acm

SIGDA NEWSLETTER

SPECIAL INTEREST GROUP ON DESIGN AUTOMATION

VOLUME 4 NUMBER 3 SEPTEMBER 1974

Contents:

MEMBERSHIP

SIGDA dues are $3.00 for ACM members and $5.00 for non-ACM members. Checks should be made payable to the ACM and may be mailed to the SIGDA Secretary/Treasurer listed above, or to SIGDA, ACM Headquarters, 1133 Avenue of Americas, New York, N. Y. 10036. Please enclose your preferred mailing address and ACM Number (if ACM member).

SIG/SIC FUNCTIONS

Information processing comprises many fields, and continually evolves new subsectors. Within ACM these receive appropriate attention through Special Interest Groups (SIGs) and Special Interest Committees (SICs) that function as centralizing bodies for those of like technical interests ... arranging meetings, issuing bulletins, and acting as both repositories and clearing houses. The SIGs and SICs operate cohesively for the development and advancement of the group purposes, and optimal coordination with other activities. ACM members may, of course, join more than one special interest body. The existence of SIGs and SICs offers the individual member all the advantages of a homogeneous narrower-purpose group within a large cross-field society.

ACTIVITIES

1) Informal technical meetings at SJSS and FJCC.
2) Formal meeting during National ACM meeting + DA Workshop.
3) Joint sponsorship of annual Design Automation Workshop.
4) Quarterly newsletter.
5) Panel and/or technical sessions at other National meetings.

FIELD OF INTEREST OF SIGDA MEMBERS

Theoretic, analytic, and heuristic methods for:
1) performing design tasks,
2) assisting in design tasks,
3) optimizing designs through

the use of computer techniques, algorithms and programs to:
1) facilitate communications between designers and design tasks,
2) provide design documentation,
3) evaluate design through simulation,
4) control manufacturing processes.

From the Editor

This issue of the SIGDA Newsletter is being sent to a large number of non-members who attended the 11th Design Automation Workshop in Denver June 17-19. If you are a nonmember with interests in DA, I hope you will join us and participate in SIGDA activities. Upcoming issues of the Newsletter will certainly justify the minimal fee involved.

We plan to bring you a series of larger, more stimulating newsletters, starting with the present number. A serious effort is being made to collect material suitable for distribution in the newsletter. If you have copy, send it to me! If you have comments (publishable or otherwise) concerning newsletter contents, please send them to me. (Note the new Lawrence California address on second cover.)

I am especially interested in promoting non-digital DA and educational topics: although I am primarily interested in digital DA, perhaps we should attempt to achieve more of an emphasis balance.

Copy which is ready for inclusion in the Newsletter is especially welcome: ideal format follows that used in the Workshop Proceedings.

- Use oversized layout sheets suitable for 25% reduction.
- Use an electric typewriter with a carbon ribbon if at all possible.
- Use photostats or xerox illustration reductions and rubber cement them directly onto the layout sheets.
- Use white liquid products to cover errors and type directly over them. Do not erase errors.
- Lightly pencil page numbers on the back of each page of your copy.
- Proof your copy carefully.

We will attempt to publish typewritten material which does not arrive in this format, but much prefer camera ready copy.

CHAIRMAN'S MESSAGE

The Systems' Approach

Design Automation systems have in the past been described to provide:

1) Means for communicating (and storing) design data.
2) Means for controlling and checking design data.
3) Application programs which allow the designer/ engineer capability to perform more functions, quicker, cheaper, and better.

In the past papers at the annual DA Workshop stressed the application programs. Many algorithms and isolated application programs which on their own were efficient were presented. However, this work was seldom tied into how the computer communicates, controls, and checks the design data.

In the past I have pressed for more of a systems approach to Design Automation. Therefore, I was pleasantly surprised to hear a number of papers stress the system aspect. In particular, the presentation, "Data Base Design for Design Automation", by Dr. Ed Hassler of Texas Instrument was greatly appreciated. Ed talked in terms of the total design data base: designers insight, marketing data, technology data, test examples, etc. He described the problem of the control of design data using a familiar example to most of us - a hand held calculator.

This fall (November 11-13, 1974) SIGDA will have a session "Data Base Systems for Design Automation Support". The session will consist of two invited papers and an invited panel.

For August or September, 1975, a Workshop is being planned and jointly sponsored by SIGDA, SIGGRAPH, and SIGFIDET (now SIGMOD). The Workshop will be structured around Interactive Data Base Systems for Design Automation. Professor James Linders of University of Waterloo, Waterloo, Ontario, Canada, will be General Chairman. The Workshop will be held at Waterloo.

I don't want to leave the impression that the programmed functions or algorithms are not important; on the contrary, they use the computer to extend the engineers' capabilities. Both without looking at the total computing and design environment, they cannot exist or grow.

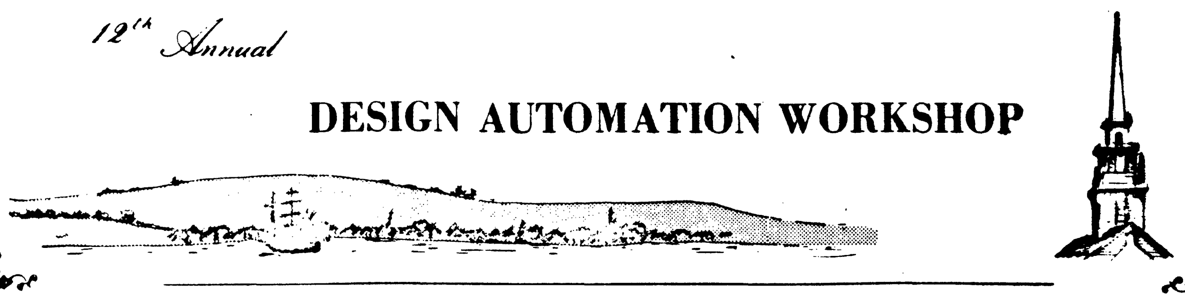
Chuck Radke/pu

Chuck Radke
Chairman, SIGDA

7/30/74

12th Annual

DESIGN AUTOMATION WORKSHOP



BOSTON, MASSACHUSETTS

REQUIREMENTS FOR SUBMITTING PAPERS

If you plan to submit a paper, you should send three copies of the paper (rough drafts are acceptable) to the program chairman no later than January 2, 1975.

Accompanying the draft should be the full name, address, and telephone number of the principal author, with whom all further direct communication will be conducted.

Notification of acceptance will be sent to you during the first week of February, 1975. After notification of acceptance, you will receive detailed instructions on the format to be observed in typing the final copy. To insure the availability of Proceedings at the Workshop, your final manuscript will be due April 21, 1975.

Final papers should be no longer than 5000 words, and the presentation should be limited to 20 minutes. Projection equipment for 35mm slides and viewgraph (overhead projector) foils will be available for every talk. Please indicate what, if any, additional audio-visual aids you require.

Program Chairman

Rough drafts are to be sent to the Program Chairman:

S. A. Szygenda
The University of Texas
Electrical Engineering Department (ENS 515)
Austin, Texas 78712
512-471-7365

Chairman of 12th DAW

R. B. Hitchcock

Sponsors

The sponsors of the Design Automation Workshop are the ACM (Association for Computing Machinery) Special Interest Group on Design Automation and IEEE (Institute of Electrical and Electronics Engineers) Computer Society.

Design Automation

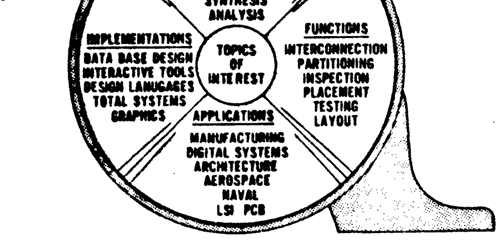
Design Automation implies the use of computers as aids to the design process.

In the broadest sense, the design process includes everything from specifying the characteristics of a product to meet a marketing objective to enumerating the details of how it is to be manufactured and tested.

Thus design automation embraces applications from one end of the design process to the other.

Site of 12th DAW

Statler Hilton Hotel
Park Square at Arlington Street
June 23, 24, 25, 1975



1974 IEEE Workshop on Design Automation

"The 1974 IEEE Workshop on Design Automation will be held on October 23-25 at Michigan State University, Kellogg Center. This workshop is devoted to the systems aspect of design automation. The overall goal is to provide the participants an environment encouraging general and specific discussion relating to the design and construction of design automation software. The sessions are:

1. Data base design

   Organization considerations, centralized vs distributed data bases, data redundancy, location of design rules, data structures, use of data base management systems.

2. Software design

   Choice of languages, structured programming, software aids such as translator writing systems, data structures, programming style, machine hardware and operation system influences.

3. The user interface

   Human factors, input and output modes such as graphics and languages, hardware impact, error analysis of input, DA and general purpose editors, the user/dA software interface.

4. Impact of new technology on D.A.

   Microprogramming automation, programmed logic arrays (PLS's), the future of IC's, impact of microprocessors on DA.

5. Managing a design project

   Project specification, case studies schedule control, module and program function interconnection, acceptance and validation, documentation, configuration project organization, software performance measurement, software control.

Attendance is by invitation only. If you would like to attend, please contact Harold W. Carter, Lawrence Livermore Laboratory, P.O. Box 808, L-156, Livermore, California 94550, (415) 447-1100, extension 8088."

SIGDA SESSION AT ACM '74

San Diego - November 11-13, 1974

SESSION:	Data Base Systems for Design Automation
ORGANIZER:	C. E. Radke

ABSTRACT:

The use of data base systems for Design Automation introduces some special problems; two of which are discussed. The first problem involves assuring the integrity of the design data as it grows and is restructured over time. Discussed are problem areas, for example, authorization of design changes, consistency checks, user notification of changes, monitoring of the design, and history recording. The second involves the evolution of interactive graphics to a computer integrated design system for designing computer hardware. Discussed are common characteristics of a design data base, use of graphics for data display, examples for which graphic concepts have been implemented, and the control of an interactive editing system.

A panel will expand upon the concepts and approaches presented in the published papers.

PAPER: Control of Design Data in the Integrated Ship Design System

AUTHOR: Dr. P. R. Bono, Naval Ship Engineering Center

ABSTRACT:

The Navy's Integrated Ship Design System (ISDS) is being designed as a collection of application program modules (for preliminary design) which communicate with a centralized set of data files. These files use the existing COMRADE Data Management System which was designed specifically for integrated systems.

Apart from providing an environment in which to operate the engineering application modules, ISDS's main role is to manage the creation, flow and archiving of the ship design data to control access to this data. Consequently, a major concern during the lengthy and complex ship design process is assuring the integrity of the design data as it grows and is revised over time.

Planning for control of the design data requires a clear understanding of the design process and the inter-relationships between the design tasks. Requirements are stated, problem areas are identified, and possible approaches for implementation are suggested.

7/30/74

SIGDA SESSION AT ACM '74

San Diego - November 11-13, 1974

PAPER:	A Graphics Window to a Data Base for Electronic System Design
AUTHOR:	Mr. Charles Alaimo, Bell Telephone Laboratories

ABSTRACT:

This paper will explore the use of "Graphics Windows" to data bases used in the physical design of electronic equipment. The comon characteristics of connectivity oriented data bases will be briefly reviewed. Then an example of a Graphics Window on this type of data base will be discussed in detail.

The discussion of the Graphics Window is in two parts. The first deals with its use as an editor of files describing printed board designs. The second deals with experiments conducted with the window which are aimed at developing display techniques useful for interactive design.

PANELISTS: Prof. Charles Rose, Case Western Reserve University

Mr. Al S. Lett, IBM Corporation

7/30/74

DESIGN AUTOMATION IN UNIVERSITIES.

Design Automation is beginning to emerge as a disipline within the university environment. In order to establish a dialogue between the universities involved in DA activities SIGDA has decided to establish a 'DA in Universities' section in its Newsletter.

The following questionaire is intended as a census of Design Automation activities in universities.

The results of this questionaire will be published in one of the following SIGDA Newsletters.

Completed questionaires as well as contributions to this section can be mailed to:

W.M. VanCleemput
Dept. of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

QUESTIONAIRE ON DESIGN AUTOMATION IN UNIVERSITIES.

1. Personal Information.

Name .... Title ....

University ....

Department ....

Address ....

Areas of interest in the DA field..........................

....

....

....

Are you a member of ACM? IEEE? SIGDA?

2. Current Status of DA at your university.

2.1 Faculty Members involved in DA.

2.2 Courses.

Give a brief outline of courses currently being taught; indicate prerequisites, level and text or references used.

2.3 Hardware available.

List the facilities available for Design Automation research (e.g. central computer system, dedicated minicomputer(s), graphics devices, etc.).

2.4 DA Software available.

List major DA software packages available to DA students, if any.

2.5 Ongoing Research.

Briefly describe each project.

2.6 Publications.

List all research reports, theses and papers on DA, published

3. Future Plans.
================
Describe briefly any plans for expanding DA involvement in your department (e.g. new courses, hardware aquisitions etc.).

4. Misc.

4.1 Some areas of Design Automation are more suited for inclusion in a university curriculum than others. In your opinion, what should be emphasized and what should not be taught?

4.2 What role should SIGDA play in DA education?

4.3 What type of research should be done by universities and what should be left to industry?

4.4 (If you are in Computer Science) Do you have any contact with faculty members, interested in DA, in other departments?

4.5 (If you are not in Computer Science) Do you have any contact with the CS department?

4.6 Do you have any contact with industry? Do you consider industrial contact necessary, useful, useless?

AGARD-CP-130

NORTH ATLANTIC TREATY ORGANIZATION

ADVISORY GROUP FOR AEROSPACE RESEARCH AND DEVELOPMENT

(ORGANISATION DU TRAITE DE L'ATLANTIQUE NORD)

AGARD Conference Proceedings No.130

COMPUTER AIDED DESIGN FOR ELECTRONIC CIRCUITS

Copies of papers presented at the 25th Technical Meeting of the Avionics Panel of AGARD held at the Technical University of Denmark, Lyngby, Denmark, 21–25 May 1973.

CONTENTS

Printed by Technical Editing and Reproduction Ltd
Harford House, 7-9 Charlotte St, London. W1P 1HD

A MINICOMPUTER-BASED LOGIC CIRCUIT FAULT SIMULATOR

by

Mark J. Flomenhoft and Brenda M. Csencsits
Bell Telephone Laboratories, Incorporated
Allentown, Pennsylvania 18103

Presented at the Eleventh Design Automation Workshop
June 17-19, 1974

ABSTRACT

A logic circuit simulator implemented on a minicomputer with 16K core handles 1000 zero- and unit-delay gates. Single "stuck-at-0", "stuck-at-1", and "short-circuit" faults are simulated in parallel seven at a time using a table-driven selective-trace fault-injection algorithm. For a typical 100-gate circuit the simulation rate for the fault-free circuit or for a group of seven faults is about 800 input patterns per minute, and a complete fault simulation run takes about 5 minutes and costs $1.

This paper, aimed at simulator program-designers rather than users, describes technical details of the implementation including minicomputer considerations, the data structure, coding three-valued-logic for efficient parallel simulation, the selective-trace algorithm, the recognition and resolution of critical races in flip-flops, the recognition and resolution of circuit oscillations, implicit fault collapsing, and short-circuit fault simulation.

INTRODUCTION

During the past few years an in-house stand-alone interactive integrated-circuit mask layout system has gained wide popularity within Bell Laboratories. At present, eighteen minicomputer systems with a common hardware configuration are in operation at four locations. To further exploit this existing computing resource, System for Logic Analysis and Test Evaluation, SLATE, has been added to provide a low-cost interactive aid for logic verification and functional test design. SLATE also provides a low-cost facility for evaluating test sequences produced by a main-frame heuristic automatic-test-generation program.

Written in assembly language for the HP2100 computer, with 16K core SLATE's capacity is 1000 gates. Each gate has either zero- or unit-delay and is allowed three possible logic values -- 0, 1, and "don't know". Don't-know is the starting value of every gate and the resulting value of critical races and oscillations. The simulation is table-driven and employs a selective-trace fault-injection algorithm. The 16-bit minicomputer word size accommodates eight two-bit logic values, and the fault-free and seven faulty circuits are simulated simultaneously in "parallel".

THE MINICOMPUTER HARDWARE CONFIGURATION AND ITS IMPLICATIONS

The mask layout system that SLATE shares utilizes an HP2100 minicomputer with 16K core, a disk, a magnetic tape unit, a card reader, a CRT keyboard terminal, and, usually, either a line printer or a hard-copy unit. Such a system costs about $50,000, or, amortized over three years and including maintenance, about $12 per hour.

In a 16K minicomputer system core is a scarce resource, and frequent disk accesses would make simulation unacceptably slow. Thus it is essential that the main simulation program and the circuit description data required for simulation are both maintained in core. (An early version of the program consisted of an "executive" and a "simulator" that were swapped from disk for each input pattern. Eliminating these disk accesses by combining the executive and the simulator into a single program module decreased run time by more than an order of magnitude.) In order to accommodate circuits of about a thousand gates, SLATE employs "parallel" fault simulation [1]-[4] rather than the newer "deductive" [5] or "concurrent" [6] techniques because the newer techniques require much more memory.

SLATE is a "table-driven" simulator -- a source circuit description is translated into tables that are directly loaded by the simulator -- rather than a "compiled" simulator -- a source circuit description is translated into an assembly language subprogram that emulates circuit behavior and includes code to perform fault-injection, selective-trace, etc. Where a table-driven simulator resides on secondary storage as a complete core-image program module, a compiled simulator must be assembled (second translation) and loaded (third translation) with an executive and input/output-routine library. Although a compiled simulator executes somewhat faster, a table-driven simulator requires less core and provides greater flexibility for introducing new features.

In-core data -- principally the circuit topology description -- is dynamically allocated to storage as the data is loaded, and pointers are generated to define the start of each data list. On the other hand, infrequently needed data -- the input sequence, symbolic gate names, and the list of faults to be simulated -- are stored on disk in a fixed-space-per-datum format so that each can be "randomly" accessed, eliminating the need for pointers. The input sequence, for example, is stored eight patterns per disk sector (128 words), and an in-core buffer is used so that disk is accessed only every eighth pattern. Both to save space and maximize execution speed, the program is written completely in assembly language.

DATA STRUCTURE

A logic circuit is composed of the following logic elements: primary inputs, equal-delay (unit-delay) gates (ANDs, NANDs, ORs, and NORs), zero-delay "tied collector" nodes (wired-ANDs and wired-ORs), and batteries (constant sources of 0 and 1). For each element a "topology word" designates the element's type (three bits), fanin (three bits), fanout (six bits), delay (one bit), and three status flags: an "activity" bit (1 when the element is scheduled for simulation), a "hold" bit (1 when the element's value is being "held" to resolve a circuit oscillation), and a "fault" bit (1 when stuck-at faults associated with the element are being simulated). Note that although the fanin and fanout of an element are limited to 7 and 63, respectively, fanin- and fanout-trees composed of zero-delay ANDs or ORs make effective fanin and fanout unlimited. (Such trees can, of course, be automatically generated, although this feature is not included in our system.)

The input connections for all the elements are contained in an "input connectivity list": first the input elements of element one, then the input elements of element two, etc. A pointer associated with each element designates the beginning of that element's sub-list of inputs. Since SLATE simulates 1000-gate circuits and the length of the input connectivity list usually exceeds 256, half-word (eight-bit) pointers would be inadequate. Then, given that a full word is required, each pointer is stored as an absolute memory address rather than as a position-in-list index, eliminating repetitive address calculations during simulation. Similarly an "output connectivity list" and a list of pointers define the output connections for every element. Note that with elements defined as above (primary outputs intentionally excluded), the total number of input connections equals that total number of output connections, a useful data check.

As is well known to users of logic circuit simulators, the coarse unit-gate-delay timing model gives rise to many circuit oscillations. Fortunately, critical races in latches (two-gate flip-flops) can be handled very simply, as described below. To enable the critical race analysis to be performed, the pairs of gates composing NAND and NOR latches are identified in a "flip-flops list". A given gate can be included in more than one latch, as occurs for example, in certain toggle flip-flops and generalized latches (three gates each feeding the other two).

In-core data also includes lists of the circuit primary inputs, primary outputs, and user-added monitor points. Space for the remaining data lists -- the CURRENT and NEXT QUEUEs and the NEW VALUEs (see below) -- is dynamically allocated during each unit-delay simulation.

THREE-VALUED LOGIC

In SLATE each element is allowed three possible logic values -- 0, 1, and "don't know". Don't-know is the starting value of every gate and the resulting value of critical races and oscillations. Truth tables for AND, OR, and NOT with don't-know (X) inputs are as follows [7]-[9]:

A	B	$A\cdot B$	$A+B$	A'
X	0	0	X	X
X	X	X	X	X
X	1	X	1	X

To represent the three logic values a two-bit coding is required. The coding

$$0 \rightarrow (00)$$
$$X \rightarrow (01)$$
$$1 \rightarrow (11)$$

allows AND and OR to be executed using the corresponding logical operations of the host computer [9]. Since AND, NAND, OR, and NOR all require AND-ing or OR-ing logic values, this coding is extremely efficient.

Taking the NOT of a three-valued logic variable is not so trivial. Simply complementing the two-bit code is invalid, for whereas $X' = X$, $(01)' = (10)$; but (10) is undefined. Allowing two representations for X [(01) and (10)] makes NOT easy, but then AND and OR are complicated. One solution for NOT is to test the existing value and complement it except when the value is X. However, this scheme is unacceptable for parallel simulation because each value would have to be tested individually, a serial procedure. Instead, a "parallel" NOT is performed by complementing the code and interchanging the bits [4]. It is easily verified that by this technique $0'=1$, $X'=X$, and $1'=0$, as desired.

The "natural" way to pack two-bit logic values in a word is to use the first pair of bits for the first value, the second pair for the second value, etc. While interchanging adjacent bits is straightforward, several program instructions are required, and this is time-consuming. However, packing the values so that the two bits are a half-word apart allows the interchange to be done with one register rotate instruction. For suppose the word bits are denoted $1, 2, \ldots, N$. Then a rotation of $\frac{N}{2}$ bits moves bit i to bit position $(i+\frac{N}{2})(\text{mod } N)$, and every pair of bits a half-word apart are interchanged. (For example, bit 5 is shifted to position $5+\frac{N}{2}$ and $5+\frac{N}{2}$ is shifted to position 5.) Thus NOT is executed in two steps -- complement and rotate. This technique is analagous to the complement and word-interchange described in [4].

For later use we call the two bits of a logic value code "Y" and "Z". Note that Y=0 implies a value is 0 or X, and Z=1 implies a value is X or 1.

CRITICAL RACES IN FLIP-FLOPS

If both the "set" and "clear" inputs of a NAND latch simultaneously change from 0 to 1, the flip-flop undergoes a "critical race", and the resulting state is indeterminate. In a three-valued-logic simulator, such an indeterminate state is conveniently represented by setting the values of the flip-flop's gates to don't-know's [3]. The recognition of critical races is an important attribute of a simulator, for a race manifests itself as an oscillation of the flip-flop's gates -- 11 → 00 → 11 → 00 → ··· -- which markedly degrades simulation efficiency.

Under unit-gate-delay three-valued-logic simulation there are additional critical race situations to be recognized. For example, if a NAND latch is in the "set" state with both its input at 1 and a unit-width 0-pulse occurs on the reset input, the effect is the same as the simultaneous multiple-input change situation. Another example is the simultaneous change of the set and clear inputs from 0 and X to 1 and 1, which produces the oscillation 1X → X0 → 1X → X0 → ··· . After considering all possibilities, one concludes that the flip-flop states 00, 0X, and X0 occur only in response to critical races, and these three states identify all critical race occurrences.

When a critical race state is observed, the computed state is overridden by XX. Taking this outcome into account, we can include XX as a state to be "overridden". We then obtain an especially simple critical race recognition mechanism, namely, both Y-bits equal to 0. (Recall that Y=0 implies a value is 0 or X.) Note, however, that XX should be excluded as a race occurrence if flip-flop races are reported to the user. Once recognized, a race is resolved simply by setting Z=1 for both of the flip-flop's gates, since X is defined by Y=0, Z=1.

We need to extend the critical race prescription so it is appropriate for parallel simulation. Effectively, many independent flip-flops are simulated simultaneously, and our goal is an efficient parallel procedure to selectively override the states of the racing flip-flops only. This is accomplished by OR-ing the corresponding Y-bits of the computed values together (in parallel) and then OR-ing the complement of the result to the Z-bits of each value (in parallel). Thus, wherever both Y-bits are 0, the corresponding Z-bits are set to 1; and wherever a Y-bit is 1, the corresponding Z-bits are unaffected.

For a NOR latch, critical races are recognized and resolved in a dual manner: When both Z-bits are 1, the computed state should be overridden by XX. Thus we AND the corresponding Z-bits of the computed values together and then AND the complement of the result to the Y-bits of each value.

CIRCUIT OSCILLATIONS

As observed previously, unit-gate-delay simulations typically give rise to circuit oscillations that somehow must be artifically terminated so the simulation can proceed. Again, it is convenient to use don't-know's to represent the indeterminate final state of the oscillating gates [3],[4].

The recognition of oscillations is simple, if not elegant: An oscillation is assumed to occur when the input pattern changes and a circuit does not become stable after some reasonable number of gate delays [3]. In our system this "oscillation criterion" is preset at 100 unit delays but can be changed by the user as part of his input sequence stream.

We now describe a technique for resolving oscillations that is optimal in that only oscillating gates are set to don't-know, all other gates being unaffected. (However, don't-know's will propagate to non-oscillating gates that are "sensitized" to don't-know gates.) Such a technique is advantageous for two reasons. First is the obvious point that only necessary don't-know's are imposed, thus preserving as much state information as possible. Second, after the oscillation is resolved, an audit of the don't-know's defines the gates that oscillated, enabling the user to determine the oscillation loops and, hopefully, infer the source of the oscillation.

Our technique is based on the two observations that in every oscillating loop at least one gate changes value at each time unit of the simulation and that oscillating loops are "sensitized paths". When the oscillation criterion is reached, each gate that just changed value is temporarily replaced by a don't-know battery (a constant source of don't-know), thus breaking each oscillation loop in at least one place. The don't-know's then propagate to all the other oscillating gates. Finally, the batteries are removed, reclosing loops that now, being don't-know at each point, are stable.

The temporary replacement of an element by a battery is accomplished easily using the HOLD BIT in the topology word for that element: When an oscillation is recognized, the HOLD BIT of each element that just changed value is set, and an element is not simulated when its hold bit is 1. After the oscillation is resolved, the batteries are removed simply by resetting the HOLD BIT of every element.

All that remains to explain is how don't-know's are applied selectively to parallel circuits so that only the oscillating circuits are affected. Given an element that just changed value, the OR of the corresponding Y and Z bits of the exclusive-OR of the LAST and NEW VALUEs is a mask whose 1's define the circuits that changed. Modifying the element's value by OR-ing that mask to the Z bits and AND-ing the complement of that mask to the Y bits imposes the appropriate X's.

ZERO- AND UNIT-DELAY SELECTIVE-TRACE SIMULATION

In order to ensure accuracy in evaluating test sequences, one would like to employ a timing model where each gate is assigned an independent propagation delay. However, since in effect a circuit has to be resimulated for each fault being considered, employing such a precise model is impractical for minicomputer as well as main-frame implementations. On the other hand, the state-variable Huffman model [11], which makes for economical simulation [1],[4], often lacks sufficient accuracy for asynchronous circuits.

In a unit-gate-delay simulator each gate is assumed to have ideal equal (unit) delay. This timing model is a reasonable approximation to real circuit performance, yet is simple enough to be practical for fault simulation. Furthermore, the model is appealing because it does not require a user to insert artificial feedback-delay elements.

Unit-delay simulation is implemented by maintaining two lists of logic values -- LAST VALUEs and NEW VALUEs. A unit-delay gate is simulated by taking input values from the LAST VALUEs list and storing the computed output value in the NEW VALUEs list.

Thus NEW VALUEs are logic states one gate delay later than LAST VALUEs. After all the gates are processed, "time" is incremented one unit delay by replacing LAST VALUEs by NEW VALUEs.

The accuracy of the simulation model is enhanced without substantially complicating the program by allowing zero-delay elements to represent tied-collector wired-ANDs and wired-ORs. A zero-delay gate is simulated by immediately storing the computed output value in the LAST VALUEs list, thus achieving a zero-delay effect.

At any point in time, logic values are changing at relatively few points in a circuit. Thus for most of the elements in a circuit, at any given time none of the inputs have just changed, and the element need not be simulated [3],[4],[10]. The technique of simulating at any given delay time only those elements with an input that just changed is called "selective trace". Selective trace is implemented most economically using "queues" to list those elements that were just simulated and those that are next to be simulated. (An experimental version of the program that polled the ACTIVITY BIT of each element, rather than employ queues, to determine which to simulate, was unacceptably slow. Changing the algorithm to the one given below reduced run time for a 1000-gate circuit by an order of magnitude.)

The complete selective-trace parallel simulation algorithm using the zero- and unit-gate-delay model is as follows:

(1) Enter the input elements in the CURRENT QUEUE and set the input values as NEW VALUEs. Set the DELAY COUNT to zero.

(2) Null the NEXT QUEUE.

(3) For each element in the CURRENT QUEUE, compare the NEW VALUE and the LAST VALUE. For each fanout element of an element that changed value, if neither the HOLD BIT nor the ACTIVITY BIT is set, add the fanout element to the NEXT QUEUE and set its ACTIVITY BIT.

(4) Is the NEXT QUEUE empty? If yes, go to step 13.

(5) Does the DELAY COUNT equal the OSCILLATION CRITERION? If no, go to step 7.

(6) For each element in the CURRENT QUEUE, do the following: Set the HOLD BIT, exclusive-OR the LAST and NEW VALUEs, and OR the corresponding Y and Z BITs of the result to produce the element's OSCILLATION MASK. Modify the element's NEW VALUE by OR-ing the OSCILLATION MASK to the Z BITs and AND-ing the complement of the OSCILLATION MASK to the Y BITs.

(7) For each element in the CURRENT QUEUE, replace its LAST VALUE by its NEW VALUE. Replace the CURRENT QUEUE by the NEXT QUEUE. Increment the DELAY COUNT.

(8) Does the CURRENT QUEUE contain any zero-delay elements? If no, go to step 10.

(9) Select a zero-delay element from the CURRENT QUEUE. Simulate the element. Remove it from the queue, and reset its ACTIVITY BIT. If its simulated value differs from its LAST VALUE, do the following: For each fanout element, if neither the HOLD BIT nor the ACTIVITY BIT is set, add the fanout element to the CURRENT QUEUE and set its ACTIVITY BIT. Replace the LAST VALUE of the simulated element by the simulated value. Go to step 8.

(10) Simulate each element in the CURRENT QUEUE. Store the simulated values as NEW VALUEs.

(11) Analyze for critical races each flip-flop both of whose elements have their ACTIVITY BIT set.

(12) Reset the ACTIVITY BIT of each element in the CURRENT QUEUE. Go to step 2.

(13) For each element in the CURRENT QUEUE, replace its LAST VALUE by its NEW VALUE. Null the CURRENT QUEUE.

(14) If the DELAY COUNT exceeds the OSCILLATION CRITERION, reset the HOLD BIT of each element.

IMPLICIT FAULT COLLAPSING

Associated with every gate in a circuit is a set of indistinguishable stuck-at faults [3],[12],[13]. For example, if i is an input and p is the output of an AND gate, then there is no test that can distinguish between i stuck-at-0 and p stuck-at-0. As proved in [14], the following algorithm implicitly generates a "collapsed" fault list in which such equivalences are excluded:

(1) For each element with fanin greater than one, include in the list an "input-open-from" fault for each input -- stuck-at-1 for an AND or NAND input and stuck-at-0 for an OR or NOR input.

(2) For each element with fanout not equal to one, include in the list "output stuck-at-0" and "output stuck-at-1".

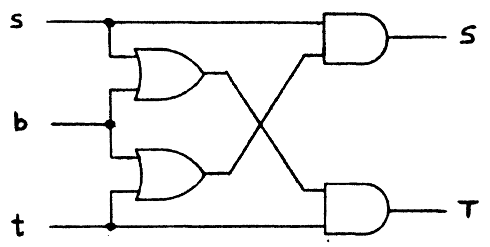
Note that step 2 allows for zero fanout, which occurs for example, when an element drives a primary output only.

FAULT SIMULATION

Fault simulation is initiated with the circuit state either all don't-know's or the same as the fault-free state after some user-specified input vector. An initial state consistent with the faults being simulated is obtained as follows: The faulty elements are entered in the CURRENT QUEUE, and the selective-trace algorithm is entered at step 8. As each faulty element is simulated, its faults are "injected" into the appropriate bits of its NEW VALUE simply by overriding the computed value by the fault-value [1], [2]. Successive vectors are simulated starting at step 1 of the algorithm, and fault injection is performed each time a faulty gate is simulated. Since the simulator is not required to generate a fault matrix [15] or a fault dictionary, simulation for a group of seven faults is terminated as soon as all of those faults have been "detected". Note that a fault is considered detected when at some circuit output the fault-free and fault-present values differ and neither value is don't-know.

SIMULATING SHORT-CIRCUIT FAULTS

To ensure high effectiveness of test sequences, it is desirable to simulate "short-circuit" faults -- pairs of elements unintentionally connected to form wired-AND's or wired-OR's -- as well as stuck-at faults. A short circuit can be simulated as a single stuck-at fault if a battery and four zero-delay gates are added to the circuit description. For an AND-short the modification is as follows:



Here, s and t are the shorted element outputs and b is a battery whose value is 1. The modification includes reconnecting the fanouts of s and t to S and T, respectively. Normally, the AND gates output unaltered values (S=s and T=t), and the circuit behaves as if the modification were not present. However, with b stuck at 0 the AND gates both output the shorted value s·t. An OR-short can be handled in a dual manner.

Since the presence of a short is simulated by a single stuck-at fault, parallel simulation can be employed for N shorts at a time, where N is the number of faults that are simulated simultaneously. In our case, the circuit size increase is a modest 35 elements for each group of seven shorts. To simplify the implementation, after a fault group has been simulated, the current circuit description is discarded, and the modifications for the next fault group are performed on an original description.

PERFORMANCE STATISTICS

For a typical 100-gate circuit the simulation rate for the fault-free circuit or for a group of seven faults is about 800 input patterns per minute. A complete fault simulation to evaluate a test sequence of 700 input patterns for a 90-gate circuit took less than ten minutes -- about five minutes for execution and the remainder for input and output -- and cost $2 based on the amortized rate of $12 per hour.

ACKNOWLEDGMENT

The authors wish to thank George W. Smith and Stephen G. Chappell for their time and effort in carefully reading this manuscript and offering helpful suggestions.

REFERENCES

1. S. Seshu and D. N. Freeman, "The Diagnosis of Asynchronous Sequential Switching Systems," IRE Trans. Elec. Comp., vol. EC-11, no. 4, pp. 459-465, August 1962.

2. F. H. Hardie and R. J. Suhocki, "Design and Use of Fault Simulation for Saturn Computer Design," IEEE Trans. Elec. Comp., vol. EC-16, no. 4, pp. 412-429, August 1967.

3. M. J. Flomenhoft, "A System of Computer Aids for Designing Logic Circuit Tests," Proc. 7th Design Automation Workshop, pp. 128-131, 1970.

4. R. M. McClure, "Fault Simulation of Digital Logic Utilizing a Small Host Machine," Proc. 9th Design Automation Workshop, pp. 104-110, 1972.

5. D. B. Armstrong, "A Deductive Method for Simulating Faults in Logic Circuits," IEEE Trans. Comp., vol. C-21, no. 5, pp. 464-471, May 1972.

6. E. G. Ulrich and T. Baker, "Concurrent Simulation of Nearly Identical Digital Networks," Computer, vol. 7, no. 4, pp. 39-44, April 1974.

7. E. B. Eichelberger, "Hazard Detection in Combinational and Sequential Switching Circuits," IBM J. Res. & Devel., vol. 9, no. 2, pp. 90-99, March 1965.

8. M. A. Breuer, "A Note on Three-Valued Logic Simulation," IEEE Trans. Comp., vol. C-21, no. 4, pp. 399-402, April 1972.

9. L. C. Bening, Jr., "Accurate Simulation of High-Speed Computer Logic," Proc. 6th Design Automation Workshop, pp. 90-93, 1969.

10. S. A. Szygenda, "TEGAS2 -- Anatomy of a General Purpose Test Generation and Simulation System for Digital Logic," Proc. 9th Design Automation Workshop, pp. 116-127, 1972.

11. D. A. Huffman, "The Synthesis of Sequential Switching Circuits," J. Franklin Institute, vol. 257, nos. 3 and 4, pp. 161-190 and 275-303, March and April 1954.

12. J. P. Hayes, "A NAND Model for Fault Diagnosis in Combinational Logic Networks," IEEE Trans. Comp., vol. C-20, no. 12, pp. 1496-1506, December 1971.

13. D. R. Schertz and G. Metze, "A New Representation for Faults in Combinational Digital Circuits," IEEE Trans. Comp., vol. C-21, no. 8, pp. 858-866, August 1972.

14. M. J. Flomenhoft, Algebraic Techniques for Finding Tests for Logical Faults in Digital Circuits, Ph.D. Thesis, Electrical Engineering, Lehigh University, 1973.

15. W. H. Kautz, "Fault Testing and Diagnosis in Combinational Digital Circuits," IEEE Trans. Comp., vol. C-17, no. 4, pp. 352-366, April 1968.

PS LANGUAGE DEFINITION

Portia Isaacson
Xerox Corporation

1. Introduction

The last few years has brought a number of changes to the digital system designer's parts inventory - mainly the addition of large scale integration components such as memories and processors. The solution of an information processing problem involves making the right choices from the variety of components offered and designing the hardware/software interface mechanisms between the components. Methods of simulating this new generation of digital systems are needed as tools. The demands placed on such a tool are great: it must (1) facilitate modeling of a digital system at various levels of architectural detail; (2) allow within a single model two components at quite different specification levels; (3) as a design progresses, allow replacement of models by more detailed models; (4) facilitate modeling of all types of components - hardware, firmware, and software; and (5) have a readily changeable parts library. PS is a tool for simulation of digital systems which meets these demands. In addition PS is designed to ease the problem of communicating hardware/software mechanisms between people by automatically producing pictures of a system in its various states. These pictures can be used as a means of describing the system. The models produced by PS are called picture-system models [1,2,3].

2. Picture-system models and PS

A picture-system model of a computer system consists of (1) a picture set containing a picture for each state of the computer system that is relevant to the mechanism being modeled, (2) a distinguished initial picture corresponding to the initial state of the computer system, and, (3) a transition graph which defines for each picture the set of pictures which may follow it. A simulation on a picture-system model is a movie, or sequence of pictures, which corresponds to a sequence of states in the computer system. All movies can be obtained by starting with the initial picture and choosing the next picture at each step from the choices defined by the transition graph.

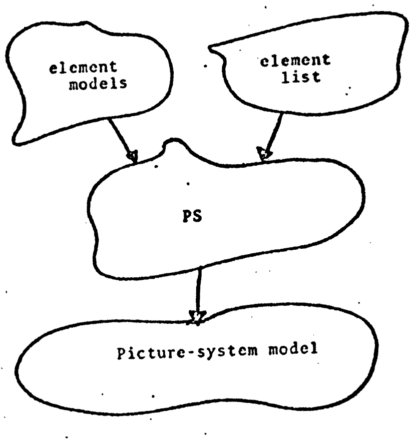


Fig. 1--A black-box view of PS

Figure 1 is a black-box view of PS. The primary inputs to PS are the element list and the element models. The element models define types of components. The element list defines particular instances of the components.

An element can have one of a finite number of states at any time. Each element is described by giving its element model and the elements which are its interfaces. Similar elements, those having the same states and transition rules, reference the same element model. Each element supplies its particular set of interfaces in its reference to the element model. An element model defines the transition rules for an element of its type in terms of the internal state of the element and the states of the interfaces to the element.

This work was conducted while the author was an instructor at North Texas State University and a Ph. D. candidate at Southern Methodist University. Present address: Xerox Corporation, 1341 W. Mockingbird Lane, Dallas, Texas 75247.

Graphics can be specified for each element in the element list and also for each element model. The graphics expressed in the element model can be dependent on the state of the element causing the pictures to vary with element states.

PS output is a picture-system model. PS forms the model by starting with the initial state of the subject computer system, the composition of the element states, and applying all possible element transition rules to determine the set of states reachable by a single element transition from its initial state. This process is repeated for each generated system state. The result is a transition graph which shows all possible transitions during operation of the computer system. Since PS rearranges the transition graph into straight order [11], much information about the computer system is immediately available, such as the identification of deadlocks whether they be single hang-up states or the more subtle cycle from which there is no escape.

Each state reachable by the modeled system has a corresponding picture in the picture-set. A picture is formed by placing in the picture, graphics determined by the states of the elements of the computer system.

3. The PS Language

The PS language is composed of two integrated sublanguages. The computer system description sublanguage is used to define a subject system in terms of its elements, their possible states, and the rules for their changing states. The graphics sublanguage, based on a method of encoding curves described by Freeman [3], is used to express curves which, when placed in the frame, form a picture. These sublanguages are integrated by associating a different curve with each element state. A state of the subject system is the composition of the states of its elements. The picture corresponding to a particular state of the subject system is the one formed by placing in the frame the curves determined by the states of the elements.

Since PS is presently implemented as a set of PL/I [8] macros and support subroutines, its syntax is PL/I determined.

An overview of the language is shown in Figure 2. A PS "program" is a picture-system specification consisting of a frame, global curves, and the element models. The frame section specifies the elements of the subject system, the size of each picture, and any graphics which appears in all pictures as background for the state-dependent graphics. Each element within the frame is given a unique name and described by a reference to the element model of the appropriate type. The reference supplies the actual interfaces of the element, an initial state,

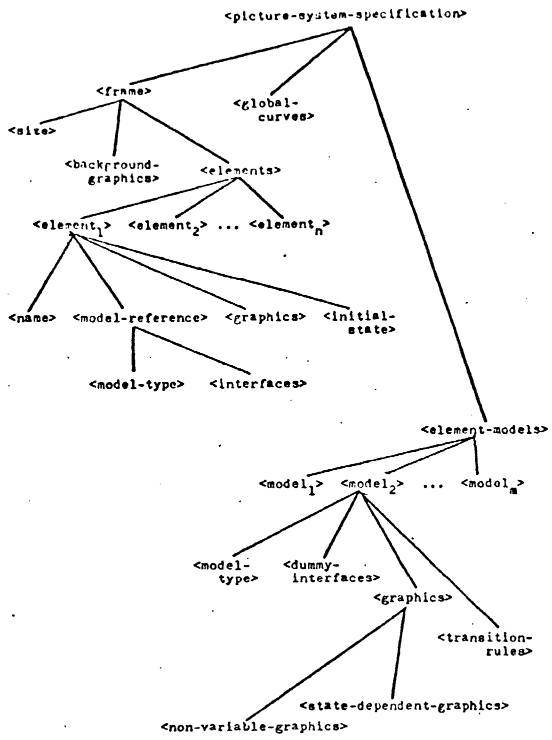


Fig. 2--PS language overview

and graphics that is unique to the element. Each element-model specification consists of the model type, the dummy interfaces, non-variable graphics, state-dependent graphics, and transition rules expressed in terms of the states of the dummy interfaces and the internal state of the dummy element.

The next several sections describe the PS language. Each language construct is described by giving a BNF-like [5] specification, a user-manual-semantic specification and examples. In the BNF-like specification brackets surround options items.

3.1. Picture-system Specification

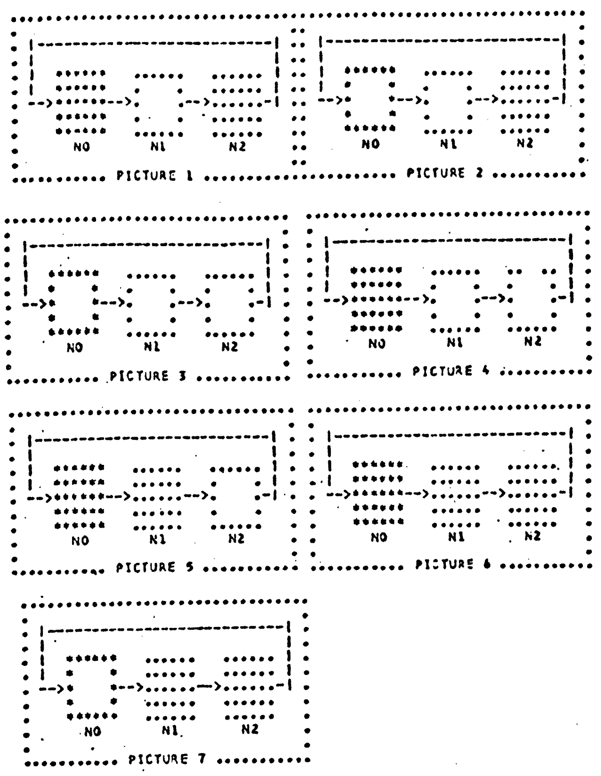
```
<picture-system-specification> ::= <frame>
          [ <global-curves> ]
          <element-model-list>
<global-curves> ::= <named-curve>
          | <global-curves> <named-curve>
<element-model-list> ::= <element-model>
          | <element-model-list>
            <element-model>
```

An example picture system specification is shown in Figure 3. The frame segment of the picture system specification supplies the size of each picture in the model and the list of element which comprise the subject system. In Figure 3 the frame size is 12 rows (vertical units) by 33 columns (horizontal units). The elements are N0 of type CHIEF and N1 and N2 of type NODE. Global curves are curves which can be used by more than one element model. In Figure 3 FILLER and OUTLINE are global curves. The element models are NODE and CHIEF.

```
FRAME( (12,33),
        C('.') AT(1,1) O R(32) D(11) L(32) U(10)
        C('-') AT(7,30) O AT(3,3) R(27)
        C('|') AT(3,3) O D(4) AT(3,31) O D(4)

ELEMENT( NAMED(N0) AT(5, 6) IS_LIKE(CHIEF,(N2) ) TEXT('N0')
        INITIAL(S(ON) ) )
ELEMENT( NAMED(N1) AT(5,15) IS_LIKE(NODE, (N0) ) TEXT('N1')
        INITIAL(S(OFF) ) )
ELEMENT( NAMED(N2) AT(5,24) IS_LIKE(NODE, (N1) ) TEXT('N2')
        INITIAL(S(ON) ) )
)


CURVE(FILLER,       AT(2,2) O R(3)D(2)L(3)U(1)R(2) )
CURVE(OUTLINE,O R(5)D(4)L(5)U(3)C('-')AT(3,-2)O R(1)C('>')R(1) )


MODEL( OF(NODE, (LEFT) )
    RULES(
        WHEN( ¬(LEFT = NODE)  ) SET(NODE = LEFT  )
        )
    GRAPHICS(
        C('.')Z(OUTLINE)
        FOR(S(ON )) DRAW(C('.') Z(FILLER) )
        FOR(S(OFF)) DRAW()
        ,AT(6,3) )


MODEL( OF(CHIEF, (LEFT) )
    RULES(
        WHEN( LEFT =S(ON) & CHIEF =S(ON) )
            SET(CHIEF = S(OFF) )
        WHEN( LEFT =S(OFF) & CHIEF =S(OFF) )
            SET(CHIEF = S(ON) )
        )
    GRAPHICS(
        C('*')Z(OUTLINE)
        FOR(S(ON)) DRAW(C('*') Z(FILLER) )
        FOR(S(OFF)) DRAW()
        ,AT(6,3) )
```

Fig. 3--A PS specification

Figure 3 is a PS specification of a computer system consisting of three elements which are nodes in a three-node self-stabilizing network [6]. If a node can change states, it is privileged. In a stabilized system only one node is privileged at a time. A node can test only its left neighbor's state and its own state when deciding whether or not to change states. There is no central clock. The initial states chosen for the nodes start the system in an unstable state as shown in picture 1 of Figure 4. All three nodes are privileged. The output of the program is the transition list and the picture set shown in Figure 4. Each picture is shown with a picture number. The transition list is in terms of the picture numbers. For example, the transition list shows that pictures number 6, 4 or 2 can follow picture number 1. A movie can be constructed from the picture set by starting with picture number 1 and following it with any sequence of pictures permitted by the transitions. One such



```
1 -->  6  OR  4 OR   2

 HEAD--<-------------------------+
2 -->  3                         |
3 -->  4                         |
4 -->  5                         |
5 -->  6                         |
6 -->  7                         |
7 -->  2                         |
 TAIL-->-------------------------+
```

Fig. 4--PS generated transition list and picture set

movie is

1 to 4 to 5 to 6 to 7 to 2 to 3 to 4 to ...

Clearly, for this initial state the network stabilizes after a single transition in all cases.

3.2. Frame

```
<frame> ::= FRAME( <rows> , <columns> ) ,
          [ <background-curve> ]
          <element-list> )
<element-list> ::= <element-specification>
          | <element-list>
               <element-specification>
<rows> ::= <integer>
<columns> ::= <integer>
```

```
<element-specification> ::= ELEMENT(
     NAMED( <element-name> )
              [ <pre-model-curve> |
              <element-model-reference>
              | <post-model-curve> ]
              [ INITIAL( <element-state> ) ]
                    )
<element-name >::= <name>
<element-model-reference> ::= IS_LIKE(
                         <model-type>
              [ , ( <model-parameter-list>) ]
                         )
<interface-list> ::= <interface>
              | <interface-list> <interface>
<interface> ::= <element-name>
                         | <element-state>
<pre-model-curve>::= <curve>
<post-model-curve> ::= <curve>
<element-state> ::= S( <name> )
```

Rows and columns are integers which define the size of each picture in the picture set. Rows is the vertical dimension of the picture and column is the horizontal dimension. Background curve is optional. If given, it is placed first in every picture in the picture set, before any of the variable element graphics are placed in the picture. Element graphics may overwrite the background curve.

The element description has a number of options. An example minimum specification is

```
   ELEMENT( NAMED(T) IS_LIKE(CLOCK) ) .
```

A unique element name must be given; in this case it is T. The element model reference must be given; in this case the referenced element model is CLOCK. Since neither a pre-model curve nor a post-model curve is given, the element model CLOCK should not contain graphics -- the graphics within the element model are relative to the frame location set by the pre-model curve. For example, the element description

```
ELEMENT( NAMED(N7) AT(3,8) IS_LIKE( NAND,
                    (A,B,C) ) )
```

sets frame location (3,8) in the pre-model curve so that this instance of a NAND will be drawn at that location. A post-model curve only, is convenient when the shape of the curve depends on the element, although the character with which the curve is drawn depends on the state of the element and is therefore determined in the element model. The element description

```
  ELEMENT( NAMED(A) IS_LIKE( LINE )
                    R(3)U(4)R(6) )
```

is of a LINE that is drawn with either a 0 or a 1 depending on the state of the line. The shape of the line is specified by the post-model curve.

The element model reference defines the type of the element and determines its interfaces. Only the state of the element and its interfaces can be used to change the element state as specified in the rules section of the element model. In other words, if the state of element A depends on the state of element B, element B must be specified as an interface to element A in A's element description. Such an element description might take the form

```
  ELEMENT( NAMED(A) IS_LIKE(THING,(B)) )  .
```

An element need not have an internal state. For example, the element description

```
ELEMENT( NAMED(N3) AT(14,2)
         IS_LIKE(NAND,(A,B,C)))
```

describes a NAND gate. The gate has three interfaces. A and B are input lines and C is an output line. The rules in NAND specify state changes in C in terms of the states of A and B.

An element state can be supplied as an interface to an element model in order to parameterize the element model. For example,

```
ELEMENT( NAMED(READER) AT(50,20)
    IS_LIKE(TASK,(S(READ)))
ELEMENT( NAMED(WRITER) AT(10,1)
    IS_LIKE(TASK,(S(WRITE)))
```

describes two TASKs that differ only in their frame locations and the operation that the task performs - READ for one, WRITE for the other.

An element may be given an initial state. In the element description

```
ELEMENT( NAMED(E) IS_LIKE( LINE )
                      INITIAL(S(LOW)))
```

the LINE is given the initial state LOW. Only those elements having no internal state, such as the NAND described earlier, need not have an initial state specified.

3.3. Element-model specification

```
<element-model> ::= MODEL( OF( <model-type>
          [ , <dummy-interface-list> ] )
          <graphics> <rules> )
<dummy-interface-list> ::= <dummy-interface>
          | <dummy-interface-list>
               <dummy-interface>
<model-type >::= <name>
```

Each element model specifies the model type, the transition rules and the graphics associated with an element of its type. The model type is used in the rules and graphics parts of the element model to refer to the internal state of an element defined as the model type. If an element of model type has interfaces, a dummy interface list is supplied. The actual interface list is supplied in an element description referencing the element model. The element model for LINE in Figure 5 has no interfaces; however, the element model for NAND in Figure 6 specifies three interfaces.

```
MODEL( OF(LINE)
    RULES()
    GRAPHICS(
        FOR(S(HIGH)) DRAW(C('1'))
        FOR(S(LOW )) DRAW(C('0'))
        )
    )
```

Fig. 5--Element model of LINE

```
MODEL( OF(NAND,(IL1,IL2,OL1))
    RULES(
        WHEN(IL1 = S(LOW) | IL2 = S(LOW) )
         SET(OL1 = S(HIGH) )
        WHEN(IL1 = S(HIGH) & IL2 = S(HIGH) )
         SET(OL1 = S(LOW ) )
        )
    GRAPHICS(
        AT( 1, 1) TEXT('---------')
        AT( 5, 1) TEXT('---------')
        C('|')
        AT( 1, 1) D(3) AT( 1, 8) D(3)
        AT( 3, 3) TEXT('NAND')
        )
    )
```

Fig. 6--Element model of NAND

3.3.1. Rules

```
<rules> ::= RULES( [ <rule-list> ] )
<rule-list> ::= <rule>
              | <rule-list>,<rule>
<rule> ::= WHEN( <state-conditional> )
           SET( <state-assignments> )
         | WHEN( <state-conditional> )
           APPLY( [ <rule-list> ] )
<state-conditional> ::= <state-term>
         | <state-conditional> <state-term>
<state-term> .:= <state-factor>
         | <state-term> & <state-factor>
<state-factor> ::= <state-operand>
         | <state-factor> "|"
              <state-operand>
    Note: the quotes are meta characters
          denoting that the bar inside
          them is not.
<state-operand> ::= <state-test>
                    ( <state-conditional> )
<state-test> ::= <element-reference>=
                 <state-reference>
               | <element-reference>¬=
                 <state-reference>
<element-reference> ::= <model-type>
                    <dummy-interface>
<state-assignments> ::= <state-assignment>
         | <state-assignments> ;
                          <state-assignment>
<state-assignment> ::= <element-reference>
                 = <state-reference>
<state-reference> ::= <element-reference>
                    | <element-state>
<element-state> ::= S( name )
```

Rules must be specified although they need not contain a rule. The element model for LINE in Figure 5 has no rule. Within a rule list each rule is independent of any others. The order in which they are specified has no effect on the picture-system model to be generated. A rule has two parts - the when part and the set or apply part.

The when part specifies a true/false-valued expression in terms of model type, dummy interfaces, element states, parenthesis, and PL/I logical operators. For each system state the when part is evaluated. The resulting true or false value determines whether or not the following set or apply parts may cause a change in the system state and therefore a transition.

The set part specifies changes to the states of model type, the internal state of the actual element, and interfaces that are to be made in any system state for which the when prefix is true. The set is specified as a sequence of assignments of states to elements. The rule

```
WHEN( IL1 = S(HIGH) & IL3 = S(HIGH))
      SET( OL1 = S(LOW))
```

in Figure 6 tests all system states to determine if any element of type NAND has interfaces IL1 and IL2 both at state HIGH. For any system state for which this when part is true, a transition to a new system state is recorded for the change in interface OL1's state to LOW. The rule

```
WHEN( LEFT = NODE ) SET( NODE = LEFT )
```

from Figure 3 tests all system states to determine whether or not the state of any element of type NODE is the same as the state of its interface LEFT. For any system states for which the when part is true, a transition to a new system state is recorded. The new state differs from the last only in that the state of the element of type NODE has been changed to the state of its interface LEFT.

An apply part specifies an entire rule list that is to be considered only when its when prefix is true. It is simply a method of factoring out an "anded" term from a set of whens. The following example codes are equivalent:

```
Example code 1
   WHEN(A = S(1) & B = S(2)) SET(B=A)
   WHEN(A = S(1) & B = S(#)) SET(B=S(5))

Example code 2
   WHEN(A = S(1)).APPLY(
         WHEN(B = S(2)) SET(B=A)
         WHEN(B = S(#)) SET(B=S(5))  )
```

3.2.2. Graphics

```
<graphics> ::= GRAPHICS(
          [ <pre-for-list-curve> ]
          [ <for-list>           ]
          [  , <post-for-list-curve> ]  )
<pre-for-list-curve> ::= <curve>
<post-for-list-curve> ::=<curve>
<for-list> ::= <for-list-entry>
          | <for-list> <for-list-entry>
<for-list-entry> ::= FOR( <element-state> )
               DRAW( <curves>)
             | <curve>
```

Graphics must be specified although the body of the graphics specification may be empty. All three parts of a graphics specification are optional. The forlist can specify a different curve for each state of an element of the model type.

All curves in the element model are relative to the place in the frame set by the pre-model curve of each element referencing the element model. This allows drawings associated with different elements of the same type to have the same graphics but at different places in the frame.

The pre-for-list curve is placed in the frame first; then the curves selected by the for-list; and last the post-for-list curve. The graphics specification

```
   GRAPHICS
     C('.') Z(OUTLINE)
     FOR( S(ON)) DRAW(C('.') Z(FILLER))
     FOR( S(OFF))DRAW()
     , AT(6,3) )
```

from Figure 3 has all three parts of the graphics specification. The pre-for-list curve draws the outline for a NODE by referencing the global curve OUTLINE. The for-list causes the outline to be filled in for ON elements and to be left empty for OFF elements. The post-for-list curve is done in either case. It causes the frame location to be set to (6,3) relative to the location on entry to the model. This location is set so that the post model curve in the element description can place the name of the particular element below the drawing.

3.4. Curves

```
<curve> ::= <curve-order>
           |<curve    curve-order>
           |null
<curve-order> ::= O
                | R( <integer> )
                | L( <integer> )
                | U( <integer> )
                | D( <integer> )
               | UR( <integer> )
               | UL( <integer> )
               | DR( <integer> )
               | DL( <integer> )
               | AT( <signed-integer> ,
                     <signed-integer> )
                | C( <quoted-string> )
                | C( <c-name> )
             | TEXT( <quoted-string> )
               | ST( <name> )
               | <named-curve>
               | Z( <curve-name> )
<c-name> ::= <element-name> <model-name>
<named-curve> ::= CURVE( <curve-name> ,
                         <curve> )
```

The picture associated with a particular system state is formed by executing all curve orders in the frame specification and in the element models referenced in the frame specification. The order in which the curve orders are executed is important to the appearance of a picture. For example, a curve order may determine the location of the following curve, or the characters used in drawing the following curve. The curve orders are executed in the order in which they are written except that for each element description the curve orders determined by the referenced element model are executed after the pre-model curve in the element description and before the post-model curve.

3.4.1. AT curve order

The AT curve order means to set the current location to the specified row and column relative to (1,1) in the upper left corner of the frame or relative to the location at which the element model was entered. When an element model is entered the location at the time it was entered becomes (1,1) for graphics inside the model. When the model is exited, the current location is relocated to (1,1) in the frame. Figure 7 shows the current location in the frame set by the curve order AT (1,1).

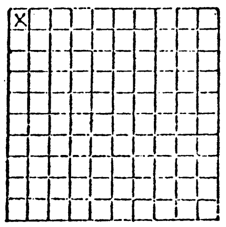


Fig. 7--Example 1 of use of AT curve order.

Figure 8 shows the current location in the frame set by the curve order AT(3,5).

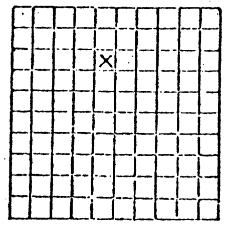


Fig. 8--Example 2 of use of AT curve order.

Figure 9 shows the current location in the frame set by the curve order AT(1,1) followed by an element description followed by the curve order AT(3,5) in the element model.

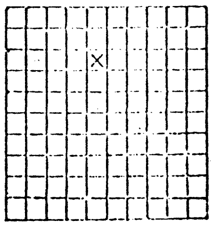


Fig. 9--Example 3 of use of AT curve order in element model

Figure 10 shows the current location in the frame set by the curve order AT(3,5) followed by the element description followed by the curve order AT(2,3) in the element model.

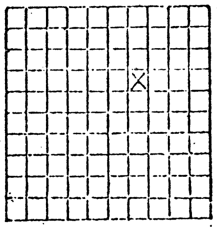


Fig. 10--Example 4 of use of AT curve order in element model.

Figure 11 shows the current location in the frame set by the curve order AT(7,6) followed by an element description followed by the curve order AT(-1,-2) in the element model.

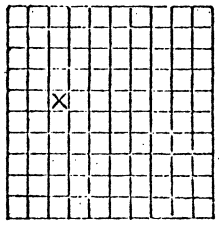


Fig. 11--Example 5 of use of AT curve order in element model

3.4.2. C Curve Order

The C curve order means to set the source of current characters to the value specified in the quoted string or the current value of c-name. The current character becomes the leftmost character of the string. The characters are used one at a time moving right until the string is exhausted. At that time the leftmost character is used again, and so on. Examples of the C curve order are

```
C('.')
C('ALPHA')
C('0')
C('1,234,782')
C('-')
c( ELE )
```

3.4.3. O, R, L, U, D, UR, UL, DR, and DL

The O curve order means to place the current character at the current location and then update the current character to the next character in the string modulo the string length. The current location is not changed.

Curve orders R (right), L (left), U (up), D (down), UR (up-right), UL (up-left), DR (down-right), and DL (down-left) all have the same algorithm except for the direction of movement. The integer in each case is the number of directional movements to be made. At each new location during the movement a character is placed and the current character is updated to the next character in the string modulo the string length. Notice that no character is placed at the initial current location and that the current location at the completion of the movement is still at the last character placed.

Figure 12 shows the curve

AT(1,1) C('X') R(2) D(7) .

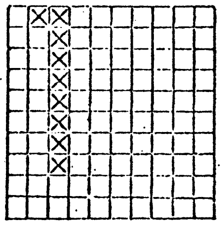


Fig. 12--Example 1 of curve movements

Figure 13 shows the curve

AT(3,2) C('*-') O R(5) .

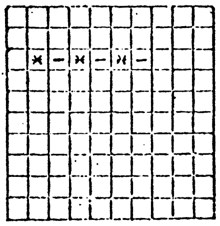


Fig. 13--Example 2 of curve movements

Figure 14 shows the curve

```
AT(1,1) C('*') DR(3) R(2) UR(2) D(4)
  DR(2) D(1) DL(1) L(5) C('-') UL(3)
     AT(7,5) O  .
```

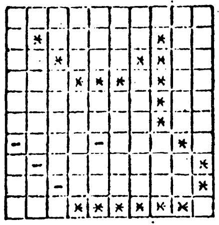


Fig. 14--Example 3 of curve movements

3.4.4. TEXT Curve Order

The TEXT curve order means to place the quoted string in the frame starting at the current location and moving to the right. Figure 15 shows the result of placing the curve

```
AT(3,3) TEXT('ALPHA')
```

in the frame.

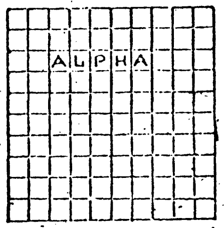


Fig. 15--Example of use of TEXT curve order

3.4.5. ST Curve Order

The ST curve order means that the state of the named element or model is placed in the frame starting at the current location and moving to the right. Figure 16 shows the result of placing the curve

```
AT(4,3) ST( SWITCH )
```

in the frame if the state of element SWITCH is OFF.

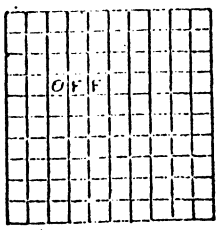


Fig. 16--Example of use of ST curve order

3.4.6. CURVE and Z Curve Orders

The CURVE curve order names a curve so that it can be referenced from several different places by using the Z curve order with the curve name. The effect of the Z curve order is to cause the curve to be placed at the current location. Figure 17 shows the result of placing the curve

```
CURVE( BOX, RN3) D(3) L(3) U(3) )
  C('*') AT(1,1) Z(BOX)
  C('.') AT(6,1) Z(BOX)
```

in the frame.

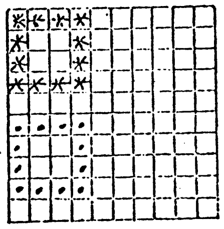


Fig. 17--Example of use of CURVE and Z curve orders.

4. <u>Conclusion</u>

We have presented picture-system models as a simulation tool for the digital system designer. The generation and analysis of picture-system models has been automated by the PS language implementation. This paper has described the PL/I-based PS language.

A wide variety of models have been developed using the prototype implementation of PS. Models of traditional digital logic components at gate level as well as functional levels have been developed [9]. These models will be a powerful teaching aid. PS digital logic models reveal races and hazards as would be expected from an asynchronous simulator. The development of digital logic models has motivated, in part, the exploration of synchronous PS models in addition to the asynchronous models which were the original subjects of our investigation.

At the high end of the architectural spectrum, a model of a channel-to-channel computer interface mechanism involving both hardware and software has been developed [3]. This very large model, more than 1,000 states, has been the subject of a study of the computer system design methodology that we are developing around PS. The idea of submodels of picture-system models has emerged as an important analysis method. Computer systems can be explored for hang-ups, inescapable cycles, races, and hazards. Fault studies have been done using the channel-to-channel model to study the potential of PS for such studies. The effect of an errant operating system in one computer, on the operating system in another computer communicating with it, has been explored as a fault study. The interface unit in the channel-to-channel connection has been subjected to fault studies which predicted its reaction to any sequence of channel signals from the two computers no matter how nonsensical. The potential for diagnostic capability based on this type of study is clear. A model of a channel-to-channel interface mechanism somewhat simplier than the one described in [3] has been used during the actual design of the interface [2]. The model had a very favorable effect on the communication between the hardware and software groups involved, in addition to pinpointing oversights which would have led to interface unit hang-ups.

Another interesting set of models that are being developed is Dijkstra's cooperating sequential processes [7]. These models give this author hope for proofs of operating systems done by "ordinary" operating system designers assisted by PS models. These proofs will be based on the structure of submodels of system behavior as defined by the transition graph produced by the PS system. The analysis of these transition graphs is one of the most useful features in PS that is not found in traditional digital system simulators. The current analysis is by ordering of nodes as described by Earnest [11] and applied by Korfhage [12]. The many other possibilities for transition graph analysis are one subject of our current investigations.

The goal of this continuing research is to incorporate within a single automated design aid the features necessary to simulate digital systems at various levels of architectural detail, in such a way that we can see (1) that it works and (2) how it works. The prototype implementation of PS has shown the feasibility and usefulness of such a system.

Acknowledgements

The author gratefully acknowledges helpful suggestions from Charles Biggs, Dennis Frailey, James Isaacson, Bob Korfhage, Ahmet Oner, Bill Nylin, Dan Scott, and Rob Smith.

References

1. P. Isaacson, "Picture-system models and computer system design," Ph.D. dissertation in preparation, Computer Science Department, Institute of Technology, Southern Methodist University, Dallas, Texas.

2. P. Isaacson, "PS: A tool for building picture-system models of computer systems," to appear, June, 1974.

3. P. Isaacson, "Picture systems, PS, and the design of a channel-to-channel computer interface," to appear, July 1974.

4. H. Freeman, "On the encoding of arbitrary geometric configurations," IRE Transactions on Electronic Computers, June 1961, 260-268.

5. P. Naur, (Ed.), "Revised report on the algorithmic language ALGOL 60," CACM 6, (Jan 1963), 1-17.

6. E. Dijkstra, Lectures at a conference on programming methodology, University of New Mexico, March 1974.

7. E. Dijkstra, "Co-operating sequential processes," Programming Languages, . Genuys, Ed., Academic Press, New York, 1968.

8. IBM, PL/I Language Reference Manual.

9. C. Biggs, "Picture-system models of digital logic elements," Master's thesis in preparation, Computer Sciences Department, North Texas State University, Denton, Texas.

10. P. Isaacson and A. Oner, "Picture-system models of Dijkstra's co-operating sequential processes," in preparation.

11. C.P. Earnest, et al, "Analysis of graphs by ordering of nodes," JACM, January 1972, 23-42.

12. R. Korfhage, "Program restructuring: garbage in, daisies out," to appear, June 1974.

A PARTITIONING TECHNIQUE FOR LSI CHIPS

Pao-Tsin Wang

International Business Machines Corporation
System Development Division

9500 Godwin Dr.

Manassas, Va. 22110

I. Introduction

A technique for partitioning LSI chips is presented in this paper. This technique is a refinement of the author's previous work [1].

The word "partitioning" is taken to mean the dividing of a chip into n sections. Each section will contain a group of circuits. The size of a section is defined as the amount of physical area occupied by the group of circuits. Each section does not have to be equal in size, however, the difference in size between any two sections should be within some prespecified number. Since the physical dimension of each circuit is not uniform, the number of circuits in a section may vary a great deal from section to section. A good partition is one such that each section has a valid size and the number of interconnections between any two sections is the smallest. In essence, one should attempt to achieve two goals in the process of partitioning: maintain a valid size for each section and reduce the number of intersection connections as much as possible.

The partitioning technique presented in this paper consists of the following steps. First, the set of circuits is ordered according to a scoring mechanism and the resulting order is called the initial order of the circuits. Next, an interchange technique is used to improve that initial order and the resulting order is called the improved order of the circuits. Last, the same interchange technique is used to partition the improved order into n sections.

II. Construction of the Initial Order

On a chip, the set of circuits is interconnected by a set of nets. A net is defined as a collection of electrically common points, where a point may be either an input gate or an output gate of a circuit. A circuit, for example, could be a 4-input-1-output NAND. Since a circuit, in general, consists of more than one gate, there is a set of nets in which the circuit is involved. This set of nets is called the associated nets for the circuit. On the other hand, a net connects a group of circuits which will now be called the group of circuits in the net.

An iterative ordering algorithm is used to establish the initial order in which one circuit follows the other. An iteration is a pass where a circuit is selected and ordered. There are as many iterations as there are circuits. To begin the algorithm, the circuit with the smallest number of associated nets is chosen as the starting circuit. This circuit now becomes an ordered circuit. At the end of every iteration, a set of candidate circuits (unordered, of course) will be constructed from the set of nets associated with the group of ordered circuits. To select for ordering the best circuit among the candidate circuits, a scoring mechanism is used. The iteration repeats until all the circuits have been ordered.

In essence, the scoring mechanism calculates a score for each candidate circuit and selects the circuit with the highest score. A net is said to be complete when all the circuits in it have been ordered; otherwise, the net is said to be incomplete. One obvious criterion in constructing the scoring mechanism is to give the highest score to a circuit that has the potential of completing the largest number of nets while, at the same time, including relatively small number of incomplete nets. Before the complete scoring mechanism is presented, some items should be defined as follows:

X	=	a candidate circuit after the ith iteration
NET(X)	=	the set of associated nets for the circuit X
CKT(X)	=	the set of distinct circuits derived from NET(X)
#CKT(X)	=	the total number of circuits in CKT(X)
SCKT(X)	=	a subset of CKT(X), representing the circuits that have been ordered
#SCKT(X)	=	the total number of circuits in SCKT(X)
NT(n)	=	the set of connected circuits in the net n
#NT(n)	=	the number of circuits in the set NT(n)
SNT(n)	=	a subset of NT(n), representing the circuits that have been ordered
#SNT(n)	=	the total number of circuits in SNT(N)
C	=	the number of common nets between the circuit X and the set of ordered circuits
Z	=	the number of nets in the set NET(X)
N	=	the number of new and incomplete nets brought in by the circuit X; N=S-C
D	=	degree of completeness, representing a measurement on the potential of the circuit X to complete its associated nets. There are two forms used to define D:

• Sum Form — $D = \frac{\#SCKT(X)+1}{\#CKT(X)}$

° Product Form — D =

$$\prod_{n \in NET(X)} \frac{\#SNT(n)+1}{\#NT(n)}$$

I = the total number of incomplete nets after the ith iteration

IC = the total number of common nets between the circuit X and the particular circuit ordered in the ith iteration

The score S for the candidate circuit X is now defined as:

$$S = \frac{(C)^2 x D x IC}{2x(I+N)}$$

Note that since there are two forms for computing D and that IC could be ignored, these are four possible scoring mechanisms, any one of which could be used for the ordering algorithm.

1. Score with D in product form and ignore IC.
2. Score with D in sum for and ignore IC.
3. Score with D in sum form and include IC.
4. Score with D in product form and include IC.

Since the score S is calculated in a heuristic manner, it is difficult to predict which scoring mechanism would produce a better result for a given chip. Experimental runs have been made on many chips and the results indicated the scoring mechanism 1, in general, produced relatively better circuit orders; hence, it was chosen for the iterative ordering algorithm.

III. Construction of the Improved Circuit Order

The initial circuit order can be described graphically, as shown in Figure 1, where the order is to be read from top to bottom. For example, net A connects circuits 78, 79 and 73, and net B, circuits 90, 88, 89 and 86. Gates are treated as if they were equal in width in order to simplify the work of displaying nets. Imaginary vertical channels are assumed, each of which is to be filled successively with nets. If a channel is full or no net fits, a new channel begins. The total number of channels required to display all the nets is a rough measurement on the "goodness" of a given initial circuit order; in other words, the smaller the number of channels required, the better the initial circuit order.

To improve the initial order, an interchange technique is required, derived from [2], and will be referred to as the K-L interchange algorithm. The strength of the K-L algorithm is its capability in identifying groups of circuits to be interchanged; furthermore, the algorithm decides when the interchange process should stop, and thus provides a dynamic stopping point. The existence of such a dynamic stopping point resolves the general problem as to when the interchange of objects ought to stop, as compared to other interchange techniques that could be employed. Due to the fact that a net, in general, connects more than two points, it was necessary to modify the K-L algorithm so that nets of more than two points would not cause the algorithm to produce misleading results. This modification was used in [1], and was also discussed in a recent paper [3]. It is assumed that the readers are familiar with [2] and hence, no detailed description of the K-L algorithm and its modifications will be presented.

The improvement of the initial circuit order is accomplished on a step-by-step basis. Refer to Figure 1. When an imaginary dividing line is drawn on Location 1, the set of circuits is split into two subsets. Application of the K-L algorithm on these two subsets results in identifying two groups of circuits to be interchanged, one group originating from the circuits above the dividing line, the other below. There is no order consideration for each circuit within each of the two groups. After the interchange takes place, one group of circuits will be placed immediately above the dividing line and pushes the remaining circuits above this line upward; the other group of circuits will be inserted below the dividing line and pushes the rest of the circuits below the line downward. As a result, the initial order has been altered; however, the number of interconnections across the dividing line was reduced. Since the goal at the time is to improve the initial circuit order, no consideration is given to the physical dimension of circuits during the process of interchange.

The dividing line now moves down to a new location, say location 2, and again the K-L algorithm is applied, resulting in a reduction of the interconnections across the current dividing line. When there are no more new locations available, the dividing line stops moving downward and starts moving up, generating new locations where the K-L algorithm will be applied. For convenience, the one round of moving the dividing line down and up again (or vice versa) is called a pass. Within a pass, the number of circuits which the line jumps over when moving

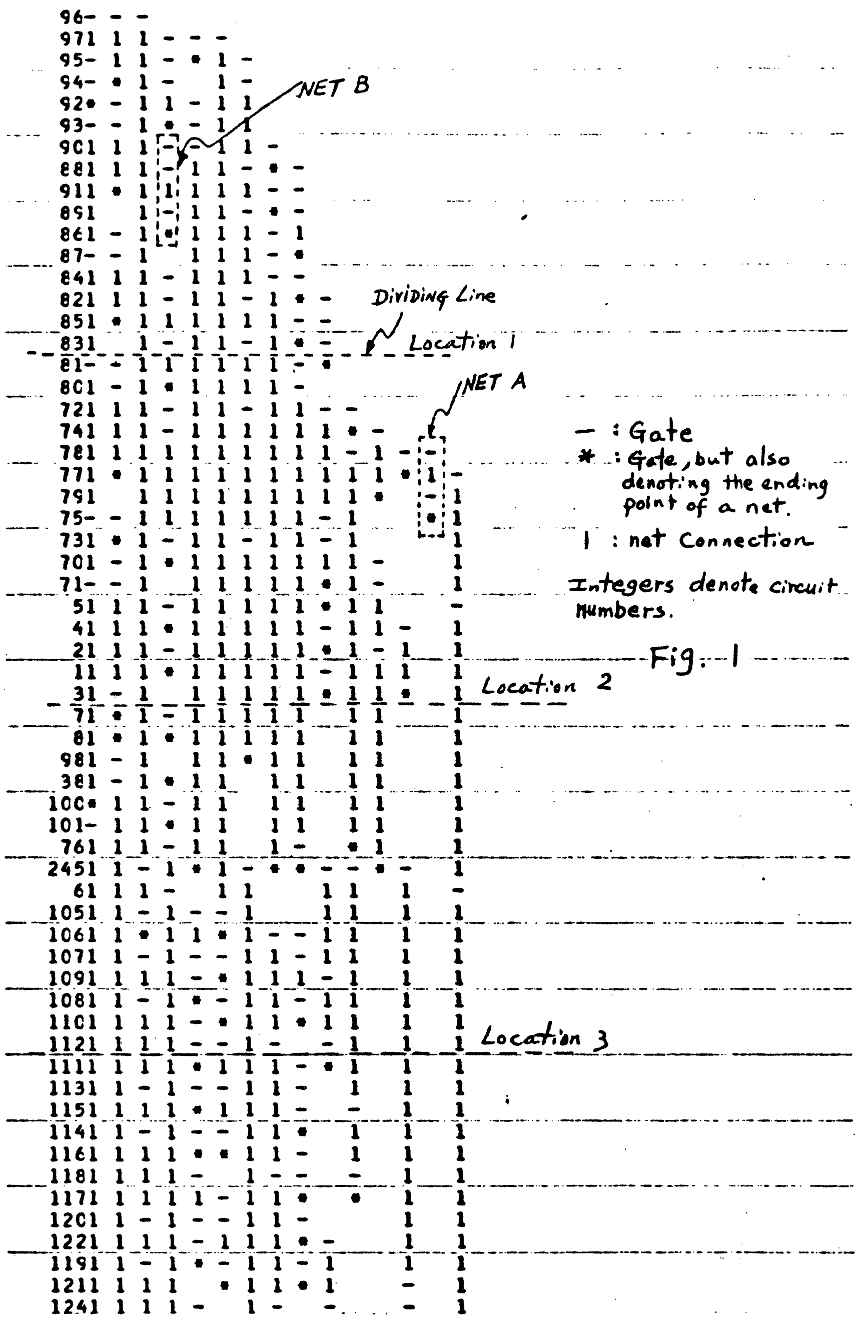

NET B
Dividing Line
Location 1
NET A
— : Gate
* : Gate, but also denoting the ending point of a net.
| : net Connection
Integers denote circuit numbers.
Location 2
Location 3

Fig. 1

from one location to another is called the step size. After applying many passes of line-moving, the initial circuit order will be greatly improved, which will now be called the improved circuit order.

The number of passes and the step size obviously has a great deal to do with the degree of improvement on the initial order; however, it is difficult to determine in advance how many passes would be enough and what step size would be proper for a given chip. For the program the author developed, it was decided empirically that the number of passes was 14 and the step size was as follows:

Step Size	Number of Circuits
5	$K < 100$
9	$150 > K > 100$
13	$200 > K > 150$

Step Size	Number of Circuits K
17	$300 > K > 200$
25	$K > 300$

IV. Partitioning of Chips

Partitioning of chips is accomplished by applying the K-L algorithm on the improved circuit order. Since a proper size is to be maintained for each section, the size of the circuits can no longer be ignored during the process of interchanging circuits. In this paper, the size of a circuit is taken to mean the height of the circuit; hence, the size of a section is defined as the sum of the heights of the circuits that are contained in the section.

There are in general, two major steps in partitioning a chip: the initial partitioning and the succeeding partitioning. Two large sections of circuits are constructed after the initial partitioning is applied; the succeeding partitioning divides each of the two large sections into a number of smaller sections until there are n sections totally where n is the number of sections required.

A. Initial Partitioning

- Case 1: n is even

The chip size is defined as the sum of the heights of all the circuits that the chip contains. To obtain a proper size for each of the two large sections to be constructed, the chip size is divided by two. Let S denote this size. The balance index is set equal to S/20; in other words, one section, at most, can only be 10% larger than the other section.

On the improved circuit order, a dividing line is to be drawn at the place where two sections of circuits can be identified with the size of one section close to the other section. When the K-L algorithm is applied, a series of interchanges of circuits takes place; the size difference between the two sections will be monitored against the balance index at each interchange of circuits. At the end of the algorithm, two large sections of circuits with proper size are constructed.

- Case 2: n is odd

As noted in the previous discussion, there is only one way to draw the dividing line on the improved circuit order and identify two sections of circuits when n is even; however, there are two ways to construct two sections of circuits when n is odd. For example, let n be 5 to illustrate the case.

The approximate size for each of the five sections is set equal to one-fifth of the chip size. Let S denote this size. For the two large sections to be identified, one section will take the size of 2S, the other 3S. The balance index is $\frac{2S}{20}$. Refer to Figure 1. A line drawn at location 1 could identify two large sections; on the other hand, a line drawn at location 3 would also identify two large sections. After applying the K-L algorithm on both configurations, the one with the smallest number of interconnections will be chosen.

B. Succeeding Partitioning

The succeeding partitioning is simply the extension of the initial partitioning; the only difference is that each of the two large sections of circuits is treated as if they were complete chips, hence, the initial partitioning is applied. In essence, n sections of circuits are constructed by repeatedly applying the initial partitioning.

The following table provides some computational results.

CHIP ID	NO. OF CKTS.	NO. OF NETS	NO. OF SECTIONS	SECTION SIZE (IN UNITS)									NO. OF INTERCONNECTIONS BETWEEN TWO ADJACENT SECTIONS								CPU TIME (SEC.)
				1	2	3	4	5	6	7	8	9	1-2	2-3	3-4	4-5	5-6	6-7	7-8	8-9	
1	514	511	6	362	369	359	325	358	354				24	22	19	16	15				76
2	620	470	5	448	440	422	338	402					50	71	60	50					160
3	1135	699	9	516	515	490	483	490	520	529	530	537	48	73	87	91	66	78	74	58	202

TABLE 1 EXPERIMENTAL RESULTS

* NOTE The program was run on an IBM System/360 Mod. 195
The Core required is 185K

V. Discussions

Since the partitioning algorithm presented in this paper is heuristic in nature, many decision-making mechanisms are constructed empirically. As such, it is difficult to assess a partitioning result produced by the algorithm.

In the course of constructing the initial circuit order, a certain circuit is chosen as the starting circuit. It appears that the selection of the starting circuit has a great deal to do with the final partition result. The application of the line-moving approach is used not only to improve the initial order, but also to diminish the impact of the starting circuit on the final result; nevertheless, the impact still remains, although to a lesser degree.

There are four scoring mechanisms presented in this paper. Many other scoring mechanisms can be found in [4]. At one time, all four scoring mechanisms were used to construct the initial order and comparisons were made on the four final partition results; however, no conclusive evidence was indicated as to which one was superior to the other three.

Number of passes and step size are the other two subjects to be further studied. The values used in this paper were selected on the basis of experimental runs.

It should be noted that no section is truly independent of the remaining sections because there are always some interconnecting nets which tend to pull circuits in one section to circuits in other sections. Effort should be made to reduce the number of the interconnections which cross many section boundaries. From a practical point of view, these interconnections could very well mean trouble during wiring. On occasion certain circuits may have to be assigned to certain sections; this pre-assignment of circuits can also be implemented in the partitioning algorithm. Due to the space limitation, no detailed discussions pertaining to these special features will be presented.

Acknowledgments

The author wishes to thank Mr. John Fischer for reviewing the manuscript and Mr. David Gontner for providing programming effort.

References

1. Unpublished Work, 1972.

2. B. W. Kernigham and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs", The Bell System Technical Journal, February 1970.

3. D. G. Schweikert and B. W. Kernighan, "A Proper Model for the Partitioning of Electrical Circuits", Proceedings of the ACM-IEEE Design Automation Workshop, June 1972.

4. M. Hanan and J. M. Kurtzberg, "Placement Techniques", IBM Technical Report RC-2846, April 1970.

ACM 74 SAN DIEGO

ASSOCIATION FOR COMPUTING MACHINERY

DYNAMIC, PERTINENT, PARTICIPATIVE
In a delightful Pacific setting

ANNUAL MEETING NOV. 11 THROUGH 13

This will be an ACTIVE meeting. Come to learn, to discuss, to explore. ACM SIG's and SIC's have already begun organizing sessions and refereeing papers. Chances are we've already arranged something pertinent and practical in more than one of your fields of special interest.

There'll be seminars, tutorials, specialized technical and commercial events. We'll meet in superb facilities where it is easy to run into those you want to see. In November San Diego will be the focal point for meeting with colleagues of stature from the California and International computing communities. San Diego is home ground for major California universities and important oceanic, medical, nuclear and aerospace research. Through ACM '74 you can establish contact with these institutions and others specializing in computer application to social problems, computers in education and hardware manufacturing.

ACM '74's published proceedings will represent papers of significant impact and pertinency, of lasting value to the computer community.

SOME PROGRAM HIGHLIGHTS

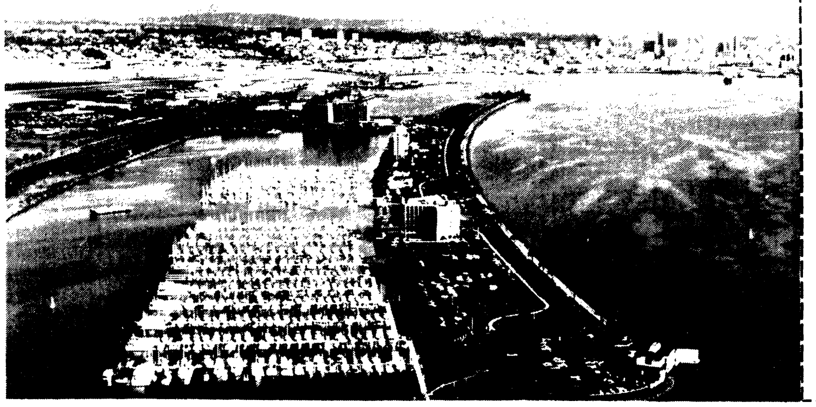
- The anual ACM Turing award.
- SICMINI session on minicomputers. Panel discussion: Will Minis Replace Large Computers?
- Special Paper: Computers and Society; Public Policies and the ACM—Daniel D. McCracken.
- Sessions on: Structured Programming, Programming Languages, Performance Measurement, Computer Aids to the Physically Handicapped, Numerical Analysis, some 50 more.
- Tutorial: Minicomputer Operating Systems.
- Unpopular Ideas in Computing; a panel.
- Security and Privacy: Their Effects on Computer Management.
- SIGGRAPH: 2 sessions, plus a tutorial and workshop.
- ACM Computer Ombudsman Program.

MOST CONVENIENT CONFERENCE EVER

All hotel space has been arranged on "walk-around" Harbor Island with marinas, restaurants and beaches in an uncommonly comfortable environment. Free shuttle buses run on Harbor Island to and from the Convention Center for technical sessions.

If you're coming from outside San Diego: treat yourself (and spouse?) to an extra day or two. Enjoy world-famed San Diego: balmy beaches, zoo, wild animal park, Old Mexico, 66 golf courses, sumptuous restaurants, scores of other things to do.

If you're coming from California: the shuttle bus service will include the airport and AMTRAK station, providing local transportation. Commute or stay over.



Clip coupon and mail to:

ACM 74—Attn. Lyn Swan
P.O. Box 9366
San Diego, Ca. 92109

Please send additional information.

Name____________________

Street____________________

City__________ State_______

Zip__________

Are you an ACM national member?
☐ Yes ☐ No

PROCEEDING FOR 9th and 10th DA WORKSHOP

The following is the rate schedule as agreed to by SIGDA and the DA Technical Committee of IEEE.

	Copies		
1)	________	9th DA Workshop Proceeding (1972) 376 Pages	@ $10.00
2)	________	10th DA Workshop Proceedings (1973) 288 Pages	

____	SIGDA Members	@ $10.00 =	________
____	ACM Members		
____	Non-ACM/Non SIG Persons	@ $16.00 =	________
____	Subscribers	@ $16.00 =	________
	(Member Number ________)	TOTAL =	________

Please send your order prepaid to:

ACM Inc.
P. O. Box 12105
Church Street Station
New York, New York 10249

Make your checks payable to Association for Computing Machinery (an added charge is made for billing).

JOIN SIGDA JOIN SIGDA JOIN SIGDA JOIN SIGDA JOIN SIGDA JOIN SIGDA JOIN SIGDA JOIN SIGDA JOIN SIGDA JOIN SIGDA JOIN SIGDA JOIN SIGDA

____
Name (please type or print)

____
Affiliation

____
Mailing Address

____
City State Zip

Annual membership dues are $3.00 for ACM members and $5.00 for others.

____ Enclosed annual dues.

____ Please send more info.

Mail to SIGDA
ACM INC.
P.O. Box 12105
Church Street Station
New York, NY 10249