Michael Monachino

# Design Verification System For Large-Scale LSI Designs

*This paper describes the changing environment of large-scale hardware designs as influenced by technology advancements and the growing use of design verification in the design implementation process. The design verification methodology presented here saved some 66% from the 3081 product schedule, when compared with a schedule utilizing a conventional verification method, on almost 800 000 LSI logic circuits. The paper discusses the use of software modeling techniques to verify LSI hardware designs, methods used for deciding when modeling should be stopped and hardware can be built with sufficient assurance to permit additional verification to continue on the hardware, methods for testing the hardware as it is assembled into a very large processor complex, and the organization of the design verification system to avoid duplicate creation of test cases for different stages of the design process. Experiences encountered in designing and verifying the 3081 system, a discussion of some shortcomings, and an endorsement of certain techniques and improvements for use in future designs are also presented.*

## Introduction

Before the advent of LSI, a designer had the ability to design a computer, build it, debug it, and correct the errors on the test floor with little help from a manufacturing facility or from vendors. With LSI, the engineer is dependent on a manufacturing entity for building the customized chips, chip mounts or modules, and boards, and for reworking these items during the building and testing phases. With these dependencies, the initial hardware delivery of a model design was projected to take a minimum of six months. An average of thirty working days were projected for reworking the hardware for each design change. Since thousands of changes were likely on a project of the 3081 scope, a design would be obsolete by the time it was ready for shipment.

On the basis of these considerations, as well as Roth's early work [1], it was determined that a *design verification system* (DVS) was essential to permitting the advantages of LSI to be realized. Through the use of the DVS, the engineers could verify the logic functionality of the design, could check that the timing constraints were met, and could track the progress of the design verification status to a preset goal before committing the design to hardware. Furthermore, they could continue to track the status when engineering changes, function enhancements, and feature additions would change the original design.

In pre-LSI designs, *ratification* (matching the design specifications), involved simulation of the logic designs with little use of high-level models. LSI ratification still uses detailed simulation but also employs intermediate simulation using higher-level models.

In the past, use of the word *validation* was restricted to the testing of a hardware model on the test floor; but *validation*, as used in this paper, means an analytic method of comparing two models [2]. The purpose of the validation process is to find the differences between the two models being compared. The validation method assumes that one of the models being compared is correct and any differences represent design errors.

Before LSI, the term *coverage analysis* was unknown. The designer had the problem of determining when the ratification of a design was completed, and to what extent design problems had been eliminated. With LSI, the design community required techniques to determine quantitatively the *quality* (adherence to specifications) of the design being released to manufacturing in order to maximize the number of design errors removed prior to the construction of hardware. The designer also needed a method to determine when to stop simulating and when to start the release process.

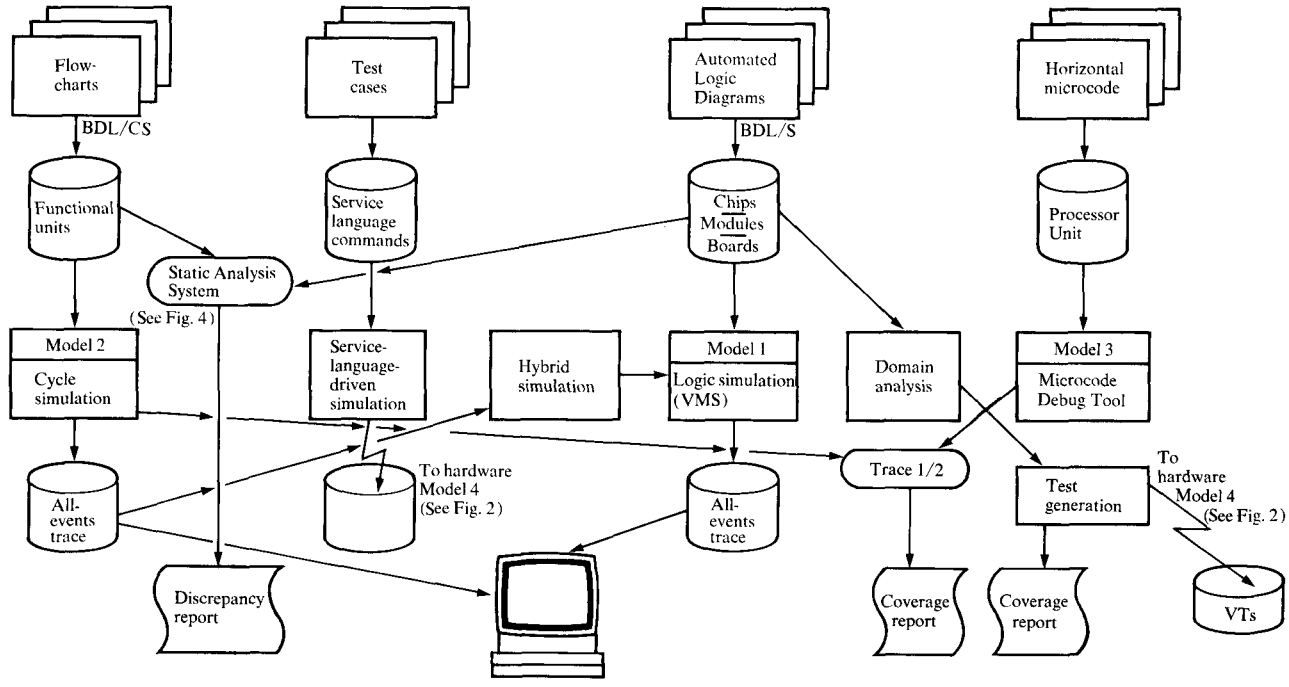



**Figure 1** Design Verification System (DVS) overview. (Note: BDL/CS = Basic Design Language for Cycle Simulation, BDL/S = Basic Design Language for Structure, VMS = Variable Mesh Simulator, VTs = Verification Tests.)

In pre-LSI designs, *timing analysis* was used only on single paths thought to be "critical" by the designers; all other timing problems were found by means of the hardware model. Also in previous technologies, clock-line tuning could be used to cover product variations. With LSI, the technology rules are of such complexity that the designer can no longer identify "critical" paths. In addition, the new technology has a wide range of tolerances, requiring a statistical timing analysis in order to maintain the functionality of the product over a large range of production variations.

With pre-LSI hardware, *model testing* was not always systematic and thorough, but was based on the knowledge, experience, and needs of the designers, who were also responsible for the hardware model debugging. They could rewire the hardware at will. LSI hardware, on the other hand, has required the designers to generate a systematic testing procedure prior to hardware building to beat the hardware change cost and turn-around times. Thus, techniques like functional matrices and cause-and-effect diagrams [3], as well as automatic result checking and automatic test-case generation, had to become a basic part of the design process. Test-case hierarchy techniques for the optimization of test-case packages were needed, and the requirement for an independent group of experts in test-case generation was demonstrated.

As a by-product of the design verification effort, all the necessary parts fell into place to create a diagnostic test package for use by manufacturing and field personnel, to identify failing hardware components, once the design was verified. Also, the ability to inject numerous error conditions into the software models and to points not accessible to hardware bugging could be exploited to verify the diagnostic package before the hardware was operational.

## Design verification system goals

The main goal of a design verification system is to establish an orderly design process with testing repeatability and test data integrity, not only for the initial design but also for any follow-on *engineering changes* (ECs), feature additions, or enhancements. The system must verify not only the logic design but also the other parts of the design process, including the architecture of the machine, the manufacturing-support package, and the field-support package. Problems must be discovered in the earliest stages of development.

Thus, a set of management goals had to be designed into DVS to evaluate the status of the quality of the hardware design being released to manufacturing. This required a method for tracking the circuits or logic blocks that had been verified, and a way of determining and presenting this information to both designers and management.

## Hardware error projections

The number of errors found in a design generally depend on the complexity of the circuits and on the number of chips. We expected to find ten errors per thousand circuits in data

path logic designs and four times that many in control logic designs. An additional factor had to be considered for the complexity of the timing problems associated with the circuit tolerances of the technology and packaging used. For the 3081, on the basis of expectations of DVS effectiveness and hardware change turn-around time, we set the design goals to catch 84% of the logic design problems and 100% of the timing problems through design verification. The remaining 16% of the logic design problems have to be isolated on the hardware.

An error-projection model incorporating the error history of previous machines as well as the characteristics of the new technology, the complexity of the new design, and the proposed schedule was used to predict the number of errors expected in the design. A contingency factor of 14% was added for non-error-type changes resulting from revised design specifications.

## Ratification techniques

The design verification system illustrated in Fig. 1 contains a structure for ratification made up of four models: Model 1 (logic simulation) consists of a detailed logic design evaluation; Model 2 (cycle simulation) consists of an intermediate-level software model built from flowcharts; Model 3 (microcode debug tool) consists of a high-level software model developed in PL/S (a Poughkeepsie laboratory adaptation of PL/I) for horizontal microcode [4] testing; and Model 4 (hardware prototype illustrated in Fig. 2) consists of a staged, physical-hardware machine constructed to be equivalent to the intermediate flowchart models. The four models are interdependent and support one another. The common link among them is the ability to run the same test cases on all four models. Each of these ratification techniques is described in turn.

A major problem with previous software models was that they never really represented the actual implementation of the hardware. The difficulty of maintaining up-to-date detailed specifications from which both the hardware and the software models could be unambiguously built, and the fact that a significant amount of programming time and computer resources were required to debug the software models, caused previous projects to drop software modeling once the initial hardware was built.

- *Ratification through cycle simulation*

*Cycle simulation* utilizes flowcharts as inputs and allows a designer to simulate both control and data path logic in flowchart form. The flowcharts are coded into a simple flowchart language called BDL/CS [5], which uses a limited number of operands and which requires no programming knowledge to use. The resultant flowcharts not only become the input source for the model but they also serve as

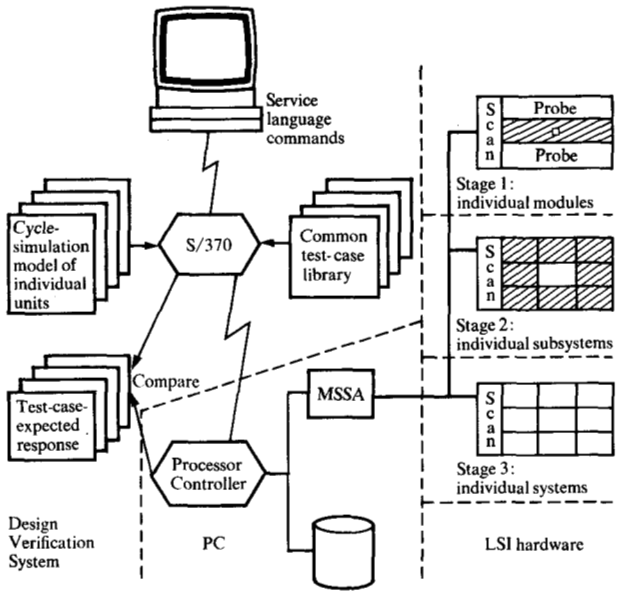



**Figure 2** Design verification system/prototype hookup. (Note: MSSA = Monitoring and System Support Adapter.)

specifications which document the hardware for engineering and manufacturing use. The use of flowcharts has eliminated the problems of specification interpretation and coding errors between a design and its high-level model. It has significantly reduced the amount of time required to debug the software model: each error found in the flowchart represents a design error, and what the designer has drawn in the flowcharts constitutes the software model. Maintenance of the cycle-simulation model is also simplified, since the model consists of altered flowcharts, with no programs to understand, modify, or change. (As part of the engineering change process, each designer updates the flowcharts in the process of updating the design.) The cycle-simulation model is built on three principles:

1. The ordering sequence is defined by the hardware designer.
2. Timewise-parallel functions may be described, although these are executed with sequential code.
3. Strict separation is maintained between the functional design and the physical packaging design. (This permits the designer to concentrate first on solving the functional problem and then on packaging a correctly designed function.)

Cycle-simulation configurations were divided into three stages, using planned increases in the size and complexity of the hardware in each stage, in order to support both the development sequence of design verification and hardware

bring-up. The three stages are illustrated in the right-hand side of Fig. 2, which further shows the way DVS is hooked up with the hardware prototype configuration (to be discussed shortly).

The cycle-simulation model is built and executed in eight steps in a very systematic manner. The first step is to create the hardware flowcharts. The second is to create the data-flow-path models. These are supplied by the engineers in the form of specifications which are then coded up on a latch-by-latch basis by an independent modeling group. Similarly, the flowcharts are transcribed into BDL/CS by the group. Step three is to compile the code and eliminate compile errors. Step four is to create arrays from a description supplied by the design engineer through an automatic software package. Step five creates the logical structure from the latch definitions using the names given to each of the latches in the machine. Step six defines the latches and flowcharts to be used with the test-case library in a particular run of the model. Step seven creates the code combining the three items, and step eight is the actual simulation run.

The simulation runs, in turn, are divided into different stages. The first stage builds a functional model of each circuit module of a processor element. The second stage takes an individual module and adds a combination of macros, which represent interface signals to the module, to simulate asynchronous combinations of data which could occur in the physical hardware. The macros consist of two formats: special flowcharts created to represent the interfaces of various functional units, or PL/S programs which decrease the interfaces of a functional unit. The use of this "driver" technique allows the engineer to simulate conditions which are highly complex, and in some cases, impossible to set up in a hardware environment. These macros and drivers remain until a full module with its natural interface signals is operational. The advantage of this staged approach is that the designers are required to simulate their own designs, using small amounts of computer time to find a large number of problems concurrently. As a by-product, functional test patterns can be saved, which are to be applied by means of a software driver to be used in the first stage of the hardware model bring-up. This stage of bring-up (Stage 1 in Fig. 2) consists of a hardware tester and a module handler with the ability to apply functional test patterns. These patterns are applied to the actual LSSD (Level-Sensitive Scan Design) [6] module scan rings to validate the functioning of the hardware design.

The second stage of the cycle-simulation modeling is to combine functional elements into software models to form subsystems. For an overview of the 3081 subsystems, refer to [7, 8]. The subsystems thus formed represent the *central processor* (CP), the *system controller* (SC), the *external data controller* (EXDC), etc. This creates larger models, requiring more computer time than the earlier smaller models and very large regions of storage (11+ megabytes) in which to execute the code. The advantage of the second-stage models is that the functional test cases can be executed at the operand level. This permits the designer to execute architectural test cases against the design the same way they are executed in a hardware environment. The test cases are generated through external means, such as running a program in memory or entering data through a console. This adds the feature of being able to check the interfaces between a functional unit and other units comprising the system. This stage of modeling was used to test a complete section of hardware housed in a full *thermal conduction module* (TCM) board.

Multiple drivers on the software model are used to emulate customer environments as well as some critical stress scenarios, obtained from previous machines as stress cases which were encountered during testing or through real customer problems collected over a five-year period. By applying these stress environments early in the simulation cycle, we eliminated some rather complex potential problems from the design. Data thus generated were also later compared against data obtained on a cycle-by-cycle basis from a hardware model, thereby making possible the evaluation of other very complex interactions.

The final stage of the simulation model consists of using a *functional simulator* (an enhanced version of the cycle simulator) to allow the engineers to construct a complete processor, in order to eliminate potential interface problems not discovered by the subsystem models. The processor model can run standalone programs of up to 128K bytes, giving the designers the ability to use system software as test cases.

In order to achieve test-case migration between the different models, a common test-case language was devised. The system to support this language was called SDS (*SLC-driven simulation*), and could be initiated through *service language commands* (SLC) from display formats identical in appearance to the operator console displays of the 3081. All levels of simulation were designed to accept test cases generated by SDS. The technique used was to write a unique driver for each of the models or hardware test vehicles rather than to write different test cases for each stage of testing. A test case, for example, was first used in cycle simulation against the functional element, then the functional unit, and finally, the functional system. It was then carried forward and used in module, subsystem, and processor testing, eliminating duplicate efforts in creating and in maintaining separate test-case libraries.

• *Logic ratification through variable mesh simulation*
Logic ratification was accomplished in two ways: unit-logic simulation and hybrid simulation. A test-pattern-driven logic simulator called the *variable mesh simulator* (VMS) [9] was available for use by the logic designers. It could be used in the following three modes:

1. AND/OR-logic simulation, ratifying the basic logic functions of a design.
2. Unit-delay simulation, to check out the timing relationships between AND/OR blocks.
3. High-level simulation.

In the last case, the model is written in a higher-level language. It allows the designer to simulate at a functional level and permits replacement of parts of the model with its counterparts in unit logic. The patterns required to drive the simulator are manually generated and are dependent on the designers' knowledge of the design, using a simulator control language called BDL/C. The output from the simulator is in the form of an "all-events trace," a cycle-by-cycle recording of every defined test point. This data may be analyzed by using available support programs to select and format the output at desired test points in an easily understood fashion.

• *Logic ratification through hybrid simulation*
Hybrid simulation is the term used when higher-level functional simulation results are converted and used to drive the lower-level models of single and multichip sections. This allows an engineer to develop test cases by executing an operation code or program against a high-level model and to use the detailed results to drive the VMS AND/OR logic. As with earlier technologies, the hybrid technique was only effective in the initial design stages, becoming impractical as larger sections of logic were put together. The main advantage of hybrid simulation was that the designer would not have to understand the values of all the inputs into the section of the logic being simulated. By use of hybrid simulation, the designer was able to tie these nets to the cycle-simulation facility names and use the high-level simulator to generate the bit patterns to drive the detailed logic.

Unfortunately, a number of disadvantages were also incurred: the detailed logic took 30 to 40 times the computer time that the higher-level models took, and the storage requirements of the jobs were three to four times as large. Furthermore, due to capacity limitations, VMS was unable to handle the very large number of circuits the designers needed to simulate at a time. To make the simulation fit, and to run for finite periods, the designers had to establish artificial boundaries and correspondences between the high-level and low-level simulation models. Discrepancies in correspondences introduced new errors that could not be detected by unit-delay simulation and complicated the picture further. Thus, we had to find another technique for establishing functional equivalences between models; this is discussed subsequently.

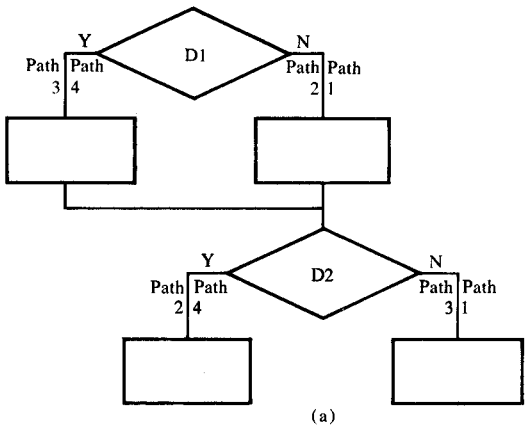
• *Microcode debugging through the microcode debug tool*
Up to this point we have discussed hardware simulation; but since the design contains both hardware and microcode, an appropriate debugging tool, called the *microcode debug tool* (MDT), was also needed. This was developed especially to test horizontal microcode in a software environment. It differs from cycle simulation in that the model is written in PL/S, and it represents the high-level function as distinct from the implementation. Its purpose is to remove gross errors, independent of the hardware design, before the horizontal microcode is merged with cycle simulation, in order to permit a complete functional test of both the microcode and the hardware simulation. MDT can fit in a normal TSO region, consists of an interactive conversational language, and can be run as a background job. It can create and execute command lists containing MDT commands from user-independent data sets. It interfaces directly with all files containing the microcode and test cases necessary to support the simulation runs. It further contains a save-and-proceed facility within TSO to allow the designer to stop at any point in the session and to save the results in one data set, together with the test environments he is using. The main characteristics of the simulator are minimally affected by hardware changes, and it can run a large number of test cases on a large amount of microcode in a very short time. Modular programming techniques used in its design permit it to be used across multiple models with minimal changes. The key feature built into the model is use of the identical naming convention and facility names of the hardware BDL/CS used in cycle simulation.

## Testing

Testing in the past had been plagued by four major problems: 1) The effectiveness of the exercisers used; 2) Eliminating obsolete code within the item being tested; 3) Determining loop conditions within the model being tested; 4) Measuring the effectiveness of test cases.

With DVS we attempted to put into place a system that would not suffer from the above problems, based on 1) A systematic testing approach using building blocks, and a method for tracking coverage; 2) Cause and effect diagrams and functional matrices to determine the test cases to be developed; 3) Automatic rather than manual verification that a test case had completed properly; 4) A test-case library which permitted test-case bucket optimization for regression test-case scheduling.

We further stipulated the following:




(a)

| | TRACE1 | | | | TRACE2 | | | |
|---|---|---|---|---|---|---|---|---|
| | D1 | | D2 | | Path 1 | Path 2 | Path 3 | Path 4 |
| | N | Y | N | Y | D1 = N<br>D2 = N | D1 = N<br>D2 = Y | D1 = Y<br>D2 = N | D1 = Y<br>D2 = Y |
| Test 1 | X | - | X | - | X | - | - | - |
| Test 2 | - | X | - | X | - | - | - | X |
| Cover | X | X | X | X | X | - | - | X |
| Coverage = 100% | | | | | Coverage = 50% | | | |

(b)

**Figure 3** Illustrating coverage with TRACE1 and TRACE2: (a) Sample flowchart, (b) Coverage analysis.

1. Test cases should be prepared concurrently with the design and should be based on the complete specifications provided by the designer.
2. The language for developing the test cases should be the same as that to be used ultimately when the processor was released to the customer.
3. The microcode used to drive the model should be the functional microcode used in the final product after having been tested on the MDT simulator.
4. All test cases should be usable for development test and manufacturing bring-up.

Expanding on the building-block approach, test cases were developed in different stages of complexity. The first stage, *single-cycle test,* is based on the control specifications or flowcharts. These are very basic tests used to check the transfer of information between functions within the models. The next stage, *multicycle test,* checks an operation within a model by executing an operation and checking the data and control flow of the machine. The third stage consists of executing nonoverlapping instructions which drive the model to performing full functions on an individual basis. The fourth stage, program testing, consists of overlapped instructions running together, allowing simultaneous conditions within the model to be activated. The fifth stage consists of interface tests designed to check the functions between individual units and the final stages of the error checkers within the machine to determine that the hardware acts correctly. The combination of all tests is used to achieve the test-case goals for path coverage.

## Coverage determination

*Coverage determination,* the systematic audit of the extent of cycle-simulation flow paths traced during simulation runs, was implemented for the 3081 with two program packages called TRACE1 and TRACE2. The TRACE facility interacts, cycle by cycle, with the cycle-simulation model execution. A pre-analysis is made of the flowcharts by recording all legs of all decision trees to create a maximum table of combinations to be analyzed. The cycle-simulation program records the legs executed when test cases are applied to it.

TRACE1 uses the output of the simulator and a list of all possible "no" decisions, as well as the state of all branch tables covered by a group of test cases or accumulated over a period of time. This provides the designers with an adequate way of measuring the effectiveness of their testing as well as a quantitative way of expressing coverage [10].

TRACE2 analyzes the same data differently because, when decisions are executed, not all legs emanating from the decision block are necessarily tested. It develops a coverage map based on *paths* rather than on *legs*. Figure 3(a) shows a flowchart and Fig. 3(b) the coverage analysis performed by TRACE1 and TRACE2. Note that TRACE1 follows the "no" path out of both decision blocks in TEST 1, and the "yes" path out of both decision blocks in TEST 2. TRACE1 concludes that all paths out of the decision blocks have been tested. TRACE2, on the other hand, concludes that there are two paths from D1 to D2, that the "no, no" and "yes, yes" paths are only half of the ways one can get through to the end, and that the "no, yes" and "yes, no" paths must also be considered.

The TRACE2 program evaluates the cycle-simulation flowcharts for at least two levels and up to five levels of path lengths beyond a decision block. The two methods together allow the engineers to establish criteria for determining when coverage is high enough to stop simulation. The criteria we chose were that 90% of all single-, 70% of all double-, and 40% of all triple-level decision paths had to be covered before software testing could be stopped. On the basis of previous analytic work, we projected that with the foregoing criteria there would be a 95% probability that the design would be free from problems, exclusive of those dependent on data and timing variations.

## Functional equivalences between models

As previously discussed, the problem of establishing the functional equivalency between different levels of models

had to be resolved. At first, hybrid simulation was considered to provide that function. But because of the large amount of computer time required, and because it depended wholly on the completeness of the test cases, a better technique was sought. Thus, SAS, the *static analysis system* program using the Boolean comparison technique, was developed (see Fig. 4). By converting the basic design language for structure (BDL/S, which the engineers used for defining the hardware in automated logic diagrams) and the flowchart language (BDL/CS, which they used to express the functional specifications) to Boolean-algebra equations and then performing the isomorphic check [2], the equivalency could be determined readily.

Since this technique guarantees a 100% equivalency check, the designer is able to relate the coverage-analysis numbers from the high-level simulator and apply them to the detailed implementation to determine the quality of the design. (This approach also identifies clerical errors introduced in the transcription of the design into the automated logic diagrams.) Thus SAS allows the designer to compare a flowchart design against a logic implementation. It validates the fact that the design which was simulated at a high level and on which trace analysis had been run is indeed the same design that is now being released to manufacturing, without making it necessary to re-invoke the costly simulation runs at the detailed design level (VMS). The Boolean comparison technique can also be used between two EC levels of the cycle-simulator model. Another practice is to invoke it before groups of chips are released to verify that the logic implementation is equivalent to the cycle-simulation model.

## Timing analysis

The problems resulting from product variations and the difficulty of problem isolation in LSI hardware require engineers to be able to analyze, in a software environment, the various timing relationships and characteristics in the physical layout implementation of the logic design. Another important ingredient of DVS thus became *timing analysis,* as described by Hitchcock *et al.* [11]. It was developed to verify that there are no long paths or race conditions in the design before the hardware is built. It allows the designer to assess any technology changes which may cause timing problems. Additionally, it provides a data base for identifying "critical paths," it aids in the development of dynamic tests (exercising critical paths), and it ensures that the cycle time is achieved in every possible path through the logic.

Since LSI does not readily permit the designer to observe signals along a path within the physical machine, and since experience has shown that all paths in the design, both short and long, are potentially critical (within tight tolerances), the required delay analysis became quite complicated. Thus, instead of using only a "worst-case design" analysis, we really needed to apply a statistical approach. The delay-equation generator was therefore designed to calculate statistical delays for both the inputs and the output of each block, and the early- and late-switching load characteristics.

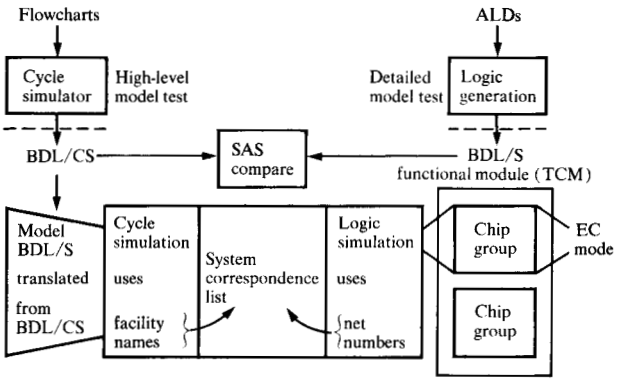



**Figure 4** Model-equivalence-checking methodology.

One of the major factors leading to the success of the timing-analysis subsystem of DVS is that it requires no test patterns to analyze the design. It is an analytical tool which checks the logic paths, measures the slack (timing tolerances in a circuit path) and provides accuracy in the delay calculations. The algorithm used treats circuits and wires as delay elements without consideration of logic functions. The basic concept is to calculate slack, based on delay equations derived from the actual geometric layout and the technology rules, to produce, statistically within three-sigma limits, a timing validation of each path.

## System diagnostics development

Diagnostics, further described by Tendolkar and Swann [12], consist of *verification tests* (VTs) (which are similar to the *fault location tests* used in System/360 [13] and are run on an off-line configuration), and *analysis routines* (ARs) that analyze error status logouts collected during on-line operations.

- *Verification tests*

Verification tests are stuck-fault tests automatically generated by a large-scale test generator [14]. Because of the use of LSSD, the *primary outputs* (POs) are easily found as *shift-register latches* (SRLs). Each PO is traced back through the logic by a program package called *domain analysis aid* (DAA) until the *primary inputs* (PIs) are found. The output of DAA for each PO, based on the trace-back, is called the *segment* for that PO. The segment is a list of nets which exist between the PO and its PIs. Another program has the capability of grouping segments, using a set

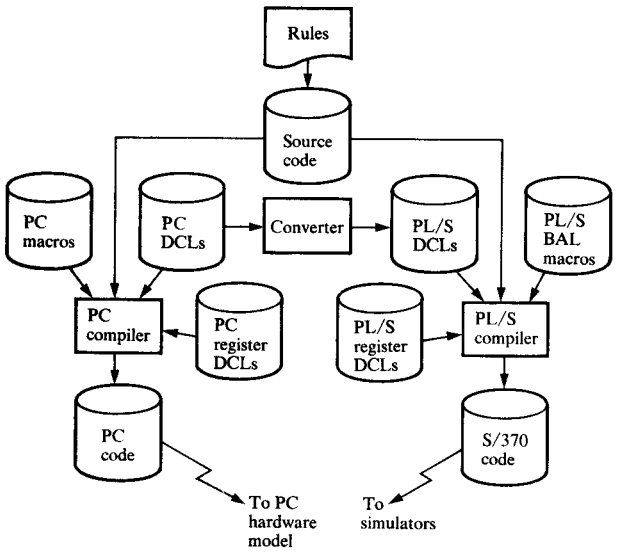



**Figure 5** Diagnostic control program debugging paths. (Note: BAL = Basic Assembler Language, DCLs = DECLARE statements, PC = Processor Controller.)

of given criteria, such as common scan-ring-accessible POs or PIs, common circuit packages (wholly within one TCM, or within one TCM board, etc.), and the number of POs desired to be tested simultaneously. Once the criteria are fed to the program, these segment groups may be formed from the logic data and passed on to the test-pattern generator, which tries to develop patterns to exercise all the PIs to test the group of segments extensively. The patterns and the logic are then fed into a fault simulator, which analyzes the test coverage provided by the patterns generated. The output of this step is a data set from which data are extracted for use by a test-case compiler which generates the machine-readable test cases to be used by the 3081.

In order to obtain a high level of *field-replaceable unit* (FRU) coverage, the test cases are developed to run in a system environment using the real TCM boards, cables, TCMs, voltages, and clocks of the system, first in manufacturing final test and later in the field. This environment allows the VTs to find problems which may pass undetected on a unit tester, since the unit tester can only check a module by itself, using the correct voltages, clocks, and data supplied by the tester, rather than by the physical bounds of the system environment. A TCM in a customer environment may not match the functional specifications exactly but still may lie within the tolerances of its design criteria.

- *Logout analysis*

*Analysis routines* (ARs) consist of a group of routines run in the processor controller which analyze the error logouts and data resulting from checking-circuit-detected "red-light" errors on the 3081 system. The product of this analysis is a "FRU call" displayed to the customer, while he continues to run his own applications, after the logout has taken place. A technique used in the analysis is called *intersection isolation* [12]. It is a comparison between single, independent logouts on multiple errors, to find a common FRU group.

The development of logout analysis as part of DVS came about because ARs originally could only be tested on a hardware machine. This imposed two problems on the diagnostic engineers:

1. They had to wait until a functional hardware model was available (at the end of the development cycle).
2. They had no simple way to inject errors into an LSI hardware model (and thereby systematically prove that the ARs are effective).

The diagnostic engineers were also affected by inaccessibility of hardware test points for bugging in the test-case validation process; also, the expected high reliability of the finished hardware did not offer them many natural test cases. The solution was to add fault injection to cycle simulation to permit diagnostic engineers to simulate hardware errors and to generate hardware logouts like those the actual machine would produce. By this means, the diagnostic engineers had available to them the first stage for validating the AR process on a software model. The next stage was to convert the diagnostic control program and the ARs into a format capable of being executed as part of the software model. This was accomplished by having the diagnostic control program written in a high-level language (PL/S) which could be compiled into both System/370 and Processor Controller code (see Fig. 5). Once the compiler was debugged, we could execute the diagnostic control program on a System/370, independent of Processor Controller hardware. By developing the ARs as elements of a table, we were able to execute them also in the System/370 environment. We had thus created a fully operational capability to test ARs early in the development cycle.

## Verification experience

Before the 3081 development, in addition to the simulation model and its associated test cases, another set of items discarded early in the development schedule, after a one-time-only use, were the numerous test cases which were only designed for a particular level of hardware bring-up. The 3081 testing philosophy was to use all tests for regression testing at all levels. We therefore needed to create and maintain a test-case library.

Modifications based on experience have to be made from time to time; this project was no exception, as was demon-

strated in the area of timing analysis. On the basis of previous design histories [15, 16], we had started with the assumption that only the "critical" paths would contain timing errors; this gave us one set of error numbers. After running timing analysis, we discovered that timing errors could occur in any path. This gave us nine times as many errors as predicted. We also learned, from experience with a five-year-old production system which suddenly began to exhibit numerous timing problems, that timing errors occurring anywhere in the design could not be tolerated. These problems were traced to a shift in a circuit fabrication process. The resultant deviations were still within specifications, but the errors occurred within various paths in the machine once the newly fabricated circuits were plugged. Naturally, we changed our design methodology and error-projection rates, and with our new statistical approach to timing analysis we expect that we will have eliminated this potential problem in the future.

Another interesting experience which differed from previous machine design histories was that, by the use of flowcharts in the early stages of the design of the 3081, numerous interface problems were uncovered. These problems used to be found late in the design cycle, when the boards and gates were wired together by previous design-automation systems [17]. They used to delay that part of the development cycle and accounted for a much higher cost than we experienced with flowcharting.

• *Error discovery vs. projections*

Figure 6 shows the projected number of errors based on the modified design estimates. From the slope of the curve, one can determine the periods when the designers were rapidly discovering design problems and where they were most efficient in their simulation efforts. The figure clearly shows when the designers should have stopped functional simulation and should have started timing analysis. The effect of introducing new functional problems as a consequence of eliminating timing problems is also clearly illustrated, usually following in three-month cycles. The other interesting result illustrated is that, with each pass of the function/timing iteration, the number of problems discovered decreases. When the design trend stabilized, and as soon as the coverage analysis showed a 90% coverage of single paths after decision nodes, we would authorize hardware construction. This does not say that all design problems were eliminated at that point, but, on the basis of the design history, we expected the hardware to be operational and to require a reasonably low number of ECs to correct the remaining errors. The general strategy, based on the effectiveness of the tools, has indicated that the use of cycle simulation and SAS makes it possible to eliminate unit-delay simulation completely. Walk-throughs and inspection [18] also are effective tools for discovering design problems early.

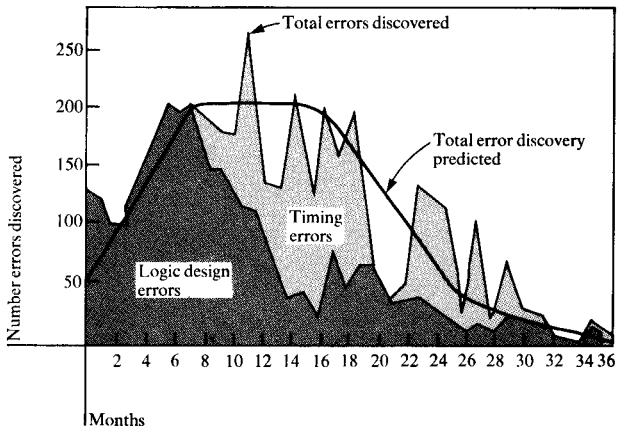



**Figure 6** Error discovery projections with and without DVS.

• *Hardware correlation to models*

The 3081 project developed a three-stage bring-up plan, consisting of module test, subsystem test, and uniprocessor test (single central processor active). These entities were built as mirror images of the simulation models discussed previously. This was the first step in positively evaluating the values of DVS. Figure 2 illustrates the test hook-up.

*Module test*

Module test consisted of a host system connected to a prototype of the Processor Controller (PC) [19] connected to an engineering test box which represented the interface between the PC and the module. This interface controlled a TCM that could be uncapped and probed with servo-controlled probes.

Module test proved that, through simulation efforts (a combination of VMS, cycle-simulation, and SAS), most functional design problems had been removed prior to hardware construction. It also showed that the automatic test-case-generating system used for simulation input would also identify manufacturing defects. Test-case migration between software and hardware models was made possible through this remote-debug facility. The number of test cases that needed to be modified or corrected due to inconsistencies between the models and the hardware amounted to less than 1%. (We had expected it to be 10%).

*Subsystem test*

The second level of testing was subsystem bring-up (Fig. 2), which consisted of three dependent models, each consisting of a fully plugged TCM board connected to an adapter (MSSA [19]) which was connected to the Processor Controller prototype and then back to a host processor. The interfaces between the TCM board and other parts of the 3081 system were emulated with test boxes.

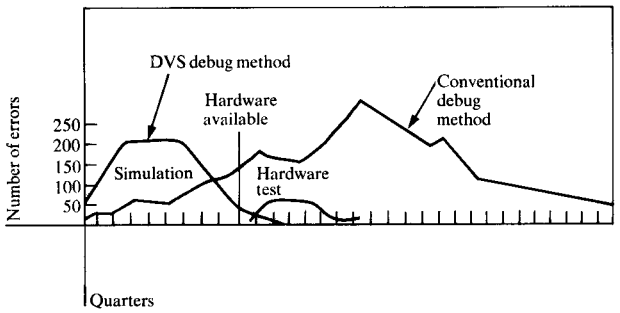



**Figure 7** Design error projection *vs.* actual discovery through DVS.

The results of this test indicated that the dynamic problems encountered (running at machine clock rates) were equivalent to the number predicted by the error-projection model. In fact, due to a delay in hardware delivery, additional software simulation was performed, and it turned out that two-thirds of the problems expected were removed before the hardware testing began. This also helped validate the assumption that dynamic problems could be discovered through DVS, especially since timing analysis is part of the verification procedure.

The subsystem test validated the approach of using a high-level model for microcode simulation, then being able to migrate the same microcode to the intermediate cycle-simulation model to test it as a complete entity, and finally applying it to the hardware machine. In fact, we discovered no problems when the hardware/microcode interface was tested after it had been completely tested earlier in a pure simulation environment. We also proved that we could migrate in the reverse direction: we could take test conditions from the hardware and apply them to the simulator, thereby identifying the cause of some hardware problems. Engineers used this capability of visibly tracing and dumping a good level simulation and comparing it against a physical machine with hardware failures, thereby identifying TCM and TCM-board failures.

*Processor test*

The third level of testing was to build a complete processor from the subsystem parts and to test it as one entity by simulating various customer environments. The purpose was to check out the subsystem interfaces and to verify more subtle interactions than could be run on a subsystem basis. We found that the number of errors discovered in this phase matched the number predicted for this environment. More than 60% of the errors uncovered were in the interfaces between subsystems.

- *Effects on the hardware designer*

With LSI, the designer was forced to become both a software and a hardware engineer. The LSI designer must have a dual background, both in logic design and in programming, generally needing less skill than before in physical hardware debugging techniques but more knowledge of system programming and software debugging techniques. The designer now uses a broader spectrum of techniques to create an error-free design.

- *Product cycle*

From previous discussions, it is clear that, with validation techniques, normal development schedules are significantly reduced. Any conventional hardware debugging, as shown in Fig. 7, is a serial process at the beginning of the test cycle. Only after a level of functionality has been achieved can it become a parallel operation. The DVS package, on the other hand, allows the designers to apply many test cases simultaneously to the design. It depends on, and is limited by, the computing resources the designer is willing to invest in simulation in order to achieve a certain quality before the hardware is assembled for the first time. Hardware debugging requires hardware rework after the corrections have been designed. This creates physical design, release, and manufacturing delays that can take weeks, and which introduce additional errors (predicted as much as 1/3 more than with the use of DVS). The changes required in a simulation model can be made immediately and interactively, and they can be tested subsequently by continuing with the simulation runs. Compared with a pure hardware debug technique, the DVS method improved the product schedule by 66%.

## Conclusions

In general, our error-prediction and verification plans were sound, and in most cases simulation worked out better than expected. The number of design errors discovered early in the design were proportional to the number of test cases that were developed and the amount of computer time spent on running the verification package. This project has demonstrated that the use of multilevel simulation has eliminated and simplified the development of test cases and test programs to be used in hardware check-out. These programs could be used on both the software models and on the hardware. Furthermore, the quality of the tests could be improved by the use of coverage-analysis-measurement techniques. We also demonstrated that migration of test results back from the hardware could be used to discover design problems on the simulator that were not easily addressed on the hardware alone. We further demonstrated that logic simulation, though cumbersome at times, was less expensive than physical hardware debugging. The cost of building a model, reworking it, and manning it two to three shifts a day does not come close to the much lower costs and turn-around times of simulation runs, which in most cases provide the same or better results. The models also have the capability of setting up customer environments or scenarios which would otherwise require extensive I/O attachments to the hardware for testing them.

On the basis of the experience gained on this project, the author believes that it is impractical to design any large-scale machine in LSI technology without the use of a design verification system which can remove 90% of the design bugs before the hardware testing of the machine begins. This becomes even more important as the technology becomes denser and as the repair and rework time is further extended due to the more complex nature of the processing problems of creating new chips, TCMs, and TCM boards after an engineering change.


## Acknowledgments

The author acknowledges the technical and management contributions of R. W. Walton and B. R. Golnek, for playing the key roles in the development of the 3081 DVS. In addition, the author wishes to recognize the contributions of key technical people on this project, such as G. L. Smith and J. E. Quigley, for timing analysis, P. A. Torney for SAS, T. R. Ferrara for VMS, B. D. Watt for MDT, M. M. Snow for VT, R. L. Price for cycle-simulation, M. C. Moore and T. S. Kho for TRACE, and W. S. Benson, Jr., for the error-prediction model. (VMS, SAS, TA, and the test generator for VT are, of course, adaptations of programs developed by the IBM Engineering Design System group.) To C. E. Sioleski, my secretary, who helped prepare the many presentations from which this paper was derived, and to R. J. Preiss, who organized and edited the manuscript, the author expresses his deep appreciation.


## References and note


1. J. P. Roth, *Computer Logic Testing and Verification,* Computer Science Press, Potomac, MD, 1980.
2. Gordon L. Smith, Ralph J. Bahnsen, and Harry Halliwell, "Boolean Comparison of Hardware and Flowcharts," *IBM J. Res. Develop.* **26,** 106–116 (1982, this issue).
3. Glenford J. Myers, *The Art of Software Testing,* John Wiley & Sons, Inc., New York, 1979, Ch. 4, pp. 36–76.
4. "Horizontal" is differentiated from "vertical" microcode usually by the complexity of the parallel and unrelated operations in the former which can take place in a single cycle.
5. 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, June 1976, pp. 48–53.
6. E. B. Eichelberger and T. W. Williams, "A Logic Design Structure for LSI Testability," *Proceedings of the 14th Design Automation Conference,* New Orleans, LA, June 1977, pp. 462–468.
7. M. S. Pittler, D. M. Powers, and D. L. Schnabel, "System Development and Technology Aspects of the IBM 3081 Processor Complex," *IBM J. Res. Develop.* **26,** 2–11 (1982, this issue).
8. R. N. Gustafson and F. J. Sparacio, "IBM 3081 Processor Unit: Design Considerations and Design Process," *IBM J. Res. Develop.* **26,** 12–21 (1982, this issue).
9. J. S. Jephson, R. P. McQuarrie, and R. E. Vogelsberg, "A Three-Value Computer Design Verification System," *IBM Syst. J.* **8,** 178–188 (1969).
10. T. S. Kho, "Measuring Simulation Effectiveness in the Design Verification Process," *Proceedings of the 1980 IEEE International Conference on Circuits and Computers, ICCC80,* Port Chester, NY, October 1–3, 1980, pp. 921–923.
11. Robert B. Hitchcock, Sr., Gordon L. Smith, and David D. Cheng, "Timing Analysis of Computer Hardware," *IBM J. Res. Develop.* **26,** 100–105 (1982, this issue).
12. Nandakumar N. Tendolkar and Robert L. Swann, "Automated Diagnostic Methodology for the IBM 3081 Processor Complex," *IBM J. Res. Develop.* **26,** 78–88 (1982, this issue).
13. W. C. Carter, H. C. Montgomery, R. J. Preiss, and H. J. Reinheimer, "Design of Serviceability Features for the IBM System/360," *IBM J. Res. Develop.* **8,** 115–126 (1964).
14. P. S. Bottorff, R. E. France, N. H. Garges, and E. J. Orosz, "Test Generation for Large Logic Networks," *Proceedings of the 14th Design Automation Conference,* New Orleans. LA, June 1977, pp. 479–485.
15. D. J. Pilling and H. B. Sun, "Computer-Aided Prediction of Delays in LSI Logic Systems," *Proceedings of the 10th ACM/IEEE Design Automation Workshop,* Portland, OR, 1973, pp. 182–186.
16. A. Nádas, "Random Critical Paths," *Proceedings of the 1980 IEEE International Symposium on Circuits and Systems,* Houston, TX, 1980, pp. 32–35.
17. 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,** 631–646 (1981).
18. M. E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Syst. J.* **15,** 182–211 (1976).
19. John Reilly, Arthur Sutton, Robert Nasser, and Robert Griscom, "Processor Controller for the IBM 3081," *IBM J. Res. Develop.* **26,** 22–29 (1982, this issue).



*Received August 19, 1980; revised July 2, 1981*



*The author is located at the IBM Data Systems Division laboratory, Poughkeepsie, New York 12602.*