Efficient Simulation of Formal Processor Models

Matthew Wilding, David Greve, and David Hardin

Rockwell Collins, Inc.
Advanced Technology Center
Cedar Rapids, IA 52498 USA

{mmwildin, dagreve, dshardin}@collins.rockwell.com

Abstract

Computer systems under development are routinely modeled by simulators, and formal verification can be integrated into conventional computer system development by reasoning directly about such simulators. However, simulators must be extremely fast to be usable in a real development effort. We have crafted a model for a simple processor in the logic of the ACL2 theorem prover that supports both formal analysis and efficient execution, with performance near that of a simulator written in C. We describe our approach using the simple model and indicate how we are applying it to our latest microprocessor.

1 Introduction

Processor simulators are routinely built as a part of computer system development because testing a model of a computer system under development is extremely useful. Simulators allow the development of different parts of the system in parallel. For example, an executable model of the microarchitecture of a microcoded processor allows simultaneous microarchitecture implementation and microcode development. Models have also been built in order to support formal computer system analysis. Most of these kinds of processor models have been created after the processor is built in order to establish the correctness of the processor design, support symbolic simulation, or to verify software for the processor to execute [3, 4, 8, 12, 10, 19, 20, 24], and typically multiple models are built to support analysis and simulation.

We believe that the unification of simulation and analysis models has advantages. Most obviously, it is easier to develop and maintain fewer models. An easily overlooked issue that arises in formal verification work is processor model validation, which is crucial to ensure that formal analysis applies to the actual machine, and with a unified model we can informally validate the formal model through its use by developers as a simulator.

1.1 Formal Model Validation

Formal processor models have been validated using a variety of approaches.

direct analysis Some efforts have generated hardware directly from the formal model being analyzed. The FM9001 verification work is an example of this approach, wherein the processor design is shown through machine-checked proof to implement a desired model [11]. Since the processor design is used to fabricate the processor, to the extent that the fabrication process is trusted the validation of the formal model is not an issue [1]. This approach is attractive but may require considerable investment for large-scale processor development and does not allow for convenient expression of computer system models at different levels of abstraction.

analysis of translated model Another approach is to use a translator to derive an analyzable model from a hardware design [5, 23]. This approach is also very attractive when the translator is trusted, but has the same drawbacks as direct analysis.

inspection Formal models can be validated by inspections, as was done for the AAMP5 and AAMP-FV verification efforts [19, 20]. This approach requires a great deal of effort in order to gain a high level of assurance in the model.

testing If the model is executable then one can execute test vectors and compare the results to the result of running the tests against the actual processor. Examples of this approach include the 68020 verification work [8] and the CAP architecture model [10]. Validation of a formal model using this approach has the standard disadvantage of testing, which is that it is difficult to construct test suites that fully exercise the model.

We are investigating another approach for validating a formal processor model: building an abstract model of our processor suitable for both formal microcode analysis and simulation. The formal model will be validated in part by having engineers use it as they develop the system. This paper describes our experience with a prototype executable processor model that we have used to experiment with this approach and details a portion of an executable model we have developed of a real processor.

1.2 A Theorem Prover that Supports Efficient Execution

Various verification projects have used theorem provers to analyze computer system models [3, 4, 7, 8, 19, 20, 10, 24, 25], which suggests that these kinds of models are amenable to the formal verification, and the use of theorem provers as digital design tools is an active area of research [15]. What is not as clear is whether the languages about which theorem proving systems reason can also be used to support simulation. Execution must be extremely fast – similar in speed to the models that are currently used – and the language must be capable of integration with debugging and simulation environments in order to become part of a real development process.

Some theorem proving systems can evaluate ground functions on constants. For example, two experimental additions to the PVS theorem prover [21] allow evaluation of these kinds of terms. One prototype PVS extension is a symbolic evaluator that uses the PVS theorem prover decision procedures in a relatively automatic manner; this significantly speeds evaluation relative to conventional theorem proving strategies but is still substantially slower than conventional program execution [22]. A second prototype PVS extension is a translator of PVS functions into Common Lisp, and preliminary results (using a version of the benchmark we present in this paper) indicate that execution speeds are within an order of magnitude of the speed of a model written in a conventional programming language. The Nqthm theorem proving system [6] also allows relatively fast execution of its Lisp-like logic on constant terms.

Execution speeds needed for real, production simulators can best be obtained by expressing the model in a real programming language where execution efficiency is of central importance. Our choice for expressing a single model for analysis and simulation is the logic of the ACL2 theorem proving system [16]. ACL2 provides a theorem proving environment for machine-checked formal analysis and its logic is an applicative subset of Common Lisp. The fact that ACL2 reasons about a real programming language suggests that an ACL2 model might be made to run fast.

Although unifying the models supporting simulators and formal analysis has advantages, we know of no other research group attempting to integrate a formal model suitable for machine-checked verification into their development methodology by using it to build a simulator for microcode developers. This paper describes our effort to build this kind of model using ACL2. We present examples of analyzable model use, but the main focus of this paper is the use of this kind of model to support efficient simulation. We describe how we overcome the inefficiencies normally associated with functional program execution to build a formally analyzable simple computer system model that runs very fast.

pop a Pop a data stack value into location **a**
pushs a Push the value at location **a** onto the data stack
pushsi i Push **i** onto the data stack
add Pop two data stack values and push their 32-bit sum
jump a Set the program counter (PC) to **a**
jumpz a Set the PC to **a** if popped data stack value is 0
call a Push the next address on the call stack and set the PC to **a**
ret Set the PC to the popped call stack value
sub Pop two values and push the maximum of 0 and their 32-bit difference
dup Duplicate the top of data stack

Figure 1: TINY Instructions

2 A Simple Formal Hardware Model

We introduce a model for a simple processor, TINY. We demonstrate an ACL2 proof about a particular TINY program, but mostly we will show how we speed TINY model execution as benchmarked against a version of the TINY model written conventionally in the C programming language [14]. We use TINY to present our technique and in Section 4 indicate how we use it on a real processor.

The TINY machine is a simple, stack-based machine whose state is composed of a program counter, a memory of integers, a data stack top-of-stack pointer, a call stack top-of-stack pointer, and a halt flag. Figure 1 lists the TINY instructions. The TINY model is represented in the ACL2 logic as an interpreter, following the convention of many previous formal verification projects [7, 8, 25]. We introduce ACL2 (and hence, Common Lisp) definitions that interpret instructions with respect to their effect on the TINY state. The ACL2 term `(tiny s n)` evaluates to a TINY state that reflects the effect of executing `n` instructions on an initial TINY state `s`. The function `tiny` models the TINY machine by describing how each possible instruction affects the TINY state.

Figure 2 presents a TINY program that calculates the integer remainder of two positive values in memory using repeated subtraction. Since we have expressed the TINY model in the ACL2 logic, we can use the ACL2 theorem prover to prove interesting properties about this code. For example, we have proved (assuming several arithmetic facts) the following ACL2 lemma:

```
(let ((s2 (tiny s (remclock s))))
  (implies
    (good-initial-remainder-state s)
    (equal (getmem (mem s2) (1+ (dtos s2)))
           (remainder (getmem (mem s) 18) (getmem (mem s) 19)))))
```

```
location      instruction

   24         pushs 18
   26         pushs 19
   28         sub
   29         dup
   30         jumpz 36
   32         pop 18
   34         jump 24
   36         pushs 19
   38         pushs 18
   40         sub
   41         jumpz 45
   43         jump 49
   45         pushsi 0
   47         pop 18
   49         pop 19
   51         pushs 18
```

Figure 2: A TINY Program for Calculating Remainder

The term `(good-initial-remainder-state s)` holds if `s` is a TINY state with the PC set to location 24, positive values are at location 18 and 19, the remainder program is loaded, and the call and data stacks are at particular places in memory. The function `remclock` calculates the number of instructions required by the remainder program, and `tiny` is the TINY interpreter. This theorem formalizes the specification that the program in Figure 2 deposits the integer remainder on the top of stack as we expect.

This lemma took one of us nearly a week to prove using ACL2. However, a system of proved reasoning rules and supporting facts could be built to reason easily about TINY programs in a manner similar to [7, 8, 24, 25].

3 Model Execution

3.1 A Need for Speed

Since definitions in the ACL2 logic are executable in Common Lisp, we can execute TINY programs using this model. How fast must a processor model execute for simulation needed in processor development work? For our purposes a model must execute no more than 2-3 times slower than our present tools in order to be usable. Some of the test suites that we run on the model of our current generation processor [13, 26] require several hours to execute, and if they could not be run overnight it would interfere with the development process.

Using the remainder program of the last section as a benchmark, the ACL2 model executes about 19,000 TINY instructions per second[1]. This may be good from the standpoint of a proof system, since proofs typically require evaluation of relatively few constant expressions. But we wish to use this model to support simulation and in this regard it is far too slow. For comparison purposes we implemented a TINY processor model in C. The C TINY program runs about 2.47 million TINY instructions per second, or 130 times faster than the ACL2 model. This degradation is unacceptable and would preclude our integrating a formal ACL2 processor model into our development process.

In this section we present several optimizations and discuss how they support faster model execution without sacrificing model analyzability.

3.2 ACL2-Provided Optimizations

ACL2 provides support for speeding execution of terms expressed in its logic. The subset of Common Lisp that ACL2 supports allows certain declarations that can be used by the compiler to improve efficiency. These declarations speed execution and generate corresponding proof obligations that ensure that the optimizations are made correctly. An example of this in the TINY model involves the arithmetic operations used to define the model. ACL2 integers are arbitrarily large numeric values, which execute less efficiently than the 32 bit operations provided by the host machine.

For example, the definition of the TINY interpreter function includes the term

```
(<= n 0)
```

We can restrict the values of n and 0 to be representable by 32 bits by replacing this term with

```
(<= (bv32 n) (bv32 0))
```

where `bv32` is the following Common Lisp (and ACL2) macro:

```
(defmacro bv32 (x) `(the (signed-byte 32) ,x))
```

This declaration has two effects when we introduce the function containing this term. First, the definition is compiled using 32-bit operations rather than Common Lisp's arbitrarily-large-integer arithmetic. Second, ACL2 requires we prove

[1]Benchmark times in this paper use ACL2 2.0 running GCL 2.1 on a Sun Ultra 30.

that `n` and 0 are representable in 32 bits. This typically requires an additional ACL2 "guard" to the function that ensures that the function is evaluated only on values that lead to this term's evaluation on values representable in 32 bits. The ACL2 theorem prover is usually able to establish this kind of property without user assistance, as it does in this case.

By sprinkling these declarations appropriately throughout the text we speed the execution of the TINY model to about 41,000 instructions per second, or 60x slower than the corresponding TINY model written in C. We use the ACL2 theorem prover to prove the generated proof obligations, so the correctness of the ACL2 system guarantees the correctness of these optimizations.

3.3 Applicative Common Lisp Optimizations

The ACL2 approach of using an existing programming language as its logic allows us another approach for execution speed optimization of ACL2 models. When we compile our ACL2 model using the GCL Common Lisp compiler we can replace ACL2 functions with fast versions. We must of course preserve the semantics of any such replaced functions to ensure that the behavior of the model corresponds to the behavior that can be derived from the ACL2 formalization of Common Lisp. We describe two uses of this kind of optimization.

3.3.1 32-Bit Operations

As we described in the previous section, we use ACL2 declarations to speed execution and use the ACL2 prover to show that the optimization is correct. However, in a few cases the functionality we wish for our processor model is missing in the ACL2 language but efficiently implemented on the machine upon which we run ACL2 programs. For example, the "add" instruction of TINY described in the previous section involves 32-bit addition, but no 32-bit addition function exists in ACL2. We can define an ACL2 function that computes a 32-bit addition in terms of the ACL2 functions "+" and "logand", but it does not execute efficiently. We resolve this problem by instructing the Common Lisp compiler to replace references to the 32-bit addition function we have defined in ACL2 with a fast 32-bit addition.

We are preserving the semantics of ACL2 with this optimization, so assuming that we implement the replacement function correctly this optimization will have no effect on the results of executing the model. We have replaced ACL2 definitions for 32-bit addition and subtraction with fast versions. Optimizing the arithmetic of the TINY model to use 32-bit arithmetic exclusively speeds execution of our benchmark to about 57,000 instructions per second, or about 43x slower than the C model.

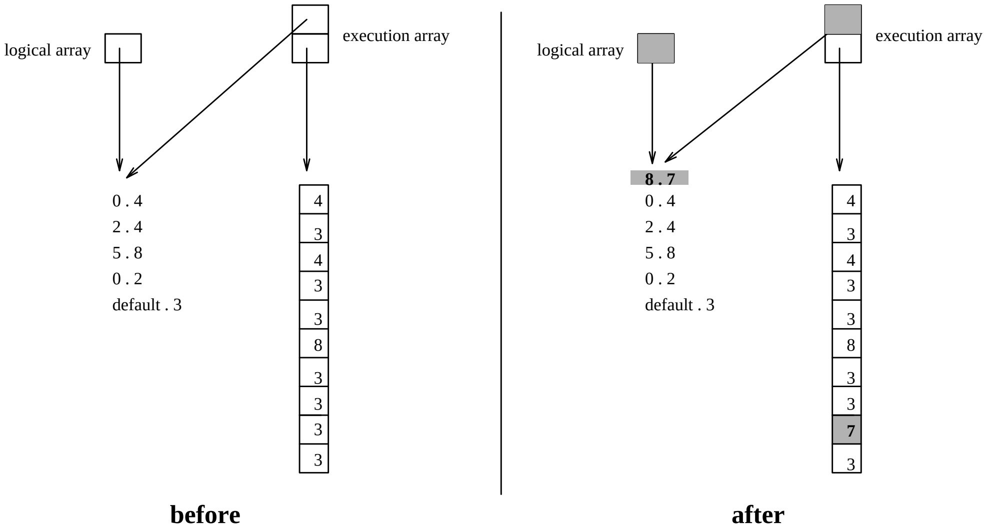

Figure 3: An ACL2 Array Update

3.3.2 ACL2 arrays

We can also optimize model execution performance by speeding array operations. Algorithms typically run faster implemented in an imperative language rather than in a functional language because destructive operations are more efficient than non-destructive operations. Non-destructive operations have overhead from unnecessary copying and the dynamic allocation and deallocation of memory. This is a particular issue when executing computer system models since the state of the system being modeled is accessed and updated frequently.

ACL2 provides a mechanism for relatively efficient execution of applicative array update and access [16]. Each array is represented in ACL2 as a list of location/value pairs, a functional style that allows for convenient reasoning. ACL2 also maintains a standard destructive array representation for each array, and when possible ACL2 uses the efficient representation when evaluating array operations. Figure 3 presents an example array as implemented by ACL2. Note that the example array has length 10 and is represented in two ways: (1) an association list of location/value pairs and (2) a conventional array together with a pointer to the list representation.

ACL2 provides one-dimensional array operations, including the ACL2-provided functions `aset1` and `aref1` for setting and referencing an element of an array. Function `aset1` updates the list representation of the array with the new value and, if the conventional array contains the same values as the list representation, updates the conventional array and list pointer. If the conventional array contains the same values as the list representation then `aref1` performs an access on the conventional array. Otherwise it searches the list representation to return the proper value, which can be very slow.

Given this implementation, how could the two representations of an array get out of synchronization while executing an ACL2 program, causing slow array access? There are two ways: manipulation of the array using a function other than the ACL2-provided array functions, and multiple-threaded array manipulation. For example, using the ACL2 "cons" function to add an element to the array would lead to slow array access on subsequent array references. Also for example, if during evaluation an array is updated before a reference is made to the original array then the reference will be slow.

Figure 3 presents an example of updating an ACL2 representation of an array. ACL2 checks if the two array representations are in synchronization by comparing the association list pointers. The update is reflected with a location/value pair added to the logical array list and a new value in the conventional array and a new logical array pointer. Note how this example could be changed to cause the two representations to go out of synchronization, and thereby cause slow array accesses. We could modify the logical representation with a non-array operation that would not update the efficient representation. We could also use a multiple-threaded array update such as

> set location 3 of array **arr** with the value of the fourth element of the array resulting from setting location 5 of array **arr** to 2

Since the results of array operations are always the same as using the list representation, we can reason about an ACL2 array as if the array were only represented as a list. The creators of ACL2 implement arrays in this fashion to preserve the applicability of their logic while providing relatively fast execution speed. Unfortunately, even after being sped by the arithmetic optimization of the previous section, as previously mentioned our simple benchmark using ACL2 arrays is 43x slower than a similar C program. This is too slow for our use as a simulator. Our challenge is to make these operations even faster without sacrificing our ability to analyze models that use them.

3.3.3 Fast arrays

We speed execution of ACL2 arrays by replacing their implementation with an implementation we call (appropriately enough) *fast* arrays. Rather than maintaining two representations of the array as in the ACL2 implementation, fast arrays are implemented using a conventional destructive array and a counter that records the number of array updates. The fast array implementations of `aset1` and `aref1` signal a LISP error when the array being accessed is not

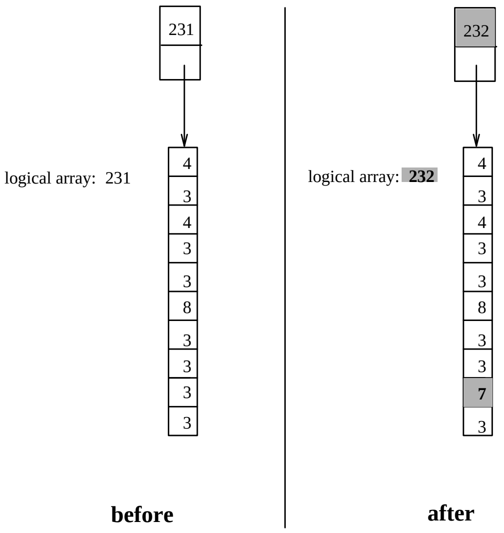

Figure 4: A Fast Array Update

current. Figure 4 presents the effect of an update of a fast array. The counter is incremented, and the corresponding array is updated appropriately.

What effect does this optimization to ACL2 have on the meaning of a model expressed in the ACL2 logic? If a model employs a function other than an array-specific operation to manipulate an array then the results are unpredictable. However, multiple-threaded access to arrays that would cause a slow array operations in raw ACL2 are detected and cause a program error with fast arrays.

We take care to avoid both of these uses of arrays in our model, so we expect fast arrays to have the same behavior as raw ACL2 arrays. However, if we should by mistake build a model that uses multiple-threaded array access, the fast array implementation would signal an error during execution. Since inaccurate results will not be generated in this circumstance we do not fear making this mistake. However, the use of non-array operations on fast arrays might lead to unpredicatable results. We have inspected our model to ensure that arrays are only accessed and updated using the standard ACL2 array functions, but we note that this is not a formal guarantee.

It should be emphasized that this opimization is installed “under the hood” from the standpoint of ACL2. When using our model to reason about TINY programs in the ACL2 theorem-proving environment this optimization is transparent and we use the axioms provided by ACL2 about array operations. It is only when

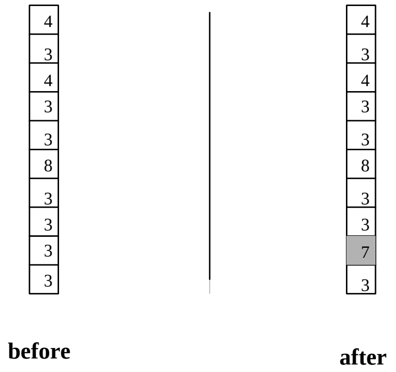

Figure 5: An Inlined Array Update

executing the model that the optimization's effect is seen.

The prototype TINY model with arithmetic optimizations and fast arrays executes 476,000 TINY instructions per second, far faster than the earlier version and about 5x slower than the TINY C model.

3.3.4 Inlined arrays

We have investigated another implementation of arrays, called *inlined* arrays, which is a conventional, destructive array. Figure 5 presents an example of updating an inlined array – we merely destructively modify the array element.

This approach omits both the association list representation of raw ACL2 arrays and the update counter of the fast arrays. Thus, inlined arrays do not provide a run-time check that ensures that multiple-threaded use of arrays will not have unintended results. If we violate our model-building rule and use a multiple-threaded array reference, then the execution of our model might differ from the behavior of the model that can be derived by reasoning about it using ACL2 axioms. We have inspected the TINY model to ensure that we have not broken this rule, but this is not a formal guarantee.

This implementation of the array operations executes the TINY benchmark at about 2,253,000 instructions per second, or about 90% of the speed of the C TINY benchmark.

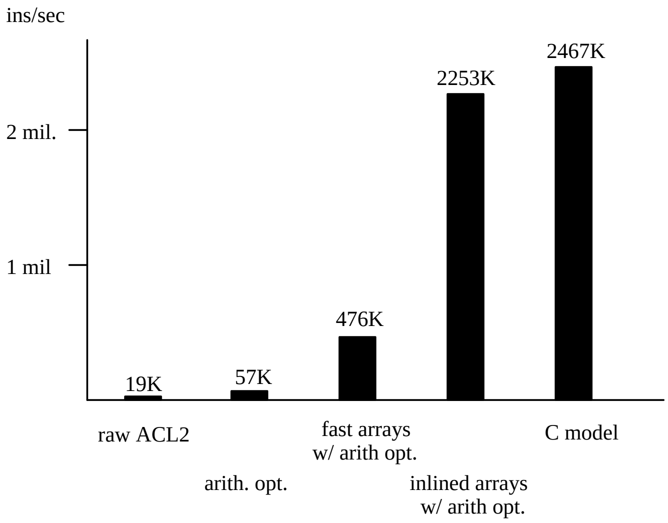

Figure 6: TINY Model Execution Speeds

3.3.5 Overview of Optimizations

Figure 6 summarizes the execution speeds of our benchmark program. In the most optimized case we have lost some of the advantage of our single model, since we have no check that our model uses arrays in the restricted sense that guarantees that our optimizations do not affect the result. However, the most optimized execution of the ACL2 benchmark is nearly as fast as our C version. We can trade faster model execution speed for a run-time check that avoids the multiple-threaded array update problem.

4 Application to a Real Processor

We are applying the techniques described in Section 3 to the construction of analyzable and fast-executing processor models at the microarchitecture level for our production microprocessor designs. The Rockwell Collins JEM1 is the world's first Java direct-execution microprocessor [13, 26]. In this section we describe a part of the JEM1 processor model that models the JEM1 ALU in order to show how a simulator for a real design can both be reasoned about and used to support mainstream development.

The JEM1 ALU is quite conventional; its operation is described at the microarchitecture level by the Common Lisp functions presented in Figure 7. This model uses several kinds of subsidiary Common Lisp functions and macros:

```
(defun adder (a b cin st)
  (declare (type (signed-byte 32) a)
           (type (signed-byte 32) b)
           (type (unsigned-byte 1) cin))
  (let ((sum (+<32> (+<32> a b) cin)))
       (declare (type (signed-byte 32) sum))
  (let ((cv  (logxor<32> sum (logxor<32> a b))))
       (declare (type (signed-byte 32) cv))
  (let ((co (logior<32>
              (logand<32> cv a)
              (logior<32>
                (logand<32> cv b)
                (logand<32> a  b)))))
       (declare (type (signed-byte 32) co))
  (let ((ov  (logxor<32> cv co)))
       (declare (type (signed-byte 32) ov))
        (jemst  (put-carry_d<st>  (if (< (bv32 co) 0) 1 0)
                (put-fout<st>     sum
                (put-alu_ovr<st>  (if (< (bv32 ov) 0) 1 0)
                 st)))))))))

(defun alulog (fout st)
  (declare (type (signed-byte 32) fout))
    (jemst  (put-carry_d<st>    0
            (put-fout<st>       fout
            (put-alu_ovr<st>    0
             st)))))

(defun alu_operate (st)
  (jemst
    (case (logand<32> #b0111 (aluf<4> st))
          ((#b0000) (adder  (s_in<32> st) (lognot<32> (r_in<32> st))
                            (cin<1> st) (jemst st)))
          ((#b0001) (adder  (s_in<32> st) (r_in<32> st)
                            (cin<1> st) (jemst st)))
          ((#b0010
            #b0011) (alulog (logxor<32> (s_in<32> st) (r_in<32> st)) (jemst st)))
          ((#b0100) (adder  (lognot<32> (s_in<32> st)) (r_in<32> st)
                            (cin<1> st) (jemst st)))
          ((#b0101) (alulog (logand<32> (s_in<32> st) (r_in<32> st)) (jemst st)))
          ((#b0110) (alulog (logior<32> (s_in<32> st) (r_in<32> st)) (jemst st)))
          ((#b0111) (if (equal (z<1> st) 0)
                        (adder (s_in<32> st) (r_in<32> st) 0 (jemst st))
                        (adder (s_in<32> st) (lognot<32> (r_in<32> st))
                               1 (jemst st))))
          (t (unspecified_aluf<st> st)))))
```

Figure 7: ACL2/Common Lisp model of the JEM1 ALU

- state-accessors: `aluf<4>`, `s_in<32>`, `r_in<32>`, `z<1>`
- state-updaters: `put-carry_d`, `put-fout`, `put-alu_ovr`
- arithmetic and logical operators: `logxor<32>`, `+<32>`, etc.

The convention followed in this code is that macro `foo<type>` expands to a call to function `foo` with declarations consistent with type `type`. For example, `lognot<32>` expands to a "logical not" function that is declared to have 32-bit arguments and return value. We have not presented enough information in this paper about Common Lisp or ACL2 for the reader to comprehend this model fully, but we wish to emphasize that, except for the selection of Common Lisp rather than C as the language, this model is typical of the kind of model that is built to support the development of any real processor.

Following the approach given in Section 3, we speed the Common Lisp execution of the model given in Figure 7 by replacing accessing and updating functions with fast, destructive versions. As with the fastest version of the TINY model, we have not added a single-threadedness check to this model.

We have integrated the ALU model into a production development environment. The JEM1 contains the Java Virtual Machine [18] opcodes in its instruction set [13, 26], and tools allow the execution of Java code on either the JEM1 hardware or a JEM1 simulator. An example of running the Averstar JWatch debugger [2] can be seen in the screen dump of Figure 8. The part of the JEM1 simulator related to the ALU is exactly the compiled code in Figure 7, so running Java code using this tool exercises our model. Like the optimized TINY code of the previous section, the optimized JEM1 model runs very fast.

We use *the same model* to prove theorems about this part of the JEM1 model. Figure 9 lists a particularly interesting result about `alu_operate`, which is the `alu_operate` of Figure 7. The function `alu-result` specifies the expected result of the ALU operations, and the theorem formalizes the notion that the overflow bit generated by the ALU is set exactly when the ALU result is unrepresentable by a 32-bit signed value. The proof of this lemma using the ACL2 theorem prover required less than a man-week of effort, and benefited from a previously-constructed library of ACL2 theorems related to hardware [9].

5 Future Work

We believe that the optimized execution of our ACL2 prototype benchmark validates the idea that a single model can support both formal analysis and simulation. Our experience using these techniques to model and analyze real processor models suggests that this approach scales.

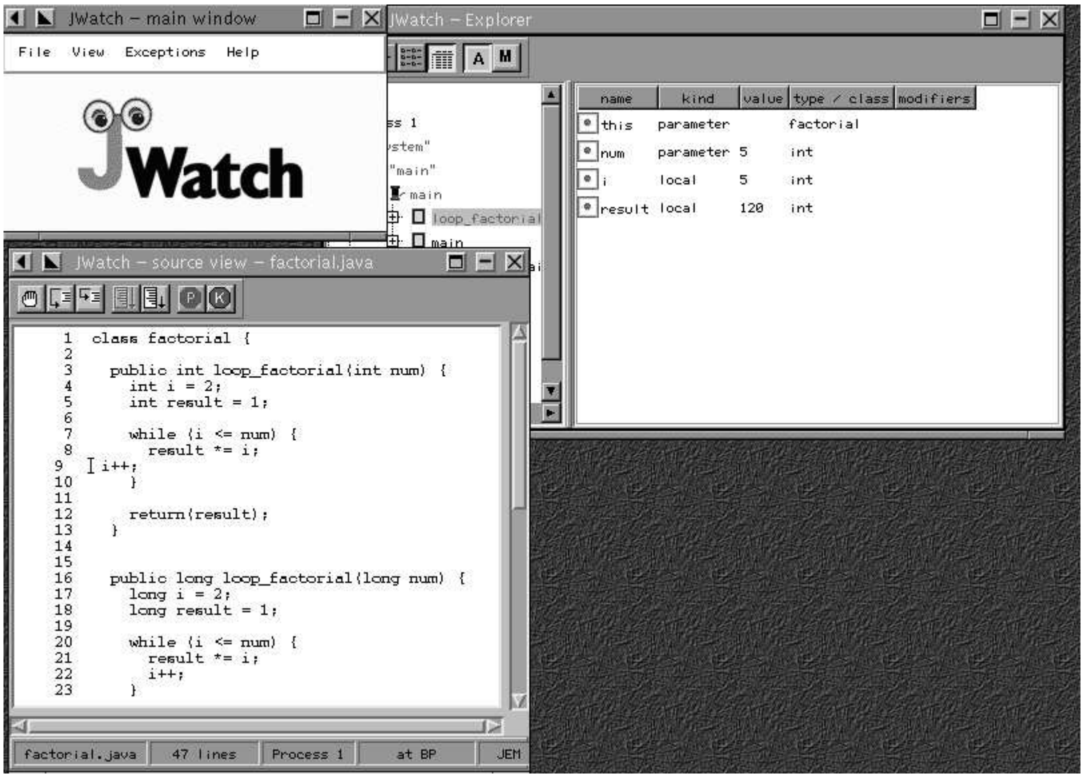

Figure 8: JEM1 simulator environment utilizing the ALU model of Figure 7

```
(defun alu-result (st)
  (let ((sin (s_in st))
        (rin (r_in st))
        (cin (cin st))
        (z (z st)))
  (case (logand (aluf st) #b111)
        ((#b000) (- (+ sin cin) (1+ rin)))
        ((#b001) (+ sin rin cin))
        ((#b010 #b011) (logxor sin rin))
        ((#b100) (- (+ rin cin) (1+ sin)))
        ((#b101) (logand sin rin))
        ((#b110) (logior sin rin))
        ((#b111) (if (equal z 0) (+ sin rin) (- sin rin))))))

(implies
 (alu-state-p st)
 (equal (OVR (alu_operate st))
        (if (signed-byte-p 32 (alu-result st)) 0 1)))
```

Figure 9: ACL2-certified JEM1 ALU overflow theorem

Overcoming the slowness of applicative data structures is crucial in order to use a single model for formal analysis and simulation. ACL2's response to this challenge for arrays is ingenious but falls short of what we need to support truly fast simulation. Our fastest optimization has sufficient speed, but could give inaccurate results if used in a multi-threaded manner. The creators of ACL2 recently have pointed out to us that this kind of check, which is at the heart of the difference between applicative and imperative execution, is done internally by ACL2 in order to speed theorem prover operations involving the ACL2 notion of "state" [17]. The right solution may be for ACL2 to support a check that ensures that definitions modify a particular datatype using only single-threaded execution. Such a facility made accessible to users of ACL2 would allow the kind of performance optimizations we describe in this paper in a way guaranteed not to impact behavior. We are currently investigating how to check that a function accesses a data-structure in a single-threaded manner by generating an obviously single-threaded version that ACL2 can prove is equivalent.

We plan to explore further how formal processor models can improve our computer system development work. In the short term this will include the integration of symbolic results into our design process and the use of symbolic results to aid our regression testing [12]. In the long term we plan to verify large portions of our designs using theorem provers. We expect that building an executable formal processor model will become a standard part of our processor development process.

References

[1] Kenneth L. Albin, Bishop C. Brock, Warren A. Hunt Jr., and Lawrence M. Smith. Testing the FM9001 microprocessor. Technical Report 90, Computational Logic, Inc., January 1995.

[2] Averstar Inc. JWatch home page, 1998. http://www.jwatch.com.

[3] William R. Bevier. KIT: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1368–81, November 1989.

[4] William R. Bevier, Warren A. Hunt Jr., J Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

[5] Mark Bickford and Damir Jamsek. Formal specification and verification of VHDL. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design – FMCAD*, volume 1166 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[6] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, Boston, 1988.

[7] Robert S. Boyer and J Strother Moore. Mechanized formal reasoning about programs and computing machines. In R. Veroff, editor, *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*. MIT Press, 1996.

[8] Robert S. Boyer and Yuan Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43(1):166–192, January 1996.

[9] Bishop Brock. ACL2 integer hardware specification (IHS) books, 1998. (available on ACL2 home page).

[10] Bishop Brock, Matt Kaufmann, and J Strother Moore. ACL2 theorems about commercial microprocessors. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design – FMCAD*, volume 1166 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[11] Bishop C. Brock and Warren 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*, 11(1):71–104, July 1997.

[12] David A. Greve. Symbolic simulation of the JEM1 microprocessor. In *Formal Methods in Computer-Aided Design – FMCAD*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[13] David A. Greve and Matthew M. Wilding. Stack-based Java a back-to-future step. *Electronic Engineering Times*, page 92, January 12, 1998.

[14] Samuel P. Harbison and Guy L. Steele Jr. *C - A Reference Manual*. Prentice Hall, 1991.

[15] David Hardin, Matthew Wilding, and David Greve. Transforming the theorem prover into a digital design tool: From concept car to off-road vehicle. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification – CAV '98*, volume 1427 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[16] M. Kaufmann and J S. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203 – 213, April 1997.

[17] Matt Kaufmann and J Moore. Personal Communication, January 1998.

[18] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, Reading, Massachusetts, 1996.

[19] Steven P. Miller, David A. Greve, Matthew M. Wilding, and Mandayam Srivas. Formal verification of the AAMP-FV microcode. Technical report, Rockwell Collins, Inc., Cedar Rapids, IA, 1996.

[20] Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, FL, 1995. IEEE Computer Society.

[21] S. Owre, N. Shankar, and J. M. Rushby. *User Guide for the PVS Specification and Verification System (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[22] Stephan Pfab. Personal Communication, November 1997.

[23] David M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division, and square root algorithms of the AMD-K7 processor. Available at http://www.onr.com/user/russ/david/, January 28 1998.

[24] Matthew Wilding. A mechanically verified application for a mechanically verified environment. In Costas Courcoubetis, editor, *Computer-Aided Verification – CAV '93*, volume 697 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[25] Matthew M. Wilding. Robust computer system proofs in PVS. In C. Michael Holloway and Kelly J. Hayhurst, editors, *LFM97: Fourth NASA Langley Formal Methods Workshop*. NASA Conference Publication no. 3356, 1997. (http://atb-www.larc.nasa.gov/Lfm97/).

[26] Alexander Wolfe. First Java-specific MPU rolls. *Electronic Engineering Times*, page 1, September 22, 1997.

Acknowledgments: We thank J Moore and Matt Kaufmann (ACL2 developers) and Bill Schelter (GCL co-developer) for developing their wonderful tools and answering many questions. We also thank Bishop Brock (IHS libraries creator) who took time to make his ACL2 libraries usable by others.