Functional Verification Methodology for the PowerPC 604™ Microprocessor

James Monaco David Holloway Rajesh Raina
Somerset Design Center
9737, Great Hills Trail
Austin, TX 78759
e-mail: {jmonaco, daveh, raina} @ibmoto.com

Abstract

Functional (i.e., logic) verification of the current generation of complex, super-scalar microprocessors such as the PowerPC 604™ microprocessor presents significant challenges to a project's verification participants. Simple architectural level tests are insufficient to gain confidence in the quality of the design. Detailed planning must be combined with a broad collection of methods and tools to ensure that design defects are detected as early as possible in a project's life-cycle.

This paper discusses the methodology applied to the functional verification of the PowerPC 604 microprocessor.

1. Introduction

Only in recent years have industry practitioners begun publishing information detailing functional verification practices and procedures [1, 2, & 3]. Building on work previously published describing the PowerPC 604 microprocessor project [4 & 5], this paper describes the functional verification methodology employed. The design methodology and specification approach are briefly discussed. Practices covering project and verification planning, testcase generation, simulation, coverage assessment, and design quality confidence are included. Finally, possibilities for future improvements to these functional verification practices are examined.

The term "verification methodology" used in this paper should be interpreted to mean "an integrated set of techniques and methods applied to produce a defect-free microprocessor design".

2. Design Methodology and Specification Approach

At the Somerset Design Center, methods and tools used in the microprocessor verification effort are significantly influenced by the tools used to develop the design model.

2.1 Development and Design Modeling

The selected hardware design modeling language (HDL) now entrenched in the design process is a proprietary superset of the Verilog language. Depending upon how a design is partitioned, portions of the logic and physical design may be synthesized while other portions may be constructed as custom logic and layout in a traditional manner. The logical and physical design characteristics can then be analyzed in parallel.

The design's logic characteristics are examined from the perspective of functional logic -- "Does the design correctly execute according to the architectural and implementation-dependent design specifications?". For the PowerPC 604 microprocessor project, design description was captured at the gate and logic device levels. Using a variety of tools for compilation and translation, models were produced in forms usable by multiple simulation systems including a Verilog event-driven simulator and proprietary cycle based simulators. Simulation was performed to verify the correctness of the simulation model.

The design's physical characteristics were examined from the perspective of physical circuit design -- "Does the design correctly meet desired timing and electronic specifications (i.e., set-up and hold constraints, switching speeds, power consumption, etc.).

An equivalency check, shown in Figure 1, was performed periodically to ensure that the hardware technology description of the design was synchronized with the high-level logical description.

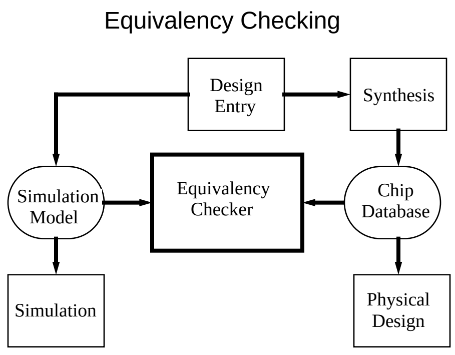

Figure 1: Equivalency Check and Logical/Physical Views

Figure 2 illustrates the design process used by the PowerPC 604 microprocessor project. It resembled a spiral-type software development approach. Most project effort involved two (iterative) steps:

- improve design, and
- evaluate improvements;

IBM, the IBM logo, AIX, PowerPC, PowerPC 604, and RISC System/6000 are trademarks of International Business Machines Corporation. Motorola and the Motorola logo are trademarks of Motorola, Inc. Windows NT is a trademark of Microsoft Corporation.

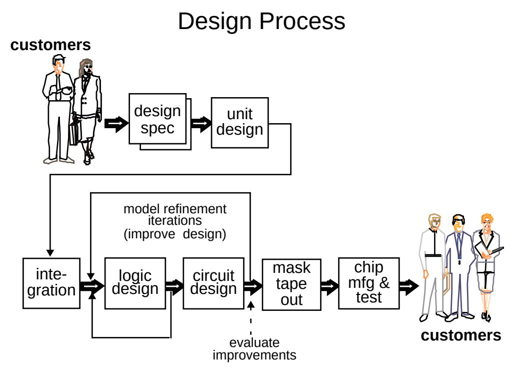

Figure 2: Design process with feedback loops

2.2 Specification Approach

The PowerPC 604 microprocessor conforms to a fundamental architecture, yet includes a set of implementation dependent features.

The architectural specification for most microprocessor is typically semiformal -- that is, it is not mathematically rigorous. The PowerPC architecture [6] provides an example of a semi-formal description of a microprocessor architecture. Semi-formal descriptions follow natural language prose and include helpful tables and block diagrams to clarify structure and behavior. Figure 3 is an excerpt from the PowerPC 604 Users Manual [7] which describes, in semi-formal terms, how memory addressing is performed.

5.1.1 Memory Addressing
A program references memory using the effective (logical) address computed by the processor when it executes a load, store, branch, or cache instruction, and when it fetches the next instruction. The effective address is translated to a physical address according to the procedures described in Chapter 7, "Memory Management," in The Programing Environments Manual, augmented with information in this chapter. The memory subsystem uses the physical address for the access. For a complete discussion of effective address calculation, see Section 2.3.2.3, "Effective Address Calculation"

5.1.2 MMU organization
Figure 5-1 shows the conceptual organization of a PowerPC MMU in a 32-bit implementation; note that it does not describe the specific hardware used to implement the memory management function for a particular processor. Processors may optionally implement on-chip TLBs and may optionally support the automatic search of the page tables for PTEs. In addition, other hardware features (invisible to the system software) not depicted in the figure may be implemented.

Figure 3: Sample Semi-formal Architectural Specification

At Somerset, the general architectural specification is combined with implementation dependent features to produce a *micro-architectural specification*. Figure 4 is an extract from the PowerPC 604 microprocessor micro-architectural specification describing a portion of the Branch Target Address Cache (BTAC) unit. From these examples it becomes evident that the model implementation itself comes closest to capturing mathematical rigor.

... outlines the inputs and outputs of the BTAC.

Signal/Bus Name	Destination Unit	Description
sbr_btac_add	BTAC	When asserted add the branch to the BTAC. When unasserted no add can occur.
sbr_btac_del	BTAC	When asserted del the branch if in the BTAC. When unasserted no delete can occur.
sdi_br_phantom	BTAC	When asserted flush the BTAC.
btac_pred_tar (0:29)	FAR	The prediction addr from the BTAC. The most significant 29 bits are used to address the ICache. The other bit specifies the word location of the branch. This data is used within the IBUF to invalidate instructions. This is always the target address.
....	...	...

The prefix of the signal name is where the signal originated. The destination column shows the destination unit for each signal, and the description column explains the purpose of the signal ...

Figure 4: Typical Semi-formal Micro-Architectural Specification

A lack of mathematical rigor in the specification process combined with the cultural acceptance of the development method outlined above and designer resistance to formal specification and verification techniques makes simulation and emulation the de facto options for demonstrating processor model correctness.

3. Project and Verification Planning

3.1 Project Milestones

As project development proceeds, model refinement follows a succession of milestones. The most significant of these are:

- Publication of the micro-architecture: the decomposition of the chip into functional blocks and definition of the basic data and control flow interfaces between these blocks.
- Preliminary model integration: blocks (units) which have been implemented and tested by individuals or a small group of designers are combined into a single processor model. Tracking of defects and counting of simulation cycles begins.
- Stable model: declared when a pre-determined set of regression testcases execute without fail. The precise goals vary from implementation to implementation but usually involves thousands of testcases executing tens of millions of simulation cycles. These testcases are retained and rerun with each successive model refinement.
- Tapeout: performed when (a) zero functional defects occurred during for a pre-determined period of time and (b) equivalency checking between logical and physical designs passes with no fails and (c) targeted simulation cycle count attained and (d) simulated boot of a simple operating system completes successfully, and (e) all physical design checks (e.g., DRC, LVS, etc.) complete with no fails. A tapeout database is produced and released to the fabrication facility mask shop.

3.2 Verification Planning

The traditional approach to microprocessor design verification allows a project to make considerable progress before verification staffing commences. At the Somerset Design Center, verification staffing was an integral part of the project plan. The verification engineer at Somerset plays a significant role in the development process and, therefore, the team begins assembly at project start.

Verification engineers were devoted to one or more logic blocks of the chip and some participated in additional support roles. For example, one verification engineer focused his or her efforts solely on verification of execution blocks such as integer and floating point units. Another verification engineer was involved with support of project progress reporting tools. Still another verification engineer was responsible for multiple tasks such as verification of logic bus interface unit plus development of the system behavioral model (i.e., the behavioral model which interacts with the processor model during simulation).

For the PowerPC 604 microprocessor, members of the verification group were divided across the functional blocks indicated in Figure 5, so that each block had at least one responsible verification engineer.

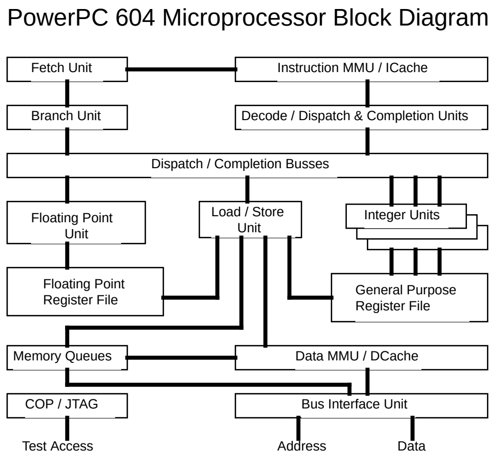

Figure 5 - PowerPC 604 Microprocessor Block Diagram

The verification team derived a series of plans from the micro-architectural specification. Plans included:

- Unit Verification Plan supporting pre-integration activities;
- Architectural Verification Plan (AVPs) demonstrating the processor meets the "black box" behavior defined by the basic architectural specification.
- Implementation Verification Plan (IVPs) demonstrating "white box" correctness by exploiting knowledge about details of the design implementation (e.g. ensure instructions proceed through the pipeline precisely as defined by the micro-architectural specification).

A plan was captured documented and maintained for each respective logic block.

4. Test Types and Testcase Generation

The verification plans produced at successive stages of project development guided the generation of testcases. The following list describes testcases constructed during project development:

Unit Phase: The unit testcases were hand generated as assembly programs and translated into a form usable for unit level simulation. The unit simulation form was a sequence of bit vectors. These testcases enumerated variable fields such as testcases containing only load word instructions using every possible destination register. Unit testcases were constructed at a high level (i.e., assembler mnemonics). These testcases formed the basis of "unit regressions". These unit regressions were reused after model integration had occurred and also during model refinement iterations.

Successive Iteration (Integrated Model) Phase: A Random Test Program Generator (RTPG) [8] generated most testcases during this project phase. It produced a baseline set of regression testcases used to qualify a stable mode. The regression set consisted of instruction streams containing single instructions, streams containing an enumeration of all pairs of instructions, and instruction streams containing complex instruction sequences employing randomly selected operands.

The verification plans developed for each logic block guided the generation of additional testcases. These testcases were generated after the verification engineer specified a file containing one or more sets of biasing parameters. These biasing parameter files were referred to as *menus*. Hundreds of biasing parameters were supported by RTPG including the following examples -- select specified groups of instructions, select specified effective address regions for load/store instructions, select probability of branch forward or backward target addresses, enable external interrupts, enable L1 caches, and suppress address translation traps. Menus were used by the random testcase generation program to produce large numbers of complex testcases. The biasing parameter files were further combined with simulation run-time parameters to produce a wide range of random events for stimulating the processor model.

5. Simulation: Getting "Megacycles"

Several hundred workstations were available at the Somerset Design Center. These computing resources were effectively exploited on a continuous basis through the use of an automated simulation environment described in [9].

As shown in Figure 6, the Automated Simulation Environment for Somerset comprised the following integrated components:

- a distributed batch job processing facility (dbj),
- a flexible random testcase generator (rtpg),
- a fast cycle-based simulator (sim),
- a common model server from which newly refined and up to date models were available (msrv), and
- a common simulation database server provided up-to-date statistics on passing and failing testcases (stat).

5.1 Run-Time Controllability and Observability of Signals

The Somerset HDL modeling environment provided two important constructs that permitted complete controllability and observability of signals (referred to as "facilities") during simulation of the compiled design model. The PUTFAC construct allowed forcing any facility to a specified logic value. PUTFAC invocations override existing logic values on the specified facility. The inverse construct, GETFAC, allowed probing a logic value for any specified facility.

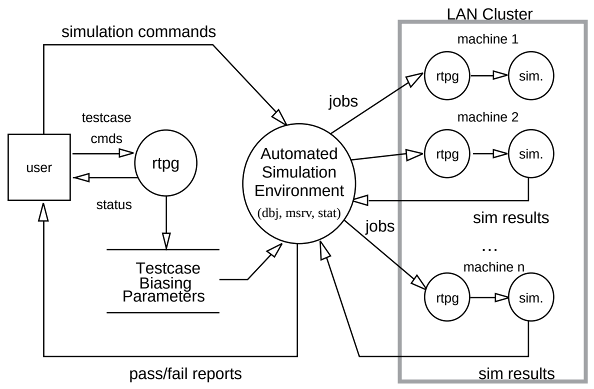

Figure 6: The Automated Simulation Environment

These two constructs afforded enormous possibilities for maneuvering the design model, thereby expediting the verification process. It was possible to stress the design by creating pathological inputs and states with the use of specific PUTFACs. Likewise, it was possible to check for specific critical events through GETFACs. These constructs formed the backbone of coverage analysis and functional design checking of the PowerPC 604 microprocessor.

Unlike typical controllability/observability procedures and functions that are written and compiled within the design model, the coverage monitor and checking procedures written with PUTFAC & GETFAC could be compiled independent of the design model and invoked at simulation time, as illustrated in Figure 7.

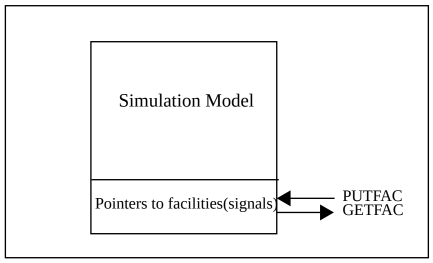

Figure 7: Design model data structure to support PUTFAC/GETFAC

A program called an IVPC (Implementation Verification Program in C) was used to interface to the model by using the PUTFAC/GETFAC constructs. Figure 8 depicts the operation of the interface.

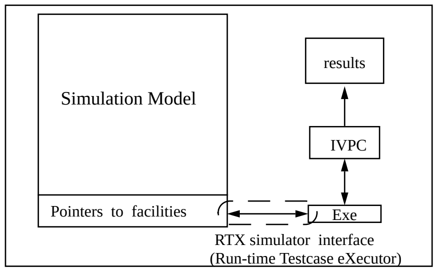

Figure 8: IVPC interface to the design model

The logic design team influenced the deployment of IVPCs and the allocation of limited verification resource. During several design verification reviews, the designers indicated areas of the design requiring additional checking and coverage monitoring. Units with a history of bugs or complex unit interfaces were also targeted for IVPC-based verification.

Since controlling and observing the PowerPC 604 microprocessor's model signals did not have to be programmed into the simulation model at compile time, verification engineers continued writing verification programs without disturbing stable versions of the model.

5.2 Stressing the Design

On the PowerPC 604 microprocessor development project, IVPCs were used for the following specific purposes:

- To generate external interrupts (random and event-triggered).
- To invalidate cache lines (random and event-triggered).
- To artificially fill the Translation Lookaside Buffer (exercise overflow functionality).
- To artificially exercise the stall mechanisms (sequencer unit pipelines).
- To verify sub-unit functionality under various conditions of instruction aborts, branches, and operand sources.

The GETFAC construct was used liberally since it was risk-free. GETFAC could not modify facility values and, therefore, was not capable of altering the normal flow of simulation. The PUTFAC construct, on the other hand, could modify the state of simulation and deserves discussion.

The three issues influencing the use of PUTFACs for stressing the design or expediting event occurrence were:

• Knowing exactly what facilities to PUTFAC:
The architecturally or even micro-architecturally visible registers/latches have buffer latches placed around them. For timing reasons, sometimes these latches serve as the source for circuits that must be stressed. They do not appear in the micro-architectural specification. If PUTFACs are used only on the visible latches, there is a possibility of obtaining unintended behavior. Therefore, it was necessary to understand the design, as modeled.

• Knowing how to PUTFAC:
Since the simulator is cycle based, depending on the facility, one needs to know whether to set the facility "before" the model is evaluated or "after" the model is evaluated. Also, if one can be sure that the chosen tests to be run during simulation will not affect the facility, one may be able to dispense with a facility PUTFAC every cycle, thereby speeding simulation.

• Understanding ramifications:
Even if PUTFACs are used on the right facilities at the correct time, it is imperative that one fully understands the ramifications of forcing signal values.

In a notable example, PUTFACs were used in an IVPC to fill up reservation stations (registers containing dispatched instructions waiting for an execution unit to become available). For fixed point units, doing so exercised the stall mechanisms in the sequencer unit pipelines as well as in the adder, multiplier and divider sub-units.

While the example IVPC succeeded in exercising the sequencer stall mechanisms, it also managed to a hide a bug at the interface of the reservation stations of the load-store unit (LSU). According to the design, all units interfacing the reservation stations (operand source/destinations) get the “RS_full” signal when the reservation stations are full. In addition the LSU would get a signal “RS_full_but_one” before “RS_full”, that lets it decide whether it can send two operands simultaneously. The bug in LSU made it ignore “RS_full” when it was in “RS_full_but_one” state while serving a misaligned exception. This allowed the LSU to erroneously send an operand to the reservation stations when they were full thereby overwriting an existing operand. The tests using the IVPC completely missed this bug. The PUTFACs directly placed the LSU in “RS_full” state, thereby preventing the LSU from sending operands to the reservation stations. However, this bug was caught during random testing with other testcases during the course of the design progress.

6. Coverage Assessment

6.1 Event monitoring and Assertion checkers

Two basic approaches were used to examine various states within the model. These were IVPC coverage “monitors” and IVPC “checkers”. Monitors extracted information from the simulation model as the simulation executed. They were built to gain visibility into the model. They were especially useful during the debug process. A simple monitor might observe a datapath only when valid data is expected to exist.

Checkers evolved from monitors and were an implementation of automated monitoring, rather than human monitoring. Checkers were sometimes referred to as “self-checking” monitors because they checked for error conditions. When illegal conditions were detected, the checker immediately failed the testcase. Checkers provided output detailing when in the simulation and where in the design failures occurred.

6.2 Coverage Analysis

Event coverage through monitors was a valuable scheme for measuring the thoroughness of logic exercise. It targeted specific desired events. The complexity of events ranged from the simple, one signal events to more complex, multi-cycle combinations of signals and states.

For example, an event could be defined as a busy resource (e.g., a bus unit or simple fixed point unit). Monitoring this simple event verifies that some amount of stress is applied to the resource in question. If the resource is never busy, then the unit is not being tested adequately. Where the busy condition is itself a net in the model, then the defined event is equivalent to simple toggle coverage of this net. A slightly more complex category of event monitoring could be to monitor for the busy condition for all possible combinations of fullness of the reservation stations for a resource. Illegal conditions (say the resource responded busy if all reservation stations were empty) are coded causing immediate failure of the testcase. Finally, complex multi-cycle events may be coded. One example of a multi-cycle event would be to watch for the start of a bus transaction, termination, then followed by a valid retry condition.

In the case of the PowerPC 604 microprocessor, this last strategy was utilized in checking the bus interface. Originally the checker was designed to just check for all bus transaction types, but quickly grew to encompass a wide range of events, ranging from the simple detection of transaction types to multi-cycle events like bus sequences (start of bus transaction, wait for response). Simple state machines that check multi-cycle operations were easily implemented in an IVPC.

Once events were coded in the IVPC checker, testcases were generated and run with the monitor and events collected in a database. Testcases were initially generated by random methods and later augmented with hand generated, focused testcases. The random testcases picked up a large proportion of the simpler events quickly, eliminating the need to hand code testcases in order to capture all the events. Some corner case were more difficult to conceptualize and took more care to uncover.

Randomly generated testcases were automatically saved along with their run-time options in an event database. Testcases were added to the database only if they contained new events or combinations not already contained in the database. Database support programs were run periodically (e.g. daily) to determine what percentage of events were covered, and what events lacked coverage. The output of these support programs consisted of combinations of covered events and not-covered events. The outputs indicated how well the collection process was proceeding, and directed the verification effort to those areas that needed better coverage.

The database itself was very useful since it captured a large collection of testcases and run-time options together. These formed an extended regression suite that was run on iterations of the simulation model in order to show that proposed bug fixes did not break other correct model behavior. Periodically the event database was reset and the regression suite of testcases was rerun to insure that all events were covered by new models.

The process was iterative and new corners were often identified while event coverage/collection was in progress. The collection of events ceased when all events had been found by the combination of random and hand generated testcases or when it could be demonstrated that an event could not occur. Table 1 contains some statistics from event coverage analysis.

Since construction of checkers and monitors was labor intensive, an automated approach for state machine analysis was applied to a subset of the bus interface unit logic as described in reference [10].

TABLE 1. Event Coverage

IVPC Coverage Checker	# of events specified	# of events found	Percent Covered
linefill buffer coverage	13	13	100
copyback buffer coverage	133	120*	100
bus events coverage	71	71	100

* 13 events specified in this checker were never found. Upon close examination with the design engineer’s help, these conditions were realized to be impossible events. The next step for these 13 events would be to recode them as checking errors, rather than treat them as events needing to be found.

7. Design Quality and Stopping Criteria

Specifically, two primary metrics were tracked after stable model milestone was achieved in order to support the decision to tape-out: cumulative simulation cycles and defect rate.

The simpler of these two metrics is cumulative cycles. Cycles are tracked on a common simulation cycle database server. The database also holds information about defects vs. model revision. On a regular basis during the project, plots are generated which show progress toward simulation cycle goals determined at the project start. A sample plot is provided in Figure 9.

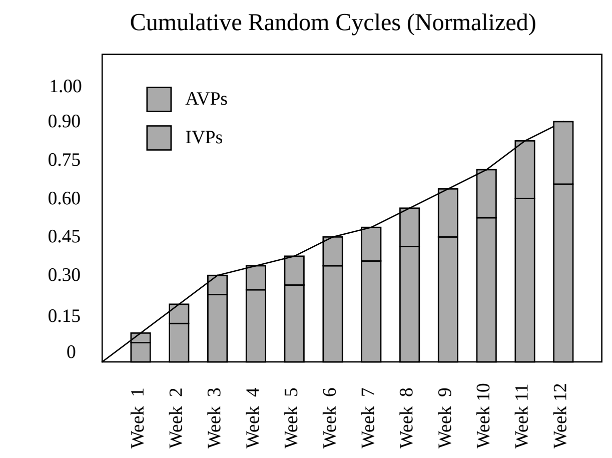

Figure 9 - Accumulated Simulation Cycles

Accumulated defects from the project start date are also plotted on a weekly basis. For the PowerPC 604 microprocessor project, the defect rate decreased logarithmically as the project proceeded.

As the rates of change in total defects decreased, a series of design reviews were held between verification and logic design engineers in an attempt to uncover events which may not have been monitored or tested. Additional testcases were constructed and run prior to tapeout.

The defect rate fell to zero during the two week period prior to projected tapeout while a constant number of simulation cycles were exercised daily. This provided confidence in the design quality. A sample chart tracking cumulative defects, shown in Figure 10, was used to determine the defect rate.

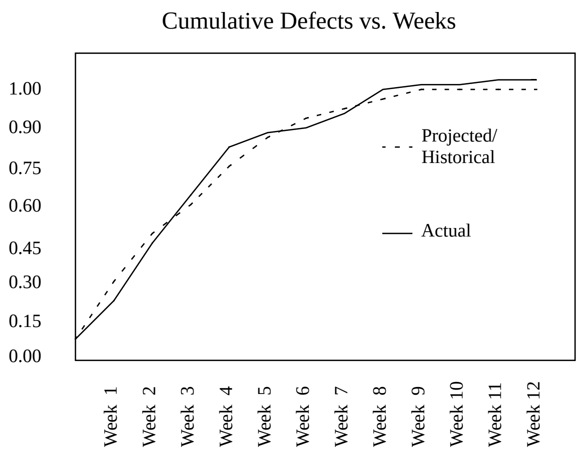

Figure 10 - Normalized Cumulative Defects vs. Project Timeline

8. Results

First pass functioning silicon was produced for the PowerPC 604 microprocessor. It was capable of booting several operating systems including MINIX, MAC O/S, AIX™, and Microsoft NT™.

9. Conclusions

The approach to functional verification described in this paper demonstrates several important advances. First, it quantifies model goodness on a sustained basis through most of the project development lifecycle. Second, it establishes a standard approach to project development. Last, it encourages discipline for process improvement.

A significant area for verification methodology improvement involves the large number of simulation cycles exercised for the PowerPC 604 microprocessor project. These numbers cannot be sustained as the variation and proliferation of this family of microprocessor grows unless a substantial commitment is made to acquire a geometrically increasing supply of computing resource.

Rather than rely solely on the volume of random simulation cycles and declining bug rates as goodness criteria, improvement to coverage measurement is proceeding. Enumerative exercise of all combinations of processor logic is impossible. Bounding the verification problem by stressing block interfaces and targeting complex state machine and combinational logic is proceeding with promising results. More optimal utilization of budgeted simulation resource should be realized without compromising model quality.

Acknowledgments

The authors are grateful to Jesse Wilson and Bill Virun of Somerset, and the Design Automation Conference reviewers for their suggestions in improving this work.

References:

[1] "The 68060 Microprocessor Functional Design and Verification Methodology", J. Freeman, R. Duerden, M. Miller, and C. Taylor, Proceedings of the Design SuperCon '95 On-Chip Design Conference, March 1995, Santa Clara, CA.

[2] "Verification Methodology and Approach for the PowerPC 603^{TM} Microprocessor Family", G. Thuraisingham, M. Pham, and S. Reeve, Proceedings of the Design SuperCon '96 On-Chip Design Conference, February 1996, Santa Clara, CA.

[3] "A New Metric for Determining Completeness of Design Vectors", L. Drucker and B. Vaughn, Proceedings of the Design SuperCon '96 On-Chip Design Conference, February 1996, Santa Clara, CA.

[4] "The PowerPC 604 ^{TM} RISC Microprocessor", S.P.Song and M. Denman, Intnl. Symposium on Computer Architecture, April 18-20 1994.

[5] "The PowerPC^{TM} 604 Microprocessor Design Methodology", C. Roth, R. Lewelling, and T. Brodnax, International Conference on Computer Design, 1994.

[6] "The PowerPC^{TM} Architecture: A Specification for a New Family of RISC Processors", C. May, E. Silha, R. Simpson, and H. Warren, Morgan Kaufmann Publishers, San Francisco, CA, 1994.

[7] "PowerPC^{TM} 604: RISC Microprocessor User's Manual", Technical Document, Motorola Order Number MPC604UM/AD,1994.

[8] "Test Program Generation for Functional Verification of PowerPC Processors in IBM", A. Aharon et. al.; pp. 279-285, Proceedings of the 32nd Design Automation Conference, June 1995, San Francisco, CA.

[9] "An Automatic Simulation Environment for PowerPC^{TM} Design Verification", J. Monaco and J. Kasha, Proceedings of the Design SuperCon'95 On-Chip Design Conference, March 1995, Santa Clara, CA.

[10] "Functional Coverage Assessment for PowerPC^{TM} Microprocessors and Chipsets", D. McKinney, B. Plessier, and J. Monaco, Proceedings of the Design SuperCon '96 On-Chip Design Conference, February 1996, Santa Clara, CA.