SOCRATES: A SYSTEM FOR AUTOMATICALLY SYNTHESIZING AND OPTIMIZING COMBINATIONAL LOGIC

David Gregory*, Karen Bartlett**, Aart de Geus*, and Gary Hachtel**

*GE Calma Company Research Triangle Park, North Carolina
**Department of Electrical Engineering University of Colorado at Boulder

ABSTRACT

This paper presents SOCRATES, a system of programs which synthesize and optimize combinational logic circuits from boolean equations. SOCRATES optimizes logic using boolean and algebraic minimization techniques, and it optimizes circuits derived from this logic in a user defined technology with a rule based expert system. This paper discusses the goals of logic synthesis and the capabilities needed in a tool to meet these goals. SOCRATES's capabilities are then presented and demonstrated with experiments run on circuits from the 1986 Design Automation Conference synthesis benchmark set.

INTRODUCTION

Logic synthesis programs are designed to improve engineering productivity by designing combinational circuits automatically. The effectiveness of such programs depends on their ease of use, and the quality of the circuits they produce. A number of factors that affect these characteristics are discussed below.

The quality of a synthesized circuit should be measured against the constraints placed on that circuit's performance and fabrication. Circuits are often constrained by the types and characteristics of components available, and by area, delay, power, and testing requirements. Synthesis programs should therefore be capable of generating circuits with competitive area, speed, power, and testablility characteristics. Different constraints are not always compatible though. For example, the smallest implementation of a design is rarely the fastest. So, an automatic synthesis program should also be able to make tradeoffs between competing constraint goals.

Logic synthesis programs must do more than synthesize good circuits. There are many circuit technologies available today, and the characteristics of these technologies change often. A synthesis program should be able to work with all of these technologies, and should also be easily updated to reflect changes in a technology.

Finally, the tool should be fast, reliable, and easy to use. It should not require expert knowledge to use, nor should it take an inordinately large computer to run. Users should also have a mechanism for verifying the correctness of their automatically produced circuits.

This paper presents a logic synthesis system called SOCRATES and shows how this system addresses each of the performance criteria mentioned above. The paper concludes with results from running SOCRATES on the 1986 DAC synthesis benchmark circuits[1].

BACKGROUND

Algebraic methods to automatically synthesize and optimize combinational circuitry have been available since the early 1950's[2, 3, 4]. These methods, and others developed since then[5, 6, 7, 8, 9], use logic minimization to reduce the amount of logic in equations which represent the desired functionality of a circuit. Once equations are reduced, they are transformed into circuits by mapping boolean operators to appropriate circuit components.

Since reduced logic normally results in fewer circuit components, logic level techniques can be very effective at reducing the size of circuits. Different components can be more or less area efficient, however, and so less logic is not always guaranteed to result in smaller circuits. Other physical characteristics like timing and power consumption vary even more substantially by component. For example, in the case of timing, circuit delays depend not only on the number of levels of logic, but also on the relative drive factors and loads of all components on critical paths. Since logic minimization methods do not model these factors, they are at a great disadvantage when optimizing their effects.

Recently, a number of circuit synthesizers have been developed which use circuit models of components[10, 11, 12, 13]. These systems operate by replacing initial circuit components with others that improve overall circuit characteristics. Since the physical characteristics of components are available, actual circuit size and performance can be measured and thus improved. However, because the design representation for these systems is more

0738-100X/86/0000/0079$01.00 ©1986 IEEE

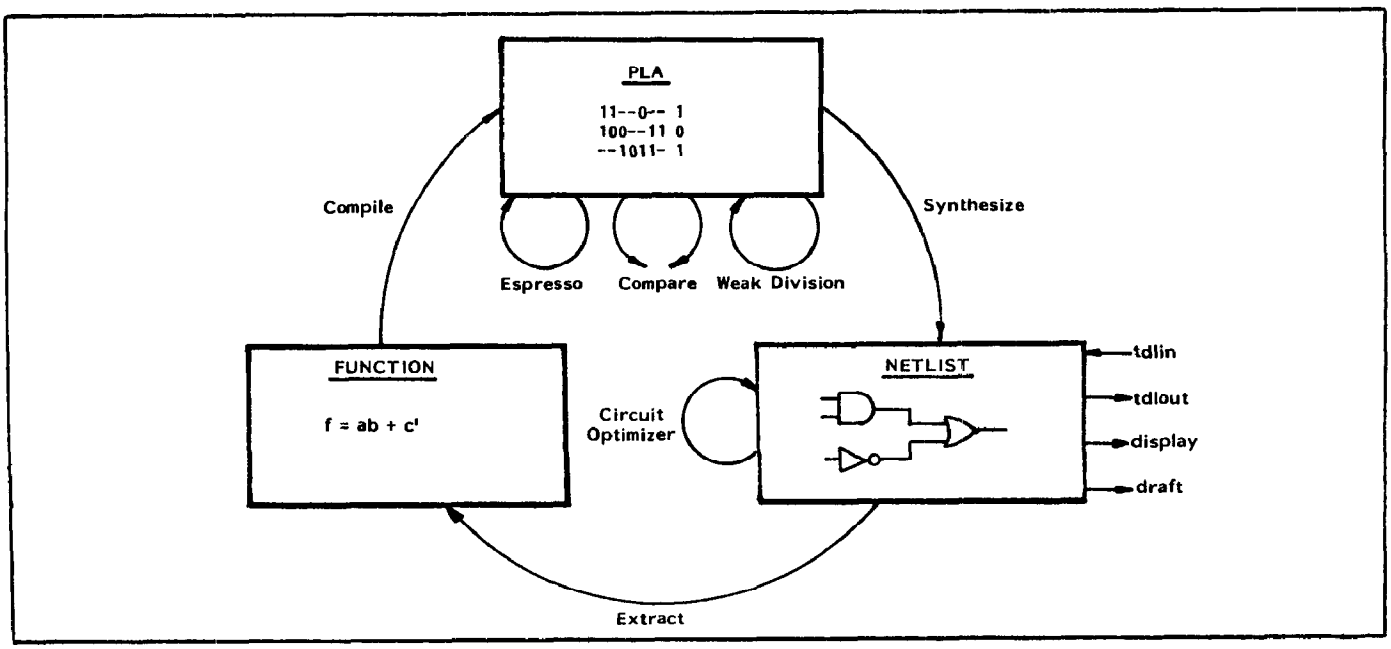

Figure 1: SOCRATES System Diagram

complex, logic level manipulations which reduce overall complexity are much more difficult to recognize and carry out.

SOCRATES takes advantage of both the approaches described above by incorporating logic and circuit level manipulation and optimization techniques. SOCRATES uses one set of programs to reduce the logic of designs, and another set of programs to choose the best components to implement designs. A common design representation and several smaller programs link the two halves together in a flexible and powerful configuration.

OVERVIEW OF SOCRATES

This section describes the SOCRATES system. A discussion of SOCRATES's design representation is followed by a description of its component programs. The section concludes with a more detailed description of the algebraic and circuit level optimizers: *weak division*, and the *circuit optimizer*.

Logic level manipulation and circuit level manipulation operate on different aspects of a design. One operates on the logic, and the other operates on the circuit components which implement that logic. To facilitate operations at each level, SOCRATES uses more than one format for representing designs. Programs performing logic level manipulation use an extended version of *espresso*'s[8] PLA format. Programs performing circuit level manipulations use a net-list format, and a third format is used for entering equations by hand. All three formats share a common constraint specification section and design history section. Translators are also provided for converting designs from one format to another.

A constraint specification allows designers to describe the desired characteristics of their circuits. Designers can specify when signals arrive at inputs, and the drive factor associated with them. Designers can also specify the maximum propagation delays to individual outputs, and the loads that must be driven at those outputs. At present, SOCRATES does not accept area constraints; it automatically synthesizes the smallest circuit it can under the timing constraints specified.

A design history contains an ordered list of the programs that have run on a design. This list is maintained automatically by all SOCRATES programs.

Figure 1 shows the three design formats used by SOCRATES and the various programs that interact with these formats. Each program is described briefly below.

Three programs, *compile*, *synthesize*, and *extract*, translate designs from one format to another. *Compile* converts boolean equations to the two-level *espresso*[8] PLA format. Multilevel equations are flattened to two-level equations in this step. *Synthesize* converts designs from the PLA format to the netlist format. Generic AND, OR, and NOT gates are used to implement corresponding logic in this step. *Extract* converts a netlist to boolean equations. *Extract* uses a boolean variable to represent each signal in the netlist and writes an equation for each gate.

Two programs, *espresso*, and *weak division*, perform logic level manipulation on designs. *Espresso* finds a minimal sum of products for each two level function. *Weak division* decomposes two level functions into multiple levels by iteratively dividing out common subexpressions algebraically. *Weak division* is discussed in more detail below.

The *circuit optimizer* program manipulates designs at the circuit level. This program improves circuit characteristics by iteratively replacing and rearranging groups of components in the circuit. The *circuit optimizer* is also discussed in more detail below.

Compare is a program which takes two designs in the PLA format, and compares them for functional equivalence. When used with translators, *compare* can verify designs in any format. This program has proved quite useful for verifying the correctness of design changes made by users, and for debugging portions of the SOCRATES system.

The remaining programs translate information to and from the netlist format. *Draft* generates a schematic from a netlist. All of the circuits shown in this paper were created using this program. *Tdlin* converts circuits from the TEGAS Design Language (TDL) to the netlist format. *Tdlin* filters out sequential circuit components and stores them so they can be added back later. *Tdlout* converts designs in netlist format back to TDL. Sequential circuit components removed by *tdlin* are automatically reinserted by *tdlout*. Finally, *display* computes and displays circuit statistics including the size and delay of a design.

Weak Division

Weak division is a method originally developed by Brayton and McMullen for algebraically decomposing a set of boolean functions[7]. Weak division decomposes functions by successively dividing the functions by subexpressions that appear more than once. The algorithm used is illustrated in figure 2.

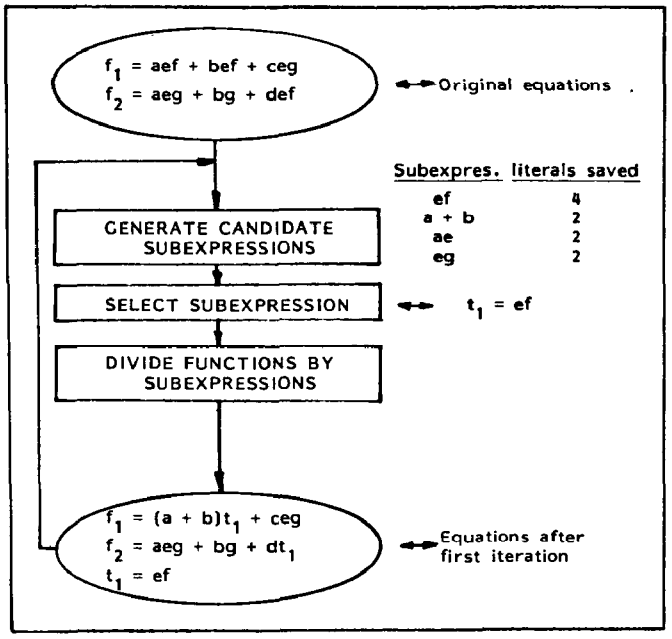

Figure 2: Weak Division Algorithm

In the first step of the algorithm, a list of candidate subexpressions is generated. The desirablity of each candidate is evaluated by a cost function in the second step. The best candidate is then chosen, and divided out of all functions in which it appears. Since the substitution of a subexpression may allow new candidates to be generated in terms of this subexpression, the entire process is repeated. Weak division stops when no more desirable candidates are found.

The cost function has a great effect on the characteristics of the final equations. When optimizing for area, *weak division* uses a cost function which estimates the total number of literals in the set of equations. More complex functions based on levels of logic and predicted circuit delays are used for timing optimization. See [BARTLETT85][9] for more details.

Circuit Optimization

The *Circuit optimizer* improves measurable circuit characteristics by iteratively replacing and rearranging small portions of a circuit. The program uses a library which describes alternative circuit implementations in a rule (*if* antecedent *then* consequent) form. The antecedent portion of the rule lists conditions that must hold before the alternative is valid, and the consequent lists actions which implement the alternative. New rules can be generated automatically from netlists of two alternatives. Figure 3 shows some example rules.

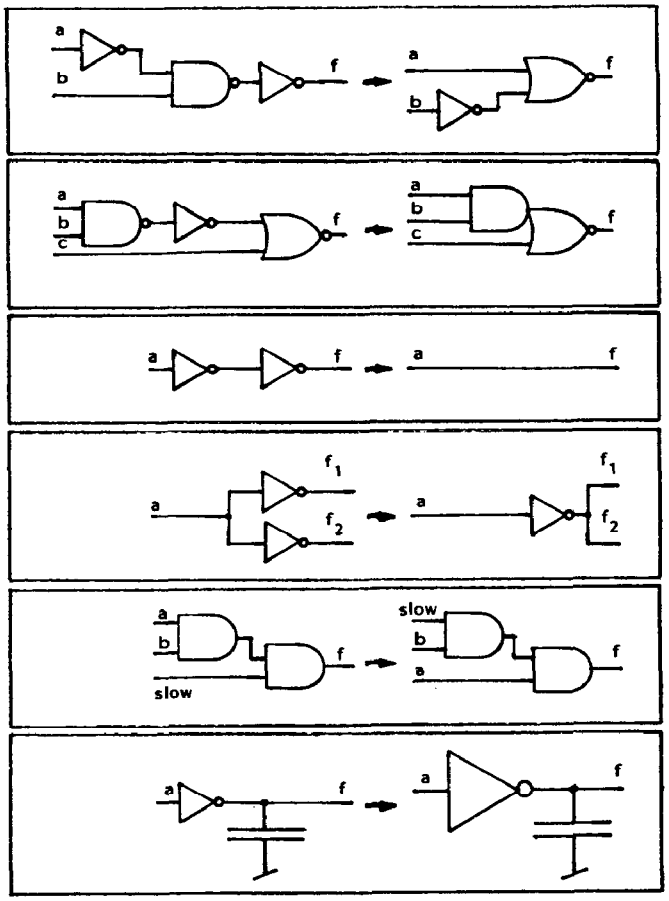

Figure 3: Example rules

Competing alternatives are evaluated by implementing each in turn and measuring the resulting circuits. The measurements are made by an incremental timing and area analysis which is performed after each rule application. These analyses are based on user supplied values for timing and area models of each gate. Different technologies use different values for the models of each gate. These values

are supplied to the program in libraries. So, when competing alternatives are compared, their actual area and performance in surrounding circuitry is used as the basis for evaluation. Also, a new technology is described to the system by inserting the appropriate area and timing parameters in a technology library file.

At present, no power or testability analysis is performed on circuits, and consequently optimization of these quantites is not possible. However, the *circuit optimizer* design representation does contain all the information needed to measure a circuit's power and testablity requirements. When the ability to measure these quantites has been added, the program should be able to choose alternatives which improve these circuit characteristics as well.

Before a rule is selected, transformations on the circuit resulting from its application are attempted. The program evaluates the effects a transformation will have on other transformations in the future by performing a state search. The depth and breadth of the search tree determine how far, and how exhaustively the program looks into the future before selecting a new rule. This look-ahead mechanism enables the *circuit optimizer* to choose transformations which do not immediately improve a circuit, but lead to other transformations which do. Using look-ahead has resulted in average area improvements of 12 percent[14].

There are currently 163 rules in the *circuit optimizer* program. These rules are grouped and ranked automatically by a program which evaluates their expected effect on a circuit. A meta-level rule-based expert system uses these groupings with feedback from circuit measurements to decide where to look for the next rule, and sets the breadth and depth of the state search performed before choosing an alternative.

Meta-rules control how area and speed are traded off against each other, and when and where CPU time is used. The current system applies rules which improve the speed of a circuit to components on critical timing paths until all timing constraints are met, or until it can make the circuit no faster. It then applies area saving rules wherever possible until no further improvements can be made. The breadth and depth of the state search used for each rule application determine the CPU time spent on a design. Currently, these quantities are set at values which have been empirically shown to yield the fastest and smallest circuits possible. A future enhancement will allow users to limit the size of these trees to allow less critical designs to be processed more efficiently. The papers [COHEN85][14] and [DEGEUS85][15] discuss the *circuit optimizer*, and its meta rule system in more detail.

DESIGN SCENARIOS

By maintaining distinct programs, SOCRATES provides a more flexible system than would otherwise be possible. A number of design scenarios are presented below to demonstrate this flexibility and to show how SOCRATES is used to solve a number of engineering problems. Figure 4 summarizes these scenarios graphically.

In the first scenario, SOCRATES is used to design a circuit from a set of boolean equations. In the first step of this scenario the user enters equations in the function format to a text file. If there are timing constraints on the circuit, they would be entered at this time as well. The *compile* program is run next. *Compile* flags syntax errors and reports

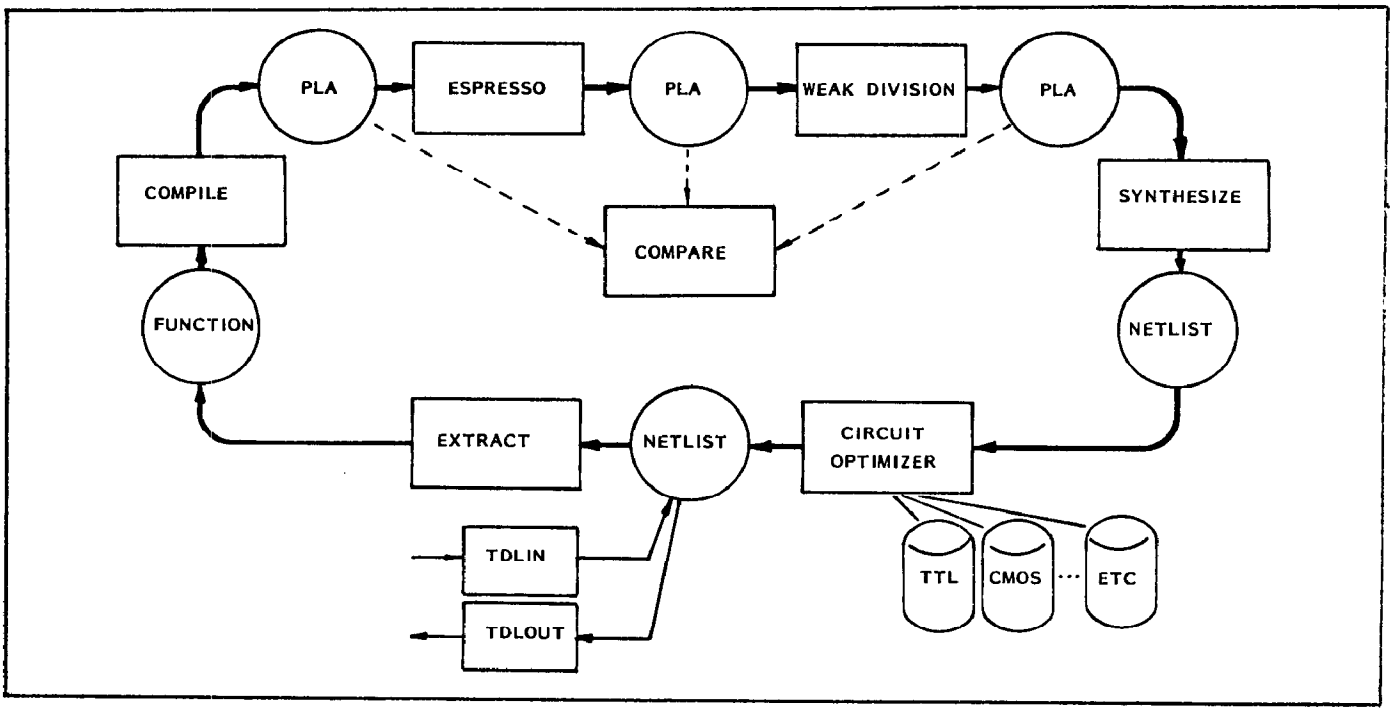

Figure 4: Example Design Scenarios

them to the user. After a syntatically correct file has been created, *espresso*, *weak division*, *synthesize*, and the *circuit optimizer* are run to create a circuit. To view or analyze the new circuit, the user runs *draft* or *display*. If the circuit meets its constraints, *tdlout* converts the design to TDL format. If the constraints are not met, new constraints must be formulated. *Weak division* and/or the *circuit optimizer* can then be rerun to create a new circuit.

In another scenario, SOCRATES starts with an existing circuit that needs to be made smaller or faster. The circuit is read in with *tdlin* and a constraint specification section is added. The design can now be massaged at a circuit level with the *circuit optimizer*, or its logic can be reworked first by running *extract*, *compile*, *espresso*, *weak division*, and *synthesize*, and then running the *circuit optimizer*. If the architecture of the original circuit has already been carefully designed and optimized, a rework by *espresso* and *weak division* may not be necessary. If the logic of the circuit has not been reduced, though, these programs should be run. The user could also try both alternatives and choose the best circuit. When finished, *tdlout* builds a TDL representation of the circuit and replaces any sequential elements removed by *tdlin*.

To convert a TTL design to CMOS, or visa versa, a design is read in with *tdlin*, and run through the *circuit optimizer* with the appropriate technology library. Any two circuits can be tested for functional equivalence by running them both through *extract* and *compile* and then using the *compare* program. SOCRATES programs can also be used individually. For example, PAL or PLA logic can be minimized using *compile*, *espresso* and *weak division*. Many other scenarios are also possible; some of the experiments described in the next section illustrate this.

RESULTS

This section presents the results of running SOCRATES on a set of benchmark circuits[1] prepared for the 1986 Design Automation Conference. Several experiments were run to demonstrate both the power and the flexibility of the system.

Circuit areas for all of the experiments are quoted in terms of two input gate equivalents, and in terms of the number of actual gates used. Circuit delays are quoted in nanoseconds. Both the sum of the delays to all outputs, and the delay of the slowest output are shown. CPU times are quoted in terms of VAX 780 minutes; the VAX 780 is a 1 MIP machine. Except were noted, all circuits were synthesized in General Electric's two micron gate array series, and delays and areas are quoted in terms of this technology. Table 1 shows the gates of this technology used by SOCRATES, and their two input gate equivalent sizes.

SOCRATES was run in its fully automatic mode for all experiments. The user's only interaction with the system was through the constraint specification section.

In the first experiment SOCRATES was run on each of the benchmark circuits to optimize area. No timing constraints were added. Only *espresso*, *weak division*, *synthesize*, and the *circuit optimizer* were used. Table 2 summarizes the results of these runs.

Gate	Description	Size
i1x	Standard inverter	0.5
i2x	Power inverter	1.0
nd2	nand(2)	1.0
nd3	nand(3)	1.5
nd4	nand(4)	2.0
nd5	nand(5)	2.5
nr2	nor(2)	1.0
nr3	nor(3)	1.5
nr4	nor(4)	2.0
aoi21	and(2) into nor(2)	1.5
aoi22	two and(2)s into nor(2)	2.0
oai22	two or(2)s into nand(2)	2.0
aoi32	and(3) & and(2) into nor(2)	2.5
aoi222	three and(2) into nor(3)	3.0

Table 1: Technology description

	Size		Delay		
Circuit	2input	Gates	Total	Worst	CPU
Alupla	125.5	125	175.2	38.0	38.5
Bw	130.5	125	389.0	20.3	77.6
Duke2	318.0	292	567.7	33.2	355.4
F2	16.0	16	27.2	6.8	2.0
Rd53	29.0	28	54.2	19.4	10.7
Rd73	101.5	100	67.3	33.1	34.9
Sao1	184.5	168	92.2	34.8	106.9
Sao2	154.5	143	106.5	34.6	88.8
Vg2	76.0	73	164.3	31.1	17.3
5xp1	104.5	105	149.2	20.56	35.4
9sym	182.0	131	41.2	41.2	37.4

Table 2: Area Benchmarks

	Size		Delay		
Circuit	2input	Gates	Total	Worst	CPU
Alupla	128.0	128	148.9	33.4	129.2
Bw	185.0	173	25.1	25.1	116.5
Duke2	-	-	-	-	-
F2	16.0	16	25.2	6.3	2.2
Rd53	73.0	62	38.5	15.6	25.4
Rd73	105.5	105	62.2	30.9	85.5
Sao1	-	-	-	-	-
Sao2	191.5	163	75.2	24.1	173.8
Vg2	82.0	84	127.6	21.2	87.3
5xp1	137.5	126	94.9	16.1	132.6
9sym	219.5	173	25.1	25.1	116.5

Table 3: Timing Benchmarks

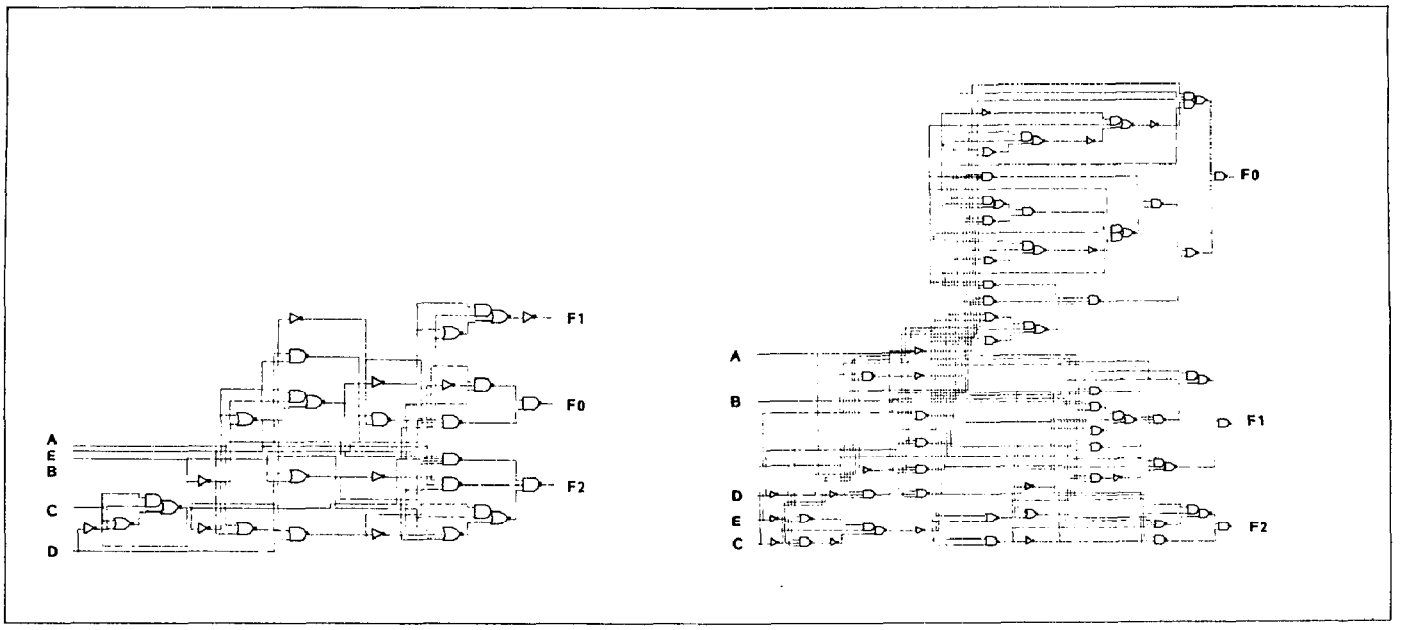

Figure 5: Area Timing Comparison for Example Rd53

In the second experiment the same circuits were resynthesized for minimum total delay by constraining the delay at each output to one nanosecond. Both *weak division* and the *circuit optimizer* were rerun since both perform timing related optimizations. Table 3 summarizes the results of these runs. On average, these circuits are 53 percent faster and 64 percent larger than those synthesized in experiment one.

Figure 5 shows the two circuits created for example RD53. On the left is the circuit optimized for area, and on the right is the circuit optimized for speed.

In the next experiment example Rd53 was run under increasingly severe timing constraints. The results in Figure 6 show the area/speed trade-offs for this example and demonstrate how SOCRATES can be used to explore a design space quickly and easily.

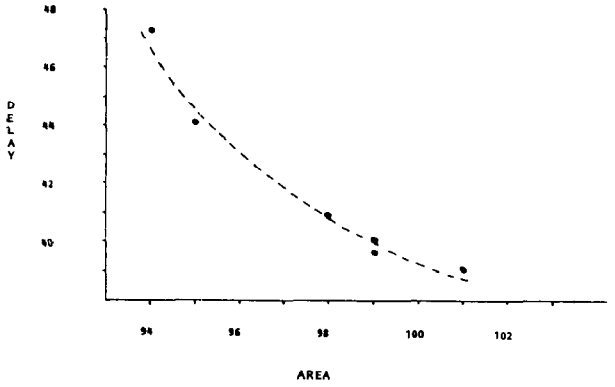

Figure 6: Area Speed Trade-offs for Rd53

To demonstrate SOCRATES'S technology flexibility, a novice user added LSI Logic Inc.'s 5000 series to the system. In addition to different area and timing characteristics, LSI's technology contained five gate types that were previously unknown to SOCRATES. To allow SOCRATES to make use of these gates twenty three new rules were added to the system. It took the new user a total of four man-hours to make all of these changes.

When *espresso* minimizes logic, it creates functions which are prime and irredundant[8]. If a circuit has redundant logic, it may also have undetectable faults. Example C432 from the benchmark circuit set compiled by Brglez and Fujiwara[16], has redundant logic which results in four undetectable faults. C432 was entered into SOCRATES, and resynthesized using *extract, compile, espresso, weak division, synthesize,* and the *circuit optimizer. Esprit*[17], a fault vector generation program, was able to generate vectors which covered all of the faults of the resulting circuit. Resynthesizing C432 with SOCRATES eliminated the untestable faults, reduced the size of the circuit by 2 percent, and increased its speed by 37 percent.

The ultimate test of the quality of an automatic synthesis tool, is to compare its circuits with those designed and optimized under the same constraints by experts. In [DEGEUS85][15], we presented the results of such an experiment in which five experienced engineers competed with SOCRATES to design the smallest circuit for each of three designs. SOCRATES won the competition for all three circuits, and beat the best area of the human designs by an average of 13.5%.

Another more recent experiment pitted SOCRATES against an experienced engineer in the optimization for maximum speed of a standard ALU circuit, TTL part SN54181[18]. Both SOCRATES and the engineer started with a TTL schematic for the part. Since the logic of this design was already well optimized, the *circuit optimizer* was run directly without *espresso* or *weak division*. The total delay

of the hand design was 169.6 nanoseconds. It s worst delay through the circuit was 31.1 nanoseconds, and it used 100 equivalent two input gates. SOCRATES's circuit had a total delay of 170.7 nanoseconds. It s worst delay was 31.8 nanoseconds, and it used 105.5 equivalent two input gates. Overall, the hand designed circuit was faster, but by less than one percent. SOCRATES used two hours of CPU time to create its circuit; the engineer worked on the design for six weeks.

SUMMARY

This paper has presented SOCRATES, a system of programs which automatically synthesizes combinational circuits from functional specifications. SOCRATES uses boolean, algebraic and circuit level optimization techniques to synthesize circuits whose size and speed compare well with circuits designed by expert circuit designers. SOCRATES is easy to use, and flexible enough to handle a variety of synthesis tasks. SOCRATES can synthesize circuits from boolean equations, or from existing circuits in the same, or different technologies. SOCRATES can also be easily changed to design in new technologies.

ACKNOWLEDGEMENTS

We wish to acknowledge and thank William Cohen, Tony Hefner, Bob Lisanke, Tim Moore, Van Morgan, and Jim Reed for their contributions to this project.

References

1. A.J. de Geus, "Logic Synthesis and Optimization Benchmarks". The benchmarks are in Logic Interchange Format (LIF) and are available from the author upon receipt of a magnetic tape or floppy diskette.

2. W. Quine, "The Problem of Simplifying Truth Functions", *American Math Monthly*, Vol. 59, No. 8, October 1952, pp. 521-531.

3. R. Mueller, and R. Urbano, "A Topological Method for the Determination of the Minimal Forms of a Boolean Function", *IRE Transactions on Electronics and Computers*, Vol. EC-5, No. 3, September 1956, pp. 126-132.

4. E. McCluskey, "Minimization of Boolean Functions", *Bell Systems Technical Report*, Vol. 35, No. 5, November 1956, pp. 1417-1444.

5. S. J. Hong, R. G. Cain and D. L. Ostapko, "MINI: A Heuristic Approach for Logic Minimization", *IBM Journal of Research and Development*, Vol. 18, September 1974, pp. 443-458.

6. T. Sasao, "Multiple-Valued Decomposition of Generalized Boolean Functions and the Complexity of Programmable Logic Arrays", *IEEE Transactions on computers*, September 1981.

7. R.K. Brayton, C. McMullen, "The Decomposition and Factorization of Boolean Expressions", *Proceedings of the International Symposium on Circuits and Systems*, 1982, pp. 49-54.

8. R.K. Brayton, G.D. Hachtel, C.T. McMullen, A. Sangiovanni-Vincentelli, *ESPRESSO-IIC: Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Netherlands, 1984.

9. K. Bartlett, G. Hachtel, "Library Specific Optimization of Multilevel Combinational Logic", *Proceedings of the IEEE International Conference on Computer Design*, October 1985.

10. J. Darringer, W. Joyner, L. Bermen, L. Trevillyan, "Logic Synthesis Through Local Transformations", *IBM Journal of Research and Development*, Vol. 25, July 1981, pp. 272-280.

11. D. Gregory, K. Bartlett, A. J. de Geus, "Automatic Generation of Combinatorial Logic from a Functional Specification", *Proceedings of the IEEE International Symposium on Circuits and Systems*, May 1984, pp. 986-989.

12. T. Uehara, "A Knowledge-Based Logic Design System", *IEEE Design and Test of Computers*, October 1985.

13. K. Enomoto, S. Nakamura, T. Ogihara, and S. Murai, "LORES-2: A Logic Reorganization System", *IEEE Design and Test of Computers*, October 1985.

14. W.W. Cohen, K. Bartlett, A.J. de Geus, "Impact of Metarules in a Rule Based Expert System for Gate Level Optimization", *Proceedings of the IEEE International Symposium on Circuits and Systems*, May 1985, pp. 873-876.

15. A.J. de Geus, W.W. Cohen, "A Rule-Based System for Optimizing Combinational Logic", *IEEE Design and Test of Computers*, August 1985, pp. 22-32.

16. F. Brglez, H. Fujiwara, "Benchmarks for Automatic Test Vector Generation". The benchmarks have been put together for the 1985 International Symposium on Circuits and Systems and are available from the authors upon receipt of a magnetic tape.

17. R. Lisanke, F. Brglez, D. Gregory, A.J. de Geus, "Testability-Driven Automatic Test Vector Generation", *Proceedings of the International Test Conference*, September 1986, pp. Submitted.

18. National Semiconductor, *Logic DataBook*, National Semiconductor, 1981.