers were saved. Both algorithms consolidate hardware operators into ALUs, but DAA tends to leave simple operators as separate modules. EMUCS only uses point-to-point interconnections and multiplexers, but will use buses if they are inserted manually, whereas DAA uses a multiplexer strategy adding buses in a limited number of situations. DAA also has the capability to change the control-step allocation during synthesis if all the delays will not fit within one state. The EMUCS program is written in C and uses much less CPU time and memory than DAA, which is written in a lisp-based system. Overall, EMUCS performs better than DAA.

2.3. PROPOSED SILICON COMPILER

The silicon compiler developed in this thesis is in some ways similar to the two compilers just discussed. Input to all compilers is an abstract specification of the design. For the compiler in this thesis, the abstract specification resembles the C programming language [KER 78]. This language was chosen instead of a register transfer language, because fewer details of the design need to be specified with a highly abstract language. This will enable the designer to design his circuits faster, but the optimization techniques used by the compiler must be more sophisticated.

Both the MacPitts compiler and the CMU-DA compiler use subcircuits from a standard cell library to implement the data path. This does not restrict the type of circuits which can be implemented, since any logic function can be synthesized by using only 2-input **NAND** gates or 2-input **NOR** gates. However, designing a circuit with only 2-input **NAND (NOR)** gates would over-tax the placement and routing routines. In this thesis, I propose to implement the subcircuits with programmable logic arrays (PLAs). PLAs provide a structured way to implement random logic. The placement of the subcircuits would be easier because each PLA would replace several gates, thus reducing the size of the placement problem. The size of the routing problem is also reduced, because only the external connections of the PLA need to be made. Another reason for using PLAs is that all the technology-dependent aspects of the design is in the PLA. By using a template-driven PLA generator, new technologies can be incorporated by simply redesigning the templates. To change technologies in the standard cell library approach, every cell in the library must be redesigned manually, which may take several weeks as opposed to several hours to redesign the PLA templates [MAH 84].

The above two compilers only perform the classical optimizing transformations and simple logic minimization. The logic minimization only considers sharing functional units in different states. In the compiler developed in this thesis, the powerful techniques employed by ESPRESSO [BRA 84a] will be performed automatically. This will free the designer from having to use hand optimization techniques, such as Karnaugh maps, which are difficult to use when the number of variables exceeds six or when several equations must be minimized together.

In order to speed up the development of the silicon compiler, many sophisticated routines from the computer-aided design approach will be used as subroutines. Many of these tools are developed at the University of California, at Berkeley. The use of these tools in the HAWK/SQUID [NEW 85] system at UCB is much different from the approach used in this thesis. At UCB, the HAWK/SQUID system is really the traditional CAD design approach of Section 1.1. The HAWK/SQUID system does have a procedural interface, but this is at a very low level. This symbolic description works with geometric data [ELL 82], but the geometries can be parameterized. This allows cells with different properties to be synthesized from the same symbolic description. My silicon compiler will insulate the designer from these geometrical problems, so that he will be able to concentrate on the algorithmic aspects of the problem.

CHAPTER 3

SILICON COMPILATION

3.1. WHY WE NEED SILICON COMPILERS

With VLSI chips becoming more complex, it is harder to design and verify the correct operation of a chip. In order to reduce the design time, we would like a design environment which reduces the complexity of the problem, and we would like to automate the process as much as possible.

Computer scientists have successfully worked with complex problems for many years. They approach the problem in a hierarchal fashion. Most programs are designed using a top-down design with a bottom-up construction. To facilitate this design approach, programmers use a structured programming style. High-level languages are ideal for structured programming since they already contain the constructs for expressing these concepts. The main features of structured programming are hierarchy and modularity. Programs are written as a collection of modules. Each module performs a well-defined function and can be built up from simpler modules. Another advantage of modules is that they can be tested separately, which reduces the time necessary to test the entire program. Modules also aid the conceptual understanding of a problem, since programs or other modules which use a particular module only need to know how to interface with it and not its implementation details.

This same approach can be used to design complex VLSI chips. First, a high-level language needs to be developed so that designers are able to express the solution in a structured form. This design approach can automate the more tedious aspects of designing a chip. A high-level language description of a circuit is also easier to understand than the corresponding logic diagram or gate description of a circuit.

3.2. TOOLS NECESSARY FOR SILICON COMPILERS

In order for silicon compilers to be totally automatic, they must be able to accept boolean equations which are not minimized. If the compiler were to implement this directly, the resulting circuit would be too large and slow. So silicon compilers must incorporate logic minimizers. The circuit can then be synthesized from these minimized boolean equations. There are several well-known methods for implementing boolean equations. Equations with a small set of variables can be implemented in read-only memory, while equations with many variables can be implemented in random logic or by using a PLA. Random logic realizations are much denser than PLAs, but they are hard to test and push placement and routing routines to their limits. PLAs, on the other hand, are easy to test. They can also be made to utilize the area much more effectively by folding the rows and columns of the PLA. Placing and routing PLAs are much simpler than placing and routing random logic because the number of PLAs to be linked together is usually small.

3.2.1. LOGIC MINIMIZATION

Logic minimization is a necessary tool for silicon compilers, because it will decrease the area and increase the speed of the circuit. Logic minimizers allow the designer to use equations in forms that best describe the problem without having to worry about how the equation should be written to get the best speed and area. For instance, if our compiler were to implement several equations in n-level logic, then we would want to find as many common subexpressions as possible to reduce the gate count, or if we implement the equations in a PLA then we would like to reduce the equations while considering the foldability of the resulting equations.

Traditional methods of logic minimization, such as Quine-McClusky [DEN 84], which achieve a minimum solution by first computing all prime implicants and then computing the minimum cover, are too slow and require too much memory. The number of prime implicants has been shown to be bounded by C^n/n where n is the number of variables. So heuristic methods have been developed which give a minimal solution. Two programs which use different heuristics are MINI [HON 74] and ESPRESSO [BRA 84a]. Both programs try to minimize the number of implicants and do not consider minimizing the complemented form or the foldability of the resulting cover.

3.2.1.1. MINI

The MINI approach to the problem is to sacrifice optimality and to use a simple cost criterion. The cost function is simplified by assigning each implicant a constant cost instead of a cost proportional to its size. MINI finds a near minimal solution by iteratively improving an initial solution. Each iteration consists of three steps: enlargement, reduction, and reshaping. MINI uses a positional cube representation for the terms since many boolean operations can be done fast and efficiently. For instance, the **OR** of two cubes is simply a list containing both cubes, the **AND** of two cubes is a bitwise **AND** of the two cubes and testing if Cube A covers Cube B is done by checking if (bitwise) **NOT** A **AND** B equals 0.

The enlargement phase merges terms if it can contain the term with a larger term in the *don't care* set or in F, where F is the logic function. The enlargement phase is where the number of cubes in the solution decreases significantly. In this phase the cubes are sorted in order of merging difficulty. This is done heuristically by noting that if a cube can be expanded to cover many cubes then the cube must have many 1s where many other cubes have 1s. The algorithm then takes each cube one at a time and finds a prime cube covering it and, hopefully, many other cubes in the list. Then all cubes covered by the prime cube are deleted and the algorithm continues with the next cube.

The prime cube in the enlargement is computed by expanding the Cube C against F. The expansion of Cube C is done by successively expanding one part at a time. The expansion of part k is done by first computing the k-conjugate sets H(C,k) and Z(C,k).

$$H(C,k) = \left\{ g_i \mid C,\, g_i \text{ are } k\text{-conjugates and } g_i \in F \right\}$$

$$Z(C,k) = bit \bigcup_{h_i \in H(C,k)} k^{th} \text{ part of } h_i$$

The single part expansion of C is now defined to be the Cube C with its k^{th} part replaced with $\overline{Z(C,k)}$. The parts are expanded in the order that will cover the most cubes that are not covered by the original cube.

In the reduction phase, each implicant is reduced to its smallest possible size while maintaining proper coverage of the minterms. This phase removes redundant terms and increases the probability of further merging in future phases.

The reshaping process finds all pairs of terms which can be converted to two disjoint terms. This is done because the disjoint F leads to an ever decreasing solution. Two cubes, A and B, are reshaped only if their distance is two and one part of A covers the corresponding part of B. If we let i and j be the parts in which A and B differ and let part j of A cover part j of B, then Cube A is **AND**ed with the complement of part j of B and Cube B is **OR**ed with part i of A. Note that this reshaping process is actually a special case of the consensus operation. The reshaped cube of B is the consensus term between A and B.

The MINI program has been improved in many of its derivatives. In order to do enlargement the algorithm needs the complement of the function. In the original MINI program this is done using the disjoint sharp operation, but has been improved by using a fast complementator based on unate functions [BRA 82]. Another program SPAM (Stanford Programmable Array Minimizer) has added partitioning to allow MINI to handle very large problems [KAN 81].

3.2.1.2. ESPRESSO

The ESPRESSO algorithm is very similar to the MINI algorithm. It too finds a minimal cover using iterative improvement. The main loop of the ESPRESSO algorithm consists of expanding each of the cubes, finding an irredundant cover for the cubes and reducing the cover. On the first iteration, the essential prime cubes are found and moved to the *don't care* set, so that they will not be needlessly processed in the minimization loop. After the function is minimized, the cubes are moved back.

The expansion phase is where the cover is reduced the most. In this phase, each cube is expanded sequentially, starting with the largest cube. Although this ordering is not as effective as the ordering used in MINI, it is fast and the cube ordering is not as critical as the part ordering for expansion. The ESPRESSO algorithm then finds a set of parts to expand simultaneously. This is done by building two matrices: a blocking matrix and a covering matrix. The blocking matrix **B** is determined by the Cube **C** to be expanded and F. Entry $\mathbf{B}_{ij}$ is 1 if the i^{th} cube of F and Cube **C** contain both true and complemented forms of the j^{th} literal and 0 otherwise. The covering matrix is defined by $\mathbf{CM}_{ij}$ equals 1 if the i^{th} cube of F and Cube **C** cover the j^{th} literal. The expanding set is now chosen to be the set of parts which cover every row of **B**, but as few rows of **CM** as possible. This heuristic choice for the expanding set hopes to maximize the number of cubes covered by the super cube and minimize the number of literals in the super cube.

The expansion phase generates a prime cover for F. In the irredundant cover phase, any redundant cubes are deleted. This is accomplished by checking if $(DC \bigcup F - \{c\} \bigcup F)$ is a tautology where DC represents the *don't care* cover.

The reduction phase generates a new cover by replacing each cube by a cube contained in it. This phase allows ESPRESSO to move away from a local minimum solution. The reduction algorithm is similar to the one used by MINI, except ESPRESSO orders the cubes by taking the largest cube and then ordering the rest of the cubes according to their distance from the initial cube. Distance is measured by the number of mismatched vertices in each of the cubes. ESPRESSO then sequentially reduces each cube in the list.

ESPRESSO iterates the above three phases until there is no improvement in the cover which is measured by the number of product terms and the number of literals. The algorithm then makes a systematic search for additional primes which are beneficial to the cover. If some are found then the main loop is reexecuted. Otherwise, the essential prime cubes are moved back to the cover and the algorithm terminates.

3.2.2. PLA FOLDING

Programmable logic arrays are regular structures for implementing random logic. PLAs consist of two planes: the AND plane and the OR plane. The AND plane forms all the minterms for the function and the OR plane forms the concatenation of the minterms. A PLA is usually specified by a personality matrix. The matrix has one row for each product term and one column for each input or output. The columns corresponding to the input represent the AND plane and the output columns represent the OR plane. The i, j entry in the matrix is a 1, 0 or X if the i^{th} product term depends on the true, complemented or *don't care* form of the j^{th} variable. A 1 or 0 indicates the presence of a transistor in the PLA (Figures 3.1 - 3.2).

Most PLAs are very sparse, so that a straightforward implementation of the personality matrix would waste a significant amount of area. However, if we let each two inputs (outputs) share a column and each two product terms share a row, then we may be able to reduce the area by 75 percent. This sharing of columns and rows is called folding.

Of course, not all columns (rows) of a PLA can be folded. Before two columns can be folded they must be disjoint. Also folding two columns introduces other constraints. Folding column i on top of column j forces all the rows containing variable i to be above all the rows containing variable j, because two signals in a PLA cannot be intermixed. So to specify a folded PLA we must specify the relative positions of the rows and columns. This specification is realizable if it does not introduce any cyclic constraints on the rows or columns.

Finding the optimal folding set is an NP-complete problem. There are several heuristics used for finding a good solution, such as bipartite folding or using graph theory techniques. Although finding the best folding set is NP-complete, optimal algorithms do exist and are widely used because additional constraints from physical considerations limit the size of the problem. Namely, the number of input/output terms will be less than 200 because of parasitic losses and delays. Most optimal algorithms use heuristic algorithms

```
   Avout    =  Uvin & cy2 & Lvin & Avin;
 overflow   =  sel0 & !sel1 & Movfl | sel0 & !sel1 & Aovfl;
  parity    =  sel0 & !sel1 & Mparity | sel0 & !sel1 & Aparity;
   Lvout    =  Uvin & Lvin & sel0 & !sel1 & Avin | cy2 & Lvin & sel0 & !sel1;
   Uvout    =  Uvin & cy2 & Lvin & !sel0 & sel1 & Avin | Uvin & cy2 & !sel0 & sel1;
 Umuxcntl   =  Uvin & cy2 & sel0 & sel1;
 Lmuxcntl   =  cy2 & Lvin & sel0 & !sel1;
 Amuxcntl   =  cy2 & !sel0 & sel1 & Avin | cy2 & !Avin;
      f_0   =  !sel0 & !sel1;
      f_1   =  sel0 & !sel1;
      f_2   =  !sel0 & sel1;
      f_3   =  sel0 & sel1;
```

where

& represents logical **AND**

| represents logical **OR**

! represents logical negation

Figure 3.1
Boolean equations representing PLA MAIN generated from silicon compiler using code in Figure A.1 (Appendix)

PERSONALITY MATRIX

```
 (inputs)     (ouputs)
-1-0------   -------1----
1111------   1-----------
----01----   ----------1-
1-1110----   ---1--------
-11-10----   ---1--1-----
11--01----   ----1-------
----00----   --------1---
-1--01----   -------1----
11--11----   -----1------
----10---1   --1---------
----10-1--   -1----------
----10--1-   --1---------
----101---   -1----------
----10----   ---------1--
----11----   -----------1
```

Figure 3.2
Personality matrix of PLA MAIN derived from the above equations

with backtracking or a branch and bound technique.

3.2.2.1. HEURISTIC FOLDING

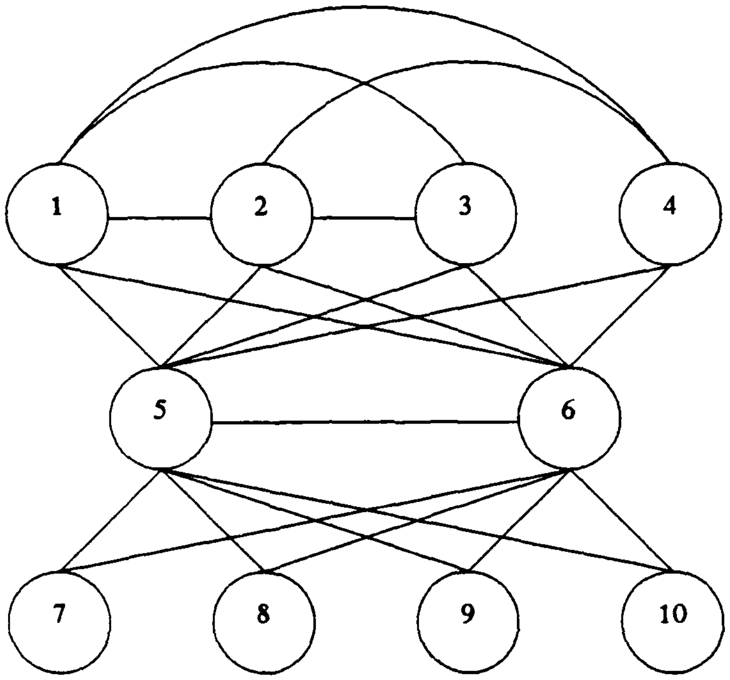


Figure 3.3
Graph representation of AND plane in PLA MAIN

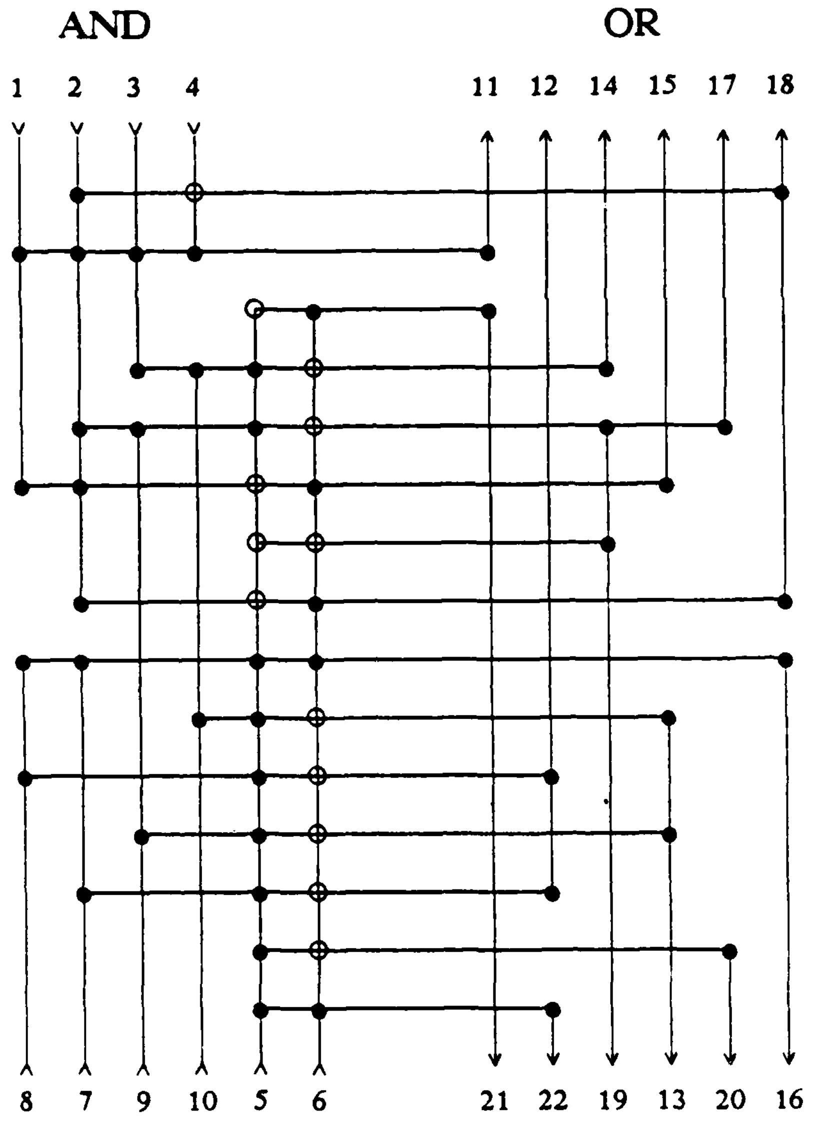
Column folding will be described first. Then we will discuss how this algorithm can be enhanced to include row folding. The optimal column folding problem is reducible to the graph problem of finding the largest set $\vec{A}$ such that $\vec{A}$ is a matching of $G(\hat{V}, E, \vec{A})$ and $G(\hat{V}, E, \vec{A})$ has no alternating cycles [HAC 82]. The vertices of the graph correspond to the columns of the personality matrix. An undirected edge is added from vertex i to vertex j if column i cannot be folded with column j (Figure 3.3). The set $\vec{A}$ consists of a set of directed edges. An edge in this set from vertex i to vertex j is interpreted to mean column i is folded on top of column j. Originally the set $\vec{A} = \emptyset$.

The algorithm works by placing all the vertices in two sets: an upper folding set. U and a lower folding set L. Then a suitable folding is found for the first vertex in U by trying every vertex in L not connected by an edge. until a folding is possible. When a folding is found. both vertices are deleted from sets U and L. The process then continues with the next vertex in U until no more foldings are possible.

To check if two columns can be folded. we must determine if an alternating cycle will be created. Checking if an alternating cycle exists if the directed edge (u. l) is added can be done efficiently by maintaining a list for each vertex v $\in \hat{V}$. The list will consist of vertices which can visit v on an alternating path. Now. to see if there exists an alternating cycle. we let F_u be the set of vertices which can reach u on an alternating path. If any of these vertices are adjacent to l (connected by an undirected edge) then we have created an alternating cycle. so the pair (u. l) cannot be folded. Updating the list of vertices on alternating paths simply consists of concatenating the lists which are adjacent to l to every list in F_u.

Since folding two columns introduces additional constraints. the order in which we fold vertices is important. Heuristically this is determined by sorting the upper folding set in descending degree and the lower folding set in ascending degree. The motivation for selecting u with minimum degree is to minimize the probability that u will be adjacent to a later l. This will minimize the creation of alternating cycles. Next. l is selected with maximum degree because this means that l is disjoint from only a few vertices. So if we can fold it. we should. because there may not be a chance to fold it later. The PLA in Figure 3.1 is folded using this algorithm and is shown in Figures 3.4a - 3.4c.

This same algorithm can be used to perform simple row folding by performing column folding on the transpose of the personality matrix. More importantly. this algorithm can also handle constrained row folding for implementing PLAs with an AND-OR-AND or OR-AND-OR architecture. In AND-OR-AND (OR-AND-OR) PLAs. when a row is folded to the left it forces all inputs (outputs) in that row to be in the left AND (OR) plane



● represents true signal
○ represents complemented signal

Figure 3.4a
PLA MAIN after folding

PERSONALITY MATRIX

-H-L--	-----H
HHHH--	H-----
----01	1-----
H-H110	--H---
-H1-10	--H-H-
HH--01	---H--
----00	--1---
-H--01	-----H
11--11	-----1
---110	---1--
1---10	-H----
--1-10	---1--
-1--10	-H----
----10	----1-
----11	-1----

Figure 3.4b
Folded personality matrix representation of PLA MAIN

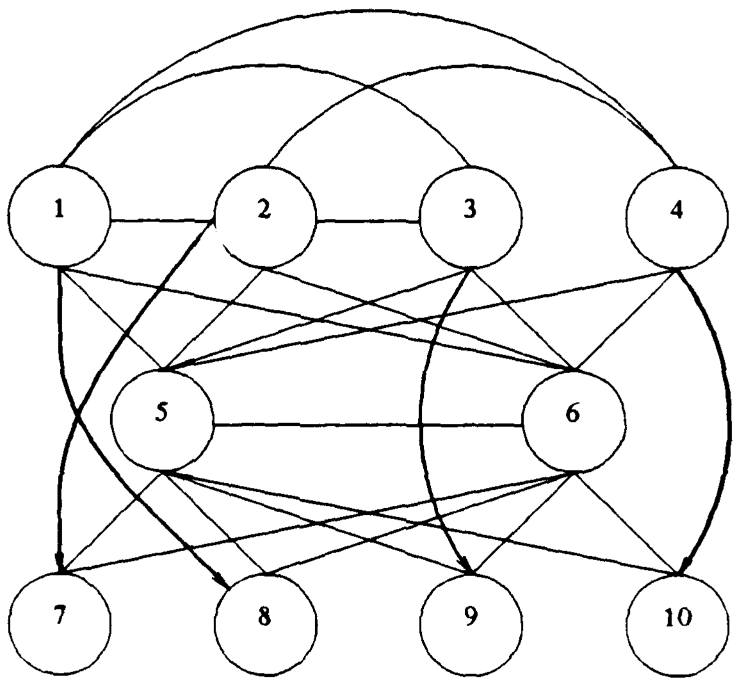


Figure 3.4c
Graph representation of folded AND plane in PLA MAIN

and likewise for the row folded right. This constraint is easily satisfied by modifying the updating algorithm for the sets U and L. Now the vertices in U which share an input with row l must be deleted and all vertices in L which share an input with u must be deleted. This step simply deletes the rows which cannot be folded left from the left candidate set and the rows which cannot be folded right from the right candidate set. The algorithm can also be used to perform mixed row and column folding. For this case, column folding followed by row folding will be considered. An additional directed graph G_{rc} is used to satisfy the additional constraints imposed by column folding. In this graph the vertices represent rows, and there is a directed edge from vertex i to vertex j if row i must be above row j. Obviously if there are any cycles in G_{rc} the folding is not implementable. When two rows are folded together, their corresponding vertices are merged. So when the row folding algorithm is run it must also check to see that no cycles are introduced in G_{rc}.

All algorithms have a worst case running time of $O(n^2)$, but by using sparse matrix techniques for computing the transitive closure of $\vec{A}$, the average running time is only O(n). Constrained row folding or mixed row and column folding have the same time bound because the additional constraints are checked by using a constant amount of time in each iteration.

3.2.2.2. OPTIMAL FOLDING

There are several reasons for considering an optimal algorithm for folding PLAs. First, the size of the PLA is limited by physical constraints. Second, the problem may be a one-time design problem and it may be worthwhile to spend several hours searching for an optimal solution and third, the classification of the problem as being NP-complete does not imply an exponential-time algorithm. A problem being NP-complete refers to its worst-case behavior. So a polynomial average time bounded algorithm may be found.

In this algorithm, the folded PLA is represented by an upper ordered set. In this representation, the unfolded lines are ignored. An n-column folded PLA is then represented as an n-tuple with each coordinate representing a column. The n-tuple represents the folded lines sorted according to the height of the cuts. Each n-tuple represents a class of PLAs.

The optimal PLA is found by examining all possible ordered sets of lines. The search proceeds by lexicographically ordering the lines and examining the ordered sets in lexicographic order. To keep search time to a minimum, several methods are used to prune the search tree [LEW 84]. First, if an ordered set of cardinality n is an implementable upper ordered set, the n columns must be disjoint from at least n other columns, since the unfolded lines are not in the n-tuple. If this condition is not met, then the subtree rooted at this node can be pruned. If the current n-tuple augmented by the maximum number of foldable columns not included in the n-tuple is inferior to the largest folding found so far, then these subtrees can be pruned.

Two upper ordered folding sets are equivalent if they are the same size. Clearly, equivalent foldings do not need to be examined. Also, if a PLA is rotated 180 degrees we have the same PLA, but a different representation. Consider the upper ordered set $(\alpha_1, \alpha_2, \cdots \alpha_n)$. If every line disjoint from this set is lexicographically less than α_1, then this subtree corresponds to a 180-degree rotation of a previously examined n-tuple.

Another method used to speed processing is to estimate a lower bound on the number of folds in the optimal solution. This will allow the algorithm to prune suboptimal solutions earlier. A good way of obtaining a lower bound greater than zero is to first fold the PLA using a heuristic algorithm! An estimate of the upper bound of the size of the optimal folding will enable the search to be terminated when an upper folding set of this size is found. A simple way to obtain an upper bound is to optimally fold two smaller PLAs. First fold the inputs, then fold the outputs. Clearly, the sum is an upper bound since there are fewer constraints to be satisfied. A significant amount of processing time can be saved because the algorithmic complexity is exponential in the size of the problem.

Using the above methods to prune the search tree have worked well for both sparse and dense PLAs. In sparse PLAs there are few restrictions on folding, so the optimal solution is found quickly. For dense PLAs the folding set is small so the optimal folding can be found quickly. For PLAs between the two types, a lot of time is spent searching for a solution, even though the optimal solution has been found. This is where the upper bound to the solution is useful. Using these techniques Lewandowski was able to optimally fold PLAs with 60 inputs and outputs using only a few minutes of CPU time [LEW 84].

3.2.3. PLA GENERATION

Generating the PLA from a folded personality matrix is straightforward. The newer PLA generators like PANDA [MAH 84], generate multiply folded PLAs. The most common approach is the building-block method [CHE 83], also known as the bristle-block method [JOH 79].

In the building-block method several templates are designed. The templates contain mask information needed to build the PLA. For instance, the generator TPLA needs 6 templates to produce a non-folded PLA. TPLA needs templates to describe the core of the AND plane, the left and right sides of the AND plane, the top and bottom sides of the AND plane and similarly for the OR plane. Many of the generators which produce multiply folded PLAs need over a hundred templates to completely describe the PLA.

The advantage of the template driven method is that the technology specific rules are all contained in the templates. A PLA can be generated in a new technology or style by simply changing the templates. For instance, the templates used by PANDA can be generated by using any layout editor. The template used by PANDA is actually a sample PLA which includes an example for every possible combination of cells. Figure 3.5 shows a PLA generated by PANDA.

3.2.4. PLACEMENT

With the advent of VLSI, the arrangement of interconnected logic blocks poses a difficult combinational placement problem. The problem is how to place the blocks on the chip in the smallest possible area subject to various electrical and positional constraints.

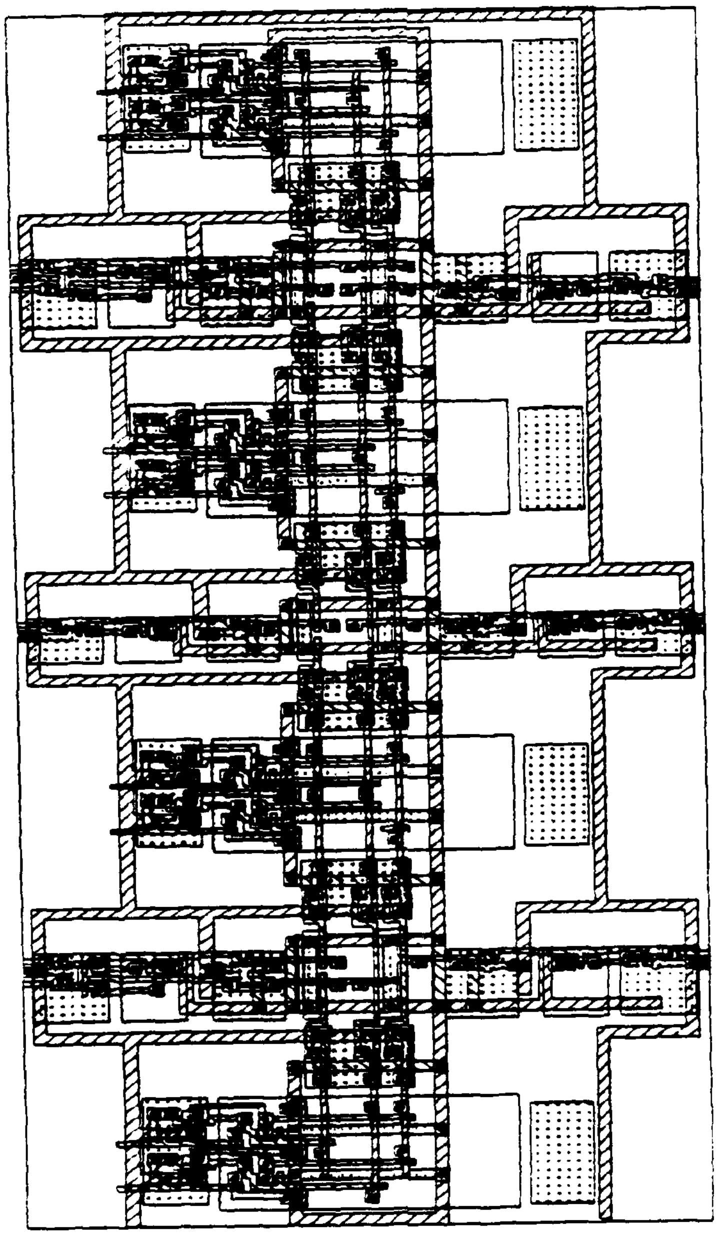


Figure 3.5
Folded PLA using PANDA

There are several approaches to VLSI cell placement. The most popular choices are algorithms based on the min-cut principle, quadratic assignment, iterative improvement, and simulated annealing.

3.2.4.1. MIN-CUT PARTITIONING

In min-cut partitioning, the blocks are represented as nodes in a graph and interconnections are represented as edges. The objective is to split the graph into q subgraphs where each subgraph is *small enough* to be placed optimally. In classical placement algorithms, an attempt is made to minimize the length of all interconnections. Usually the distance cannot be computed exactly so the length of the minimal spanning tree is used or one-half the perimeter of the minimal enclosing rectangle or the steiner tree length is used. Min-cut partitioning attempts to minimize the number of edges which have vertices in two different subgraphs.

This new objective function is motivated by the fact that it is easier to route a signal through sparse channels and that the density varies along some areas of the signal net. This objective function is similar to functions which try to minimize wiring density since the number of signals cut by line c is a lower bound on the number of routing tracks which must cross c when the circuit is routed.

Since true min-cut partitioning is NP-complete [GAR 79] a heuristic algorithm must be used. The most popular heuristics are group migration and direct methods.

3.2.4.1.1. GROUP MIGRATION

Group migration was developed by Kernighan and Lin [KER 70]. The algorithm is developed for bisecting a graph, but can be modified to produce q subgraphs. The algorithm is usually used recursively in VLSI placement programs to bisect each subgraph repeatedly until they are *small enough*.

The algorithm works by successively improving a partition by reassigning nodes. In order to bound the processing time, a node is only reassigned once so that the number of nodes to consider is reduced. The algorithm for group migration is:

1) Partition the graph into two subgraphs A and B. This can be done randomly.

2) For every node in A calculate the net change in edges cut if this node were moved to B. Do the same for every node in B. Set i equal to 0.

3) Choose the pair of nodes (a_i, b_i) that gives the best improvement in the number of edges cut. Save the net improvement as g_i.

4) Remove a_i from A and remove b_i from B. Recalculate the net change in the number of edges cut for each node that is connected to a_i or b_i.

5) If $|A| > 0$ and $|B| > 0$, then increment i and go to 3.

6) Find a k satisfying

$$G = \min_k \sum_{j=0}^{k} g_j$$

and move $a_0, a_1, \ldots a_k$ to B and move $b_0, b_1, \ldots b_k$ to A. If G is less than zero and at least one node is moved, go to 2.

Since most VLSI placement problems are very sparse, a slight modification can be made to Steps 3 and 4. Instead of considering all pairs of (a,b), only the elements with the best improvement in the number of edges cut are considered. This implies that the nodes in A, B are sorted, so in Step 4, all the nodes that are updated must be resorted. This can be done in time n log N, where N is the number of nodes in the subgraph and n is the number of modified nodes, by using height-balanced trees [KNU 73]. This will only improve the running time of the algorithm if N >> n.

3.2.4.1.2. DIRECT PARTITIONING

Direct partitioning algorithms have 3 main steps: initial grouping, seeding and allocation. The initial grouping can simply be taken as one node per group, but better partitioning can be achieved by using strongly connected components as groups. To determine the strongly connected components, a directed graph is used to represent the circuit. The direction of the edge is taken to be from driver node to driven node. The strongly connected components are then found by marking all the nodes in a depth first search manner.

In the seeding phase the groups with the most external edges (edges with vertices in two different groups) are assigned to each of the subgraphs. Next, the rest of the groups are allocated. A merit function, such as:

$$\min dP_{ij} * dA_{ij}$$

where

dA_{ij} = change in Area of subgraph i if group j is included

dP_{ij} = change in number of external edges in subgraph i if group j is included

is used to determine which group will be allocated. Each subgraph also has a maximum capacity based on the number of external edges and on the area of the graph. This is used to insure that the subgraphs will be of nearly equal size. The allocation phase terminates when all the groups have been assigned or when no more groups can be assigned. If there are still groups to be assigned when the allocation phase finishes, then the seeding phase is repeated.

3.2.4.2. QUADRATIC ASSIGNMENT

The quadratic assignment problem [GAR 79] can be stated as: given nonnegative integer costs c_{ij}, $0 < i,j \leqslant n$, and distances d_{kl}, $0 < k,l \leqslant m$, find a one-to-one function $f:\{1,2,\ldots,n\} \rightarrow \{1,2,\ldots,m\}$ such that

$$\sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} c_{ij} d_{f(i)f(j)}$$

is minimum. For our placement problem, n = m = number of modules to be placed and we let c_{ij} be the weighted sum of the signals common to modules i and j. In VLSI there are a large number of sets of modules which are unconnected. The cost of placing any module in this set is independent of where the other modules in the set are placed. So the cost function is reduced to

$$\sum_{i \in A} c'_{i,f(i)}$$

where

$$c'_{i,j} = \sum_{k=1}^{n} c_{i,k} d_{j,k}$$

A = *set of unconnected modules.*

But, this is simply the linear assignment problem, which can be solved fast and efficiently [MUN 57]. So the Steinberg algorithm for finding a good placement is to start with an initial placement and to improve the placement of each unconnected set by applying the linear assignment algorithm.

3.2.4.3. ITERATIVE IMPROVEMENT

This placement procedure works by systematically generating a set of modules to be interchanged. If the proposed combination improves the placement then it is accepted. There are several variations on which modules are chosen to be exchanged. In pairwise interchange all possible pairs of modules are selected, whereas in neighborhood interchange only those modules which are *close* to the primary module are chosen. In λ-interchange, λ modules are chosen to be interchanged. For λ-interchange only a predetermined number of trials is attempted because the total number of possible combinations increases rapidly.

Usually the algorithm is run on ten initial random placements and the best is chosen. Although this method takes ten times longer to generate a placement, it is not a problem as long as the number of modules to be placed is under 250. It has been suggested that many initial random placements be generated, but to only use the best initial starting position. In general, this method generates inferior placements because the optimality of the final placements generated by iterative improvement schemes are not correlated to the optimality of the initial placement.

3.2.4.4. SIMULATED ANNEALING

Simulated annealing [WHI 84] is similar to iterative improvement, but instead of only accepting states which improve the placement, those that are inferior are accepted with probability exp (-abs(dc) / T), where dc is the difference in cost between two states and T is an arbitrary parameter with the property that as T approaches 0 the placement approaches the optimal solution. Simulated annealing generates each new state randomly. The new states fall into four categories: pairwise interchange, rotations, mirroring, and translations. An important property of simulated annealing is that the new state generated can be illegal (ie., allowing the modules to overlap). Violations are accounted for in the cost function by a penalty term. The penalty function has the property that as T approaches 0 the penalty function also approaches 0. Simulated annealing programs usually consist of two loops. The outer loop decreases T and checks for convergence, while the inner loop evaluates the new states (Figure 3.6).

```
state = InitialState();
WHILE (penalty != 0 && !converged) {
      T = k * T;        /* k < 1.0 */
      WHILE (!converged) {
            newstate = MakeNewState();
            dc = cost(state) - cost(newstate);
            if (dc > 0 || random() < exp (dc/T))
                  state = newstate;
      }
}
```

Figure 3.6
Structure of simulated annealing programs

3.2.4.5. HIERARCHAL ITERATIVE IMPROVEMENT

The placement method chosen for the silicon compiler is a new hierarchal iterative improvement which has been implemented in a program called PLACE. This technique is similar to iterative improvement, but it allows the modules to vary greatly in size. The modules are sorted into a set of standard sized blocks. The block sizes are chosen so that the next larger block is twice as large. This way four small blocks can be grouped together to form a large block.

The algorithm works by generating an initial grouping, which is then iteratively improved for each level. An initial placement is then chosen for the top level blocks. This placement is then iteratively improved.

3.2.4.5.1. PLACE DETAILS

The input to PLACE is a directory which contains all the modules in Cal-Tech Intermediate Form (CIF) [MEA 80] and an interconnection file. The interconnection file has the format

filename.label = filename.label

The output is a two level CIF file. Each symbol in the output file represents a module.

First all the modules are parsed and the size of the bounding box is determined. The average height and width is used as the standard size block, unless the standard deviation is within five percent of the average. In that case, the standard size block is taken to be the average plus the standard deviation. Let h_s and w_s be the height and width of the standard size block respectively. All the modules are now assigned to blocks of size

$$h_s * 2^i \ by \ w_s * 2^j$$

where

$$i - 1 \leqslant j \leqslant i + 1.$$

The blocks now enter the grouping phase. Here four small blocks are joined together to form a large block, which is then recursively joined to three other large blocks to form a larger block. This continues until the largest size block is formed. The largest size block is the smallest block that will completely enclose the largest module.

After the blocks are grouped together, the initial grouping is iteratively improved one level at a time, starting with the smallest block. The cost is based on the number of external connections to different modules minus the number of connections to group members. This cost function was chosen so that modules in the same signal net will be grouped together.

A random initial placement is now chosen for the top level blocks. This placement is then iteratively improved. The cost function for improving the placement is now based on the total length of all signal nets. The length of the net is approximated by point-to-point distances.

3.2.5. ROUTING

After the logic blocks are designed and placed, they need to be connected together. There are several constraints on the interconnections between blocks. Obviously, wires of different nets must occupy different sections in the wiring track. Tracks may also be blocked by logic blocks (no over the cell routing). Often, the technology limits the positions of vias adjacent to one another which limits the number and density of dog-legs (bends in the net).

The wiring problem is also an NP-complete problem [GAR 79]. So, many heuristic techniques have been tried to solve this problem. The figure of merit used to evaluate the effectiveness of the design is usually based on track overflow, total wire length, the number of vias, and the maximum net length. To simplify the problem further, the routing is done using only horizontal and vertical wiring tracks with a limited number of dog-legs.

The wiring problem cannot be solved by just using a series of shortest pathfinding steps. Instead, most routers use a fast pathfinding algorithm as a subroutine. Routers try to route each wire while considering congestions on the chip. They choose the lowest *cost* path and not the shortest path. Here cost is based on an estimate of the congestion, density of tracks, number of free tracks and the length of the path. It is very difficult to compute a good cost function for track to track resolution, so the problem is done in two stages: global wiring and detailed wiring.

In global routing the chip is divided into an array of cells or regions. Each region contains a manageable number of horizontal and vertical wiring tracks. In the first stage, nets are allocated to the channels (set of wiring tracks along the boundary of a region). The global wiring stage ensures that all nets are routed without exceeding the channel capacity of the regions.

One method of solving the global wiring problem is by building a steiner tree which is the minimum weighted tree that connects all the terminals in a net. This tree can be built fast and efficiently by using a greedy algorithm to select the edges.

Simulated annealing has been used by Vecchi et al. [VEC 83] to produce uniform global routes. The global routing problem is modeled by lumping all terminals into a regular grid of N_x by N_y points. The edges represent the channels. The nets are then routed along the edges. In this representation, several nets may be allocated to an edge. This does not violate the constraint that nets must be electrically isolated since each edge represents a channel with several wiring tracks. Blockages in the wiring channel are accounted for by prefilling the edges. The nets are grouped into sets of twos and then randomly routed by using only straight and L-shaped paths (paths with only one bend). A new routing is obtained by randomly selecting a section to move. The new state is accepted if it has lower cost. States with higher cost are accepted with an exponential probability, so that the system may move out of local minimum states. The global wiring is first annealed using only L-shaped paths and then reannealed allowing Z-shaped paths (paths with two bends). This method of annealing empirically produces better results faster than starting the annealing with Z-shaped paths. The cost function used is the sum of the squares of the number of wires on each edge. This cost function penalizes nonuniform wiring densities.

The second stage, detailed wiring, assigns each net a specific track within the channel. Here the routing is usually done by using the Lee-Moore algorithm [LEE 61] for finding the shortest path. The Lee-Moore algorithm uses a breadth first search technique to find the shortest path. The shortest path between nodes A and B is found by starting at Node A and marking all neighboring nodes with a direction to Node A. Then all unmarked nodes adjacent to the marked nodes are marked with a direction back to Node A. This process is repeated until the outwardly propagating wave reaches Node B.

Simulated annealing can also be used to perform detailed wiring. Here the nets are only shifted between tracks in the same or adjacent channels, since the global wiring stage has already found a near optimal placement for the nets.

CHAPTER 4

SILICON COMPILER DETAILS

4.1. SILICON COMPILER LANGUAGE DETAILS

The compiler developed in this thesis is based on the C programming language enhanced with some new features. An additional data type signal has been added. The standard boolean operators (and, nand, or, nor, exclusive-or, exclusive-nor, and not) can be used with this new data type. The data type signal behaves differently from the normal type of variables. When an assignment is made to a signal, the assignment does not take on the value of the expression, but rather it is wired or'ed with the signal. Another difference with the type signal is that the value of a signal is the value that was passed at the beginning of the subroutine. If the most recent value is needed, then the operator NEW is applied to the signal. For example

```
x = a & b;          |  z = x | y;
y = c & d;          |  y = c & d;
z = x | y;          |  x = a & b;
```

will generate the same circuit. While the code

```
x = a & b;          |  x = a & b;
y = c & d;          |  y = c & d;
z = new x | new y;  |  z = a & b | c & d;
```

are equivalent. Circuits with memory can be described by simply equating a signal with itself. All memory elements are implicitly synchronized with the system clock.

4.2. SILICON COMPILER IMPLEMENTATION DETAILS

The silicon compiler is actually written as a collection of several programs which are transparent to the user by using fork and exec on the UNIX operating system. The user only interacts with the front-end program. The following functions are performed by separate programs:

compiling and code generation

object module linkage

executing run-time module

logic - minimization†

PLA folding and generation†

placement of PLAs

routing of PLAs†

The subprograms used in the compiler which are not developed in this thesis are from the University of California at Berkeley (UCB) CAD tools. The UCB CAD tools were chosen because the tools and documentation were easily obtainable from UCB.

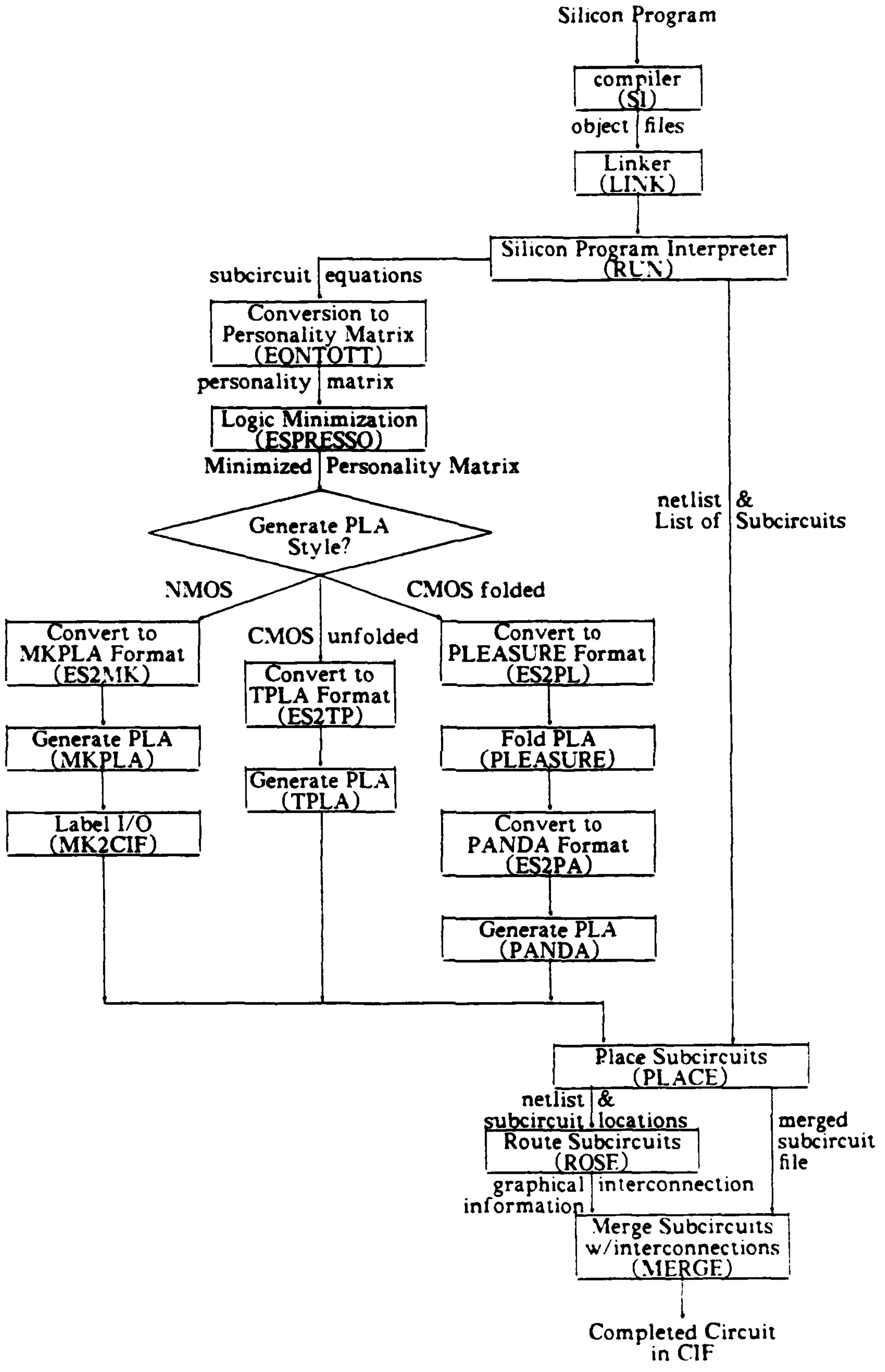
The compiler is written in C. The compilation is done in four phases. In Phase One the source program is preprocessed, lexical analysis is done and the program is parsed. In Phase Two the symbol table is built, code generation and object module linking is done. In Phase Three the code is executed and PLAs are generated. In Phase Four each of the PLAs is placed on the chip and wired together. The calling sequence for the front-end processor is shown in Figure 4.1.

The execution of silicon compiler code is different from the execution of a normal program. A log of all subcircuits executed along with the values of all non-signal parameters and external variables is kept. So when a subcircuit is called its parameters are checked against the subcircuits already generated. If the subcircuit already exists then only the interconnection information needs to be produced, which will decrease the time necessary to generate a chip and will also reduce the memory requirements of the chip.

4.2.1. PHASE I

The compiler simultaneously performs preprocessing, lexical analysis and parsing. These three processing steps are similar to standard programming language compilers. The

† Subprogram from UCB CAD tools.



Silicon Program
compiler (SI)
object files
Linker (LINK)
Silicon Program Interpreter (RUN)
subcircuit equations
Conversion to Personality Matrix (EQNTOTT)
personality matrix
Logic Minimization (ESPRESSO)
Minimized Personality Matrix
Generate PLA Style?
NMOS
CMOS unfolded
CMOS folded
Convert to MKPLA Format (ES2MK)
Generate PLA (MKPLA)
Label I/O (MK2CIF)
Convert to TPLA Format (ES2TP)
Generate PLA (TPLA)
Convert to PLEASURE Format (ES2PL)
Fold PLA (PLEASURE)
Convert to PANDA Format (ES2PA)
Generate PLA (PANDA)
netlist & List of Subcircuits
Place Subcircuits (PLACE)
netlist & subcircuit locations
Route Subcircuits (ROSE)
merged subcircuit file
graphical interconnection information
Merge Subcircuits w/interconnections (MERGE)
Completed Circuit in CIF

Figure 4.1
Control flow for front-end processor SS

parser is written using YACC [JOH 75] which generates an LR(1) parser [AHO 74]. As the parser builds the parse tree it calls YYLEX to get the next token. YYLEX is written using LEX [LES 75] a lexical analyzer generator. The lexical analyzer calls the preprocessor for the next inp racter. The preprocessor allows the user to define constants, define macros, include files, to perform conditional compilation, and have nested comments.

The silicon compiler builds a parse tree for each subroutine. It then builds a symbol table and allocates space for variables, performs type checking and does code generation. The compiler then repeats these steps for every subroutine in the input file. The code generated for integer statements is similar to code generated in standard programming languages and will not be discussed further. Each signal valued statement is represented as a boolean cover. A boolean cover is simply a set of boolean cubes. Each cube represents one product term in a boolean expression written as a sum-of-products. The cubes consist of a fixed length part and a variable length part. The fixed length part corresponds to scalar signal variables and the variable length part corresponds to array elements or to integer values which cannot be determined at compile time. Each variable is represented as a 2-bit code which is positionally dependent within the cube. The codes used are

11	-	does not depend on this variable
10	-	depends on true form
01	-	depends on complemented form
00	-	null cube

These encodings were chosen so that if two cubes need to be ANDed together then a bitwise AND is done on the two cubes. To AND two covers together, each cube in one cover is ANDed with all the cubes in the second cover. Also the true and complemented representations of a variable are complements of each other. In this representation a 00 in any position represents a null cube. This representation also allows two covers to be ORed by simply concatenating the two list of cubes (Figure 4.2). The compiler only performs distance-one merging of each cover and subsumption since global minimization during compilation is

expression	boolean cube
	a b c
b	11 10 11
ac	10 11 10
abc	10 10 10
$a\bar{b}c$	10 01 10
$b\bar{b}$ (null)	11 00 11

Figure 4.2
Boolean cube example

a waste of time because not all covers will be included in the final PLA. and different parts in the variable length part of the cube may represent the same variable. Before the cover is merged. the nonessential vertices are removed. This greatly reduces the size of each cube, which in turn speeds up the merging process. In distance-one merging the cubes are sorted and any adjacent cubes which differ in one or less parts are ORed together. This is similar to merging two implicants in a Karnaugh map (Figure 4.3).

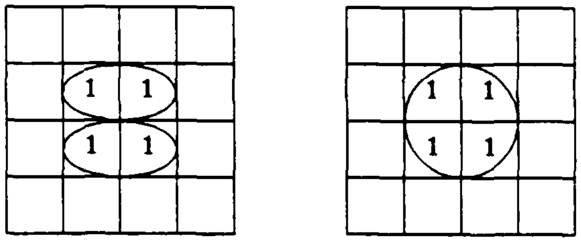


Figure 4.3
Karnaugh map representation of distance-one merging

Distance-one merging is a fast operation requiring O(kn log n) time where n is the number of cubes and k is the number of parts. While the cubes are being sorted, subsumption of cubes is checked. Cube A covers Cube B, if (bitwise) **NOT** A **AND** B is zero. Theoretically, performing subsumption after distance-one merging should produce better results. but in practice performing subsumption before merging produces equally good results.

Complementation is performed by recursively applying the Shannon expansion.

$$f = xf_x + \bar{x}f_{\bar{x}}$$
$$\bar{f} = x\bar{f}_x + \bar{x}\bar{f}_{\bar{x}}$$

where f_x and $\bar{f}_{\bar{x}}$ are cofactors.

If f is unate and monotone increasing then

$$\bar{f} = \bar{x}\bar{f}_{\bar{x}} + \bar{f}_x$$

or if f is unate and monotone decreasing then

$$\bar{f} = x\bar{f}_x + \bar{f}_{\bar{x}}.$$

Since complementation of a unate function is well understood and can be performed quickly, we try to make f_x and $f_{\bar{x}}$ unate as quickly as possible. The heuristic suggested by Brayton [BRA 82] is to choose the most binate variable to do the splitting. Choosing the most binate variable attempts to keep cubes which are both part of f_x and $f_{\bar{x}}$ as small as possible. Also since the cofactors may not be represented by prime implicants, a cofactor may actually be unate even though the covers are not unate. This method of complementing a function is about twice as fast as complementation by computing the disjoint sharp [HON 74]. This method also produces fewer product terms.

4.2.2. PHASE II

After all the subroutines are compiled, they are linked together. The linker merges all the object files and resolves all subroutine entry points and global variables. The resultant file can then be executed. The execution of a silicon program is different from conventional programs. In the silicon program a hashed-list of all subroutines and the value of all integer parameters and integer global variables is kept. If a subroutine is called again with identical integer parameters then the PLA specification is not regenerated, and only the new interconnection information is generated. This speeds up the execution time and saves memory, since 90 percent of a VLSI circuit are regular structures.

The PLA specification file is written as a set of two-level boolean equations using AND, OR and negation of single variables. This format was chosen instead of a personality matrix representation because this form is easier to read and modify.

4.2.3. PHASE III

Logic-minimization is performed by a separate program so that new programs to do logic-minimization and partitioning can be easily incorporated. This way logic minimizers with different critera, such as minimizing the number of implicants, area or speed, can be added. For instance, if one decides to use true static CMOS PLAs then the logic-minimizer must partition each equation so that no implicant has more than four terms, otherwise the speed of the PLA will be severely degraded.

The logic equations generated from executing the silicon program is converted to a personality matrix using EQNTOTT. This personality matrix is then minimized using ESPRESSO. The resultant file is then used to generate the PLAs. Up to this point, all circuit specifications have been technology independent. At this point the circuit is tied to a particular technology. Currently, the circuit can be designed in two technologies: NMOS and CMOS. The PLAs are generated from three different PLA generators: MKPLA, TPLA, and PANDA. MKPLA is used to generate unfolded NMOS PLAs. TPLA is used to generate static CMOS unfolded PLAs using p-channel devices as resistive pull-ups. PANDA is used to generate multiply folded static CMOS PLAs. Both TPLA and PANDA are technology independent PLA generators which are template driven. So to modify the compiler to generate circuits in new technologies or to improvements in technology, one just has to design a new template. Or, an entirely new program can be used in place of the three PLA generators as long as the new program generates the subcircuit in CIF format. Also, since the subcircuits are in CIF format, they can be edited using any layout editor [†] if the performance of the subcircuit does not meet specifications.

Each of the three PLA generators accepts the PLA specification in a different format. The front-end program automatically invokes the proper conversion routines before generating the PLA. For instance, the PANDA program generates a PLA from a folded personality matrix. The front-end program first takes the output from ESPRESSO and converts it to a format compatible with PLEASURE which will do the folding. Another conversion routine is then automatically called to convert the PLEASURE output into a form usable by PANDA. Finally, PANDA is invoked and a PLA is generated.

4.2.4. PHASE IV

The PLAs are then sent to PLACE, which merges all the individual CIF files and places the PLAs on the chip. PLACE uses hierarchal iterative improvement to determine a near optimal solution. PLACE generates three output files: rose.in, ckt.stat, and ckt.plc. The file rose.in contains the netlist and location of the subcircuits and the terminals. The file ckt.plc contains the definitions for all the subcircuits and the file ckt.stat contains information necessary to merge the graphical interconnection file with ckt.plc. The graphical interconnection file is generated from rose.in by using the program ROSE. ROSE routes all the interconnections using a Steiner-tree-on-graph algorithm. The interconnects are routed using two layers: polysilicon and metal. Another program MERGE then merges the graphical interconnection file with ckt.plc to generate a completely wired circuit. The circuit can now be sent to a silicon foundry for fabrication.

4.3. DESIGN EXERCISE

For this study, I have designed a 16-bit fixed point processing element using the silicon compiler. The processing element is to be used for performing LU factorization of parti-

[†] Most layout editors do not work with CIF files directly, but provide conversion routines to translate the CIF format into an internal format and from the internal format to CIF.

tioned systems [LUI 84]. The processing element performs four functions:

$$(1)\ A_{ij}^{m+1} = A_{ij}^{m} - L_{ik}^{m} * U_{kj}^{m}$$

$$(2)\ L_{ik}^{m} = A_{ik}^{m} \,/\, U_{kk}^{m}$$

$$(3)\ U_{kj}^{m} = A_{ij}^{m}$$

$$(4)\ A_{ij}^{m+1} = A_{ij}^{m} + L_{ik}^{m} + U_{kj}^{m}$$

Function (1) updates the matrix elements. Functions (2) and (3) compute the lower and upper triangular factors respectively. Function (4) sums the elements together at the interconnection level.

From the above functional specification, we need to design a 16-bit multiplier, divisor, and adder. A top-down approach is chosen to design the processing element. The arguments to the main circuit specify the I/O pin assignments to the integrated circuit. Then the logic block diagram (Figure 4.4) is converted into silicon compiler statements. Each of the functional blocks in the diagram is implemented as subcircuits.

Now each of the subcircuits needs to designed. Both the multiplier and the reciprocator are implemented as cellular arrays. Both subcircuits make use of two-dimensional signal arrays to specify all the interconnections between cells. If we look at the PLA generated from the multiplier specifications (Figures 4.5, 4.6), we see that the outputs correspond to signals that are the result of some boolean operation which is input to another subcircuit. So the use of temporary signal variables does not affect the size of the PLA that will be generated.

The silicon compiler also checks to see if the non-signal arguments to a subcircuit have changed. If they are the same as in a previous call the code is not reexecuted, but instead only new interconnection information is generated. For instance, reciprocator calls subcircuit CAS 256 times, but only one equation file is generated. This not only speeds up execution time of the silicon program, but it also saves time in minimizing the equations, time

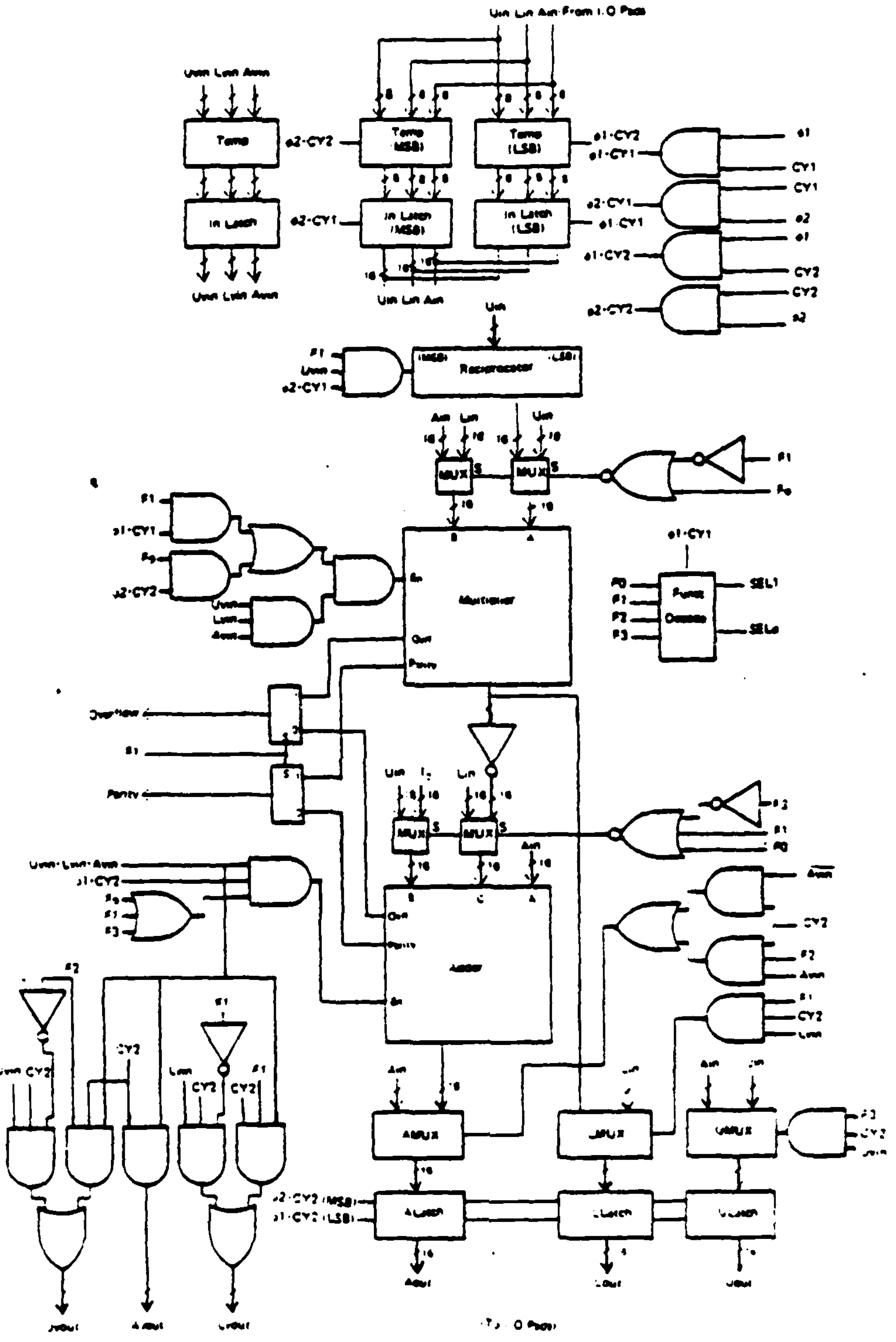


Figure 4.4
Logic block diagram for CMOS processing element [LUI 84]

```
# define WORDSZ 16


mult (a,b,c,ovfl)
input signal a[WORDSZ], b[WORDSZ];
output signal c[WORDSZ], ovfl;
{
/*              form the multiplication of a and b by using an array
 *            of full adders
 *                        c = a * b
 *                        ovfl = 1 if overflow
 */
int i, j;
signal p[WORDSZ+1], s[WORDSZ][WORDSZ], co[WORDSZ][WORDSZ+1], gnd;

c[0] = a[0] & b[0];
for (i=WORDSZ-2; i--; )
            add0 (a[i+1] & b[0], a[i] & b[1], gnd, s[i][0], co[i][0]);
c[1] = new s[0][0];
for (j = 1; j < WORDSZ - 2; j++ ) {
            for (i = WORDSZ - 3; i--; )
                        add0 (new s[i+1][j-1], new co[i][j-1], a[i] & b[j+1],
                                    s[i][j], co[i][j]);
            add0 (a[WORDSZ-2] & b[j], new co[WORDSZ-2][j-1],
                        a[WORDSZ-3] & b[j+1], s[WORDSZ-2][j], co[WORDSZ-2][j]);
            c[j+1] = new s[0][j];
}
for (i=0; i < WORDSZ-3; i++)
            add0 (new s[i+1][WORDSZ-3], new co[i][WORDSZ-3], gnd,
                        s[i][WORDSZ-2], co[i][WORDSZ-2]);
add0 (a[WORDSZ-2] & b[WORDSZ-2], new co[WORDSZ-2][WORDSZ-3], gnd,
            s[WORDSZ-2][WORDSZ-2], co[WORDSZ-2][WORDSZ-2]);
for (i=0; i < WORDSZ-2; i++)
            add2 (new s[i][WORDSZ-2], a[WORDSZ-1] & b[i], a[i] & b[WORDSZ-1],
                        s[i][WORDSZ-1], co[i][WORDSZ-1]);
c[WORDSZ-1] = new s[0][WORDSZ-1];
add2 (new co[WORDSZ-3][i], a[WORDSZ-1] & b[i], a[i] & b[WORDSZ-1],
            s[i][WORDSZ-1], co[i][WORDSZ-1]);
add2 (new s[1][WORDSZ-1], new co[0][WORDSZ-1], gnd, p[0],
            co[0][WORDSZ]);
for (i=0; ++i < WORDSZ-2; )
            add2 (new s[i+1][WORDSZ-1], new co[i][WORDSZ-1],
                        new co[i-1][WORDSZ], p[i], co[i][WORDSZ]);
add2 (a[WORDSZ-1] & b[WORDSZ-1], new co[WORDSZ-2][WORDSZ-1],
            new co[WORDSZ-3][WORDSZ-1], p[WORDSZ -1], p[WORDSZ]);
chkovf (p,ovfl);
}                       /* end of mult( ) */
```

Figure 4.5
Silicon program code for multiplier

spent attempting to fold the personality matrix, and time spent generating the PLA.

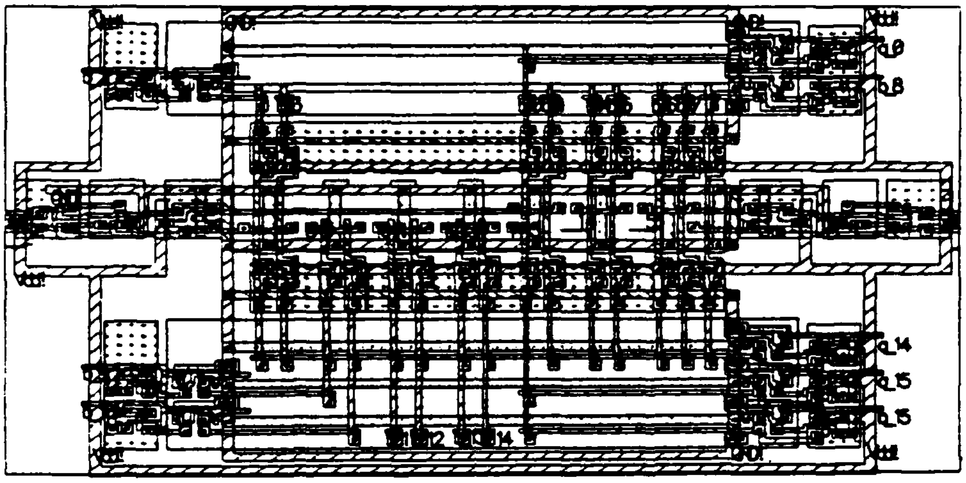


Figure 4.6
Multiply folded PLA mult_5

After running the silicon compiler program, 14 equation files corresponding to 14 unique calls to subcircuits are generated. The equation files are then converted into a personality matrix and sent to ESPRESSO to do logic minimization. The minimization done at run time considers each logic function separately and is used mainly to keep peak memory requirements down. The algorithm used at run time is of order O(nk log n) where n is the number of product terms and k is the number of variables in the function. On average, the run time algorithm is able to reduce peak memory requirements by 75 percent. The minimization done by ESPRESSO considers all output functions together and is able to reduce the number of product terms in the personality matrix by another 20 percent.

At this point, the user must decide on the technology to implement his chip. Currently, the chip can be implemented in NMOS or CMOS p-well. The CMOS PLAs are generated by using two template driven PLA generators: TPLA and PANDA. TPLA is used

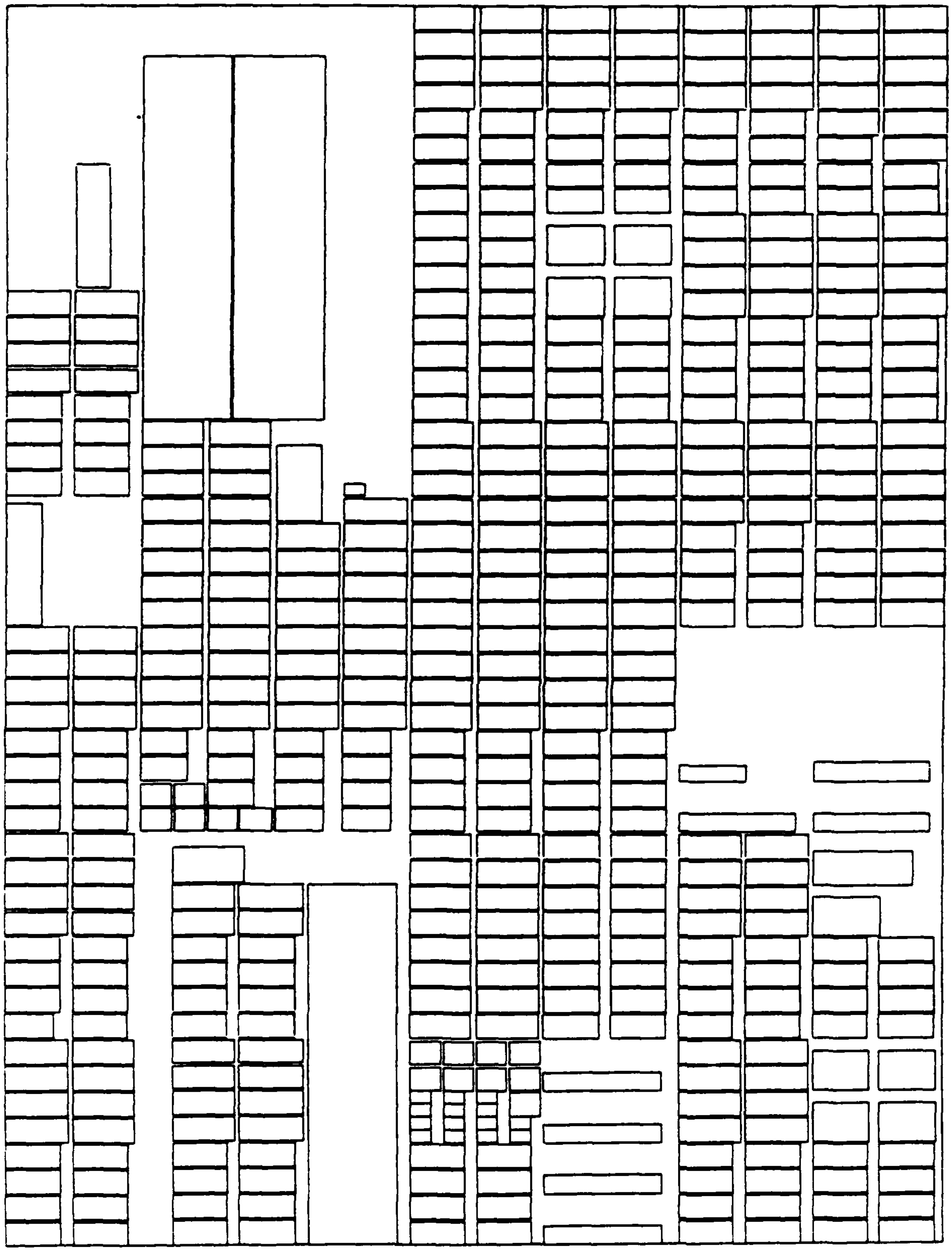


Figure 4.7
Chip designed using silicon compiler

for nonfolded PLAs and PANDA for multiply-folded PLAs. The CMOS PLAs are static PLAs using CMOS pullups. Currently, only the CMOS templates have been designed so the NMOS PLAs are generated by using MKPLA, which also generates static PLAs. In the future, templates for both static and dynamic NMOS and CMOS dual-well PLAs will be designed. If the user chooses to generate his PLAs using PANDA, the personality matrix is first folded by using PLEASURE, a heuristic folding program. The user may wish to generate both folded and nonfolded PLAs and choose the smaller of the two. In this design example, CMOS multiply-folded PLAs are chosen. The PLA generated for the multiplier code in Figure 4.5 (p. 49) is shown in Figure 4.6 (p. 50).

The CIF files are then read by PLACE which concatenates the files and finds a good placement for the subcircuits using a hierarchal iterative improvement technique. The concatenated CIF file is then sent to ROSE which interconnects the subcircuits using a steiner-tree algorithm. The output from PLACE and ROSE is sent to MERGE, which merges the subcircuits with the interconnections. The completed circuit can be seen in Figure 4.7 (p. 51).

```
#    compile and load silicon program
% ss add.si mult.si recip.si main.si mux.si register.si
#    run program
% ss -run
#    minimize equations
% ss -min
#    generate CMOS folded PLAs
#          -fold == specify folded PLAs
% ss -cmos -fold
#    place subcircuits
% ss -place
#    interconnect subcircuits
% ss -route
#    circuit is now in ckt.all.cif
```

Figure 4.8
Command sequence to generate circuit from behavioral specification (Comments in italics)

The resultant chip can then be extracted and simulated. If the circuit does not meet specifications, then the silicon program can be modified to improve performance. Such a modification is the subject of further research.

The circuit used in this study was generated using the command sequence shown in Figure 4.8 (p. 52). The front-end processor, SS, automatically chooses the correct filters at each stage of the circuit design. For example, all three PLA generators require a different input format. Also, the entire circuit could have been generated with a single call to SS instead of 6, by concatenating the arguments. The front-end processor will automatically invoke the other programs in the correct order.

CHAPTER 5

CONCLUSIONS

This thesis investigates the feasibility of designing circuits using a high-level language to specify only the behavioral aspects of the circuits. A silicon compiler is used to take this description and transform it into the mask-layers for fabrication. This approach to designing circuits is expected to be a powerful tool which will greatly reduce the design time. The silicon compiler could successfully compete with the semi-custom approach to designing circuits.

The compiler synthesizes the subcircuits as PLAs. PLAs are chosen because they are regular structures which are easy to customize and readily adaptable to changes in technology. The new features can be incorporated in a matter of hours by just redesigning the templates used by the PLA generator. However, the PLA generator used in this thesis does not allow the user to use different transistor sizes. With this restriction, the circuits produced are only as fast as gate-arrays. In future versions of the compiler, the transistor sizes will be based on fan-out and parasitic line capacitance. In addition to the speed penalty, the area penalty is much too high. Even with multiply-folded PLAs, the area penalty is still large. The problem with multiply-folded PLAs is that the area saved by folding may be completely negated by the additional buffers placed on the other sides of the PLA. This usually happens in small PLAs since folding only reduces the size of the core. To solve this problem, the programming style of the designer could be changed to promote large PLAs, or the compiler could restructure the code to produce large PLAs, or a cell library like those used in the semi-custom design approach could be included.

The first alternative is the easiest to implement, but is the least desirable method. Forcing the designer to think in unaccustomed ways may lower productivity. The best approach would be to both restructure the code and to use a cell library. The cell library approach is effective because many small subcircuits are in a cell library. Also most bit-slice circuits waste a lot of area when implemented as PLAs. A disadvantage to the standard cell library approach is that the library is technology dependent, so changes in technology will be harder to implement.

Currently the only method of verifying the correct operation of the chip is to extract the circuit after the subcircuits have been generated or after the entire chip is generated. To aid the designer, the compiler should generate code so that logic simulation can be performed without having to generate the subcircuits. This way the designer would not have to waste hours of CPU time generating a circuit that was logically incorrect. Then when the designer is confident that the circuit is logically correct, he could generate the circuit and verify the performance with a timing simulator.

The majority of the time spent in synthesizing the circuit is in placement and routing. One way to speed up the placement and the routing would be to reduce the number of subcircuits and the number of terminals that need to be connected. One method of doing this is to generate the subcircuits inside loops such that they will abut. Then the entire loop should be treated as a single large subcircuit. Since most of the circuits designed are bit-slice architectures with word sizes of 16 or 32, the potential savings in time could be great.

APPENDIX

The following program (Figures A.1 - A.14) is used to generate the special purpose processor for LU factorization, which is described in the design example in Chapter 4. All PLAs and mask descriptions used in this thesis are derived from the following program.

```
# define WORDSZ 16

main (Avin,Ain_out,sel1,sel0,Uout,Uvout,Lvin,Lin,cy1,cy2,Uin,Uvin,Lout,
        Lvout,parity,overflow,ck1)
input signal Avin, sel1, sel0, Lin[WORDSZ / 2], Lvin, cy1, cy2,
        Uin[WORDSZ / 2], Uvin, ck1;
output signal Uout[WORDSZ / 2], Uvout, Lout[WORDSZ / 2], Lvout, parity,
        overflow, Avout;
signal Ain_out[WORDSZ / 2];
{
signal Amuxcntl, Lmuxcntl, Umuxcntl, Mparity, Aparity, Movfl, Aovfl,
        abmux, ulmux, NotS[WORDSZ], BM[WORDSZ], AM[WORDSZ], BA[WORDSZ],
        AA[WORDSZ], Ain[WORDSZ], Aout[WORDSZ], f[4], Addout[WORDSZ],
        Mulout[WORDSZ], Lin16[WORDSZ], Uin16[WORDSZ], Recout[WORDSZ];
int i;

/*          define function switches                */
f[0] = ~sel1 & ~sel0;
f[1] = ~sel1 & sel0;
f[2] = sel1 & ~sel0;
f[3] = sel1 & sel0;
/*          latch inputs          */
register (Ain_out,Ain,cy1,cy2);
register (Lin,Lin16,cy1,cy2);
register (Uin,Uin16,cy1,cy2);
overflow = Movfl & new f[1] | Aovfl & ~ new f[1];
parity = Mparity & new f[1] | Aparity & ~ new f[1];
/*          form 1/Ain or 1/Uin16          */
Umuxcntl = new f[3] & cy2 & Uvin;
mux (Ain,Uin16,new Umuxcntl,Uout);
recip (Uin16,Recout);
/*          form Ain/Uin16 or Lin16 * Uin16                */
abmux = ~(~new f[1] | new f[0]);
mux (Ain,Lin16,new abmux,BM);
mux (Recout,Uin16, new abmux,AM);
mult (AM,BM,Mulout,Movfl);
Lmuxcntl = new f[1] & cy2 & Lvin;
mux (Lin16,Mulout,new Lmuxcntl,Lout);
/*          form Ain + Lin16 + Uin16 or Ain - Lin16 * Uin16                    */
ulmux = ~(~new f[3] | new f[1] | new f[0]);
for (i = WORDSZ; --i; ) BA[i] = Uin16[i] & new ulmux;
BA[0] = Uin16[i] &new ulmux | ~new ulmux;
muxinv (Lin16, AM,new ulmux,AA);
add3w (BA,AA,Ain,Addout,Aovfl);
Amuxcntl = ~Avin & cy2 | cy2 & new f[2] & Avin;
mux (Ain,Addout,new Amuxcntl,Aout);
Uvout = Uvin & cy2 & ~(new f[2]) | new f[2] & cy2 & Uvin & Lvin & Avin;
Avout = cy2 & Uvin & Lvin & Avin;
Lvout = Lvin & cy2 & ~(new f[1]) | new f[1] & Uvin & Lvin & Avin;
}                     /* end of main() */
```

Figure A.1

```
recip (divisr,quot)
input signal divisr[WORDSZ];
output signal quot[WORDSZ];
{
signal Gnd, VDD, C[WORDSZ][WORDSZ], s[WORDSZ][WORDSZ];
int i, j;
/*          form the reciprocal of divisr and place result in quot
 *          the reciprocal is formed by using an array of controlled
 *          add/subtract cells (cas)
 */

cas (VDD,divisr[WORDSZ-1],VDD,VDD,s[WORDSZ-1][0],C[WORDSZ-1][0]);
for (i = WORDSZ-1; i--; ) {
            cas (VDD,divisr[i],Gnd,new C[i+1][0],s[i][0], C[i][0]);
}
quot[0] = new C[0][0];
for (j = 1; j < WORDSZ - 1; j++) {
            cas (new C[0][j-1],divisr[WORDSZ-1],new s[WORDSZ-1][j-1],
            new C[0][j-1], s[WORDSZ-1][0], C[WORDSZ-1][0]);
            for (i = WORDSZ-1; i--; ) {
            cas (new C[0][j-1],divisr[i],new s[i+1][j-1],new C[i+1][j],
                      s[i][j], C[i][j]);
            }
            quot[j] = new C[0][j];
}
}                       /* end of recip() */
```

Figure A.2

```
CLAgen (x,y,p,g)
input signal x[4], y[4];
output signal p[4], g[4];
{
/*          generate p & g for carry look-ahead                  */
int i;

for (i=0; i < 4; i++) {
            p[i] = x[i] ^ y[i];
            g[i] = x[i] & y[i];
}
}                       /* end of CLAgen() */
```

Figure A.3

```
register (DataIn,DataOut,cy1, cy2)
input signal DataIn[WORDSZ / 2], cy1, cy2;
output signal DataOut[WORDSZ];
{
signal A[WORDSZ], B[WORDSZ];
int i;

/* latch high order bits */
for (i = WORDSZ; i-- > WORDSZ / 2; ) {
            A[i] = A[i] & ~cy2 | DataIn[WORDSZ - 1 - i] & cy2;
            B[i] = B[i] & cy2 | A[i] & ~cy2;
}
/* latch low order bits */
for (i = WORDSZ / 2; i--; ) {
            A[i] = A[i] & ~cy1 | DataIn[i] & cy1;
            B[i] = B[i] & cy1 | A[i] & ~cy1;
}
for (i=WORDSZ; i--; )
            DataOut[i] = B[i];
}                       /* end of register() */
```

Figure A.4

```
BCLA (p0,p1,p2,p3,g0,g1,g2,g3,cin,c0,c1,c2,c3)
input signal cin,p0,p1,p2,p3,g0,g1,g2,g3;
output signal c0,c1,c2,c3;
{
/*            4-bit Block Carry look-Ahead unit                  */

c0 = g0 | cin & p0;
c1 = g1 | new c0 & p1;
c2 = g2 | new c1 & p2;
c3 = g3 | new c2 & p3;
}                       /* end of BCLA() */
```

Figure A.5

```
Bsum (c,p,s)
input signal c[WORDSZ],p[WORDSZ];
output signal s[WORDSZ];
{
/*              add two words together using output from BLOCK look-ahead
 *             unit
 */
int i;
signal gnd;

s[0] = p[0];
for (i = WORDSZ; --i; )
              s[i] = p[i] ^ c[i-1];
}                           /* end of Bsum() */
```

Figure A.6

```
add3w (x,y,z,s,Cout)
input signal x[WORDSZ], y[WORDSZ], z[WORDSZ];
output signal s[WORDSZ], Cout;
{
/*              s = x + y + z
 *             reduce three operands to one by using two two-word adders
 *             in series
 */
signal tmp[WORDSZ], p[WORDSZ], g[WORDSZ], pt[WORDSZ], gt[WORDSZ], gnd,
              c[WORDSZ], ct[WORDSZ];
int i;

/* tmp = x + y */
CLAgen (x,y,p,g);
BCLA (p[0],p[1],p[2],p[3],g[0],g[1],g[2],g[3],gnd,c[0],c[1],c[2],c[3]);
for (i = 4; i < WORDSZ; i += 4)
              BCLA (p[i],p[i+1],p[i+2],p[i+3],g[i],g[i+1],g[i+2],g[i+3],c[i-1],
              c[i],c[i+1],c[i+2], c[i+3]);
Bsum (c,p,tmp);
/* s = tmp + z */
CLAgen (tmp,z,pt,gt);
BCLA (pt[0],pt[1],pt[2],pt[3],gt[0],gt[1],gt[2],gt[3],gnd,ct[0],ct[1],
              ct[2],ct[3]);
for (i = 4; i < WORDSZ; i += 4)
              BCLA (pt[i],pt[i+1],pt[i+2],pt[i+3],gt[i],gt[i+1],gt[i+2],gt[i+3],
                        ct[i-1], ct[i],ct[i+1],ct[i+2], ct[i+3]);
Bsum (ct,pt,s);
}                           /* end of add3w() */
```

Figure A.7

```
chkovf (p,ovfl)
input signal p[WORDSZ+1];
output signal ovfl;
{
/*              or bits together to see if result has overflowed
 */
int i;

ovfl = p[WORDSZ];
for (i = WORDSZ; i−−; )
                ovfl = new ovfl | p[i];
}                           /* end of chkovf() */
```

Figure A.8

```
add0 (x,y,z,s,c)
input signal x, y, z;
output signal s, c;
{
/*              1−bit full adder                    cs = x + y + z
 *           where       c = carry
 *                       s = sum
 */

s = z & (~x & ~y | x & y) | (x ^ y) & ~z;
c = x & y | y & z | z & x;
}                           /* end of add0() */
```

Figure A.9

```
add2 (x,y,z,s,c)
input signal x, y, z;
output signal s, c;
{
/*              1−bit full adder                    (−c)s = −x − y + z
 *           where       c = complement of carry
 *                       s = sum
 */

s = z & (~x & ~y | x & y) | (x ^ y) & ~z;
c = x & y | y & ~z | x & ~z;
}                           /* end of add2() */
```

Figure A.10

```
mult (a.b.c.ovfl)
input signal a[WORDSZ]. b[WORDSZ];
output signal c[WORDSZ]. ovfl;
{
/*              form the multiplication of a and b by using an array
 *              of full adders
 *                          c = a * b
 *                          ovfl = 1 if overflow
 */
int i, j;
signal p[WORDSZ+1]. s[WORDSZ][WORDSZ]. co[WORDSZ][WORDSZ+1]. gnd;

c[0] = a[0] & b[0];
for (i=WORDSZ-2; i--; )
            add0 (a[i+1] & b[0]. a[i] & b[1]. gnd. s[i][0]. co[i][0]);
c[1] = new s[0][0];
for (j = 1; j < WORDSZ - 2; j++ ) {
            for (i = WORDSZ - 3; i--; )
                        add0 (new s[i+1][j-1]. new co[i][j-1]. a[i] & b[j+1].
                                    s[i][j]. co[i][j]);
            add0 (a[WORDSZ-2] & b[j]. new co[WORDSZ-2][j-1].
                        a[WORDSZ-3] & b[j+1]. s[WORDSZ-2][j]. co[WORDSZ-2][j]);
            c[j+1] = new s[0][j];
}
for (i=0; i < WORDSZ-3; i++)
            add0 (new s[i+1][WORDSZ-3]. new co[i][WORDSZ-3]. gnd.
                        s[i][WORDSZ-2]. co[i][WORDSZ-2]);
add0 (a[WORDSZ-2] & b[WORDSZ-2]. new co[WORDSZ-2][WORDSZ-3]. gnd.
            s[WORDSZ-2][WORDSZ-2]. co[WORDSZ-2][WORDSZ-2]);
for (i=0; i < WORDSZ-2; i++)
            add2 (new s[i][WORDSZ-2]. a[WORDSZ-1] & b[i]. a[i] & b[WORDSZ-1].
                        s[i][WORDSZ-1]. co[i][WORDSZ-1]);
c[WORDSZ-1] = new s[0][WORDSZ-1];
add2 (new co[WORDSZ-3][i]. a[WORDSZ-1] & b[i]. a[i] & b[WORDSZ-1].
            s[i][WORDSZ-1]. co[i][WORDSZ-1]);
add2 (new s[1][WORDSZ-1]. new co[0][WORDSZ-1]. gnd. p[0].
            co[0][WORDSZ]);
for (i=0; ++i < WORDSZ-2; )
            add2 (new s[i+1][WORDSZ-1]. new co[i][WORDSZ-1].
                        new co[i-1][WORDSZ]. p[i]. co[i][WORDSZ]);
add2 (a[WORDSZ-1] & b[WORDSZ-1]. new co[WORDSZ-2][WORDSZ-1].
            new co[WORDSZ-3][WORDSZ-1]. p[WORDSZ -1]. p[WORDSZ]);
chkovf (p.ovfl);
}                           /* end of mult() */
```

Figure A.11

```
mux (A.B. cntl.out)
input signal A[WORDSZ]. B[WORDSZ]. cntl:
output signal out[WORDSZ]:
{
/*              multiplex inputs A and B                  */
int i;

for (i = WORDSZ; i--; )
            out[i] = A[i] & cntl | B[i] & ~cntl;
}                          /* end of mux() */
```

Figure A.12

```
muxinv (A.B. cntl.out)
input signal A[WORDSZ]. B[WORDSZ]. cntl;
output signal out[WORDSZ];
{
/*              multplex inputs A and not B               */
int i;

for (i = WORDSZ; i--; )
            out[i] = A[i] & cntl | ~B[i] & ~cntl;
}                          /* end o muxinv() */
```

Figure A.13

```
cas (P.Bi.Ai.Ci.Si.Cip1)
input signal P, Bi, Ai, Ci;
output signal Cip1, Si;
{
/*              Controlled Add/Subtract cell
 *                        P = 1  subtract
 *                            0  add
 *                        Ci = carry in
 *                        Si = result
 *                        Cip1 = carry out
 */

Si = (Bi ^ P) ^ Ai ^ Ci;
Cip1 = Ai & (Bi ^ P) | (Bi ^ P) & Ci | Ai & Ci;
}                          /* end of cas() */
```

Figure A.14

REFERENCES

[AHO 74] A. V. Aho and S. C. Johnson, "LR Parsing," *Computing Surveys*, June 1974.

[BAR 81] M. Barbacci, "Instruction Set Processor Specifications (ISPS): The Notation and its Specification," *IEEE Transactions on Computers*, vol. C-30, pp. 24-40, January 1981.

[BRA 82] R. K. Brayton, J. D. Cohen, and G. D. Hachtel, B. M. Trager, D. Y. Y. Yun, "Fast Recursive Boolean Function Manipulation," *International Symposium on Circuits and Systems*, pp. 24-40, May 1982.

[BRA 84a] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. Sangiovanni-Vincentelli, "ESPRESSO-II: A New Logic Minimizer for Programmable Logic Arrays," *Custom Integrated Circuits Conference*, pp. 370-376, May 1984.

[BRA 84b] R. K. Brayton, N. L. Brenner, C. L. Chen, G. DeMichelli, C. T. McMullen, and R. H. J. M. Otten, *The Yorktown Silicon Compiler*, IBM Technical Report, 1984.

[BRE 84] Norman Brenner, "The Yorktown Logic Language: An APL-like Design Language for VLSI Specification," *International Conference on Computer Design*, pp. 11-15, October 8-11, 1984.

[CHE 83] N. P. Chen, C. P. Hsu, E. S. Kuh, C. C. Chen, and M. Takashi, "BBL: A Building Block Layout System for Custom Chip IC Design," *International Conference of Computer-Aided Design*, pp. 40-90, 1983.

[CHE 84] Edmund K. Cheng, "Verifying Compiled Silicon," *VLSI Design*, pp. 70-74, October 1984.

[DEN 84] L. Z. Deng, E. L. Leiss, and B. C. McInnis, "A Refinement of the Quine-McClusky Algorithm by Using Presorting," *submitted IEEE Transactions on Ciruits and Systems*, November 1984.

[ELL 82] S. A. Ellis, K. H. Keller, A. R. Newton, D. O. Pederson, A. L. Sangiovanni-Vincentelli, and C. H. Sequin, "A Symbolic Layout Design System," *International Symposium on Circuits and Systems*, vol. 2, pp. 670-676, May 10-12, 1982.

[FEL 83] Stuart I. Feldman, "The Circuit Design Language Xi," *International Conference on Computer Design*, pp. 652-655, 1983.

[GAJ 84] D. D. Gajski and J. J. Bozek, "ARSENIC: Methodology and Implementation," *Proceedings of the International Conference on Computer-Aided Design*, pp. 116-118, November 1984.

[GAR 79] Michael R. Garey and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Murray Hill, NJ: W. H. Freeman and Company 1979, pp. 209,218.

[HAC 82] G. D. Hachtel, A. R. Newton, and A. L. Sangiovanni-Vincentelli, "Techniques for Programmable Logic Array Folding," *19th Design Automation Conference*, pp. 147-153, 1982.

[HAY 84] Frederick Hayes-Roth, "The Knowledge-Based Expert System: A Tutorial," *IEEE Computer*, pp. 11-28, September 1984.

[HON 74] S. J. Hong, R. G. Cain, D. L. Ostapko, "MINI: A Heuristic Approach for Logic Minimization," *IBM Journal of Research and Development*, pp. 443-458, September 1974.

[JOH 75] S. C. Johnson, *Yacc: Yet Another Compiler Compiler*, Computer Science Technical Report no. 32, Bell Laboratories, Murray Hill, NJ 1975.

[JOH 79] D. Johanssen, "Bristle Blocks: A Silicon Compiler," *16th Design Automation Conference*, pp. 310-313, 1979.

[KAN 81] S. Kang and W. M. vanCleemput, "Automatic PLA Synthesis from a DDL-P Description," *Proceedings of the 18th Design Automation Conference*, pp. 391-397, June 1981.

[KER 70] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell System Technical Journal*, vol. 49, pp. 291-307, 1970.

[KER 78] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Englewood Cliffs, NJ: Prentice-Hall, INC. 1978.

[KNU 73] Donald E. Knuth, *The Art of Computer Programming: Sorting and Searching*, Reading, MA: Addison-Wesley Publishing Company, 1973.

[LEE 61] C. Y. Lee, "An Algorithm for Path Connections and Its Applications," *IRE Transactions on Electronic Computers*, vol. EC10, pp. 346-365, September 1961

[LES 75] M. E. Lesk, *Lex - A Lexical Analyzer Generator*, Computer Science Technical Report no. 39, Bell Laboratories, Murray Hill, NJ, October 1975.

[LEW 84] J. L. Lewandowski and C. L. Liu, "A Branch and Bound Algorithm for Optimal PLA Folding," *21st Design Automation Conference*, pp. 426-433, 1984.

[LUI 84] Kin-man Ivy Lui, *Special Purpose Computer Architecture for LU Factorization of Partitioned Systems*, University of Illinois at Urbana-Champaign, Report R-1015 UIUL-ENG 84-2209, August 1984.

[LUR 84] C. Lursinsap and D. Gajski, "Cell Compilation with Constraints," *Proceedings of the 21st Design Automation Conference*, pp. 103-108, June 1984.

[MAH 84] Grace H. Mah, "PANDA: A PLA Generator for Multiply-Folded PLAs," *IEEE International Conference on Computer-Aided Design, ICCAD-84*, pp. 122-124, November 12-15, 1984.

[MEA 80] Carver Mead and Lynn Conway, *Introduction to VLSI System*, Reading, MA: Addison-Wesley Publishing Company, 1980.

[MUN 57] J. Munkres, "Algorithms for the Assignment and Transportation Problems," *Journal SIAM*, vol. 5, pp. 32-38, March 1957.

[NEW 85] A. R. Newton, D. O. Pederson, and A. L. Sangiovanni-Vincentelli, "Design Aids for VLSI: A Perspective Revisited," *IEEE Design and Test of Computers*, pp. 106-115, April 1985.

[SOU 83] Jay R. Southard, "MacPitts: An Approach to Silicon Compilation," *IEEE Computer*, pp. 74-82, December 1983.

[THO 83] Donald E. Thomas, Charles Y. Hitchcock III, Thaddeus J. Kowalski, Jayanth V. Rajan, and Robert A. Walker, "Automatic Data Path Synthesis," *IEEE Computer*, pp. 59-64, December 1983.

[VEC 83] Mario P. Vecchi and Scott Kirkpatrick, "Global Wiring by Simulated Annealing," *IEEE Transactions on Computer-Aided Design*, vol. CAD-2, no. 4, pp. 215-222, October 1983.

[WHI 84] Steve R. White, "Concepts of Scale in Simulated Annealing," *International Conference on Computer Design*, pp. 646-651, October 8-11, 1984.

[YOU 85] Jeremy Young, "IC-Design Automation Strides into Silicon-Compilation Era," *Electronics*, pp. 58-63, June 24, 1985.