Arnold S. Tran
Richard A. Forsberg
Jack C. Lee

A VLSI Design Verification Strategy

With the ever-increasing density, development cost, and turn-around time of VLSI chips it becomes increasingly important to have a design verification methodology which enables first-pass chips to be fully functional. The strategy discussed in this paper exploits the best attributes of the two traditional methods of design verification (i.e., software simulation and hardware modeling). Software simulation was chosen for its capability in the area of delay analysis and early functional checking. An automatically generated nodal-equivalent hardware model was built to provide the vehicle on which exhaustive functional checking could be performed. The model also operated as early user hardware on which functions such as operating systems, I/O adapters, and a floating-point feature could be tested. A technique known as interface emulation was used on certain well-defined subsystems to facilitate a shorter verification schedule through parallel debug efforts.

Introduction

Achieving a first-pass [1] fully functional design in the current world of Very Large Scale Integration (VLSI) has become an increasingly difficult task due to the highly complex systems and large logic circuit counts in state-of-the-art technologies. In the *VLSI microprocessor* module described in the paper by Campbell and Tahmoush [2] there are approximately 15 000 circuits and an imbedded 1K × 50-bit ROS contained on four custom bipolar chips. In the storage control unit there are another 5000 circuits contained on five gate-array chips. With relatively long turn-around times for the custom chips and the high cost involved in their design and manufacture, it becomes even more important that first-pass chips be fully functional.

A design verification strategy was developed to achieve the desired goal. The approach used has some basic concepts which, when rigorously applied, ensures not only a fully functional design but also VLSI chips which are logically equivalent to the various models used to verify the design. These concepts are the following:

- Using single-source data bases.
- Minimizing manual intervention.
- Using software simulation for early functional checking and critical-path analysis.
- Using a hardware model for exhaustive functional testing.
- Using interface emulation on well-defined subsystems.
- Using the hardware model to debug the test bed and tools prior to the arrival of the VLSI hardware.

This paper describes a comprehensive design verification strategy which was used successfully in the design of the referenced microprocessor system [2]. However, the strategy developed is applicable to most digital systems of various types and sizes. Methods for the development of a software simulation and the automatic generation of a hardware model are described. The tools necessary to complete the design verification are essential and are described in this paper.

The first section describes the overall strategy which was employed. Next, the generation and description of the hardware model are discussed, followed by a section on the software simulation. Sections on the functional test programs, processor test station, and the I/O test adapter describe the tools used to drive the hardware model. Next, a section on interface emulation describes how a well-defined subsystem may be verified in parallel with the main system. Finally, sections on detailed timing analysis and engineering changes complete the overall strategy.

Design verification strategy

The design verification strategy was to employ a comprehensive set of tools which would detect as many errors as early as possible, thereby providing a reasonable assurance that the manufactured hardware would function as intended. All of these tools use the data bases from which the hardware was eventually manufactured. Both software and hardware simulations were performed. Functional testing was provided for both software and hardware models, in varying degrees of rigor, from basic operations to exhaustive programs, with input/output equipment attached. Detailed timing analysis was performed to ensure proper performance at rated speed. To achieve the required schedule, the microprocessor module and storage units were developed and verified independently; there was minimum opportunity for joint subsystem verification. Finally, engineering changes were strictly controlled to guarantee the equivalence of the verified and manufactured designs.

Random logic, *read-only storage* (ROS), and *programmable logic arrays* (PLAs) are used in the processor design. The personalities of the ROS, PLAs, and logic interconnections are contained on data bases. The term "single-source data bases" refers to the method in which the common data bases for the VLSI chip design are used to generate both the hardware model and software simulation. Through this procedure, any change made to a VLSI data base directly affects the software simulation and hardware model. Use of the same data throughout all stages of the design provides continuity, familiarity, a common set of tools for each type of data (*i.e.*, ROS, PLA, random logic), and a guarantee that the design which was built was the same one that had been tested and verified.

The data base for registers and random logic was in the form of a standard logic-description language used throughout IBM, known as BDL/S [3]. Logic macros defined in this language were verified individually by a separate group which was responsible for their physical and electrical design. This provided a set of logic primitives with assured functions from which logic designers could choose. This also allowed logic and circuit design to proceed in parallel, a necessity in light of the aggressive schedule.

The control store of the processor (ROS personality) was maintained in a single, specially formatted data set. A set of program tools was developed which allowed rapid changes and generation of an easy-to-use listing.

Programmable logic arrays were handled similarly. Special programs were used to enter PLA data, check for logic redundancy, and aid in hardware model personalizing.

The central strength of this methodology lies in the fact that the same data base, for each data type, is used during all phases of the design. This common data base was the source from which ROS and PLA personalities and logic models were created for software simulation. It was the source from which hardware modules were personalized when the hardware model was being built. It also served as the source of the design from which the VLSI chips were manufactured. This commonality guaranteed that the VLSI hardware which was built was identical to the hardware model which had been tested and verified.

Throughout the generation of the hardware model there are many steps in which mistakes can be introduced by human error. To ensure against these errors, the process of building the hardware model was fully automated. Automatic conversion programs convert the VLSI design to a model consisting of TTL (transistor-transistor logic) gates, PLA (programmable logic array) modules, and EPROMs (erasable programmable read-only memories). To illustrate this type of error, suppose during the process of personalizing a PLA module the operator types a wrong key. This results in a discrepancy between the VLSI design and the hardware model. To prevent such an occurrence, an automatic personalizing tool was designed. This tool reads the PLA data base, converts the data to the required format, and controls the burn-in tool. Thus, all manual intervention is eliminated from this process.

It is essential to the design verification strategy that the model be an exact logical representation of the VLSI design. This is ensured by the automatic procedures developed, but another source of error still remains. After the model is debugged and modified in the lab, there is no longer a guarantee that the model in the lab is equivalent to the VLSI design. To correct this, a method of reconversion is used whereby the model is recreated from the VLSI data bases. The reconversion can be performed on the whole machine, or only on certain areas if desired (*e.g.*, only one chip of the final system might be converted). The reconverted model is reverified with a complete regression test. This guarantees that the VLSI data bases are correct.

The software simulation provides an excellent medium for early functional checking and verifies the critical delay paths. Before the initial VLSI logic is released for use in building the hardware model, it is checked via simulation to be functional to some degree; the checkpoint is to verify the system reset function. By doing this, time can be saved in initial bring-up of the hardware model. However, the major role of the simulation exists in the area of critical-path analysis. Since the hardware model does not run at the final product-rated speed and is built from different technologies than the final product, it is not useful for delay analysis. The software simulation is capable of exercising the machine in a nominal or worst-case environment. For delay calculations,

the actual physical data from the VLSI chip design are used. These data consist of line lengths and line capacitances.

Simulation of the design was further extended beyond the processor. A software model of the *random-access memory* (RAM) was developed to simulate the local high-speed store of the processor and a similar model was used to test the I/O channel. Later, the processor model was joined with the storage subsystem model and combined system checking was performed by running some small functional test programs.

Hardware simulation by means of a functionally equivalent hardware model was the backbone of the logic design verification methodology. It provided a practical way to accomplish the exhaustive testing necessary to ensure correct operation. With the use of the functional test programs (FTPs) the machine can be exhaustively tested logically. Approximately 300 000 lines of code are contained in the FTPs, and many times this number of instructions were executed due to the iterative nature of the FTPs. In addition to these programs, the operating systems and other system programs were run. Product I/O adapters, a floating-point feature, and an I/O test adapter were just a few of the devices that were attached to the hardware model. This type of functional testing could only be performed using the hardware model. One of the most important concepts in this strategy is the use of the hardware model to debug the test bed and tools prior to the availability of the VLSI hardware.

If the methodology only used a software simulation and a hardware model was not available, the VLSI chips would have to be tested in a test bed which had not yet been debugged. In addition, all the software and hardware tools needed would not have been tested. The hardware model is the vehicle on which the test bed and all the tools may be debugged long before the VLSI hardware is received. The same hardware and software tools which were developed and used on the hardware model are fully functional when the VLSI hardware becomes available, saving much time in the development schedule.

The hardware model was automatically generated from the VLSI data bases using the logic conversion programs previously described. Various technologies were used in building the model. Two types of PLA modules were used to accommodate the custom PLAs in the processor. Two medium-scale integrated chips were built to model the LSSD (*level-sensitive scan design*) [4] latches. This helped to increase the circuit density on the model card. The hardware model was controlled via a test adapter which used the SPCTS (*single point of control test station*). Through use of SPCTS, the test bed control became fully automated and was capable of running overnight tests. An error log-out allowed analysis of errors encountered.

An equally important aspect of this methodology is the ability to accomplish verification on each machine subsystem without excessive dependencies on the others. This provides an important degree of flexibility when scheduling the large number of tasks required in such a design effort. Through the use of separate hardware models just of the adjacent subsystems and/or their interfaces, these independent verifications can be accomplished.

The processor subsystem required models of the adjacent memory and I/O subsystems. During software simulation, the memory was modeled with a VMS/PL/S [5] behavioral model written especially for this purpose. The equivalent function was provided during hardware model testing by a special "pseudostorage." It was designed using a storage technology which could be made to function according to the planned interface with a minimum of effort. The "pseudostorage" played a major role in processor testing while the planned storage subsystem was still under development.

To verify the design of the storage control card, models of both the processor and the storage array cards were required. During software simulation using VMS [5], the storage array was modeled with a behavioral model written in a hardware description language used within IBM. In the hardware model, the storage array was a combination of the planned RAM module and a TTL equivalent of the LSI array support logic.

Hardware model

The hardware model is the vehicle used to design-verify the system. The various functions of the machine can be exhaustively tested with the use of the functional test programs. The model runs at approximately one-fifth the real processor speed; at this speed, exhaustive testing can be performed which would be too costly and time-consuming for software simulation. Each VLSI chip was modeled on a single board, with eight cards per board. A total of four boards and twenty-nine cards were needed for this processor [6].

Problems associated with building a hardware model and ensuring a one-to-one relationship with the VLSI design can be eliminated by using the automatic procedures described herein. The key concept was using a minimum of manual intervention in generating the model, thereby ensuring the relationship. This was realized with the use of automatic conversion programs, automatic card wiring, and use of data bases common to the VLSI design (see Fig. 1).

The conversion program operates on the user-supplied data set that contains the VLSI logic design. Output of the programs represents another data set that contains both the VLSI and converted logic. The output includes logic prints, module placement, and card-wire lists. The wire list is used

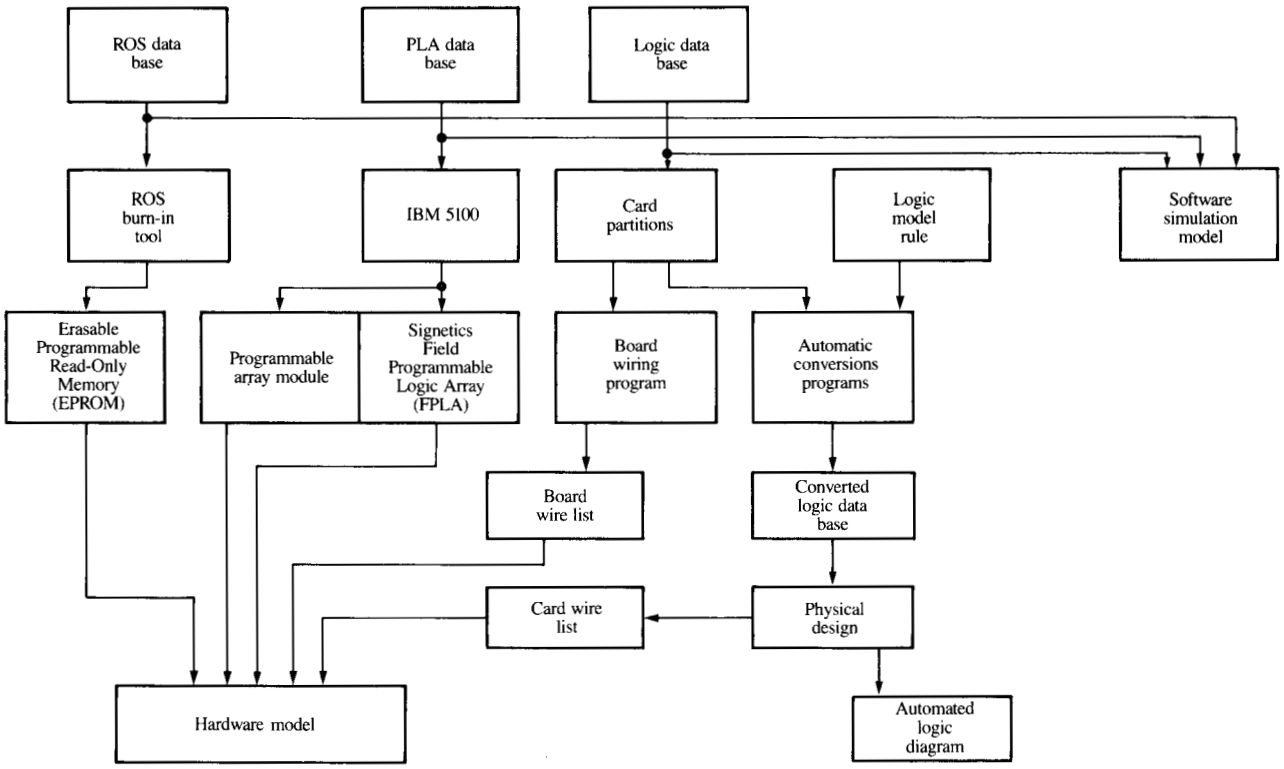

Figure 1 Hardware model generation flow.

for automatic wiring on the wire-wrap machine. Many technologies may be converted to the modeling technology; the conversion program is not limited to a single technology.

The conversion process using VLSI logic design as input is performed through the following steps:

- Space is reserved on the *automatic logic diagrams* (ALDs) for printing the expanded logic.
- The Engineering Design System (EDS) [7] expands the logic from VLSI to the modeling technology using a set of transformation rules defined for the technology being converted.
- Card I/O pins are assigned for inter-card wiring.
- Entry blocks or buffer blocks are assigned to all card input nets to prevent electrical overloading.

Each modeling logic circuit is processed according to rules contained in a conversion table that determines which type of TTL module is to be used in the hardware model. For example, the correct circuit is selected depending on the number of inputs wired, the circuit is changed to an open-collector output if two or more outputs are dotted together; selected circuits are left unchanged if they have position-sensitive inputs such as clock inputs to a latch; and a noninverting "AND" circuit with only one input wired is removed and replaced by a wire. The converted logic is updated to include the physical location, circuit portion, and module rotation for each logic statement.

EDS PD (Physical Design) is used to produce a list of modules used and their location, a list of modules with unused circuits, a list of card I/O pins used, a list of wire nets, ALD sheets, and a module map which shows module placement on the card. The output, a from-to wire list, is used for the automatic wiring tool.

The card used allows placement of 14- or 16-pin modules, 24- or 28-pin modules, IBM MSI [8] modules, and 25-mm PLA modules. The card holds a maximum of 90 14- or 16-pin TTL modules. The TTL count is reduced when IBM 25-mm or other-sized modules are used. Any card may be used for modeling. A card rule (defining the physical parameters of the card) and module placement algorithms are required.

The modules are placed on the card in an order determined by a placement-sequence table. A single placement-sequence table is generated manually for the cards used. When more than one module-placement table refers to the same physical card location, the first module is placed and the other

modules use the next entry in their table. A different set of module-placement tables would be used if other physical card types were used.

The card pin connections are assigned automatically, and this information is placed in the logic data base, by the conversion program. Nets that require card pins are identified by a program.

Card I/O pins are sequentially assigned by the conversion programs. When a card is converted a second time, it is sometimes desirable to be able to plug the new card in the old position on the board. A HOLDPIN program holds card pins and is used to set card pins equal to the card pins from an earlier design level.

Since a VLSI design usually is converted to more than one model card, some manual preparation of the logic is needed prior to conversion. First, a copy of the VLSI logic is made. Then the logic is partitioned into several data sets so that the logic contained in each data set can be packaged on a card. The wires (card pins) between the partitions are identified via a card-pin program. About 60% of the card area is used; the area remaining allows for logic added by conversion programs and for engineering changes. The conversion programs create logic by adding pull-up resistors to internally dotted nets, and by powering some card inputs.

The technology chosen to model the processor and storage subsystems was TTL (7400 series) and other TTL-compatible devices. This technology was chosen for its high noise insensitivity, high speed, and single power supply. Two MSI (medium-scale integration) parts were designed to model the LSSD latch and register macro. PLAs were modeled using FPLAs [9] and an IBM programmable-array module. This was done to accommodate the modeling of the custom-size PLAs in the design. An EPROM was used to model the 1K × 50-bit ROS. This made it simpler to modify the processor microcode during debug.

Several tools were developed to personalize PLAs [10, 11] and ROS [12]. The PLA data set was transferred to an IBM 5100 computer controlling one of the two personalizing tools. A program was written to convert the VLSI data format to that required for the PLA tools. In a similar fashion, the ROS data were transferred to a separate system which took the bit patterns and controlled the ROS burn-in tool.

As the VLSI chips become available, each is substituted one at a time for its model equivalent. In this model, one chip replaced one board (eight cards) of logic. When complete, this procedure results in a model which consists of all VLSI chips. After the test bed contained all the VLSI chips, an increase in speed was achieved. However, this configuration did not obtain the full speed of the final package.

Software simulation

The software verification of any design in a custom VLSI environment is performed on three levels, each equally important. These levels are verification of the logic function, verification of clock timing delays, and verification of the macro rules library. The need for logic verification is self-evident. Delay analysis is critical because, to date, there does not exist a hardware model that can truly duplicate the timing and performance of VLSI technology. Also, in a custom design environment, the line length, width, and placement can have a tremendous impact on delays [13]. Finally, the verification of the macro rules library is unique to custom design. Since physical design relies heavily on automatic checking, and since manufacturing requires test patterns to produce good parts, the rules library plays an important role in the design process. In a gate-array technology, these rules would have been verified before release for general use. However, in a custom design environment, these rules are written and are constantly updated by the in-house designers, and therefore require the added verification of software models to guarantee their accuracy.

The software model is a simplified system model of the total machine and has the following structure. It consists of logic models made up of primitive functions such as AIs, XORs, and latches along with behavior models for representing PLAs, RAM, and main storage. The total structure has an equivalent logic of over 20 000 circuits. In addition, an adapter model was created for extended testing. A high-level control language is used to drive the system model, and simulation is performed by a standard IBM simulator. It should be emphasized that the source data bases used to create the software model are the same data bases used to generate the hardware model and the VLSI physical design (see Fig. 2). Certain tools were used to facilitate debug, including printed and interactive timing charts, other EDS statistical tools, a TRACE feature of PLAs, and automatic conversion of simulation runs to test patterns for dc and ac testing. Even though the total model is unique in its application and function, the whole system conforms to IBM design methodology as prescribed by EDS. This is especially helpful in the later manufacturing, release, and testing stages of the final product.

A few years ago, IBM introduced the idea of *structured processing* in its design methodology [14]. This was the keystone to many improvements in the areas of simulation, behavioral modeling, and test-pattern generation at the higher integration level. The goal of structured processing is to design and process using the building blocks that were created at the lowest level of design. This saves the large CPU time required for model re-build at each higher package level. Another advantage is the ability to mix behavioral models and micro-block logic models into a hybrid model.

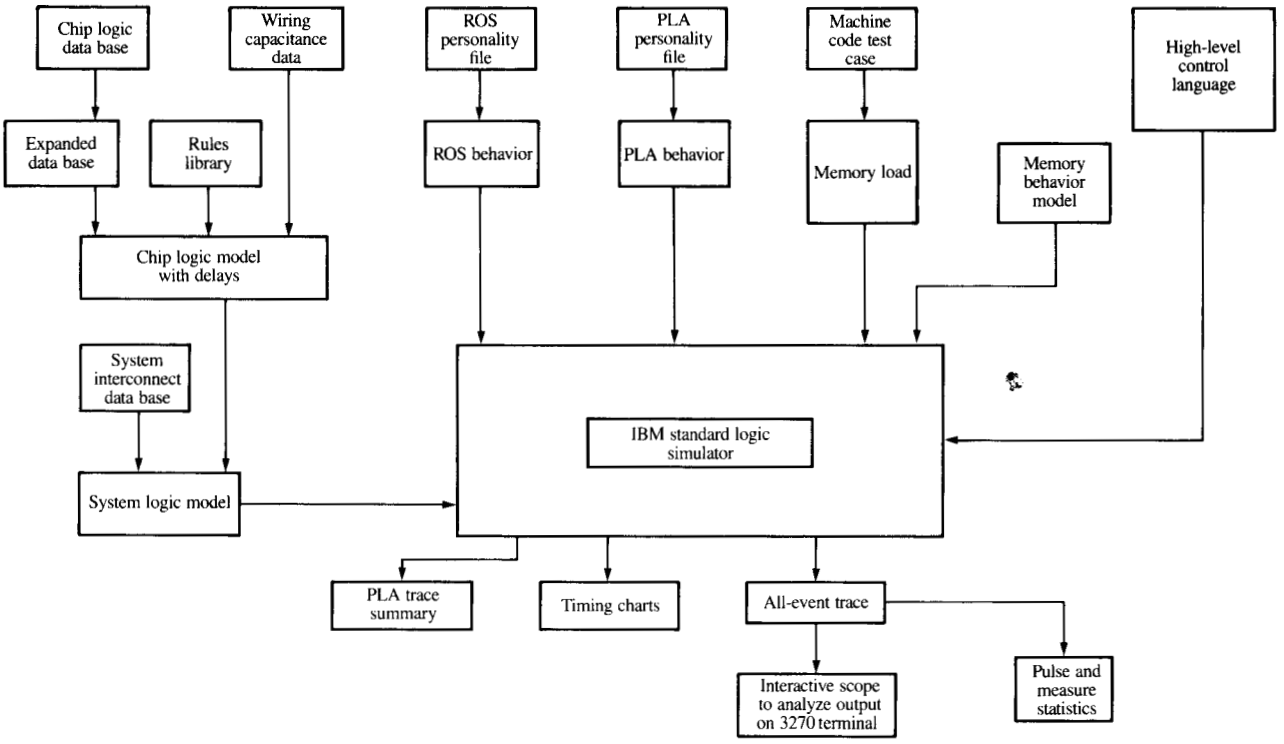

Figure 2 Flowchart of software system simulation model.

This gives the designer the flexibility of using a faster behavioral in place of micro logic for all except the area of design being worked on. Also, the behavioral allows early simulation to take place before the micro logic design is completed. For example, the processor made use of a behavioral model for the storage subsystem at the early stage of design; once the detailed storage subsystem model was debugged, the behavioral model was replaced with the detailed model and testing continued.

The software system model was created over a period of time concurrent with the design and debug of the hardware model. The initial simulation was performed on functional units with unit delays [15]. As the pieces were interconnected, the accuracy of the delay simulation was improved. Nominal delays were first calculated using equations supplied by the technology designer. Critical nets were analyzed with ASTAP [16] and the delay numbers were added to the model. Finally, as the on-chip wiring information became available from the physical design, it was incorporated in the final delay calculations. Worst-case delays were generated by scaling the nominal delays with a multiplier specified by the technology designer. This process represents stage-of-the-art logic delay modeling and simulation [17, 18].

The test cases used to exercise the system model are a critical subset of the functional test programs used by the hardware model. Because of the limitations on the size and run time of a simulation model of this scale, the test cases were reduced. This, however, does not limit the effectiveness of the model to duplicate any unique test condition under investigation. The model is able to exercise each instruction once (to prove it works), but exhaustive testing is left to the hardware model [19].

The software simulation structure can be easily adapted for creating ac functional testing at the higher package level, where problems have come up in the past due to lower test coverage. This functional testing leads to a simplified manual test-pattern generation procedure, in addition to the automatic test-generation scheme [20] to obtain the required test coverage.

Functional test programs

Functional Test Programs (FTPs) are programs designed to verify that the processor conforms to design specifications. Design requirements selected for the FTPs ensure high confidence that the tested processor is fully functional. Such an exhaustive level of testing requires that the FTPs run on a vehicle with an execution speed approaching that of final hardware. Special features of the functional test programs allow them to determine if design objectives have been met.

Each functional test program verifies a particular aspect of the processor design, for example, the priority level switching mechanism. In the FTP, a set of driver routines

precedes one or more test cases. The driver routines, under control of data bits in a control table, can vary the execution of the test case(s). For example, the test case may be executed using different register sets, on different priority levels, from different locations in main storage, and using different memory addressing modes. Each of these options may be independently varied by setting the value of certain flags in the FTP control table.

Within each test case, variations of the basic test are also stored in a table format. For example, in testing a Load Register Immediate instruction, the test table might contain all possible variations of such an instruction, in which both register operands and immediate fields are cycled. Parameters from the test table are dynamically moved from the test table into the instruction stream of the executing test case. Parameters from the test table are also used to check the results of each test.

The table-driven nature of the FTPs has three main advantages. First, the code is efficient. Since test parameters are used to self-modify executing code, relatively little storage is necessary to contain an exhaustive test. This same feature increases programmer productivity. Second, the FTPs are flexible. By altering the constants that define the test tables, an FTP may be set by the user to run using a subset of all its possible test variations. This allows the FTP to be used as a debug tool during initial hardware bring-up, and as a regression test tool after installation of a hardware change. Third, the structured design of the FTPs allows them to be run under an automated test system. Each test case signals completion or failure using the same routines, and the main storage location for control tables is the same in all FTPs. Therefore, using a table parameter, FTPs may be set for maximum testing duration and left to execute unaided in an automatic test station.

Processor test station

Functional test programs run under control of a *single point of control test station* (SPCTS). The SPCTS performs three basic functions. First, it stores the FTP's object code and tables on a disk file. Second, it controls execution of FTPs in the processor either under user control or in an automatic test environment via a command list facility. Finally, it allows access to internals of the processor under test via a test adapter and its CRT terminals. The test adapter continuously monitors the processor through testing and allows the processor to run under different modes, *e.g.,* single instruction or single cycle.

Figure 3 shows a typical processor test-bed layout. It shows the processor under test connected via a test adapter to the SPCTS test station. The SPCTS may connect to other devices under test. FTPs are coded and assembled off line

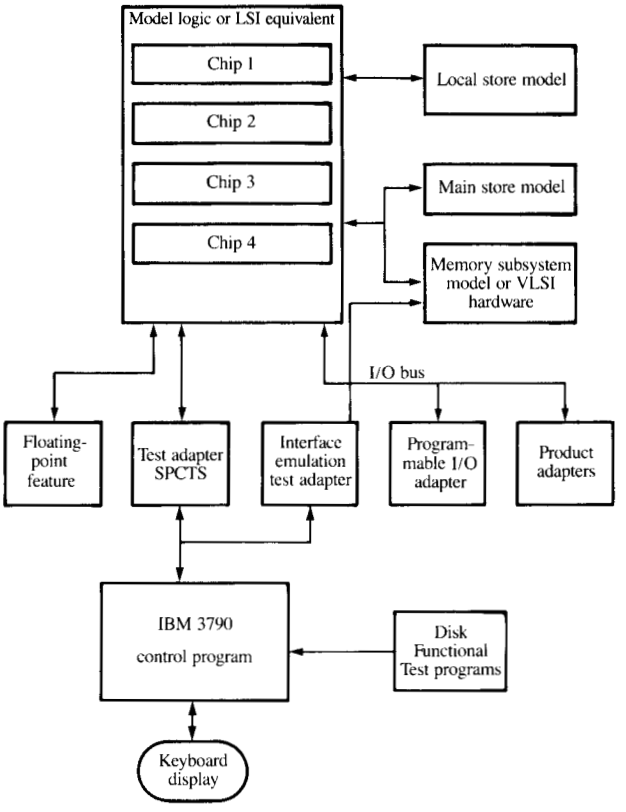

Figure 3 Hardware model structure.

from the test environment. Object code for the FTPs is transferred to the SPCTS via a BSC link [21]. Under control of either a user or a command list, programs are loaded to the processor under test and executed. The test adapter has a "stop on instruction fetch" facility that allows the test station to signal either completion or error. In the case of unattended test runs, relevant data from the processor are logged on the SPCTS line printer.

The processor test station was partially debugged without the hardware model where possible. Debug of the test bed was completed concurrently with the bring-up of the hardware model.

I/O test adapter

The I/O interface of the processor is tested on the hardware model by an I/O test adapter. This adapter is attached to the I/O port of the device under test. This is essentially a programmable test adapter that may be set up to act as any kind of I/O device which is allowed by the processor architecture. The I/O adapter is capable of performing all valid and invalid types of programmed I/O and channel I/O operations. Other features include the ability to program control-tag responses by the I/O test adapter, and to delay

these tag responses by different programmable time delays. These features accomplish further checking of the channel's function. Interference from the adapter may also be set up by the functional test programs, *i.e.*, I/O interrupts, channel requests, and EMA (*external memory access*) interference.

Memory subsystem driver

The processor-to-storage interface is the driving stimulus to the storage subsystem. Thus, the need arose for a means of driving this interface. Furthermore, testing had to be done on early software models and again on hardware as it was developed. The conventional method for modeling such an interface is to develop a model of the entity on the other side of the interface (the processor in this case), and obtain the values of the interface lines as a by-product of its behavior. Because of the complexity of the processor, and because neither software nor hardware processor models were available, an alternate technique called *interface emulation* was used.

Interface emulation

Interface emulation models only the behavior of the logical lines that make up the interface, rather than the entity which generates them. This technique is accomplished with two components—a test case description, and a means of converting it into a form which is compatible with the entity being verified, the storage control card in this case.

The test case descriptions are written in a language which uses predefined mnemonics to describe how the interface lines change with time. Some mnemonics, or keywords, represent the passage of a fixed amount of time each time they appear. As an example, consider an interface with a four-phase clock:

C1 C2 C3 ADDR(00F302) READ HW C4 C1 SEL C2 C3 C4 . . .

The keywords C1 through C4 represent the passage of one clock time and also specify the values of the four clock lines. A halfword read from address F302 is specified starting at C3 time and the storage select goes active at the following C1 time.

Each keyword mnemonic applies to one of six logical spaces describing the interface. *Pattern space* represents random control lines. Their values are repeated each fixed time interval unless changed. *Delay space* represents the propagation delay of each interface line in pattern space. *Address space* represents a group of address lines which change value together when a single keyword with parameter [*e.g.*, ADDR(00F302)] appears in the test case. *Output data space* represents a group of data lines on the storage interface when "write" data are being sent to the storage subsystem. *Compare data space* represents the same group of storage interface data lines when a storage read occurs. *Static space* represents control lines which remain static throughout the duration of a test case.

Converting the test case description into a form that is compatible with the entity under test is done in two stages. First, the test case is run through a PL/I program which converts it into a series of bit patterns. The program is driven by a keyword table which defines, by means of control bits and parameters, each mnemonic appearing in the test case description source code.

The second stage of test-case conversion uses the bit patterns generated by the PL/I converter program. Its implementation is different when driving software or hardware models, although the function is analogous. In the case of software simulation, the bit patterns are converted to simulation net changes by a behavioral model written in a hardware description language used within IBM.

When the transition is made to hardware, the same bit patterns are converted to electrical signals by an interface emulation test adapter specifically designed and built for this purpose (see Fig. 3). Its function is analogous to that of the software behavioral test-case converter, including all the logical spaces previously described. Furthermore, it can generate signals which were observed and recorded during software simulation, thereby simulating nonexistent hardware for which only a software model is available.

In the case of the storage subsystem, where interface emulation was employed, the same test cases were used during both software and hardware testing. This provided several significant benefits. First, test-case definition began early in the design cycle, so that both test cases and early software models could be used to verify each other. Second, these existing test cases were used to validate more detailed software models as they became available, by observing that they functioned similarly. At the same time, progress was continuing toward writing a complete set of tests for all basic functions of the storage subsystem. These test cases were again used when bringing up the hardware model and VLSI hardware. Due to the relative expense of software simulation, it was limited to testing basic functions. In software, each processor instruction and storage control sequence was verified in a limited subset of all possible operating conditions. As software verification progressed, however, a point was reached where increasingly exhaustive testing became less and less practical using a software model. For this reason, more exhaustive testing was performed on a hardware model of the storage control subsystem using the previously described functional test programs. At various points in the development process, the processor and storage hardware models were connected together and tested with these exhaustive functional test programs.

Detailed timing analysis

As previously mentioned, software simulation was used to help identify critical timing relationships in the machine. In addition, paths which were known to be critical to the design were identified for further analysis. Finally, all lines between chips were systematically examined. Logic paths which appeared to be critical were then analyzed in more detail using a collection of computational tools and techniques.

Most of this effort was accomplished using a series of delay calculators written in the APL language. Foremost among these was the APL delay calculator written specifically to support the custom bipolar technology macro set. The delay paths were defined manually, but a program automatically obtained data such as loading, wire length, etc. for use by the delay calculator. Approximately 350 delay paths were calculated. The delay equations were based on the electrical characteristics of the logic circuits and were generated by the group responsible for their design. The APL delay calculator used delay equations for random logic, registers, PLA, and ROS macros. Existing APL delay calculators were used for IBM gate array logic.

APL delay calculators were used for well-defined combinations of technologies, but there are a host of other physical configurations for which delay equations do not exist. Networks of this type include those which exceed wiring rule length restrictions, interface between technologies with different switching levels, employ unusual dotting combinations, exceed usual parameter limits, etc. In these cases, propagation delays must be determined through the use of electrical circuit analysis programs such as ASTAP. ASTAP was also used extensively for determining the constant values in the custom macro delay equations.

Engineering changes

At selected times in the development process, the design was checked and subsequent changes were controlled by a formal engineering-change procedure. This was the primary mechanism for keeping the software and hardware models at the same and latest change level. This is essential to prevent finding the same problem twice when both types of verification are occurring in parallel. It also provided valuable change-history information, which was frequently useful in solving design problems. The formal change procedure was based on a standard well-defined form which was used each time a change was made. The form is intended to be easy to use by providing places for the following information:

- description of the problem;
- previous, related problems;
- test being performed when the problem was found;
- list of all available documentation where those requiring updates may be indicated;
- description of the actual logic change that was made to solve the problem;
- tests that were run to verify the correctness of the fix;
- places to be initialed by the person completing each part of the update; and
- place where any related future problems can be indicated.

Conclusion

A complex set of VLSI chips requires a comprehensive design verification strategy and set of tools to successfully produce operational first-pass chips. The strategy chosen consisted of software simulation, hardware modeling, and interface emulation. Software models are most effective for initial and critical function verification and for performance estimation. Hardware models are most effective when performing a complete set of functional tests. Also, hardware models are most effective for debugging a test bed prior to the arrival of VLSI hardware. Interface emulation provides flexibility in verifying the design of multiple subsystems concurrently. The use of common data bases is essential to maintain control and compatibility of the hardware and software models. Through the use of functional test programs, test station adapters, and delay calculators, the total verification was accomplished.

The validity of any strategy or methodology can only be judged by the results which are produced. The objective here was to design a processor in a custom VLSI technology and to ensure that it follows the architecture and meets the design schedule. Using the strategy described in this paper, the first-pass VLSI hardware was designed and built, and is fully functional at full speed.

As the size of the machine design increases, the difficulties in building a hardware model increase much more rapidly. To accomplish this, some additional macro chips might be built to help in keeping the model small. PLAs are another device which will aid in dense model packaging in the future.

In conclusion, the importance of software simulation and hardware modeling of any design in a VLSI environment needs to be emphasized, especially with regard to delay analysis [22] and functional testing. In the past, problems associated with timing or functional deviation have been corrected in the field with some expense, but nevertheless corrected. In VLSI, that option is closed; to take additional time and resources to recycle a chip could prevent a product from ever being shipped.

Acknowledgments

The authors would like to thank the following people and groups: Floyd Petersen and the Systems Modeling group for their work in building the hardware model and for contribu-

tions to this paper; Joe Kavaky, Paul Lakin, and the Design Verification group for writing the functional test programs and for contributions to this paper; Mark Johnson for his design of the I/O test adapter, test-station adapter, and interface emulation test adapter, and for contributions to this paper; Courtny Barnett, Lisa Goodwin, and other contributors to the development of the hardware model conversion programs; Charlie Winn for writing the APL delay calculator and other programs; Gerry Thompson for writing the program to automatically generate input to the APL delay calculator; and Joe Tahmoush and John Campbell for developing the methodology described in this paper. In addition we would like to thank Fred Weiss for recognizing the need for hardware modeling and his support of it. Finally, we thank all the others involved in this project for the successful achievement realized.

References and notes

1. First-pass—initial release and manufactured hardware.
2. John E. Campbell and Joseph Tahmoush, "Design Considerations for a VLSI Microprocessor," *IBM J. Res. Develop.* **26,** 454–463 (1982, this issue).
3. BDL/S (Basic Design Language for Structure) is an IBM logic-description language.
4. E. B. Eichelberger and T. W. Williams, "A Logic Design Structure for LSI Testability," *Proceedings of the 14th Design Automation Conference,* New Orleans, LA, 1977, pp. 462–468.
5. J. S. Jephson, R. P. McQuarrie, and R. E. Vogelsberg, "A Three-Value Computer Design Verification System," *IBM Syst. J.* **8,** 178–188 (1969).
6. For a description of card and board packaging technology used at IBM, the reader is referred to D. P. Seraphim and I. Feinberg, "Electronic Packaging Evolution in IBM," *IBM J. Res. Develop.* **25,** 617–629 (1981).
7. 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).
8. IBM MSI—bipolar gate array medium-scale integrated circuit with approximately 100 gates and 45 signal I/O pins.
9. *Signetics 82S100 FPLA,* Signetics Corporation, 811 East Arques Ave., P.O. Box 409, Sunnyvale, CA 94086.
10. IBM Programmable Array Module Personalization Tool—program and hardware to automatically personalize IBM programmable array modules, R. Harr, IBM Lexington, KY.
11. FPLA—program and hardware to automatically personalize FPLA modules on a Data I/O Corporation PLA burn-in tool, A. Tran and C. Winn, IBM Kingston, NY, 1978.
12. EPROM—program and hardware to automatically personalize an erasable programmable read-only storage module on a Prolog Corporation PROM burner, G. Salyer, IBM Kingston, NY, 1977.
13. Amr M. Mohsen and Carver A. Mead, "Delay-time Optimization for Driving and Sensing of Signals on High-Capacitance Paths of VLSI Systems," *IEEE Trans. Electron Devices* **ED-26,** 540–548 (1979).
14. W. M. Vancleemput, "Hierarchical VLSI Design," *IEEE Computer Society 20th International Conference,* COMPCON Spring 80, San Francisco, CA, February 25–28, 1980, pp. 83–87.
15. Unit delay—a VMS simulation mode in which the propagation delay through each logic block is one time unit.
16. *Advanced Statistical Analysis Program (ASTAP),* General Information Manual, Order No. GH20–1271; available through IBM branch offices.
17. P. Losleben, "Utilizing Semiconductor Technology of the 80s—A Design Problem," *Proceedings of the Conference on Computing in the 1980s,* Portland, OR, 1978, pp. 237–243.
18. Raymond P. Capece, "Tackling the Very Large Problems of VLSI: A Special Report," *Electronics* **51,** 111–125 (November 23, 1978).
19. W. M. Vancleemput, "Design Automation Requirements for VLSI," *IEEE Computer Society 18th International Conference,* COMPCON Spring 79, San Francisco, CA, February 26–March 1, 1979, pp. 2–6.
20. Keiji Muranaga, "Utilization of Logic Simulation and Fault Isolation Software for Practical LSI and VLSI Component Test Program Generation," *Semiconductor Test Symposium on LSI and Boards,* Cherry Hill, NJ, October 23–25, 1979, pp. 193–202.
21. J. W. Cullen, "Binary Synchronous Communications," *Technical Report No. TR–29.0029,* IBM Raleigh, June 1968.
22. W. C. Holton and G. Brown, "Potential Barriers to Very Large Scale Integration," *Proceedings of the Conference on Computing in the 1980s,* Portland, OR, 1978, pp. 213–218.

Received September 14, 1981; revised February 5, 1982

The authors are located at the IBM System Products Division laboratory, Neighborhood Road, Kingston, New York 12401.