Formally verifying a microprocessor using a simulation methodology

Derek L. Beatty
Cadence Berkeley Laboratories
Cadence Design Systems, Inc.
beatty@Cadence.com

Randal E. Bryant
Carnegie Mellon University
bryant+@cs.cmu.edu

Abstract

Formal verification is becoming a useful means of validating designs. We have developed a methodology for formally verifying data-intensive circuits (e.g., processors) with sophisticated timing (e.g., pipelining) against high-level declarative specifications. Previously, formally verifying a microprocessor required the use of an automatic theorem prover, but our technique requires little more than a symbolic simulator. We have formally verified a pre-existing 16-bit CISC microprocessor circuit extracted from the fabricated layout.

Introduction

Previously, symbolic switch-level simulation has been used to verify some small or simple data-intensive circuits (RAMs, stacks, register files, ALUs, and simple pipelines) [2, 3]. In doing so, the necessary simulation patterns were developed by hand or by using ad-hoc techniques, and it was then argued that the patterns were sufficient, and that their generation could be automated. We have developed sufficient theory to fully support such claims, and used this methodology to verify a representative set of operations (initialization and interrupt as well as instructions) of a microprocessor [1].

To verify a circuit, a designer writes assertions to specify high-level operations. He or she also writes mappings illustrating how abstract state and IO is realized by the circuit. From these, our verifier generates symbolic simulation tests, which are localized in time and space. In other words, they are short patterns (typically single operations) which exercise parts of the circuit. Our theory then guarantees, from these local tests, that an arbitrary sequence of operations will work correctly.

In developing our methodology, our touchstone was a microprocessor called Hector [10, 5]. The complexity of a microprocessor implies that ad-hoc techniques will not suffice, so its verification requires careful attention to methodology. Hector is a 16-bit CISC fabricated in 1985. Its 2-address architecture is similar to the PDP-11, but with more (16) registers and fewer addressing modes. System state is held in the register file and a few condition code bits. The implementation is microcoded; at the microcode level it is slightly pipelined, but at the instruction set level it is not.[1] The bus interface is similar to the Motorola 6800. In addition to a reset line, there is a wait line, DMA, prioritized interrupts, and a single-step facility. Hector has no cache and does not support virtual memory.

Hector includes an ALU with condition codes. Of its addressable registers, 7 are completely general-purpose, and 9 are sometimes specialized (e.g., PC, stack pointer, interrupt vectors). Hector has four addressing modes: register, indirect, indirect with post-increment, and indexed. Additional addressing modes can be synthesized since the stack pointer and program counter are addressable. (This also gives some pathological addressing modes.)

We did not participate in Hector's design. In modeling Hector for verification, we first extracted a switch-level circuit from layout. We made few changes to the extracted circuit—just those necessary to model it at the switch level. (We used the switch level due to our expertise there, but our methodology could be easily used at a higher level instead. The key requirements on the simulation are that it be symbolic, and efficiently support an "unknown" signal value, such as the switch-level X value.)

Thus, we started with a pre-existing circuit. A C program simulating Hector's instruction set was also available, but we did not use it directly. Instead, we wrote a higher-level, declarative formal specification of the instruction set. (It might also be possible to extract such higher-level descriptions from HDL programs [8]. We have not pursued this.)

[1] Actually, there is a very slight degree of pipelining: the processor senses the state of the interrupt lines as it completes execution of each instruction.

Formal verification

"Formal verification" (or simply FV) consists of establishing that a mathematical relation holds between two descriptions of a system. The particular relation established varies with the approach to verification. A high-level description, called the "specification," is taken to be correct and a lower-level description, called the "realization," is checked.[2]

Our approach to FV divides the specification into 2 parts. The main part is a set of assertions, which are descriptions of desired state-transition behavior. The other part describes IO encodings. Once we have verified a circuit, mathematically we have established a relation between IO sequences of the specification and IO signals of the circuit. Informally, everything the circuit does must be something allowed by the specification.

Some researchers make a distinction between "property checking" (or design verification) and "machine comparison" (or implementation verification). The assertions comprising one of our specifications define (implicitly) a state machine. Thus, in a technical sense, we are performing machine comparison. However, strict division between two kinds of verification is unhelpful in evaluating this work—surely one sees a significant difference between the statements "this microprocessor implements this instruction set" and "these two state machines are similar."

Other work

Previously, microprocessors have been formally verified using automatic theorem provers [6, 7, 4, 13]. Our methodology requires little more than a symbolic simulator. Processors have been verified by Madre and colleagues without using theorem provers [8, 9], but the specification and circuit were required to have identical latch structures and timing, and a unique reset state. We do not have these restrictions. We can also use more detailed circuit models, and handle detailed circuit timing including pipelining. Our specifications are at the instruction-set level, a higher level than most previous work. Unlike approaches based purely on logic, we can structure our specifications to avoid the danger of antecedent failure.

Verifying circuits

We give two examples: a latch, to illustrate ideas in simplest form, and Hector, to illustrate their actual application. The first step in formally verifying a design is to specify it formally.

Abstract specification

Data-intensive systems perform operations on data values. Their state transitions implement such operations. A latch has two kinds of operations: it can load a new data value, and it can store an old one. Formally, we can describe such operations using assertions. An assertion consists of two logical formulas, which describe sets of states. Its antecedent, or pre-condition, describes states before the operation occurs. For example, before a "load" operation in a latch, some new value must be given to the latch, and the latch must be told to load the value. An assertion's consequent, or post-condition, describes states afterward. After a "load" operation, the new value will be stored in the latch. For example, we can describe a latch using the two assertions

$$\text{op} = \text{load} \wedge D = v \stackrel{\delta}{\Rightarrow} Q = v$$
$$\text{op} = \text{hold} \wedge Q = v \stackrel{\delta}{\Rightarrow} Q = v$$

These say that if the operation is a "load" and the input D has value v, then afterward the state Q will have value v. If the operation is a "hold" and the state has value v, it will remain v. The symbol $\stackrel{\delta}{\Rightarrow}$, which we read as "then implies" or "leads to," indicates both that the left-hand side implies the right, and that time passes.

A microprocessor has more kinds of operations: it can be reset, it can respond to interrupts, and it has many types of instructions. The Hector microprocessor is reset by applying an external reset signal. This signal must be applied for at least 7 clock cycles,[3] but this detail is extraneous to the *behavior* of the processor, so here we will concentrate on the abstract reset operation, writing the assertion

$$\begin{aligned}\text{control} = \text{reset} \stackrel{\delta}{\Rightarrow}\ & \text{invariant} = 0 \\ & \wedge\ R[\text{PC}] = 0 \\ & \wedge\ R[\text{SP}] = 0 \\ & \wedge\ R[\text{INT}] = 4 \\ & \wedge\ R[\text{NMI}] = 2\end{aligned}$$

It states that if the processor is given its reset signal, it will then enter a state where:

- It will be ready to execute instructions or respond to interrupts, as reflected by an invariant condition, and
- Several registers will have specified initial values, including the program counter, the stack pointer, and the interrupt vectors.

Hector's response to an interrupt is specified by a more complicated assertion. Part of the added complexity is in specifying the conditions under which an interrupt occurs, for we must include more of the processor's state than was necessary in specifying reset behavior. (Since the reset operation makes no use of existing processor state, we did not need to describe initial state in the reset assertion.)

The antecedent of the assertion describes initial state:

$$\begin{aligned}&\text{control} = \text{nmi} \wedge \text{invariant} = 1 \wedge M[J] = d \\ &\wedge\ (r \neq \text{NMI} \wedge r \neq \text{SP} \wedge r \neq \text{PC}) \rightarrow R[r] = v \\ &\wedge\ R[\text{NMI}] = n \\ &\wedge\ R[\text{SP}] = s \\ &\wedge\ R[\text{PC}] = p \\ &\wedge\ \text{cyCC} = \text{cy} \wedge \text{ovCC} = \text{ov} \wedge \text{ngCC} = \text{ng} \\ &\wedge\ \text{zeCC} = \text{ze} \wedge \text{intCC} = \text{int} \\ &\wedge\ (r \neq 0) \rightarrow R[0] = w\end{aligned}$$

[2] In general the specification might either describe the system completely (but abstractly), or give some properties the system should have, or both.

[3] This number was found empirically, then verified formally.

It describes the conditions in which a non-maskable interrupt occurs. A non-maskable interrupt occurs when the abstract input is "nmi." Any arbitrary memory location l holds some arbitrary data word d. Any arbitrary register r (other than the special registers: NMI, which holds the address of the interrupt service routine; SP, the stack pointer; or PC, the program counter) holds some arbitrary value v. The special registers and condition codes hold arbitrary special values n, s, and p. Register 0 also holds some arbitrary word w (unless register 0 was the arbitrary register r selected above; if so, we have already stated that it has a value, namely v).

The consequent of the assertion describes the conditions that follow receipt of a non-maskable interrupt.

$$
\begin{aligned}
&\text{invariant} = 0\\
&\wedge\ (l \neq s-1 \wedge l \neq s-2) \rightarrow M[l] = d\\
&\wedge\ M[s-1]\langle 4:0\rangle = \text{int}\,\text{ze}\,\text{ng}\,\text{ov}\,\text{cy}\\
&\wedge\ M[s-2] = p\\
&\wedge\ R[\text{SP}] = s-2\\
&\wedge\ (r \neq \text{SP}) \rightarrow R[r] = w\\
&\wedge\ \text{cyCC} = \text{cy} \wedge \text{ovCC} = \text{ov} \wedge \text{ngCC} = \text{ng}\\
&\wedge\ \text{zeCC} = \text{ze} \wedge \text{intCC} = 1\\
&\wedge\ R[\text{PC}] = n\\
&\wedge\ R[\text{NMI}] = n
\end{aligned}
$$

After an interrupt is received memory will be unchanged, except for the stack, which will hold the previous condition codes and program counter. The stack pointer will have been updated. Most condition codes will be unchanged, but the "interrupt" flag will be asserted. The program counter will now point to the interrupt service routine (whose address also remains in the NMI register). Since Hector does not allow instructions to be interrupted, i.e., interrupts are sensed only between instructions,[4] this assertion captures all possible conditions in which an interrupt could occur.

It is important to observe that in these assertions there are two different kinds of variables. Some variables—those on the left of "=" signs—correspond to abstract system state (or IO), such as Q in the latch, or R in the microprocessor. Other variables are used in representing the values that these abstract states can take on, such as v in the latch, or d, n, s, w, and v in the microprocessor. We must consider the first kind of variables, which represent components of abstract system state, when we define mappings from abstract state onto circuit state.

Observe that assertions are local properties in two ways. First, each describe a very short computation: an isolated, single state transition. Second, each describes the operation of only part of the circuit. For example, the "hold" assertion for the latch does not involve the D input, and for the processor, the specification of the initialization operation says nothing about the memory. Thus, individual assertions are partial specifications, and do not define the transition behavior of the specification machine, but they constrain the transition behavior,[5] so that the set of all assertions taken together does define a transition relation. Though the assertions are localized, by checking them we can guarantee global properties which hold for computations of any length involving the entire circuit.

Mappings

IO and timing

An abstract description of state-transition behavior is insufficient to specify a circuit. Circuits do not take abstract inputs and produce abstract outputs. Instead, they have input and output signals. Thus, it is also necessary to specify the way in which abstract IO is encoded as circuit IO. Input variables in the assertions (D and op for the latch) will be mapped onto particular voltage levels at particular times. The zero point with respect to which time is measured is the "nominal beginning" of the operation. We will also specify the possible durations of the operation by specifying the "nominal ending" of the operation. (The beginning of any successive operation will, of course, be the ending of the current operation.)

Although we use a textual representation to express mappings in the actual verifier, it is easiest to show this with timing diagrams. Rather than giving nominal beginning and ending time points as numbers, we can more easily give them as identifying markers—vertical lines sketched on a timing diagram. Figure 1 shows how we map the abstract latch operations onto one particular circuit.

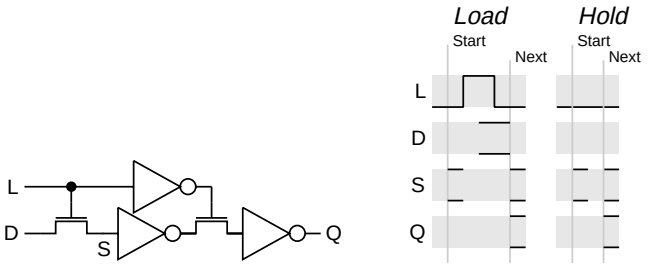

Figure 1: A simple latch and its mapped assertions as timing diagrams. For each signal, a double horizontal line indicates that either a high or a low value might be present; the absence of any line indicates that we don't know or don't care about the value.

State mappings

It is insufficient to consider only the inputs and outputs of sequential systems. Operation depends crucially on stored internal state. Thus, we also describe the way in which abstract state is encoded as circuit state.

[4] This is verified during verification of instructions, by checking that once an instruction has begun execution, it is completed regardless of subsequent activity on the interrupt line while the instruction is completed.

[5] For example, to show that unintended state changes do not occur, it is necessary to express this condition, but it could be written as a new assertion, or incorporated into existing ones.

State variables in the assertions (Q for the latch *assertions*) will be mapped onto charge stored on circuit nodes, over intervals of time measured relative to a "marker." This is illustrated with the last two rows of the timing diagram. Note that, abstractly, the value stored in the latch is also its output value, but in the circuit, the node controlling the stored value (S) and the output (Q) are separate nodes.

For the microprocessor, the mapping is more complex, and it must take into account the separation of processor state and memory state, but it is constructed similarly. Processor state is mapped normally, while memory state is mapped onto memory operations. We illustrate this in Fig. 2. For example, the initial program counter value is mapped onto register 15, Hector's PC register, at the beginning of the operation.

Given such a specification and a symbolic simulation model for a circuit, we can verify it. We check each assertion separately. For each assertion, we use the mappings that we have defined to generate symbolic simulation patterns.

We generate the stimulus using the precondition of the assertion. Mapping the abstract input variables of the specification yields a short circuit input sequence, which also contains two markers. Mapping the abstract state variables of the specification yields a circuit state sequence, defined relative to a marker. We align this marker with the *first* marker of the input sequence to get the entire stimulus.

We generate the response using the postcondition of the assertion. Mapping the abstract state variables of the specification (which also serve as outputs) yields another circuit state sequence defined relative to a marker, but we align this marker with the *second* marker of the input sequence, to get the desired response.

This explains what happens to the abstract input and *state* variables that appeared in the specification. The specification also contained another kind of variables: those used to represent values that the abstract state could take on (e.g., v in the latch, d in the processor). We have not explained what happens to these variables because nothing happens to them: they are preserved, so that they appear in the simulation patterns. (Since we are using a symbolic simulator, variables can appear in simulation patterns.)

We check the generated patterns using symbolic trajectory evaluation, a form of symbolic simulation which allows precise constraining and checking of system state during sequences of operation [12]. This exploits the power of the switch-level model's ternary X value in reducing extraneous analysis of circuit components that do not participate in a calculation (thereby reducing precision, but remaining accurate, i.e., not producing incorrect binary values).[6]

[6] Of course, a simulator that propagated X values indiscriminately would be too blunt a tool.

More about mappings

The specification here is simplified. In the actual specification [1, appendix B], references to values stored in memory are given with an extra parameter, a small integer. It is a "hint," used to establish the clock cycle on which a memory operation takes place. Formally, hints are unnecessary. To be most general, assertions should be mapped so that they allow *any* sequence of memory operations that yields the desired effect. For example, the order in which locations are read from memory does not matter. However, checking all possible orders is expensive. For Hector, it is easy to identify the specific order that actually is used, by examining the instruction level simulator. The generated assertion is then specific to the particular sequencing that was assumed, and we would be unable to verify a processor that attempted to perform the memory operations in a different order. Hints do not compromise the validity of verification.

Definition of "implementation"

The relation established by our verification is one between IO sequences of the specification and IO signals of the circuit. This is a global property. However, the illustration above has discussed only individual assertions in isolation—local properties. It is not particularly interesting to guarantee that a processor will execute one instruction correctly—we must show that it will execute entire programs correctly.

The theory underlying our verifier explains how establishing the local property is sufficient to establish the global property. Before explaining it, we should be more precise about exactly what is established. We give an abstract definition and an example.

Formally, circuits (realizations) and their formal specifications are both computational agents, which are nondeterministic, Moore-type, finite-state machines without defined initial states. (A nondeterministic machine is one whose response to a stimulus is not entirely determined. For example, we don't know what values most of a processor's registers will have after we reset the processor, and we may not know what output a circuit will produce when it is first powered up.) However, to define what we really mean when we say that the circuit implements its specification, we need very few details about agents. All we need to know is that each agent takes an input sequence and produces an output sequence (which may not be uniquely determined). Since the specification is more abstract than the realization, we also have a "mapping" which "fills in the details," i.e., for each specification sequence, it produces a corresponding circuit sequence. This mapping may also be nondeterministic, for there may be more than one circuit sequence that corresponds to a single specification sequence.

Given this notion of computational agents, we say that a circuit implements a specification if the following holds for every specification input sequence: for every corresponding circuit input sequence, every possible resulting circuit output sequence is within the image, under the mapping, of some specification

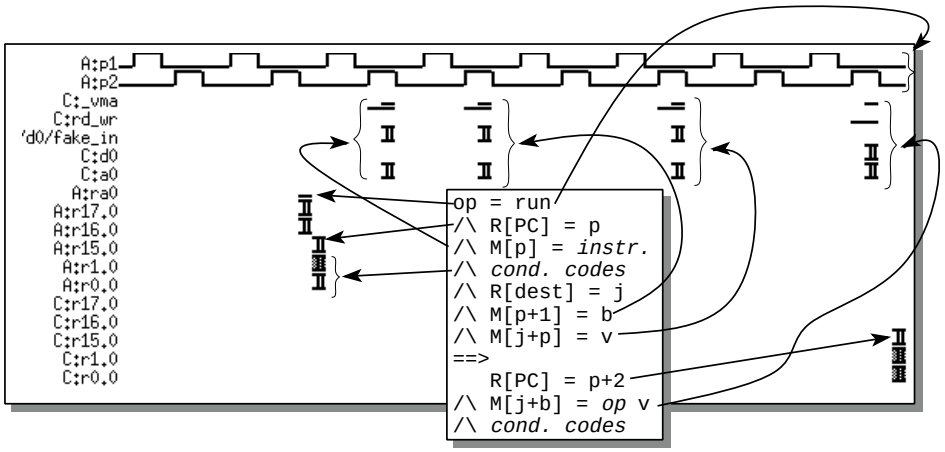

Figure 2: Example of an assertion mapped onto the microprocessor. Signals beginning with A are applied, and those beginning with C are checked.

output sequence which the specification could have produced from the original input sequence.

Details of "implementation"

The property we have just discussed is actually not implementation. Instead, it is what we call *obedience*. Implementation comprises obedience plus two other technical conditions, *conformity* and *distinction*.[7]

Without these additional conditions, mathematically trivial solutions to the obedience condition are possible. We impose the extra conditions to avoid these trivialities. Conformity requires that for every specification input sequence there be a corresponding circuit input sequence. Distinction requires that any two different specification output sequences cannot have the same corresponding circuit output sequence. These are properties of the mapping, rather than of the circuit, so they are easy to check.

Theory

The theory underlying our methodology, which allows us to conclude the implementation relation between IO sequences of a circuit and its specification by performing symbolic simulation tests on only individual transitions, is actually rather simple but mathematically abstract in its essential form. Details appear elsewhere [1].

[7] A convenient mnemonic is a fictitious military boarding school with motto "Conformity, obedience, distinction."

Conclusion

Experimental results

As we developed the methodology sketched here, we verified a number of different operations and instructions of the Hector microprocessor. We verified initialization and response to the non-maskable interrupt. We verified instructions including all addressing modes of the CLR instruction, some branch instructions, and the register addressing mode for the unary operations (NOT, INC, DEC, SHL, ROL, SHR, ROR, LDF, STF and SWAP) and binary operations (ADD, ADDC, SUB, SUBC, AND, OR, XOR and test-and-branch).

Our goal was to demonstrate the feasibility of symbolic simulation for formal verification, so we have not concentrated on performance. Fig. 3 shows the performance of the verifier for several instructions and operations. The table is useful for an indication of the magnitude of the numbers involved, but not for a detailed analysis of a factors contributing to the verifier's performance. As shown, checking an assertion is not a fast process.

Checking each assertion involves a significant amount of work. Consider the "clear" instruction with indexed addressing. Referring back to the timing diagrams of Fig. 2 is instructive: in order to verify this instruction, 7 cycles of system operation must be simulated.

Debugging

Pinpointing suspected errors during verification seems straightforward in retrospect, when their causes can be simply stated. However, locating these errors was most tedious. First, either the specification or the circuit may be in error. Second, without schematics, it was difficult to even know what circuitry surrounded the node exhibiting the error. Third, there is error latency, the activity between an error's occurrence and its

Instr.	Addr Mode	Time (s)	BDD size final	BDD size max
clr	reg.	518		240000
clr	ind.	341		31000
clr	inc.	380		
clr	indexed	853		159000
clr	reg.	559		241000
clr	indexed	819		156000
clr	indexed	611	6930	
add	reg.,reg.	1711		
xor	reg.,reg.	647	22500	86000
sub	reg.,reg.	1090		122000
subc	reg.,reg.	1068		62000
add	reg.,reg.	644		103000
or	reg.,reg.	741		53000
xor	reg.,reg.	893		67000
clr	ind.	534	23000	45000
initialization		303	496	2376
nmi		790	4783	15445
initialization		369	256	2051

Figure 3: Performance of verifier on several assertions. Time was measured in user CPU seconds on a DECstation 5000/200 with 32 MB memory (25 MHz R3000 CPU, 19.9 SPECmark) under the Mach 2.6 operating system. The operations shown more than once were re-verified at different stages of tuning the verifier's performance.

manifestation.[8] Fourth, understanding the state of a symbolic simulator is difficult.

A symbolic simulator represents not a single state for the system being modeled, but many states, one for each valuation of the symbolic variables. Understanding even a simple Boolean function of three variables takes some thought, which is more difficult if the function is expressed in some automatically generated form (e.g., as an ordered sum of products, or as a BDD) rather than an expression designed for exposition. Understanding a large set of even more complicated functions, and the structure of a circuit, and the relation between the two—at the same time—is all but impossible. Thus, when errors are detected by symbolic simulation, a different strategy is required to analyze them. Selecting a valuation for the symbolic variables which manifests the error is the first step. Although in principle any such valuation will do, simpler ones—such as those in which most of the variables take the value 0—are often easier to understand. Examining the symbolic simulation state under this valuation becomes tractable, for the state values become 0, 1, and X rather than complex functions.

In our experience, when verifying toy circuits, the most difficult part of our methodology is writing the mapping. For real circuits, finding errors in the abstract specification itself also becomes a significant task. For example, Hector's "SUBC" instruction was difficult to verify because its use of the carry flag is quite subtle to specify correctly, since it is actually necessary to allow *18*-bit arithmetic in one case.

Ultimately we found no errors in the instructions we examined. Hector's designers later confirmed that the only known bug in Hector affects an instruction we had not tried to verify. We did find that precisely stating instruction semantics was a challenge. For example, Hector has a "push" instruction and the PC is addressable, so there is a "push PC" instruction which stores into instruction memory then decrements the PC (causing a loop). This is a useless instruction, but it is necessary to either specify its behavior, or specify that its behavior does not matter (e.g., with an assertion whose consequent is the constant formula **true**).

Observations

We have verified that a microprocessor circuit implements its intended instruction set using symbolic simulation. There are some key differences between Hector and modern, commercial designs, including size, pipelining, exceptions, and caches.

One of the principal dangers of formal verification is what we have called antecedent failure.[9] Formally, implications have an *extensional* meaning: they are also true if the antecedent condition is *not* true. Antecedent failure means that we can speak nonsense and not realize it.[10] One strength of our approach is that we can check for antecedent failure, and structure our specifications so that antecedents never fail. While structuring a specification this way lengthens it, it improves our confidence that the specification actually makes sense.

The danger of antecedent failure should not be underestimated. If a logical proposition a is false, every implicative statement having a as antecedent, "if a then c," is always true. When a is easy to understand, this poses no problem. If wishes were horses, then beggars would ride. However, when statement a concerns the very circuit being verified, it may not be obvious that a is in fact false—the falsity of a might well be due to a circuit bug. "If the input is high, then the output is low" is not only true of an inverter, but also true of any circuit whose *input* is accidentally shorted to ground. This is an example of antecedent failure.

One way to avoid antecedent failure is to check that the antecedent is, in fact, always true. For the inverter, we would discover the problem when we check that we can indeed set the input high. Incorporating such a check restricts the class of assertions that can be verified to just those whose antecedents are true. When we find an antecedent failure, we rewrite assertions so that their antecedents remain true. (This is always possible.) For example, the restrictions in the second line of the interrupt antecedent, earlier in this paper, were added for this reason.

The methodology has some distinguishing features worth noting.

[8] This is a good general rule of debugging which bears some repeating.

[9] This phenomenon is called one of the "Lewis principles" by logicians.

[10] Antecedent failure can be a problem in any approach to formal verification. Techniques that are based entirely on pure logic, such as HOL, are especially prone to it.

- It is most suited for verifying functional properties of data intensive systems, i.e., those whose operation can be thought of as updating data values stored as components of a large stored state, in response to a relatively small number of operations.
- Our specifications are given at a high level. This requires an unconventional format. We have defined a new language, based on assertions rather than imperative commands, that is higher level than most hardware description languages. It is possible to derive assertions from more-conventional HDL descriptions [11], but such assertions would be too highly constrained for verification.

Future work

The goal of FV is to eliminate harmful design errors without compromising design goals. The work outlined here is a step toward this ideal. Further work could solidify the theory, implement it in robust tools, and demonstrate their usefulness on real circuits. Definition of mappings is an ideal task for a graphical specification language. The full generality of trajectory evaluation should be explored at other levels. Supporting existing HDL's in some way would facilitate acceptance of FV techniques in industry.

When we choose circuits to verify, we must take care to ensure that they differ from previous circuits, to advance the state of the art in verification. In addressing the differences between Hector and more modern processors, one place to start would be with simple pipelines that allow interrupts or exceptions. The ultimate goal of verification is to attain currency with state-of-the-art design techniques, so it is imperative to deal with superpipelined and superscalar designs. Although we have given multiple-issue systems some thought while developing our theory, we have not given it sufficient serious study. The approach of starting with simple examples is useful, but so far we are unaware of any simple superscalar designs.

References

[1] D. L. Beatty. *A Methodology for Formal Hardware Verification, with Application to Microprocessors*. PhD thesis, published as technical report CMU–CS–93–190. Comp. Sci. Dept., CMU, Aug. 1993.

[2] D. L. Beatty, R. E. Bryant, and C.-J. H. Seger. Synchronous circuit verification by symbolic simulation: an illustration. *Advanced Research in VLSI: Proc. 6th MIT Conf.*, pages 98–112. MIT Press, Mar. 1990.

[3] R. E. Bryant, D. L. Beatty, and C.-J. H. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. *28th DAC*, 1991.

[4] A. Cohn. Correctness properties of the Viper block model: the second level. Technical report 134. University of Cambridge Comp. Lab., May 1988.

[5] K. W. Fernald, T. A. Cook, T. K. Miller III, and J. J. Paulos. A microprocessor-based implantable telemetry system. *IEEE Computer*, **24**(3):23–30, Mar. 1991.

[6] W. A. Hunt, Jr. *FM8501: a Verified Microprocessor*. PhD thesis. Univ. of Texas, Austin, Dec. 1985.

[7] J. J. Joyce. *Multi Level Verification of Microprocessor-Based Systems*. PhD thesis, published as technical report 195. Univ. of Cambridge, Comp. Lab., May 1990.

[8] J.-C. Madre and J.-P. Billon. Proving circuit correctness using formal comparison between expected and extracted behavior. *25th DAC*, pages 205–10, 1988.

[9] J.-C. Madre, O. Coudert, M. Currat, A. Debreil, and C. Berthet. The formal verification chain at BULL. *EURO ASIC* (Paris, 28 May–1 June 1990), pages 474–9. IEEE, 1990.

[10] T. K. Miller III, B. L. Bhuva, R. L. Barnes, J.-C. Duh, H.-B. Lin, and D. E. Van den Bout. The HECTOR microprocessor. *ICCD*, pages 406–11, 1986.

[11] J. D. Oakley. *Symbolic Execution of Formal Machine Descriptions*. PhD thesis. CMU, Apr. 1979.

[12] C.-J. H. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. Technical report 93–8. Comp. Sci. Dept., Univ. of British Columbia, 1993.

[13] M. Srivas and M. Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, **7**(5):52–64, Sep. 1990.