Hardware design and description languages in IBM

by L. I. Maissel
H. Ofek

Hardware design languages (HDLs) allow computer hardware to be described in sufficient detail to be simulated and built, such a description being at a sufficiently high level of abstraction to make the complete design readily intelligible to anyone skilled in that language. A number of HDLs have been developed and are in use in IBM. To date, no overwhelming case can be made for choosing any one HDL over the others. The major trends in HDL are discussed. Several examples of HDLs are presented in some detail. VHDL, the yet-to-be released HDL which is to serve as a front end to the U. S. Government's Very High Speed Integrated Circuits program, is among these.

Introduction

This paper describes IBM activities relative to the development of hardware design and description languages (HDLs) and their usage as part of several design automation systems. The paper is limited to efforts carried out at IBM laboratories in the United States and it focuses on languages which have a product history or are of strategic importance.

Digital hardware may be described in a variety of ways. At one extreme, for example, it could be described as a series of I/O patterns (output values that result from a given set of input stimuli). Such a description would be characterized as purely functional. At the other extreme, the description could be the detailed information needed to actually implement the hardware as an integrated circuit in silicon. In this case, the description would be characterized as purely structural. Between these two extremes we can identify a continuum of description formats that are characterized by a mix of functional and structural information in varying proportions.

From a human perspective, a given piece of digital hardware is first conceived in terms of the function that it will perform. The process of digital design then consists of translating this into silicon hardware, usually in a series of steps. The purpose of hardware design and description languages is to facilitate this process. Hardware description is facilitated by enabling structure to be described in a way that makes the function of that structure easier to perceive. Hardware design is facilitated by enabling function to be described in a format that implies a certain amount of structure. We refer to the level within the functional-to-structural spectrum in which HDLs fall as the RT (register-transfer) level of hardware description.

Even if they are considered to be at the same RT level, two HDLs may differ from one another in a variety of ways. One way in which they could differ is in the details of the type of structure that is implied by a given description. In general, it is possible to implement any design (coded in a given RT language) in any given technology but, in practice, the structure that is most easily generated from a description in a given HDL often favors a specific technology. Regardless of any such built-in bias, however, it is usually the goal of the RTL developers to generate a structural description that is as technology-independent as possible (even though that is something of a contradiction in terms).

In practice, HDLs allow for the simultaneous inclusion of descriptions at the functional and structural level as well as at the RT level. The functional component tends to dominate during the early stages of a design but must be translated to the RT level, or directly to the structural level,

before the design can be implemented. This translation requires human intervention. In some cases additional software that is not part of the HDL may be used to translate the final structural description produced by the HDL to final hardware. This process is termed *technology transformation*. The combination of RTL-to-structure translation together with technology transformation is termed *logic synthesis*.

Historically, at IBM as elsewhere, initial attempts to create design languages were directed more towards structure than function. In addition, hardware designers have generally preferred the graphic form of expression rather than the textual form. An early form of abstraction used in IBM design was the ALD (Automated Logic Diagram) [1]. This represented a modest piece of logic (such as a NOR or NAND gate). Connections between ALDs could be specified through naming conventions used to identify the input and output lines on the ALD. ALDs are easy to draw on a simple (nongraphic) printer, and substantial pieces of logic can be represented in a booklike format in which ALDs appear on successive pages, the connections between ALDs being shown as pointers to the appropriate page(s).

With the advent of alphanumeric interactive terminals, the need for a textual form equivalent to the ALDs became apparent and BDL/S (Basic Design Language for Structure) was defined. A structure described in BDL/S can be simulated in a simulator called VMS (Variable Mesh Simulator). Furthermore, as the state of the design automation art matured, it became apparent that functional descriptions could effectively be used in conjunction with BDL/S. Thus the final logic did not have to be designed down to the gate level before being simulated for the first time. The language used for functional descriptions in conjunction with BDL/S is called EPL/S; it is the first of the IBM languages we describe.

EPLS (Extended Programming Language for Systems)

EPL/S was defined about 1973. Its purpose was to provide the engineer with a way to describe functional models of hardware, allowing for the early modeling and simulation of such hardware. No process for transformation from EPL/S to BDL/S was defined, although descriptions written in these languages may be compared by applying identical stimuli to both and then checking the corresponding responses for equivalence.

EPL/S was defined as an extension of PL/S (Programming Language for Systems). Historically, it was an extension of VMS–PL/S, which is itself an extension of PL/S. PL/S, a variation of PL/I, is a procedural programming language which is suitable for writing systems programs. It includes a macro facility which allows a PL/S programmer to define new statements. Such new statements are processed by the PL/S macro compiler and transformed to regular PL/S programs. EPL/S consists of the entire PL/S language plus a set of additional commands. Thus, when a hardware designer describes a logic function in EPL/S, it is translated into PL/S code, which in turn is compiled into machine-executable code. The executable modules which are created from EPL/S descriptions are executed under the control of a simulator. It should be noted that an engineer using EPL/S must be familiar with PL/S, making the language less attractive to those engineers who believe that they should not have to become "programmers." One convenient feature of EPL/S is the availability of commands which allow the model to communicate with the simulator.

EPL/S descriptions are executed under VMS, which is part of EDS (the Engineering Design System) [2]. VMS was originally designed to simulate logic network structures described by BDL/S. In order to allow for EPL/S modules to be handled by VMS, one needs to specify "structures" which encapsulate the behavioral descriptions. To that end, a set of structure commands is defined as part of EPL/S. The two basic structure commands, which enable a user to break the description into different types of subsets, are

- BLOCK—a group of statements analogous to a procedure, and
- MINIMOD—a related set of statements within a block.

These are used to identify the behavioral description to VMS by providing it with the function name, and to generate the internal interfaces needed for the EPL/S model to communicate with VMS.

A second set of EPL/S commands is defined to aid the engineer in telling the simulator how to account for the timing and synchronization involved in executing several behaviors or sub-behaviors. The three basic timing-related commands are

- WAIT, which suspends execution by a specified number of time units,
- SCHED, which schedules a MINIMOD to execute at a prescribed time, and
- POST, which is used to bypass WAIT and SCHED commands asynchronously.

The timing commands are used to help the simulator order the execution of the behavioral elements in the EPL/S model. They are not describing the actual timing and delay characteristics of the real logic.

Another set of commands is used to define data and data attributes, including scope. Examples are

- INPUT, which describes input lines to a block,
- OUTPUT, which describes output lines from a block, and
- TABLEOPT, which lists variables to be accessed during simulation.

Some of these commands are used to create data entities which are shared by the EPL/S descriptions and other parts

of the system, such as the stimulus description language which is used to specify test cases for VMS.

In addition to these command types, there are a number of commands whose purpose is to make the task of writing code easier for the user. For instance, the BIT statement enables the user to specify bit manipulation of bit strings defined in one of the data definition commands. Another statement of importance is the TRACE statement. It helps the user to debug the EPL/S model by returning control to the simulator and by allowing display and modification of the variables. This feature is vital to interactive simulation.

In summary, EPL/S is a reasonable modeling language. However, it lacks sufficient hardware description capabilities to allow for hierarchical hardware documentation and design support. Its style is readily acceptable to engineers who are also programmers, but it is not considered an "engineer's language."

Although EPL/S was, and still is, widely used within IBM, it is not a true HDL (as defined above). As the state of the design automation art matured, however, languages at the RT level did begin to appear within IBM. Starting around 1970, a number of such languages were developed, most of them short-lived. Three of them have, however, survived the test of time. They are all about the same age and each has been used at several IBM locations. All have had success in the generation of real hardware. These three languages are IDL, BDL/CS, and SDL.

IDL (Interactive Design Language)

IDL was conceived about ten years ago [3]. Its initial purpose was to facilitate the design of PLAs that embodied highly complex algorithms, but it is now being used for the design of random logic as well [4]. IDL is nonprocedural; the order in which IDL statements are listed is unimportant. IDL is hierarchical with some limitations. Though entry of data into IDL may be graphical (flow chart form) or textual, the great majority of IDL users have preferred to use the textual form. IDL is implemented in APL, although certain key routines which are CPU-intensive are coded in 370 assembler.

IDL is particularly well suited for self-documentation. If they choose, users can, in effect, create a syntax which makes their code look like standard English (or any other language). IDL is used for design as well as description, since it generates two-level logic from the high-level description. Multi-level logic can also be generated; the format that has been adopted is that of a series of two-level "boxes" connected by signals.

Design verification under IDL is achieved via simulation; three simulators are available. Two of these are of the zero-delay type and are intended for single boxes. The third is a multi-box simulator in which the boxes have specific internal delay times, clock rates, etc., associated with them. All three simulators accept four possible input values—zero, one, unknown, and high impedance. Additional design tools associated with IDL, although not formally part of the language, include minimization (number of product terms and number of feedbacks), partitioning, merging, and logical equivalence checking (between two pieces of two-level logic).

IDL is suitable for both structural and behavioral descriptions, although structure is usually used to indicate connectivity between pieces of logic whose structure was synthesized by the system from a behavioral description.

Some of the key features and constructs of IDL are as follows: Sequence control is achieved through the use of labels; every IDL statement that is not a declaration must be associated with a label. No restriction is placed on how many labels may be active at a given time. If two or more labels are simultaneously active, simulation treats them as parallel processes. Multiple simultaneous assignment is permitted and is treated as the OR of the individual assignments. The general action statement in IDL is IF THEN ELSE. Input conditions can be quite complex and many complex functions such as relational operators, incrementers, etc. are built in, i.e., synthesized for the user. Output statements can also be more complicated than merely assigning values to outputs. They can, for example, imply complicated control actions such as register transfers, memory accesses, etc.

IDL allows blocks of two-level logic to be represented in IDL code as truth tables. Since the logic that is synthesized from IDL code is two-level logic, a given IDL design can itself be used as a truth table within a larger design. This form of hierarchical representation can be used to as many levels as desired, but during simulation the hierarchy is first flattened to the lowest level. Alternatively the user may temporarily represent a truth table as a truth function. Under these circumstances, the table is simulated (as a function) directly and the hierarchy is not flattened.

Functional descriptions (APL programs) that do not represent truth tables are also permitted within IDL. These are directly executed during simulation. Subroutines are also available. These are blocks of IDL code that are executed more than once in the course of running an IDL program. They are automatically linked to ensure return to the correct state after execution. Subroutines offer a "busy-protect" feature so that competing processes may or may not share the same subroutine simultaneously, depending on the user's choice.

A useful feature of IDL is the ability to represent a particular action as a sequence of actions taking place over several cycles and then synthesize this as an action executed in a single cycle. A refinement of this feature is the mechanism which allows IDL to be used for multi-level logic. In IDL, multiple pieces of two-level logic that are in series can be described as a single piece of two-level logic connected by "zero-delay" feedbacks. These behave like multi-level logic during simulation and also convey

information to existing programs within IBM (but outside IDL) for the synthesis of optimal multi-level logic [5].

IDL's strongest features are its self-documenting abilities and its suitability for highly sequential designs that embody complex algorithms, particularly those in which many parallel processes are occurring. It is weakest when applied to designs whose optimization depends on very close attention to the details of the ultimate physical embodiment.

IDL is routinely used for problems of the order of 10 000 random logic gates and has been used successfully for designs of at least 40 000 random logic gates [6, 7]. At this time four chips that have gone all the way through manufacturing (and ended up in products) have been designed by IDL. This is in addition to many other designs that "died" before reaching final manufacturing.

IDL has found limited use outside IBM, e.g., at M.I.T. [8].

BDL/CS (Basic Design Language for Cycle Simulation)

BDL/CS was conceived in 1971 by an engineering group in Poughkeepsie [9]. The language was enhanced in 1978 to be used with the newly developed EFS (Experimental Functional Simulator) rather than the VMS simulator [10]. A hardware simulator is currently in development which uses BDL/CS as its input language [11]. BDL/CS is used as an input for a static analysis system to verify the equivalence of two logical models, one in BDL/CS and one in BDL/S [12]. BDL/CS is also used as an input for a technology-dependent logic synthesis process [13].

BDL/CS was initially designed as a flowchart language with a one-to-one correspondence between the flowchart and the code generated for the model. Graphic entry, in the form of flowcharts, is available, or BDL/CS may be created as text. BDL/CS is well suited for self-documentation, with the designer using the flowcharts as his master high-level document.

The emphasis is on behavior rather than structure. BDL/CS is designed for cycle simulation with no provision for simulating delays within a cycle. A timing analysis system is used to determine delays within the logic [12]. BDL/CS is a nonprocedural language, allowing statements to be placed in any order. Algorithms can be fragmented without explicit connections between the various parts. Currently, BDL/CS is nonhierarchical, although there are plans to accommodate some form of hierarchy in the language and simulator.

The compiler is implemented mostly in PL/I with a few modules in PL/S. The simulator is implemented totally in PL/S with some imbedded System/370 assembler for performance.

Some of the key features and constructs of BDL/CS are as follows:

1. The order in which statements are executed is determined by means of automatic signal ordering to ensure that no signal is used before it is properly generated.
2. The general action statement in BDL/CS is IF THEN ELSE.
3. Multiple simultaneous assignments are allowed and are treated as the OR of the individual assignments.
4. Functional descriptions, written in PL/S, are allowed.
5. Parallelism during processing is correctly simulated to faithfully duplicate the action of the hardware.
6. Various facility types are supported to allow the designer to define the action he desires without explicit coding in the model. Some facility types are *set dominant, reset dominant, signal, one-cycle, bit array, and byte array.*
7. Two- or four-value simulation is supported. The four possible values are *zero, one, uninitialized,* and *unknown.*

The strongest points of BDL/CS are as follows:

1. It is practical for models representing up to 1 000 000 random logic gates.
2. It is technology-independent, allowing the designer to evaluate algorithms prior to implementing them in BDL/S.
3. It is self-documenting in terms of flowcharts.

The weakest points of BDL/CS are as follows:

1. It does not currently support phased clocking.
2. It does not currently support hierarchical designs.
3. It is difficult to simulate asynchronous logic.
4. It is of limited value for describing complex sequential logic.

The principal use of BDL/CS has been in the large-processor area, such as the 308X, although projects much smaller in scope have begun to use it also.

SDL (System Design Language)

SDL is a technology-independent, register-transfer-level, hardware design and documentation language. It is nonhierarchical, alphanumeric, list-oriented (nongraphic), and free-format. It was conceived in 1971 at IBM Rochester. John Reed developed the language syntax and Bill Steingrandt developed the first simulator. Some of the early work on SDL was influenced by DDL (Digital Design Language) [15]. SDL is implemented with PL/S and System/370 assembler code.

SDL is used by logic designers to describe hardware that will be synthesized into VLSI chips [16]. It is also useful for creating simulation models of existing functions and for limited microcode debug of existing systems. SDL may be used to describe logic from a structural point of view (counters, arithmetic logic units, data flows, macros, etc.), from a behavioral point of view (sequences, procedures, control flows, operations, etc.), or a combination of both.

SDL consists of hardware declarations called FACILITYs and hardware processes called OPERATIONs and AUTOMATONs. FACILITYs are explicit hardware elements and include INPUT, OUTPUT, OPERATOR (combinational logic), REGISTER, STORAGE (arrays), MACRO (custom logic structures), DELAY (timing control), TIMER (oscillators), and INTERFACE (temporary structural connections). OPERATIONs and AUTOMATONs describe the interactions between hardware elements, and each OPERATION and AUTOMATON operates in parallel with all other OPERATIONs and AUTOMATONs.

Within OPERATIONs and AUTOMATONs, IF ELSE, DECODE (select), and DO END statements are used for decision making. Data are assigned to REGISTER, STORAGE, DELAY, and INTERFACE facilities by means of transfer statements (load, set, reset, and connect), and references may be made to other FACILITYs or groups of FACILITYs imbedded in logical expressions.

SDL has twelve logical operators: AND, OR, XOR, binary subtract (twos complement), ones complement, parity, concatenate, binary addition, gate, equal, less than, and greater than. They are used in OPERATORs, OPERATIONs, and AUTOMATONs to generate logic expressions or comparisons of any desired complexity.

OPERATIONs describe events which are not sequentially related to other events. During simulation all the events defined within an OPERATION execute simultaneously. An OPERATION might be used to describe what happens during "power-on reset."

AUTOMATONs describe events which are sequentially related to other events. An unlimited number of STATEs may be defined for each AUTOMATON. Each STATE has a set of user-defined events and all events in a STATE execute simultaneously. During simulation only one STATE of an AUTOMATON is active at a time and control is passed between the STATEs of the AUTOMATON by the user-defined events. An AUTOMATON might be used to describe what happens during the "instruction fetch" sequence.

For debug, SDL provides the DUMP, EXIT, and EVENT statements. These statements allow the user to print information about activities within the model. In addition, trace information regarding model activity may be dumped to the facility and operate trace files.

SDL is an integral part of a larger design, verification, and synthesis methodology [16]. Simulation of SDL models is done using EDS. SDL models, either alone or as part of a larger group of VMS-compatible behaviors (VMS–PL/S, EPL/S, BDL/S, etc.) are simulated by executing user-written control statements under control of the EDS VMS simulator.

Topological analysis of an SDL model may be done using the SDL–PRIME analysis routines [16]. These routines provide cross-referencing information, combinational logic equation listings, LSSD scan path information, design data flow information, and design control flow information.

Logic synthesis of an SDL model occurs in two steps. First, the SDL–PRIME synthesis routine converts the SDL model to technology-independent BDL/S [16]. Then EDS LTS (Logic Transformation System) converts the TI BDL/S to technology-dependent (TD) BDL/S [5]. The TI and TD BDL/S created by these programs may be used as input to any EDS-supported program (Static Analysis System, LSSD Design Rules Check, Timing Analysis, VMS, etc.).

The strengths of SDL lie in its descriptive flexibility, its ability to model concurrent processes, the state concept embodied in the AUTOMATON concept, and its capacity to model logic designs of large size (in one design, 40 000 random logic gates were modeled along with 128K bytes of storage).

The weaknesses of SDL include its poor timing specification capabilities and its inability to communicate with other behaviors or programs except through INPUT and OUTPUT statements.

VHDL (VHSIC Hardware Design and Description Language)

VHDL is a language whose most important characteristic at this point in time is that it represents the future [17]. While it has not yet been released, it could pervade the entire industry, since it represents more current thinking in the area of HDLs, i.e., the state of the art.

The U.S. Department of Defense (DoD) is funding the development of VHDL. The work will result in a DoD standard hardware description language, an analyzer/compiler, and a VHDL mixed-mode simulator. The scheduled completion date is December 1, 1985. Contractors will then be required to describe their VLSI designs (design specification or completed design) in the standard VHDL. The new language has implications for the development of design tools and for the way designers approach new designs or use existing ones.

The language organization will provide a hierarchical design capability that allows the designer to describe, evaluate, and utilize design alternatives. The key conceptual element within VHDL is the design entity. It is composed of a unique interface description and a design body. The interface description represents the intended external interface of the hardware being designed. As design progresses, the interface description will be refined to match the real hardware exactly. Within the design body, the designer may describe one or more design alternatives, called *variants*, for the desired hardware.

Within each variant, several design aspects may be described in terms of function, RT level, or pure structure. Other user data relative to a design may also be conceptually held within a variant. Functional descriptions are to be

written in a subset of the Ada language. The RT-level aspect allows the designer to describe a hardware design in a nonprocedural language where the structure may be depicted as a combination of structural statements and Boolean conditions for the execution of those statements.

In addition, the language will provide "assignment" constructs. The underlying conceptual model is that all conditions are evaluated, in parallel, at "interesting" points of time. Interesting points of time are determined on the basis of transactions (implicit handshake), events (explicit handshake), and periodicity (implied clock). All statements that have conditions that evaluate to "true" will be executed in parallel. Conditions may be labeled, the label values being evaluated as part of the condition, thus allowing the designer to create any desired sequence of concurrent or parallel operations.

Through the use of the condition, action, and effect parts of the function statement, the designer can easily separate control flow from data flow. This approach is much used by designers today as a means of comprehending and partitioning a complex function to be designed. Since the RT level in VHDL implies a structure, tools could be developed to create a graphic picture of the implied structure.

An explicit structural description represents the interconnection of design entities. It is key to the hierarchical nature of VHDL design descriptions. Design entities may be decomposed into subfunctions, giving rise to a hierarchy of design entities. The structural description implies the hierarchy. A hierarchy of design entities may take the form of a graph, since a given design entity may be used many times throughout the hierarchy.

A simulation model may be created, in the simplest case, by selecting a design entity and choosing the desired design alternative from that design entity. If structural description is chosen, a model tree may be created by looking at the structure contained within the chosen design entity and at each lower-level design entity that is part of the structure. This process continues until all the design entities in lower-level structures for which functional- or RT-level descriptions exist have been chosen. The model tree is then "flattened," keeping only the structure of connected behaviors and the root node (the topmost design entity with the chosen design alternative).

A library capability will be provided so that frequently used design abstractions written in Ada may be accessed by a designer not wishing to write his own. This will allow experimentation with newly created design function in the context of a surrounding abstract functional environment.

The primary use of VHDL is intended for, but not limited to, the design of VHSIC class components (10K to 100K equivalent logic gates per chip). The designer will use the VHDL design system to specify designs, to create new design entities, and to utilize existing design entities. VHDL may be used by different designers employing various methodologies.

Concluding remarks

As part of the development of HDLs, many languages have been proposed; only a few have survived. The reasons for their survival are often as dependent on factors such as the persuasiveness of their developers, the timing of their release, etc., as upon technical merit. As a consequence, none of the HDLs currently in widespread use in hardware manufacture can be regarded as the last word. In almost all cases, their language definitions have been informal, and extension of these languages to support innovations in the design automation art (such as mixed-mode simulation, for example) is difficult if not impossible. Furthermore, the existing languages tend to be specialized, each having evolved inside a relatively small part of the design community.

Thus, a language that can be used by a broad spectrum of designers, that promises to be readily extensible in the future, and that has been designed in a formal manner, would be highly desirable. VHDL may have these properties.

Acknowledgments

The authors would like to thank H. Fleisher for his general encouragement in the writing of this paper and for many stimulating discussions. The section on BDL/CS was written by R. L. Price, the section on SDL was written by L. F. Saunders, and R. Waxman and A. D. Savkar wrote the section on VHDL. We thank the U.S. Department of Defense for giving us permission to publish the latter.

References

1. P. W. Case, H. H. Graff, L. E. Griffith, A. R. Leclerq, W. B. Murley, and T. M. Spence, "Solid Logic Design Automation," *IBM J. Res. Develop.* **8,** No. 2, 127–140 (March 1964).
2. P. W. Case, M. Correia, W. Gianopulos, W. R. Heller, H. Ofek, T. C. Raymond, R. L. Simek, and C. B. Stieglitz, "Design Automation in IBM," *IBM J. Res. Develop.* **25,** No. 5, 631–646 (September 1981).
3. L. I. Maissel and D. L. Ostapko, "Interactive Design Language: A Unified Approach to Hardware Simulation, Synthesis, and Documentation," *Proceedings of the 19th Design Automation Conference,* Las Vegas, NV, 1982, pp. 193–201.
4. Leon Maissel and Raymond L. Phoenix, "IDL (Interactive Design Language) Features and Philosophy," *Proceedings of the IEEE International Conference on Computer Design,* Port Chester, NY, 1983, pp. 667–669.
5. J. B. Bendas, "Design Through Transformation," *Proceedings of the 20th Design Automation Conference,* Miami Beach, FL, 1983, pp. 253–256.
6. K. R. Woodruff and P. S. Balasubramanian, "Top-Down Design Using IDL," *Proceedings of the IEEE International Conference on Computer Design,* Port Chester, NY, 1983, pp. 670–673.
7. Mark W. Brown and M. Jay Kimmel, "Multiple Implementations of a Microprocessor from a Single High Level Design," *Proceedings of the IEEE International Conference on Computer Design,* Port Chester, NY, 1983, pp. 674–677.
8. Steven K. Heller and Arvind, "Design of a Memory Controller for the MIT Tagged Token Data Flow Machine," *Proceedings of*

the IEEE International Conference on Computer Design, Port Chester, NY, 1983, pp. 678–682.
9. G. J. Parasch and R. L. Price, "Development and Application of a Designer Oriented Cyclic Simulator," *Proceedings of the 13th Design Automation Conference,* San Francisco, CA, 1976, pp. 48–53.
10. L. N. Dunn, "IBM's Engineering Design System Support for VLSI Design and Verification," *IEEE Design & Test of Computers* **1,** No. 1, 30–40 (February 1984).
11. G. L. Pfister, "The Yorktown Simulation Engine," *Proceeding of the 19th Design Automation Conference,* Las Vegas, NV, 1982, pp. 51–54.
12. M. Monachino, "Design Verification System for Large-Scale LSI Designs," *IBM J. Res. Develop.* **26,** No. 1, 89–99 (January 1982).
13. J. A. Darringer and W. H. Joyner, "A New Look at Logic Synthesis," *Proceedings of the 17th Design Automation Conference,* Minneapolis, MN, 1980, pp. 543–549.
14. R. Hitchcock, G. Smith, and D. D. Cheng, "Timing Analysis of Computer Hardware," *IBM J. Res. Develop.* **26,** No. 1, 100–105 (January 1982).
15. J. R. Duley and D. L. Dietmeyer, "A Digital System Language (DDL)," *IEEE Trans. Computers* **C-17,** 850–861 (1968).
16. L. F. Saunders, "An Approach to VLSI Logic Design," *Electronic Design Automation (EDA '84), Electronics Division of the Institution of Electrical Engineers of Great Britain, Conference Publication 232,* March 26–30, 1984, pp. 33–34.
17. Contract No. F33615-83-C-1003, U. S. Air Force Wright Aeronautical Laboratories, Avionics Laboratory VHSIC Program Office, Air Force Systems Command, Wright-Patterson Air Force Base, OH 45433. Work being performed by Intermetrics, Inc. (prime contractor), IBM Corporation, and Texas Instruments, Inc. (subcontractors).

Received February 13, 1984; revised April 30, 1984

Leon I. Maissel *IBM Data Systems Division, P.O. Box 390, Poughkeepsie, New York 12602.* Dr. Maissel is a senior technical staff member in the office of IBM Fellow Harold Fleisher. He received his Ph.D. in physics from the University of London, England, in 1955. He joined IBM in 1960. For ten years he was active in the area of thin films. In 1970, his interests changed to computer systems, in particular hardware design and description languages. He has served on the editoral boards of several journals, currently including the *IEEE Design and Test Magazine.* Dr. Maissel is a Fellow of the Institute of Electrical and Electronics Engineers.

Hillel Ofek *Silval-Lisco, Menlo Park, California 94025.* Since the beginning of 1984 Mr. Ofek has been Senior Vice President of Research and Development at Silval-Lisco. Prior to that, he had joined IBM in 1965 in the Systems Development Division, where he designed special hardware for NASA and experimental logic for prototype digital machines. He also did work in the area of reconfigurable systems. In 1969, Mr. Ofek joined the engineering design system, where he worked on logic partitioning, test generation, and Boolean analysis. From 1973 to 1976, he worked at the Thomas J. Watson Research Center, where he participated in design automation research addressing the topics of analysis and symbolic simulation techniques for hardware design verification, and the definition of the language for computer design (LCD). In 1976 Mr. Ofek returned to the engineering design system, where he was responsible for advanced development work in design verification, hardware description languages, and design transformation aids. In 1983, he joined the Corporate Technical Committee in Armonk, New York. Mr. Ofek received his B.Sc. in electrical engineering from the Technion (Israel Institute of Technology) in 1963 and his M.E. in electrical engineering from New York University in 1965. He is the recipient of a First-Level IBM Invention Award. Mr. Ofek is a member of the Institute of Electrical and Electronics Engineers.