Verification of the IBM RISC System/6000 by a dynamic biased pseudo-random test program generator

by A. Aharon
A. Bar-David
B. Dorfman
E. Gofman
M. Leibowitz
V. Schwartzburd

Verification of a computer that implements a new architecture is especially difficult since no approved functional test cases are available. The logic design of the IBM RISC System/6000™ was verified mainly by a specially developed random test program generator (RTPG), which was used from the early stages of the design until its successful completion. APL was chosen for the RISC System/6000 RTPG implementation after considering the suitability of this programming language for modeling computer architectures, the very tight schedule, and the highly changeable environment in which RTPG would operate.

The ultimate goal of design verification is to ensure equivalence between a design and its functional specification. Strictly speaking, we can say that this goal can be achieved by exhaustive simulation or formal proof of correctness. The exhaustive simulation, in which all possible combinations of all inputs and memory elements of the design should be applied, can be done only for very small designs. Also, the state of the art of the formal techniques and the complexity of designs and specifications, usually written in English, do not allow utilization of the formal techniques in most industrial applications.[1] Despite significant progress achieved in recent years in formal verification, it has been pointed out that formal verification is not intended to replace simulation completely and that simulation is presently the major tool for the (partial) validation of hardware designs.[2]

In practical applications only a relatively small subset (as compared to the exhaustive set) of selected test cases is simulated. The challenge, then, is to create a subset that provides high confidence in the correctness of the design. We discuss how biasing techniques, combined with the dynamic approach to random test program generation, help to solve this problem.

This paper describes the concepts behind and the implementation of the IBM RISC System/6000* random test program generator (RTPG) developed to assist in the interactive creation of the adequate subset, as well as to automatically produce a vast

number of test programs for the comprehensive verification of the design.

At the moment the design was launched, no functional test cases existed for RISC System/6000 architecture. It was obvious that the traditional way of writing test cases could not provide the required level of confidence in the design. The design verification methodology developed at the IBM Haifa Research Group (HRG)[3,4] had already been successfully applied to several smaller designs such as floating-point units and a microcontroller. It was decided to adopt this approach for verification of the RISC System/6000 computer system.

APL was chosen as the programming language for the RISC System/6000 RTPG after considering the suitability of this language for modeling computer architectures, the very tight design schedule, and the highly changeable environment in which RTPG would operate. Originally the RISC System/6000 RTPG was developed in VS APL on the virtual machine (VM) operating system. It was later "migrated" to APL2 on the same system. It is currently being used for verification of follow-on products and is running mainly in batch mode on a cluster of over 30 RISC System/6000 machines, using IBM's APL2/6000.

The second section of the paper discusses some aspects of processor verification and describes a test program format suitable for this purpose. The subsequent section presents the main RTPG concepts and ways to realize them. The RTPG structure and the basic operation modes are described in the fourth section. Highlights, conclusions, and results of the RTPG experience are summarized in the last section.

The nature of processor verification

The RISC System/6000 RTPG and its predecessors. Logic verification of VLSI (very large-scale integrated) designs has always been an intrinsic part of the design process; however, the complexity of verification grows much faster than the complexity of designs. The problem is widely recognized in the case of microprocessors, as they present the leading edge of single-chip design complexity. Verification of microprocessors is considered to be a bottleneck of the entire design process, with crucial impact on the schedule for delivery of new systems.[5] An automated approach is essential for verification of a system that consists of several VLSI chips including a processor, a floating-point unit, a storage control unit, and caches.

In most applications, a test case for a processor is a program written in assembly language. The main goal of the tool called *RTPG*,[6] which is actually a *dynamic biased pseudo-random test program generator*, is to make the test program generation process more productive, comprehensive, and efficient.

A random approach to automatic test generation has proved to be successful.

A random approach to automatic test generation for software[7] and hardware[8] verification has proved to be successful. It was applied to the verification of selected design units such as a floating-point unit[9] and even a complete processor,[8] but very strong restrictions were imposed on the generated test programs. As a result of those restrictions, many parts of the design could not be accessed and, thus, could not be verified.

For some designs, such as a floating-point unit, the main part of the verification task can be fulfilled by programs that consist of only one instruction. The generation of such programs is relatively straightforward: The generator (or the user) selects an instruction and the required controls, generates the operands randomly (or provides them), and then invokes a reference model of the design to get the expected results. The generation of multiple instruction test programs is much more complicated, especially when such features as program control instructions, interrupts, and address translation are to be verified. There are approaches[8,10] that present a way to generate multi-instruction test programs. They are based on creating special tables of operands, and each instruction may select operands only from the relevant tables. Although much more productive than manual test writing, these approaches have drawbacks. The tables used must ensure that the generated test programs are worthwhile, that they would create the required instruction stream, and that they would not quickly end up with an

interrupt. These conditions imply use of the utmost caution in creating the tables and make this task very cumbersome. The generated test programs are relatively simple, and again, must obey many restrictions. For example:

- Some instructions are always preceded by specially inserted instructions, e.g., for initialization of base registers to get the allowed memory addresses. Thus, some sequences of the instructions can never be generated.
- For the same reason, a register may not be used as a source for different types of activity, such as an operand in an arithmetic instruction and a base register for addressing.
- To avoid creating endless loops, only branch forward instructions are generated. In branch conditional instructions, where it is not known *a priori* whether the branch is taken or not, instructions for both possible paths must be generated. As a result, the generation of test cases with many branch conditional instructions is quite difficult.

Such an approach to test generation may be classified as a static one, since the test programs are assembled first and executed afterwards. There is no relation between the intermediate machine states during execution of the test program and the test generation process. In RTPG the test generation is interleaved with the execution of every instruction as soon as it is generated. This dynamic nature of RTPG allows us to overcome drawbacks of the static approaches.

Because RTPG makes it easy to write test programs, it encourages the logic designer to create appropriate test programs while the logic design is still fresh in the designer's mind, whether these programs can be simulated at that early stage or not. Thus, the design verification is naturally integrated into the design process.

There are two challenges in having a test program generator ready at the early stages of design. The first is simply the time required to implement the test program generator. The second is a requirement for high flexibility. Frequent changes are required to the generator because the architecture specification is often very much in flux at this time and because implementation-specific details of the test programs are decided as the design progresses. These requirements are two of the reasons that APL was chosen for the RISC System/6000 RTPG implementation. APL provides quicker implementation than many other languages. It is also easy to modify to meet changing requirements as well as to handle various designers' requests. Another reason is the special suitability of APL for describing and modeling computer architectures. From its very beginning, APL was used for this purpose. Iverson's original book[11] contained a description of the IBM 7090 machine, and in 1964 the complete System/360* was formally described in APL.[12] All Boolean and relational functions are supported, and these functions provide very efficient bit-per-bit execution for bit arrays of any length. The language has the ability to individually address each bit in an array. It is often necessary to work with bit fields and subfields within instruction or data words, including double-precision floating-point data which are 64 bits long in the RISC System/6000. APL allows the needed fields to be easily split out, whereas many languages do not support bit operations at all (and especially not in more than 32-bit words). Because bit operations are so common, RTPG keeps values for all of the instruction and data words in Boolean form. Since APL stores a Boolean value as a single bit in the host processor storage, there is no penalty for keeping data in this convenient form. The APL rotate function together with the selection functions (like **take** and **drop**) are natural for implementing bit-shifting operations that are required in any computer processor model and are especially powerful in the IBM RISC System/6000 architecture. For example, the result of a Shift Right Algebraic Immediate (SRAI) instruction[13] is calculated by the following concise expression:

```
GPR[RA;]←32↑(SH/GPR[RS;0]),GPR[RS;]
```

and the Carry bit (CA) is:

```
XER[2]←GPR[RS;0]∧∨/(-SH)↑GPR[RS;]
```

Here *SH* is the shift amount, *RS* and *RA* are the numbers of the general-purpose registers involved in the instruction, and the second bit of *XER* (fixed-point exception register) contains CA. A description of the same instruction in any other programming language would be much longer and less easily understood. Note that the description of this instruction in English takes 10 lines in the architecture document.

Finally, although RTPG was initially planned to run on an IBM 3090-type processor (under VM), it was recognized early-on that it would also be necessary to run RTPG on workstation platforms. In fact, RTPG is now running under IBM's APL2/6000 on the very platform that it helped to verify. The transfer of the RTPG APL code from APL2 on the VM operating system to APL2/6000 on the Advanced Interactive Executive* (AIX*) operating system was trivial. The only change required was to the four file I/O programs and to the display screen programs. As mentioned earlier, RTPG is now running on a cluster of over 30 IBM RISC System/6000 processors to do the work of verifying new processors under development for the RISC System/6000 family of computers.

Test programs for processor verification. The most natural way of processor verification is to run assembly programs through the design model and to compare the simulated results with the expected ones. Usually the test programs are written as self-checking programs that return only a "go/no-go" flag. This concept is simple; however, its usage faces difficulties since:

- It can be used only when the design model is at an advanced stage, or at least when load and compare instructions are implemented.
- The test programs should obey the restrictions imposed by the supervisor that runs them.
- It requires more simulation cycles (running time) because of additional load and compare instructions that are simulated.

The RTPG approach is different. A test program generated by RTPG consists of three parts:

1. *Initial state* defines the contents of all registers, control flags, tables, caches, and memory locations (called "facilities") that influence, explicitly or implicitly, the execution of a test program. The instruction pointer (IP) register provided in this part contains the program initial address.
2. *Instructions* are given as the contents of caches or memory locations or both. The instructions part may be included in the initial state but is separated for better readability.
3. *Expected results* present the final state of all facilities that were changed during the test. The user may request that the final state of additional facilities also be included in the expected results. The IP register provides the test program breakpoint.

The collection of these three parts is referred to as a *test program* in this paper. Such test programs are self-contained: they include all information required for their independent and completely predictable execution. This feature enables them to freely migrate between test libraries and to be executed in any order.

A small test program generated by RTPG for the IBM RISC System/6000 processor is shown in Figure 1. As usual, asterisk "cards" (or lines) are used for comments. Comments may also be included in any line after the required data. The header (H) card contains the test number and indicates the beginning of the test. The register (R) cards specify register names and initial values. The instruction (I) and data (D) cards provide memory addresses and their contents. The IP values (both the initial value and the result) are given as effective addresses, i.e., before any address translation is performed. All other addresses are given as real memory addresses. The I and D cards are essentially the same and have different tags for readability only.

In addition to the data required for the processing, an RTPG-generated test program contains the following information:

- User comments to record the purpose for which the test has been created
- Corresponding assembly code (in the I cards) for readability
- Calculated effective address of data and target instructions, included as comments in I cards of load, store, and branch instructions
- The translation path of each address when a test program is running in address translation mode (not demonstrated here for reasons of clarity)
- The initial value of the Random Link (□RL) used to create the test and other control parameters required for regeneration of the test program (Only a few of these are shown in Figure 1.)
- Hooks for handling the program in test libraries

When requested, intermediate results of each instruction are included as comments in the *instructions* part of a test (the debug mode described in the fourth section).

Realization of the RTPG principles

RTPG realizes the dynamic approach to test generation in the following way. A test program is built step by step (instruction by instruction). Each step

Figure 1 An RTPG-generated test program for the RISC System/6000

```
* -------------------  RISC System/6000 RTPG  -------------------
H 10000;
* Created by: UserId                          Mar 28 12:38:17 1990
* Title  :  A simple test program for RTPG paper
* Comment:  Add, Load, Branch, and Store instructions
* Number of tests: 1;   Instructions in test: 4;
* Instructions: a  lx b sth
* Seed: 228656141;  FN: example;  Instr. order: f;  New_Reg: y;
*
* --------------------  Initialization  ------------------------
*
R IP        00010000
R R1        03642998
R R8        0000000F
R R10       1E12115F
R R22       0129DFFF
R R30       800000BA
R MSR       00008000
R CR        8CC048C8
R XER       2000CD45
D 0129DFFC 4E74570E
D 03640B90 7D280411
*
* --------------------  Assembly Program  ----------------------
*
I 00010000 7C48F415  ao.      R2,R8,R30
I 00010004 7CE0B02E  lx       R7,R0,R22           * E/A 0129DFFF
I 00010008 49BBB904  b        *+29079812          * T/A 01BCB90C
I 01BCB90C B141E1F8  sth      R10,X'E1F8'(R1)     * E/A 03640B90
*
* --------------------  Expected Results  ----------------------
*
R IP        01BCB910
R R2        800000C9
R R7        4E74570E
R MSR       00008000
R CR        8CC048C8
R XER       0000CD45
D 0129DFFC 4E74570E
D 03640B90 115F0411
END
```

consists of two main stages: a generation stage and an execution stage. At the generation stage a new instruction is chosen, and the required facilities are initialized. The execution stage is then invoked to execute the instruction and to update the affected facilities.

Dynamic test generation. The generation stage starts by inserting the operation code into the instruction word and establishing the rest of the instruction fields. At any point in the process each facility may be either free, which means that no value has been assigned to it yet, or have a value, in which case one is assigned to it by the initialization part of the process or by the execution of previous instructions. All facilities that influence the execution of the generated instruction are inspected, and those that are free are initialized. This principle works regardless of the complexity of a generated instruction and the initialization that it requires. For example, a single store instruction, besides initialization of data, base, and offset registers, may require initialization of an entire address translation path.

As soon as the instruction and all of the associated facilities are defined, the instruction is executed and all facilities that are changed during the exe-

cution are updated. Therefore, at the beginning of the generation of the next instruction, RTPG has the exact information about the current state of all facilities in the system. This information allows RTPG to:

- Select an instruction and its fields to gain the best effect from the execution (various biasing strategies help to achieve this goal)
- Bias data and operands, depending on the instruction being generated
- Reject the instruction if its execution would render the test program invalid
- Include any number of branch instructions in the test
- Control eventual interrupts
- Protect the required areas of memory and certain registers from being used by the generated test
- Define, on the fly, all required entries in the system tables (such as the page frame table)

A trace of the most recently updated facilities is also available and is used for implementation of some useful RTPG options.

Biasing. The generation of test programs is biased in order to increase the probable occurrence of events that otherwise have very low chances of being created. Biasing is both the strong point and the vulnerable point of RTPG. Strong, because it allows the generation of test programs with the required features. Vulnerable, because the selection of biasing strategies cannot be completely formalized and depends on the experience of the RTPG developer and that person's knowledge of the design. The goal of the biasing is not the creation of unique or very rare situations, but rather is to direct the generation process toward selected design areas so that most of the events in these areas are tested when the number of generated test programs is reasonably large.

The biasing functions are employed in the process of selecting instructions, instruction fields, registers, addresses, data, and other components that construct the test program. The starting set of the RTPG biasing strategies is derived from the architecture. Each instruction or process (such as interrupt action or address translation) specified in the architecture is represented by a block diagram composed of decision and execution blocks. In every decision block the data affecting the decision are selected in such a way that the subsequent blocks are entered with user-specified or RTPG-controlled probability. However, in multi-instruction test programs it is not always possible to get the required data. Let us say that the user asked for a 10 percent probability of floating-point overflow, and in the current instruction the decision was made to create it. If all floating-point registers already have values assigned by previous instructions, it may happen that no pair of registers will produce an overflow. Thus, in the generated test cases the actual probability of overflow might be less than the requested one.

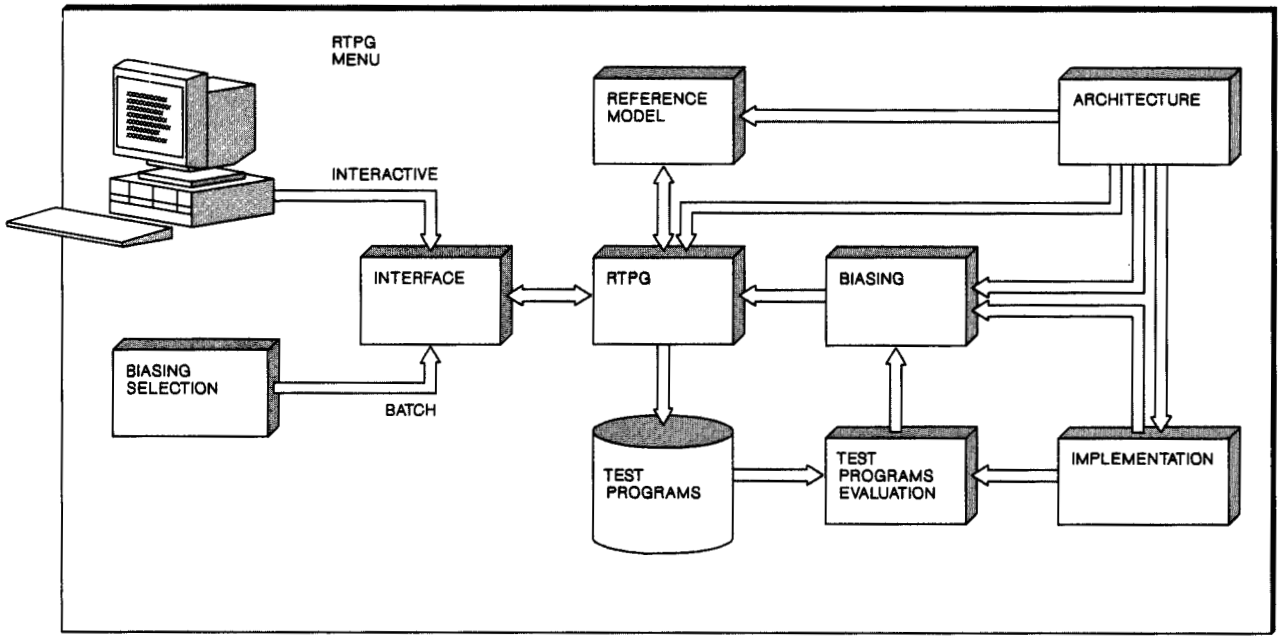
To some extent RTPG is a system that gathers into its biasing strategies all of the experience gained during the verification of several processor designs.

RTPG gathers into its biasing strategies all of the experience gained during verification of several processor designs.

As an example, consider the strategy of register selection, which is very important, especially for a RISC-type architecture where a large number of general-purpose registers is available and as many as three or four registers may be used in one instruction. The RISC System/6000 RTPG allows the selection of any one of the following three strategies:

1. Selection of free registers only. Here RTPG has complete freedom in biasing the instruction operands. Also, the result of each instruction will never be overwritten by the actions of subsequent instructions. Only relatively short test programs can be generated when this option is chosen.
2. Random selection (the default strategy). A register is selected randomly with biasing toward:
   - Increasing the probability to use the same register more than once in an instruction.
   - Preventing usage of a register as a target if it has been a target during its previous usage. This feature increases the test program observability, i.e., the probability to propagate any intermediate errors to the observable expected results.

Figure 2 The RTPG environment



3. High probability to select the target register of the previous instruction as a source or target of the current instruction. This option is useful in the verification of the register bypass logic as well as in the verification of synchronization between instructions when multiple instructions are issued and executed concurrently.

The starting set of biasing strategies is revised based on test coverage analysis of the generated test programs on both the RTPG reference model and the design models.[4] Coverage evaluation helps to detect and remove "holes" in the biasing. The final set of strategies ensures that there is a generation process with a reasonable probability of covering every architectural feature and every design block.

RTPG supports two biasing levels: local and global. The local biasing is involved in selecting immediate fields of instructions and selecting data for the operands. Many local biasing functions, specific for every class of instruction, are implemented in RTPG. For example, the generation of operands for add class instructions ensures a high probability of getting long chains of carries. In a "count leading zeros" instruction the biasing ensures the creation of operands with equal probabilities for any number of leading zeros. Some more sophisticated local biasing functions are implemented in more complicated cases, e.g., floating-point instructions.

Examples of global biasing control parameters that have a primary effect on the generation process are:

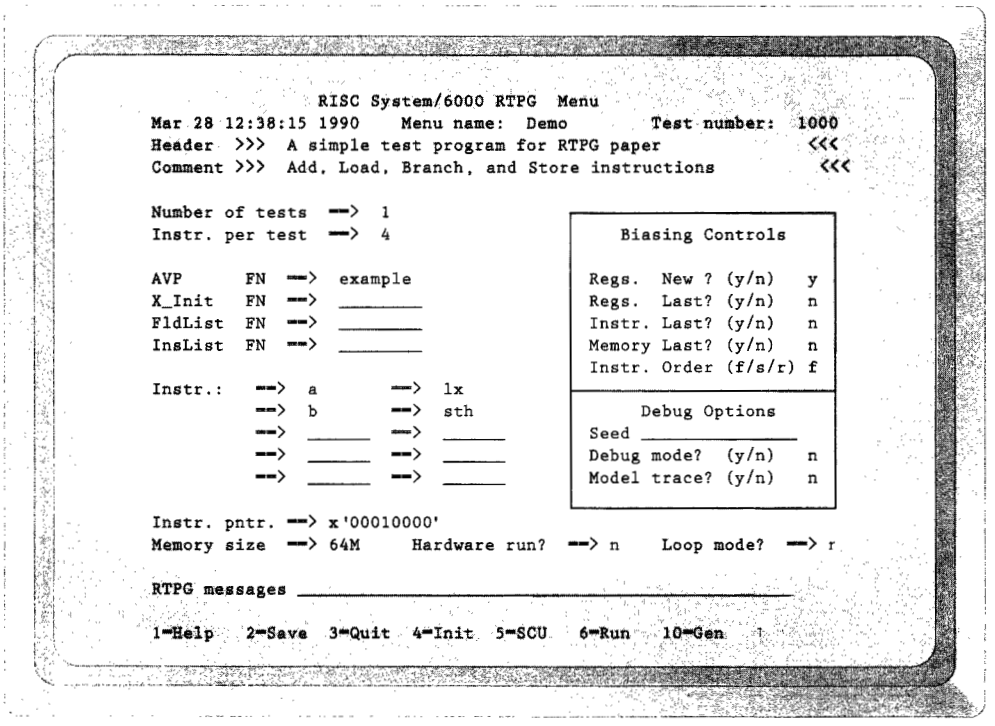
- Instruction selection strategy
- Initial value of the machine state register (MSR)
- Strategy for selecting general-purpose registers
- Memory areas that the test program is allowed to use

Each global and most of the local biasing strategies may be specified by the user. They are selected randomly by RTPG if not provided.

RTPG structure and basic operation modes

Environment. The RTPG design verification environment is shown in Figure 2. The *interface* and *biasing* blocks are actually parts of RTPG but are shown separately because of their connections to the external world. RTPG includes a *reference model*, a high-level architectural model of the processor to be tested. The lighter-shaded lines indicate flow of

Figure 3 The main RTPG menu



information with manual work involved in the process, and the darker lines indicate automatic flow of data between the units.

The architecture specification document is the primary source of information for the RTPG developers, and most of its features are embodied in RTPG. RTPG has to know all of the instruction format, and for each instruction, all of the parameters that influence its execution. This information is required for generating the instruction fields and for checking that all necessary facilities were defined before the execution of the instruction. RTPG has to keep a record of all facilities changed during instruction execution. The final state of all of the changed facilities provides the expected results.

The architecture may leave the handling of certain situations to the implementation. For example, unaligned storage access may cause an alignment interrupt in some implementations and may be handled by hardware in others. All such situations are handled in RTPG so that the test program that is created is correct for the implementation being tested.

User interface. The RTPG *user interface* includes several screens that allow the user to define the initial state of the processor and to control the test program generation process. The main screen used to generate the test program of Figure 1 is shown in Figure 3.

All screen parameters are optional, and if not specified, the default values are used (e.g., the default for InsList contains all instructions). The interface provides a way for documenting the generated test programs, selecting biasing strategies, initializing instruction fields, registers, and memory, and executing existing test programs.

The user may initialize any of the required facilities within an X_Init file. The file has the usual test program format. Thus, a prototype of a test program may first be created by RTPG and thereafter used for initialization and generation of many new test programs on top of the prototype. The X_Init file may also contain blocks of instructions and data that become parts of the test program. This feature is useful for including interrupt handler routines in the generated test programs.

The Debug Mode is a powerful by-product of the dynamic nature of RTPG. Including it in RTPG is almost free since RTPG already scans all changed facilities after every executed instruction. When this option is employed, the expected results of each intermediate instruction are included in the generated test program. This option is very useful in locating problems when a test fails. However, it requires much more space for storing the test programs.

RTPG offers two modes of operation.

RTPG offers two modes of operation: the generation mode (Gen) and the execution mode (Run). In the first mode, RTPG is used to generate one test file per invocation. The file contains the requested number of test programs, each program generated according to the control parameters specified on the screens. The batch version of the Gen mode is used for mass production where a large number of test programs is created for predefined sets of initial conditions. Such generation is performed as overnight runs or during weekends. Since the "porting" of RTPG to the IBM RISC System/6000 platform, RTPG runs as a background process concurrently on many RISC System/6000 machines connected in a local area network. The programs generated on the workstations as well as the programs generated on the VM host machines are submitted automatically to various simulators connected to the same local area network. Tests that do not expose any design problems are discarded.

In Run mode RTPG executes an existing file of test programs and returns it, including correct expected results. The original file may or may not include expected results. If they are provided, they are compared with the expected results created by RTPG, and any discrepancies are reported. Run mode is used to define the expected results for manually written test programs. In case of changes in the architecture, it is used to confirm or update the expected results of programs imported from other sources or generated previously by RTPG. Another use of Run mode is to rerun an existing test program in Debug Mode. This use is done frequently when a test fails and when it was originally generated with the Debug Mode turned off.

Test coverage evaluation on the architecture level. The quality of verification is improved significantly when there is a means to estimate test coverage on both architecture and implementation levels. In regard to RTPG, the results of coverage analysis provide feedback for improving the biasing functions. Also, the coverage analysis on the architecture level is used in the preparation of a relatively small subset of tests that include test programs for every architectural feature. In the RISC System/6000 RTPG the high-level reference model was implemented within RTPG. The interpretive nature of APL considerably facilitated the implementation of some coverage analysis techniques, including techniques that require fault injection.

One of the simplest coverage techniques is ensuring that each line of the code has been executed at least once. A special function analyzes the character representation of an APL function and splits each labeled line into two lines. The first one contains the label and an assignment statement that sets the corresponding bit in a trace vector associated with this function. The second line created by the split contains the APL statement that was on the line before the split (but without the label). Assignments of the bits of the trace vector are also inserted after each statement with an APL right arrow (**branch**). A bucket of test programs is then executed (using the Run mode) on the "trace-modified" RTPG. Zero values in the trace vector indicate blocks that were not reached.

Another coverage technique called "skip mutation,"[14] which requires injection of faults into the code and thus provides much higher confidence in the generated test programs, may also be easily implemented. Skip mutation means that one line of the analyzed function is not executed. To make this technique more sensitive, an original APL function may be replaced by its more detailed version. Skip mutation is performed by another function that takes the character representation of an APL function to be checked and precedes the required line by the comment symbol "⍝". The information from the previous step (line coverage) is used to select only those test programs that pass through the

skipped line. The procedure is repeated for each line in the function.

The coverage analysis is done automatically as soon as both the function to be analyzed and the test file are specified. However, because of performance considerations, only functions for which a low coverage is suspected are analyzed. In the RISC System/6000 RTPG environment, only functions that implement the floating-point unit were analyzed by both techniques.

RTPG implementation. The RISC System/6000 RTPG is implemented as a single workspace which is able to create test programs of up to several thousand instructions on a six- to eight-megabyte virtual machine. The user interface was written in REXX to simplify some Conversational Monitor System (CMS) file-related checking that was not so easy to implement in VS APL. The RTPG functions may be grouped into:

- Utility and service functions
- Functions for instruction execution
- Biasing functions
- Simulation of interrupts
- Address translation

The service functions prepare the initial machine state, manage instruction selection, prevent the creation of endless loops in the generated test programs, and mask undefined results. These functions also handle the Run option, including comparing the actual results with the expected ones that are provided in the original test program.

Each RISC System/6000 instruction has a corresponding APL function with the same name that operates on the architectural facilities (defined as global variables) to perform the behavior of the instruction. Each "instruction function" is partitioned into a biasing section and an execution section which are used as required for biasing and setup or reference model operation. As soon as the instruction to be generated is selected, the required APL function is invoked by "executing" the character representation of the instruction mnemonic. Thus, when a new instruction is added to the system, or in case of a change in the instruction mnemonic or behavior, only one function must be added or changed. A rich set of utility functions that perform many of the required common tasks, such as incrementing the instruction pointer or adding two register values, facilitates writing of the "instruction" functions.

The register arrays of the processor are modeled as APL Boolean matrices. The memory is modeled as another Boolean matrix with a companion address vector that maps processor memory addresses to APL matrix indices.

The RISC System/6000 RTPG is implemented as a single workspace.

When the project was started, very little reusable RTPG software existed from previous projects (although the concepts were well understood). At any moment no more than four persons were working on RTPG. One of them was dedicated to the user interface written in REXX, and one was involved only part time in RTPG development. In less than four months the first version of RTPG, which supported almost all branch and fixed-point instructions, was given to the designers. Floating-point instructions were delivered a month later. From that time only two people on average were involved in RTPG development, working on the storage control unit, on cache modeling, on imbedding architecture changes, and on implementation-dependent features. They also supported RTPG in the field, responding to numerous designers' requests. This activity was completed exactly one year after starting the project, and since then, only one person is involved in RTPG support and enhancements. This person's responsibility includes porting RTPG to APL2/6000 and upgrading it to support follow-on designs.

Concluding remarks

We described a comprehensive procedure for biased random test program generation and the RTPG implementation of the approach. This approach has been adopted as a main technique in the design verification process of several IBM designs. The designs varied from floating-point coprocessors to the complete complex of the IBM RISC System/6000

computer. In all cases the corresponding VLSI products came out fully functional on the first pass. In the case of the RISC System/6000, once the final design was completed, no new bugs were found.

RTPG accompanies the design process from its very early stages through its successful completion. At the beginning of the process, RTPG is used by the designers to generate simple test programs directed toward recently developed logic. This use results in a significantly lower error detection rate at the advanced stages of the design. At the system level, RTPG is used mainly in batch mode, where a significant volume of test programs, some of them consisting of up to several thousand instructions, is generated and simulated on the design model. The employed biasing strategies have evolved, based on requests coming from the designers and the feedback from the coverage analysis.

The RTPG development effort is not considered to be negligible. However, it is incomparable with the amount of resources required to achieve similar verification quality with manually written or purely randomly generated test programs. At the moment, RTPG is notably tailored to the architecture it serves. Nevertheless, existing RTPGs provide a good groundwork for the development of new ones, even when the architectures are different.

Choosing APL for the implementation of RTPG allowed us to provide this tool to the designers on a timely basis, and it also allowed us to keep up with many changes and modifications to RTPG necessitated by the novelty of the approach and also by frequent changes in the architecture at that time. Shoulder-to-shoulder work with the designers contributed to the success of the tool but required instant response to their requests. Again, APL, with no compilation and linkage overhead, allowed us to quickly respond to these requests. The interpretive nature of APL was used to implement some test coverage evaluation techniques that work on the architecture level directly in RTPG.

In the beginning, RTPG was used mainly in the interactive mode. With a capability to generate 20 to 30 instructions per second, it provided fairly good response time to the users. The design model simulator running on the VM host machine was slower. Porting of the simulator to the RISC System/6000 platform and the use of hardware assist for the simulation required more and more test programs to feed all available simulators. Moving RTPG to APL2/6000 solved the problem of limited available computer time, and now RTPG, running on a cluster of RISC System/6000 machines, is able to produce the required number of test programs.

Acknowledgments

The authors deeply appreciate the help of their colleagues at the VLSI testing and verification group in HRG, especially Raanan Gewirtzman and Yossi Malka for their work on test coverage evaluation. The authors gratefully acknowledge cooperation with IBM design and simulation teams in Essonnes, Burlington, Boca Raton, and Austin that enabled us to develop this verification methodology and to experiment with it on their designs. Special recognition goes to Israel Berger (HRG) for his innovative contribution to the methodology and his supervision and support, together with Yoav Medan (HRG) and Jerry Long (Austin), during the development process of the RISC System/6000 RTPG.

Cited references

1. P. Camurati and P. Prinetto, "Formal Verification of Hardware Correctness: Introduction and Survey of Current Research," *Computer* **21**, No. 7, 8–19 (July 1988).
2. *Formal Verification of Hardware Design*, M. Yoeli, Editor, IEEE Computer Society Press, Los Alamitos, CA (1990).
3. A. Aharon, A. Bar-David, E. Gofman, M. Leibowitz, and V. Schwartzburd, *RTPG—A Dynamic Biased Pseudo-Random Test Program Generator for Processor Verification*, Technical Report 88.290, IBM Israel Science and Technology, Technion City, Haifa 32000, Israel (July 1990).
4. A. Aharon, R. Gewirtzman, E. Gofman, and Y. Malka, *Hardware Design Verification with Task Models*, Technical Report 88.289, IBM Israel Science and Technology, Technion City, Haifa 32000, Israel (June 1990).
5. N. Tredennick, "Trends in Commercial VLSI Microprocessor Design," *VLSI CAD Tools and Applications*, W. Fichter and M. Morf, Editors, Kluwer Academic Publishers, Norwell, MA (1987).
6. A. Aharon, A. Bar-David, R. Gewirtzman, E. Gofman, M. Leibowitz, and V. Schwartzburd, *Dynamic Process for the Generation of Biased Pseudo-Random Test Patterns for the Functional Verification of Hardware Designs*, Israel Patent Office, Patent Application No. 94 115 (April 1990).
7. D. L. Bird and C. U. Munoz, "Automatic Generation of Random Self-Checking Test Cases," *IBM Systems Journal* **22**, No. 3, 229–245 (1983).
8. A. S. Tran, R. A. Forsberg, and J. C. Lee, "A VLSI Design Verification Strategy," *IBM Journal of Research and Development* **26**, No. 4, 475–484 (July 1982).
9. P. M. Maurer, "Design Verification of the WE32106 Math Accelerator Unit," *IEEE Design and Test of Computers* **5**, No. 6, 11–21 (June 1988).
10. C. Bellon et al., "Automatic Generation of Microprocessor

Test Programs," *ACM/IEEE 19th Design Automation Conference Proceedings* (June 1982), pp. 566–573.
11. K. E. Iverson, *A Programming Language*, John Wiley & Sons, Inc., New York (1962).
12. A. D. Falkoff, K. E. Iverson, and E. H. Sussenguth, "A Formal Description of System/360," *IBM Systems Journal* **3**, Nos. 2 and 3, 198–261 (1964).
13. *POWER Processor Architecture*, IBM Corporation, Advanced Workstation Division, 11400 Burnet Road, Austin, TX 78758 (1990).
14. A. Aharon, I. Berger, E. Gofman, and M. Yoeli, "Testing a Microprogrammed Control Unit," *VLSI and Computers, COMPEURO Conference Proceedings*, Hamburg, Germany (May 1987), pp. 390–393.

Accepted for publication June 20, 1991.

Aharon Aharon *IBM Israel Science & Technology Ltd., Technion City, Haifa 32000, Israel.* Mr. Aharon joined IBM at the Haifa Research Group (HRG) in 1983 and since 1989 has been the manager of the VLSI Testing and Design Verification Group. From the time he joined the HRG he has been involved in developing a methodology for logic verification of hardware designs and applying it to various designs within IBM, among them the IBM RISC System/6000 for which he received an IBM Outstanding Technical Achievement Award. Mr. Aharon received his B.Sc. degree in 1981 in computer engineering and his M.Sc. degree in electrical engineering in 1983 from the Technion, Israel Institute of Technology. Since 1984 he has been an Adjunct Teaching Associate in the Electrical Engineering Department at the Technion. Mr. Aharon is a coauthor of several papers and technical reports. He also wrote several teaching books published by the Open University of Israel in the areas of digital systems, logic design, computer organization, and microprocessors.

Ayal Bar-David *IBM Israel Science & Technology Ltd., Technion City, Haifa 32000, Israel.* Mr. Bar-David joined IBM at the Haifa Research Group (HRG) in 1983 and has been involved in the development of tools for logic verification of hardware designs. He received a general award for his participation in the verification of the IBM RISC System/6000. During 1987–1989 he was visiting at Qualcomm Inc., San Diego, California, working on the design of ASICs for digital communications. Presently he is working on the development of CAD tools for VLSI. Mr. Bar-David received his B.Sc. degree in 1983 and M.Sc. degree in 1986, both in electrical engineering, from the Technion, Israel Institute of Technology.

Barry Dorfman *IBM Advanced Workstations Division, 11400 Burnet Road, Austin, Texas 78758.* Mr. Dorfman is an advisory programmer in the simulation/verification area of the Central Electronics Complex Engineering Center. He joined IBM at Austin, Texas, in 1976 after receiving a B.S.E.E. degree that year from Arizona State University, Tempe, Arizona. He held various assignments in circuit and logic design and received an IBM informal award for his design and APL implementation of an engineering processes system. He worked in the information systems area of the Systems Technology Division where he was responsible for developing several software systems. In 1987 Mr. Dorfman joined the IBM RISC System/6000 team where he works today with processor verification and the RTPG program.

Emanuel Gofman *IBM Israel Science & Technology Ltd., Technion City, Haifa 32000, Israel.* Dr. Gofman joined IBM at the Haifa Scientific Center and Research Group in 1977. For his work on projects on the Hydraulic Network Solver (1977–1979) and Computer-Aided System for Scheduling School Time-Tables (1979–1981) he received two awards from the Information Processing Association of Israel, first in Implementations of Science and Technology and second in Data Processing in Administration. Since 1981 he has been involved in developing a methodology for logic verification of hardware designs and application of this methodology to various designs in IBM. He spent 1986–1987 as a visiting staff member in the IBM Independent Business Unit, Austin, working on verification of the IBM RISC System/6000. For this work he received an IBM Outstanding Technical Achievement Award. Dr. Gofman received an M.Sc. degree in applied mathematics from the Moscow State University, USSR, in 1968, and a Ph.D. in technical cybernetics in 1975 from Riga Politechnical Institute, Latvia.

Moshe Leibowitz *IBM Israel Science & Technology Ltd., Technion City, Haifa 32000, Israel.* Mr. Leibowitz is a research staff member in the Haifa Research Group (HRG) VLSI group. He graduated from the Technion, Israel Institute of Technology with a B.S.E.E. degree (cum laude) in 1975 and received an M.S.E.E. degree from the Technion in 1989. He joined IBM in March 1985 at Haifa, Israel, and took part in and led several projects in VLSI testing and simulation and in physical design. Currently he is on international assignment at IBM Boca Raton working on VLSI synthesis and design verification.

Victor Schwartzburd *IBM Israel Science & Technology Ltd., Technion City, Haifa 32000, Israel.* Mr. Schwartzburd joined IBM at the Haifa Scientific Center in 1984. He has been involved in developing a methodology for logic verification of hardware designs and application of this methodology to various designs in IBM. From 1986 to 1988 he worked on development of an automatic verification system for the IBM RISC System/6000. For this work he received an IBM Outstanding Technical Achievement Award. He also has a patent (with other IBM employees) on the method of automatic verification and testing. He spent 1990 and 1991 as a visiting staff member in the IBM Storage Systems Products Division, Tucson, Arizona, working on testing of the IBM 3990 Control Unit. Mr. Schwartzburd received an M.A. degree in electrical engineering from the Moscow Power Institute, USSR, in 1957.

Reprint Order No. G321-5451.