VERIFYING THE FM9801 MICROARCHITECTURE

DESIGNERS USE FORMAL LOGIC AND A THEOREM PROVER TO VERTIFY THAT A COMPLEX MICROARCHITECTURE ALWAYS EXECUTES ITS INSTRUCTION SET CORRECTLY.

Warren A. Hunt, Jr.
IBM Austin Research Laboratory

Jun Sawada
University of Texas at Austin

Hardware verification accounts for a considerable portion of the costs in the microprocessor design process. Traditionally, designers have verified microprocessor designs using simulation techniques that help find most design faults. However, simulation never guarantees the correct operation of the final product. Some design faults are very difficult to detect by simulation; they may slip through the verification process into manufactured chips, raising costs. We believe that verification costs can be reduced by the judicious application of formal methods, which should lower the overall costs of design.

Our approach

Formal verification is an alternative to the simulation process. It mathematically analyzes the hardware design and verifies that it is functioning correctly. Typically, when a formal verification process fails because of design faults in the target hardware, it provides clues that identify bugs. In fact, hardware designers are more interested in finding bugs than in verifying their design, and formal verification is a powerful method for finding the most elusive bugs. (See the "Simulation vs. formal verification" sidebar on the next page.)

The key to formal verification of a microprocessor is the use of completely precise models. We have used the logic of the ACL2[1] theorem prover to formally specify abstract models of the FM9801: a pipelined, superscalar microprocessor of our own design. The use of formal logic for specifying our microprocessor permits us to formally verify our design. Since our models are also executable specifications, we can test the design by simulation as well.

To investigate our ability to model and verify a microprocessor, we created a pipelined, multi-issue design. It includes Tomasulo's algorithm, memory-write buffering, load-bypassing, external and internal exceptions, a branch prediction mechanism, speculative execution, and privileged instructions, and permits the self-modification of program code. We included these features to ensure that we had specified a system containing a number of the features commonly found in modern microprocessor designs. Using the ACL2 logic, we specified this design both at a microarchitectural (MA) level, where all of the features just mentioned are apparent, and at the instruction-set (ISA) level, where only the instruction set is specified. Using the ACL2 theorem-proving system, we mechanically proved that the MA description implemented the ISA specification.

Using formal techniques as an aid in hardware design is spreading.[2-4] The precision of using formal techniques can improve the

Simulation vs. formal verification

The industry often applies the word *verification* to performance, functional, test, timing, and so on. Here, we mean that we can exhibit a mathematical proof that some specification or design has a desired property. For our work, we represented all properties and designs using the ACL2 logic, and carried out proofs within the ACL2 formal logic using its associated mechanical theorem prover.

To clarify, consider the following specification: $3x$. Imagine that the only "gates" available to a designer are an adder and a storage device. Typically, industrial designs are specified using a hardware description language such as VHDL, and a designer might record that design as:

```
latch <- x + x;
output <- latch + x;
```

assuming that the signal assignments both occur at the same time as some implicit clock.

A designer would typically check that the design in question is correct by trying different numbers for x in both the specification and the design to see if the same result is computed: this is simulation. Another way to check that our design implements $3x$ is to prove their equality using algebra; this is verification. No cases need be tried, and no "vectors" need be run. Our implementation requires two steps to compute its result, therefore, we would need to prove that after two steps the variable *output* does indeed contain $3x$. Thus, we need to prove that $(x + x) + x$ is $3x$.

We use formal verification to prove that $(x + x) + x$ does implement $3x$ for all values of x. This kind of approach can be much less expensive than testing. Just to ensure that our circuit design really triples a 64-bit number, we must try all 2^{64} combinations. The number of test cases often grows geometrically with size of circuit designs.

In a nutshell, a single mathematical proof can provide the confidence provided by an exhaustive set of functional tests, often at a cost far lower than the cost of exhaustive simulation. For something as large as a microprocessor, complete functional testing is not an option because the number of different states is so large that no full testing regimen can be completed. Note that these proof techniques may be used in different ways, for example, for architectural and behavioral validation, comparison of implementations with specifications, and comparison of one implementation with another.

We verified the FM9801 microprocessor in a manner similar to that of our simple example. The FM9801 is specified with an ISA step function, which returns the state after executing one instruction or, if there was an interrupt, returns the state after vectoring to the interrupt handler. The complete MA specification of the FM9801 is also defined with a step function, but in addition to the programmer visible state, all of the additional internal states required for our specific implementation are present. By repeatedly stepping these specifications, we can (symbolically) simulate the behavior of both the ISA and the MA over a number of steps.

Each step of our MA specification can be thought of as one "clock tick" for a physical implementation. Our MA specification runs at a different rate than our ISA specification. Its rate is determined by its design, the speed at which the memory unit responds, and by the quality of the branch prediction. Because of the different rate at which the design executes instructions as compared to the ISA execution rate, we are forced to create a rate-mapping function, which we call a witness function, between the two specifications. Then we compare the MA with the ISA at certain points. In the case of the FM9801, our proof approach ensures that the program counter, registers, and memory are identical when the implementation is in a flushed state.

design process. Historically, hardware description languages have been used to specify hardware designs, but these languages do not have a formal basis. Various hardware description languages have been formalized,[2,6,7] but without great impact due to the complexity of the resultant specifications. Circuits with the complexity of simple microprocessors[3,8] have been given mathematical specifications, and their designs have been mechanically proved to implement their specifications. The construction of these specifications has limited their usefulness; that is, these models were often constructed with only verification in mind. This kind of one-dimensional approach hampers the transfer of hardware verification techniques to commercial engineering practice.

The commercial use of formal verification has, to this point, been limited to the verification of properties local to small pieces of a design. To our best knowledge, the industry has not verified that a specific microarchitecture implements its ISA specification.

Our approach is to build executable models, but our resulting models are formal, and therefore, can be mathematically analyzed as well as executed. Thus, our models have at least dual use: as a simulator and as the object of our formal verification.

Informal machine description

We have specified our microprocessor design at the MA and ISA levels. Both levels are specified as executable interpreters, but the MA specification contains a number of cooperating finite-state machines (FSMs) that implement various processor features such as the reorder buffer and the different ALUs. The ISA specification is an abstraction of the design, only specifying the possible changes to the programmer visible state, which are shown as shaded boxes in Figure 1.

The ISA interpreter function takes as arguments the current ISA state and an external interrupt input, and it returns the programmer visible state after executing a single instruction or responding to an interrupt. All components shown in Figure 1 are included in the MA next-state function. This function takes a current MA state and its external inputs as arguments, and returns the total state after one clock cycle. The ISA model is not pipelined; that is, a complete instruction is interpreted with each invocation.

Our ISA model specifies the behavior of 11 different instruction classes. The ISA specification also includes actions for external interrupts and internal exceptions, as the changes caused by these exceptions are visible to the programmer. When an exception occurs, the FM9801 ISA model saves some states in special registers, changes its internal state to supervisor mode, and finally jumps to an address specific to the exception type.

The MA specification consists of a number of cooperating FSMs, including the exception mechanisms, the branch prediction unit, and the memory-write buffers. This design fetches and commits instructions in program order, but it can simultaneously issue as many as three instructions to the execution units. The machine design can have as many as 15 instructions in flight, and as many as 12 of these instructions may be executing speculatively. Each instruction is first fetched, then decoded and dispatched to an appropriate reservation station, to await its operands. Once the operands of an instruction are available, the instruction and its operands are issued to an execution unit, after which the result is written to the reorder buffer. Instructions are committed to the memory and register file in program order. Speculatively executed instructions from a mispredicted execution branch proceed no further than the reorder buffer.

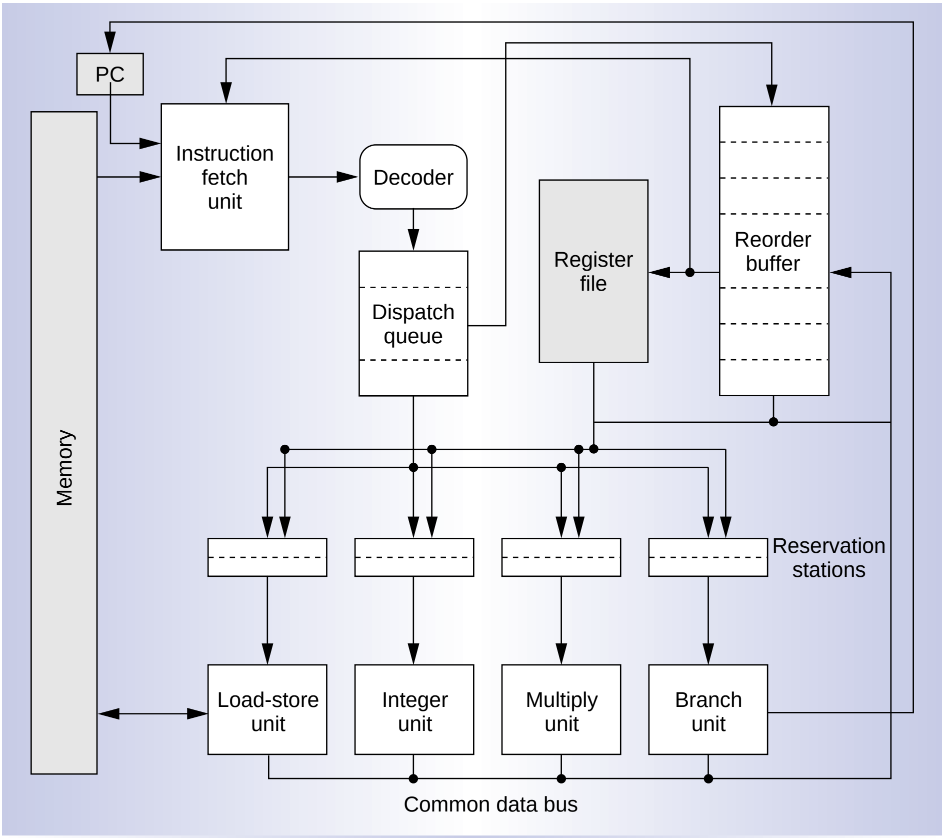

Figure 1. Block diagram of our pipeline machine design.

Our MA design includes four types of exceptions: fetch errors, decode errors, data access errors, and external interrupts. The first three exceptions are generated internally. Exceptions are handled in a precise manner: all instructions preceding an exception must complete their operation, and all partially executed instructions following an exception must be abandoned with no side effects. Because the synchronization requires completion of partially executed instructions, a large number of machine cycles may be required before our MA design actually starts exception handling. If multiple exceptions are detected in the pipeline, only the earliest exception in program order is processed.

Machine proof requirements

To demonstrate that the MA design correctly implements its ISA specification, we prove that the symbolic execution of the MA always produces the same result in the programmer visible state. The relationship between the MA and ISA machines is subtle because of the different rates at which they operate. Each step of the ISA specification completely executes one instruction, where each step of the MA specification models one clock cycle of execution that may or may not retire an instruction while many instructions may make some progress. The key difference is the complicated time abstraction between the two models; the data abstraction is a simple projection of the programmer visible state from the MA state.

Several different methods to specify the cor-

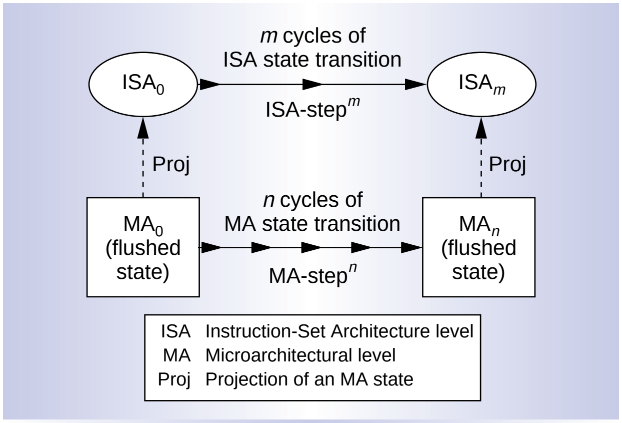

Figure 2. Correctness diagrams.

rectness of a pipelined processor have been proposed. Most methods are a variant of Burch and Dill's correctness criterion involving pipeline flushing.[9] Although this criterion with flushing has been extended to cover superscalar processors,[10] it does not address speculative execution and external exceptions. For our work, we used the correctness criterion shown in Figure 2.

Our verification criterion states that by starting the two models in related states, the programmer visible state changes in the MA design are exactly related to the programmer visible state changes in the ISA specification. The problem in comparing the states of a pipelined MA and its ISA is the existence of partially executed instructions. A simple projection of an arbitrary MA state can produce an inconsistent ISA state that does not appear during the corresponding ISA execution. Instead, we restrict the initial and final states to a flushed pipeline state; that is, no partially executed instructions are in those states. We do not directly compare the intermediate MA states to the ISA, but these states must be correctly executing instructions so that the final result is always correct. In fact, our verification process obliges us to verify the correctness of intermediate states by proving invariant conditions discussed later.

Correctness criterion: For an arbitrary MA execution sequence from a flushed state MA_0 to another flushed state MA_n, there exists a corresponding ISA execution sequence from ISA_0 to ISA_m. This sequence executes the same instructions as occur in the MA execution sequence, and satisfy $ISA_0 = proj(MA_0)$ and $ISA_m = proj(MA_n)$.

The proof of this criterion requires us to show the existence of m, which is the number of instructions that are executed during the MA execution from MA_0 to MA_n. We exhibit such m by constructing a *witness function*. This function calculates the exact number of executed instructions, excluding speculatively executed instructions that are abandoned in the middle of their execution.

Our correctness criterion checks whether instructions are correctly executed independently of how branches are predicted. In a correctly designed MA, speculatively executed instructions will have no side effects on the programmer visible state. Since the ISA specification exactly characterizes the machine behavior without speculation, comparison of the MA execution against its ISA specification should reveal incorrect side effects from the invalid speculation.

We want to show that the MA design implements exceptions correctly, but we also have to show that the MA implements precise exceptions. Since the ISA specification executes instructions sequentially, it exactly specifies the nature and order of exceptions. Since the MA design may execute instructions both speculatively and in an out-of-order fashion, it is possible for exceptions to occur that really do not occur in the program. For instance, if an exception occurs in a speculatively executed instruction, it may be that this instruction is never actually executed. The MA design must not process such an exception until it is known that the exception-causing instruction will actually be executed.

External exceptions make the problem more complicated. The ISA interpreter function takes an external interrupt signal as one of its arguments, and this specification describes the action of an external interrupt in a similar manner to the way it does for internal exceptions. Unlike internal exceptions, however, the ISA specification does not specify which instruction should be interrupted by an interrupt signal. There can be multiple correct MA implementations that interrupt different instructions for the same interrupt signal. Interference between the external interrupt, internal exceptions, and the speculative execution further complicates the exception behavior. The sequential execution by the ISA always defines the ideal MA behavior, because the MA behavior should appear to the programmer as if it were a sequential machine.

The nondeterminism of the ISA step function introduced by the external signal can lead to different final ISA states, as shown in Figure 3. The commutative diagram holds only for the ISA state transitions, which interrupt the same instructions as the MA does. Since supplying different environments to the MA will cause different instructions to be executed and interrupted, we need to find the corresponding ISA sequence for each MA state sequence with different input signals. Our correctness criterion has been expanded to incorporate this aspect.

An interesting problem in the verification of pipelined microprocessors is the issue of self-modifying code. All processors, in some sense, must permit the self-modification of the program code, as just the act of loading a program into memory is a modification of the program memory. During program execution by our MA, there are a number of instructions inside the machine at one time.

Consider an instruction that modifies the next instruction in the program memory. Our ISA specification will modify the program memory, and when the ISA specification is called upon to execute the next instruction, it will read the modified instruction. Conversely, our MA design executes an instruction over a number of clock cycles, and there are a number of instructions in various pipeline stages. When the MA design reads an instruction, it takes several clock cycles to finish executing this instruction; at the same time the MA design fetches the next instruction before completing the previously fetched instruction. In this case, our MA design does not read the same instruction as the ISA specification.

In the proof of our correctness criterion, we assume that the sequence of instructions between flushed states is not modified. Multiple sequences of this type can be appended, and our correctness criterion is maintained. This will be sufficient to guarantee the correct behavior of self-modifying code with appropriate synchronization. For instance, program code can be loaded by a small program that

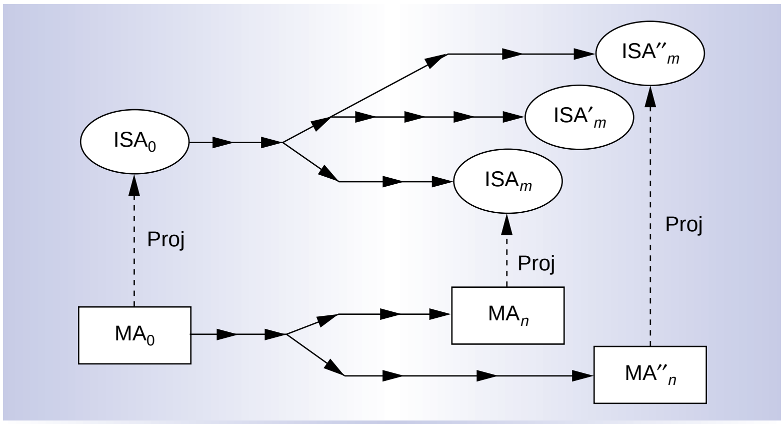

Figure 3. Correctness diagrams for external interrupts.

does not modify itself and flushes the pipeline by executing a SYNC instruction. Then, the loaded program code can be safely executed.

Obviously, our correctness criterion does not imply that our microprocessor design works correctly in every environment. For instance, the liveness of our processor design is not part of our correctness criterion. It could be separately proven that no deadlock will occur in the pipelined machine. The criterion does specify that external interrupt signals are processed correctly, but it does not guarantee that all the interrupt signals actually interrupt our machine design. For instance, if there is an interrupt every clock cycle, the MA cannot process them all. For a real-time system, we would want to prove that our processor does respond to an external interrupt in a bounded amount of time with appropriate memory responses.

Design invariants

Invariant conditions are the properties that the machine satisfies in every reachable state. Invariant properties often realize the designer's idea of correct machine behavior. Finding and verifying invariant conditions was our most difficult task; we had to manually analyze the hardware to identify invariant conditions.

Identifying and verifying invariant conditions is a key to our approach. We start by defining a small set of invariant conditions that specifies the correct state of the programmer visible components. We then incrementally strengthen this initial set of conditions. For instance, while proving an invariant property about the register file, we may have to know that the reorder buffer is

ACL2

ACL2 (A Computational Logic for Applicative Common Lisp) is both a programming language and a system for analyzing models. The ACL2 language implements a subset of Common Lisp. We use ACL2 much like a hardware designer would use VHDL or C; that is, we use it to specify our designs at various levels of abstraction. Also, like VHDL and C, we execute (simulate) our models to evaluate their properties. The critical difference between ACL2 and a language like VHDL is that there is a formal definition of every ACL2 language construct. We use these formal definitions to deduce properties of our models.

Consider the following definition of a ripple-carry adder in C:

```
bit  a[64],  b[64],  c_in;    //inputs
bit  sum[64],        c_out;  //outputs
bit c;

c = c_in;
for( i=0 ; i < 64 ; i++ )
    {
        sum[ i ] =  xor3( a[ i ], b[ i ], c );
        c = maj3( a[ i ], b[ i ], c );
    }
c_out = c;
```

We can execute this program (fragment) by initializing arrays *a* and *b* and the variable *c_in*. The results will be placed in the array *sum* and variable *c_out*. We can do the same thing by defining several ACL2 functions.

```
(defun adder (a b c i n)
 (if (and (naturalp i) (naturalp n) (< i n))
     (let ((a_i       (logbit i a))
           (b_i       (logbit i b)))
         (logcons  (xor3 a_i b_i c)
                   (adder a b (maj3 a_i b_i c)
                              (+ i 1) n)))
      c))

(defun c_out (a b c_in)
    (logbit 64 (adder a b c_in 0 64)))

(defun sum (a b c_in)
    (loghead 64 (adder a b c_in 0 64)))
```

Imagine that we would now like to prove a commutative property of our adder; that is, we want to prove that any two initial values for *a* and *b* can be swapped without affecting the result. In C, the only mechanism for "proving" this is to execute this program with all of the possible initial values of *a* and *b*. This would require a great deal of simulation time as there are roughly 10^{38} possible combinations. C does not have rules for manipulating programs; in fact, C does not provide mechanisms for specifying programs. VHDL is only slightly better in this regard as it provides the assert command as a runtime check, but it does not provide specification nor reasoning mechanisms.

The meaning of every ACL2 built-in primitive has been given a formal meaning—these are called axioms. We know that each axiom is valid. In addition, a number of axioms about propositional logic, rational numbers, and lists establish initial truths about the ACL2 primitive functions. For instance, zero is an integer, and zero is different than every other integer.

ACL2 also provides rules of inference to derive additional truths (theorems) from the initial axioms. By repeated application of these rules (such as the commutativity of addition), we can establish the validity of very complicated models. For instance, ACL2 has been used to prove the correctness of a number of things: the FM9801 microprocessor, the divide algorithm of the AMD K5 microprocessor, and Motorola's CAP DSP pipeline, among others. These proofs involve very large models with extremely large state spaces. Both the ISA and MA models of the FM9801 processor include the full memory.

To facilitate the validation of very large models, the ACL2 authors have mechanized the application of rules of inference. That is, the authors have provided a computer program that manages the definition of models and checks the proof requests a user submits. In this sense, ACL2 is like model checking.[5,12] However, ACL2 has a much more expressive logic than that provided by a model-checking system. As such, ACL2 has been used to specify the operation of an entire microprocessor—something that would be extremely difficult to handle with modeling checking.

The verification of theorems using ACL2 is much like proving properties in high-school algebra. The theorem prover is led to a proof by providing it with a graduated sequence of steps (lemmas) sufficiently "close together" so that the theorem prover can fill in the missing parts of the proof. If a user asks the ACL2 system to consider a proof that is "too hard," it will either fail, run forever, or exhaust the available computing resources. In this way, the ACL2 system differs from model checking, which provides an algorithm that runs to completion, assuming there are adequate memory and computational resources.

A drawback of using the solely algorithmic approach provided by model-checking systems is that the size of the models that can be checked is quite limited. ACL2 can handle larger models: ACL2 can reason about arbitrarily large models but requires that the user structure the proof so that ACL2 can check it. We use ACL2 to check the validity of very large models. In the FM9801 verification, we used the ACL2 system to check that the MA model correctly implements the ISA model. We, as well as many other researchers, are looking for ways to combine model checking with a theorem-proving approach, such as provided by ACL2.

working correctly. This can be characterized by a new design invariant. To show that the newly defined invariant holds, we may further have to define new invariants about execution units, and so on. Eventually, we have a sufficiently strong set of invariants, from which we can show the validity of the MA design.

The verified invariants must satisfy six requirements:[11]

- if the MA is in a flushed state, every instruction in the MA is committed;

- the program counter must correctly point to the next instruction to be fetched, unless it is fetching instructions speculatively;
- the register file contents must match at the two levels when the instructions are committed;
- the memory contents must match when instructions complete their memory operations;
- the initial (flushed) state must be well-formed and satisfy the invariants; and
- each time the MA steps it returns a well-formed state satisfying the invariants.

We can prove our correctness criterion from the invariants satisfying these requirements. In this sense, our proof approach reduced the problem of checking the correctness criterion to the problem of verifying our design invariants. During the invariant strengthening process, we have used the ACL2 theorem prover to verify every invariant condition. (See the "ACL2" sidebar.) The verification of invariant conditions is also the place where we discovered faults in our example design.

Verification summary

We have used the ACL2 theorem prover to completely verify that the entire architecture is working correctly. We first ran test programs on our executable specification to eliminate easy bugs, and then applied our formal verification techniques. Our ACL2 specification of the machine and the proof scripts are available at http://www.cs.utexas.edu/users/sawada/FM9801.

The human cost of conducting such verification is a serious practical concern. To guide the ACL2 theorem proving system to a proof, we had to manually write many lemmas. In Table 1, we list the size of the ACL2 proof script files and the time to run the ACL2 theorem proving system on these files using a 200-MHz PentiumPro processor. We have found that the human effort devoted to the corresponding task is generally proportional to the size of the ACL2 prover scripts, not to the CPU time. We wrote the ISA and MA specifications in one month, but the whole verification project took about 15 person months.

Although the verification task is labor intensive, our technique seems to scale well with the machine size. We have compared the size of our machine specification and the verification script for two other projects[13] in which we employed a similar approach, but with different example machine sizes. Table 2 lists these results.

During the creation of the verification scripts for the FM9801, we found 14 design faults in our MA design. Most of these design faults were subtle bugs, since easy bugs had already been eliminated by simulation. We believe some of the bugs would have been difficult to discover with simulation techniques. For instance, we found a bug that required seven instructions issued in certain timing with an unusual memory delay to be exposed. We also found a performance bug that did not expose incorrect behavior visible to the programmer.

All of these bugs were found during the invariant verification. When the theorem prover failed to prove an invariant condition, we found that some microarchitectural components did not behave as we had expected. This often revealed a bug that may cause incorrect behaviors visible to programmers.

Other analyses and documentation

The level of precision that we have employed here is necessary to permit the formal verification of microprocessors of this

Table 1. ACL2 script size and CPU time for different verification phases.

Type of ACL2 script	Script size (Kbytes)	Certification time (minutes)
Definitions of ISA and MA	140	14
Intermediate abstraction	55	6
Definitions of invariants	89	7
Proof of shared lemmas	481	58
Proof of invariants	1,034	211
Proof of criterion	37	11

Table 2. Sizes of ACL2 proof scripts for different machine design examples.

Verified machine design examples	Specification only (Kbytes)	Total script (Kbytes)
Small example machine	13	169
CAV '97 design	78	757
FM9801 design	140	1,909

We hope efforts like the FM9801 example will persuade industry to consider using formal verification techniques at the microarchitectural level as well as at the register-transfer level for property checking.

complexity. This level of precision also provides a basis for other analyses. An important product of our work is that exactly the same models can be used for simulation and other analyses. Thus, the cost of producing a special (possibly inaccurate) model is unnecessary.

For instance, to carry out our functional proof obligation, we were required to construct a witness function that calculates the exact number of instructions executed by the pipelined design. It is possible to prove bounds on clocks per instruction (CPI) from the witness function, given that the memory delays and the branch prediction delays are known. Also, the use of formal models does not imply lower simulation performance. Brock reported that his ACL2 model simulated a digital signal processor faster than its equivalent Cadence-based specification.[13]

The formal specification of microarchitecture is necessary for formal verification. Sharing the same model for simulation and other analysis can reduce the cost of creating such models. The executable nature of our formal models is especially important for the acceptance of formal verification by design engineers. In this way, the formal specification of high-level design does not take anything away from a designer, but it does add another verification procedure.

We also want to see our formal specifications used as documentation for a hardware system. We are working toward the point where documentation for a hardware system is a formula manual—a collection of fully formal specifications that provide a precise and complete basis for using and designing a specified device. To assure that the formula manual is correct, it will be necessary to *prove* that a design meets *all* of its specifications. This formula manual can be used as a reference for the system programmers who need the precise specification of the device. It can also be used as a reference for the register-transfer-level designers, so that the ambiguity of microarchitectural specification written in natural languages can be avoided.

The FM9801 microprocessor is one of most complex designs ever submitted to a complete formal verification. We verified that our MA design would always get the answer predicted by the FM9801 ISA specification.

The use of executable formal logic to represent our hardware design has provided us with means to perform both simulation and formal verification. It also provided a basis for other analyses such as performance analysis. Breaking the verification problem into a number of smaller invariant proofs was critical to the verification of such a large design. We used the ACL2 theorem-proving system to mechanically certify the correctness of all proof scripts. During the course of our verification, we found 14 subtle bugs in the design, which had escaped detection during the simulation.

Our verification technique seems to scale well with respect to the size of the implementation, even though the process is labor intensive. At this moment, the lack of the integration of other automated verification techniques, such as model checking, required us to verify all the invariant conditions with the ACL2 theorem prover. Reducing the cost of verifying invariant conditions is an important problem for future research. An interesting approach to combine theorem proving and model checking is already emerging.[14]

Industrial microprocessor specifications are written in natural language augmented with charts, graphs, and tables. Formal definitions of high-level specifications are rare even for small sections of industrial designs and nonexistent for large designs. Since there are no formal specifications with which to compare RTL or MA designs, industry does not employ formal verification at the level we have presented here; however, this level of verification is arguably the area of the design process that needs the most assistance. We hope efforts like the FM9801 example will persuade industry to consider using formal verification techniques at the microarchitectural level as well as at the register-transfer level for property checking. MICRO

References

1. M. Kaufmann and J.S. Moore, ACL2: An Industrial Strength Version of Nqthm, *Proc. 11th Ann. Conf. Computer Assurrance, COMPASS-96,* IEEE Computer Society Press, June 1996, pp. 23-34.
2. B.C. Brock and W.A. Hunt, Jr., "The DUAL-EVAL Hardware Description Language and Its Use in the Formal Specification and Verification of the FM9001 Microprocessor," *Formal Methods in System Design,* Kluwer Academic Publishers, Dordrecht, Netherlands, Vol. 11, Issue 1, July, 1997, pp. 71-104.
3. M.K. Srivas and S.P. Miller, *Formal Verification of an Avionics Microprocessor,* Tech. Report CSL-95-04, SRI Int'l Computer Science Laboratory, Menlo Park, Calif., June 1995.
4. B.C. Brock, M. Kaufmann, and J.S. Moore, "ACL2 Theorems About Commercial Microprocessors," *Formal Methods in Computer-Aided Design (FMCAD'96),* Lecture Notes in Computer Science 1166, M. Srivas and A. Camilleri, eds., Springer Verlag, Berlin, 1996, pp. 275-293; also http://www.cs.utexas.edu/users/moore/acl2/index.html.
5. K.L. McMillan, *Symbolic Model Checking,* Kluwer, Boston/Dordrecht/London, 1993.
6. *Formal Semantics for VHDL,* C.D. Koos and P.T. Breuer, eds., Kluwer, 1995.
7. M.J.C. Gordon, "The Semantic Challenge of Verilog HDL," *Proc. 10th Ann. IEEE Symp. Logic in Computer Sci.,* (LICS'95), IEEE CS Press, pp. 136-145.
8. B.C. Brock, W.A. Hunt, Jr., and M. Kaufmann, *The FM9001 Microprocessor Proof,* Tech. Report 86, Computational Logic, Inc., Austin, Tex, Dec., 1994, http://www.cli.com/reports/files/86.ps.
9. J.R. Burch and D.L. Dill, "Automatic Verification of Pipelined Microprocessor Control," *Proc. Computer-Aided Verification (CAV '94),* Lecture Notes in Computer Sci. 818, D. Dill, ed., June 1994, Springer-Verlag, pp. 68-80.
10. J.R. Burch, "Techniques for Verifying Superscalar Microprocessors," *Proc. Design Automation Conf. (DAC '96),* ACM Press, N.Y., June 1996, pp. 552-557.
11. J. Sawada and W. Hunt, Jr., "Processor Verification with Precise Exceptions and Speculative Execution," *Proc. Computer-Aided Verification (CAV '98),* Lecture Notes in Computer Science 1427, A.J. Hu and M.Y. Vardi, eds., Springer Verlag, 1998, pp. 135-146.
12. C-J.H. Seger and R.E. Bryant, *Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories,* Tech. Report 93-8, Computer Sci. Dept., Univ. of British Columbia, Vancouver, 1993.
13. J. Sawada and W. Hunt, Jr., "Trace Table Based Approach for Pipelined Microprocessor Verification," *Proc. Computer-Aided Verification, CAV '97,* Lecture Notes in Computer Science 1254, Springer Verlag, 1997, pp. 364-375.
14. K.L. McMillan, "Verification of an Implementation of Tomasulo's Algorithm by Compositional Model Checking," *Proc. Computer-Aided Verification (CAV '98),* Lecture Notes in Computer Science 1427, A.J. Hu and M.Y. Vardi, eds., Springer Verlag, 1998, pp. 110-121.

Warren A. Hunt, Jr., is a research staff member at IBM's Austin Research Laboratory and a research fellow at the University of Texas at Austin. His research interests include hardware verification, circuit design, and mechanized theorem proving. Hunt holds a BSEE from Rice University and a PhD in computer science from the University of Texas. He is a member of the IEEE, IEEE Computer Society, and ACM.

Jun Sawada is a PhD student at the University of Texas at Austin. His research interests include hardware design and verification, automated theorem proving, model checking, and hardware specification languages. He holds an MS in mathematical science and a BS in mathematics from Kyoto University, Japan.

Direct comments about this article to Warren Hunt, IBM Corporation, Mail Stop 9460, Building 904, 11501 Burnet Road, Austin, TX 78758; whunt@austin.ibm.com.