

Verification-Aware Microprocessor Design

Anita Lungu
Dept. of Computer Science
Duke University
anita@cs.duke.edu

Daniel J. Sorin
Dept. of Electrical and Computer Engineering
Duke University
sorin@ee.duke.edu

Abstract

The process of verifying a new microprocessor is a major problem for the computer industry. Currently, architects design processors to be fast, power-efficient, and reliable. However, architects do not quantify the impact of these design decisions on the effort required to verify them, potentially increasing the time to market. We propose designing processors with formal verifiability as a first-class design constraint. Using Cadence SMV, a composite formal verification tool that combines model checking and theorem proving, we explore several aspects of processor design, including caches, TLBs, pipeline depth, ALUs, and bypass logic. We show that subtle differences in design decisions can lead to large differences in required verification effort.

1. Introduction

The computer industry has acknowledged that design verification consumes the majority (60-70%) of the resources—engineers, time, and money—devoted to the creation of a new microprocessor [9, 15]. Despite this effort, the most recent processors from Intel and AMD have been shipped with dozens of design bugs that have since been documented [1, 16, 17]. Many of these bugs can crash the computer or corrupt important data (e.g., the Pentium’s floating point division bug [7]).

Many activities are considered part of the verification effort, including *design testing* and *formal verification*. Design testing is the process of running the simulator of the microprocessor, with either random or directed inputs, for as many cycles as possible in order to try to uncover bugs. Design testing is not, strictly speaking, verification, but the industry convention is to consider it part of the “verification” process. Formal verification is the process of proving that the design is correct, and it can be performed with tools such as model checkers and theorem provers.

There are two reasons why design testing and formal verification cannot currently catch all design bugs. First,

design testing will always be incomplete [6], because we cannot exhaustively test a system with as many states as a microprocessor. Second, both primary techniques for formal verification are limited. One technique, *model checking* [11], explores the entire state space of a system and formally validates that the user-specified definition of correctness is true in all situations. Model checkers—such as Mocha [2] and Mur ϕ [14]—can verify systems with a fairly small number of state elements (e.g., 100-200). However, for microprocessors with an enormous number of states, model checkers are thwarted by the *state explosion problem*. The other type of formal verification, *theorem proving*, can, in theory, handle massively complex systems. However, theorem proving demands an unbounded amount of user effort to define and guide the proofs. Due to its lack of automation, pure theorem proving is not tractable for large, sophisticated designs.

In this work, we use a promising composite formal verification tool, Cadence SMV [19]. Cadence SMV (which we will refer to as SMV, for brevity) uses a combination of theorem proving and model checking. SMV requires the user to specify correctness properties that each involve no more state than can be handled by the model checker, and then SMV’s theorem prover invokes its model checker to verify these properties. If the model checker cannot verify a given property, because of state space explosion, then the user must add “helper properties” that are easier to verify and that aid in verifying the main property. Despite the power of SMV, using it to prove the correctness of complex systems, such as processors, is extremely challenging. The difficulty is mostly due to the effort involved in the verification procedure—breaking up the specification of correctness into a large number of properties and guiding the theorem prover to help it demonstrate the validity of all of the proofs—as well as the state explosion problem itself.

One key to reducing verification effort is to exploit *symmetry* among states (i.e., uncover states that are equivalent from a verification perspective). For example, in a multiprocessor with 8 processors, the state in which

Processor 1 has block B with Exclusive coherence permissions and the other 7 have no access to block B is symmetric to the other 7 states in which a single processor has B Exclusively and the other processors have no access to B. However, current microprocessors often lack enough symmetry to be verifiable. One contributing factor is that microarchitects have not usually treated verifiability as a first-class design constraint. In general, microarchitects design processors to be fast, power-efficient, reliable, etc., and *then* they pass their designs over to the verification team. Thus, we may have scenarios in which microarchitects tweak a design to eke out a small amount of additional performance, without considering the possibility that this modified design may require significantly more verification effort than the original.

Our long-term research goal is to design microarchitectures with verifiability as a first-class design constraint. This end goal is not yet attainable, largely because of the extremely steep learning curve required to perform even trivial verifications. However, in this paper, we take a first step on this path. We examine microprocessor structures, such as caches and ALUs, that are already verifiable using current formal tools. Using SMV, we quantitatively compare the effort necessary to verify different implementations of these structures. We believe that this paper is the first work to explicitly address the issue of *verification-aware design* for microprocessors. Our contributions are:

- A set of general guidelines for designing verifiable microarchitectural components.
- Quantifying the verifiability impact of certain design decisions for the caches, TLBs, pipeline depth, ALUs, and operand bypass network.

In the rest of this paper, we first discuss in Section 2. how we perform formal verification with SMV. In Section 3., we provide general guidelines for how to design systems that are more easily verifiable. In Section 4., we describe our experimental methodology. Then, in case studies presented in Sections 5. and 6., we use these guidelines to provide intuition for why certain design options for specific microprocessor components are more easily verifiable than others. In Section 7., we discuss related research, and we conclude in Section 8..

2. Background on Formal Verification

Cadence SMV is a composite formal verification tool that exploits the best features of model checking and theorem proving. Like model checking, SMV allows us to specify systems as state machines (rather than mathematical entities), and it also provides helpful counter-examples when a verification fails. Like theorem proving, SMV can verify more complicated sys-

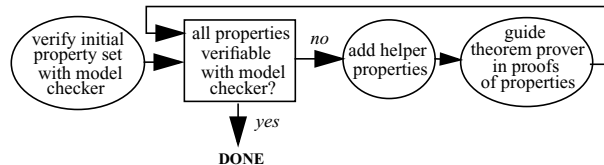


Figure 1. Verification Procedure

tems and parameterized systems (e.g., a processor with any size reorder buffer).

In SMV, the theorem prover invokes the model checker to verify properties about the system, and it is incumbent upon the user to specify a set of properties that each involve no more state than can be handled by the model checker. If verifying any property involves more state than the model checker can keep in memory, then the user must add “helper properties” that can be proven with the model checker and that help to prove the otherwise intractable property. A key technique is trading model checking effort for user effort. A flow-chart of this process is illustrated in Figure 1.

In this section, we discuss the model checker (Section 2.1.), theorem prover (Section 2.2.), and how we quantify verification effort (Section 2.3.). We conclude this section with a simple example of a verification and how much effort it requires (Section 2.4.).

2.1. Model Checking

A model checker [11] takes two inputs, a model (i.e., description) of the system to be verified and a specification of correctness and then outputs whether the model meets the correctness specification. A correctness specification consists of multiple properties that must all hold for the design to be correct. For a microprocessor, it is common for the specification of correctness to be an abstract or simplified version of the microprocessor, such as its instruction set architecture (ISA). We refer to this specification of correctness as the *abstract processor*, and we refer to the processor model being verified as the *implementation processor*. At a high level, a model checker walks through the implementation processor’s entire state space and checks that it is always adhering to the behavior of the abstract processor.

To prove that a specific property holds, the model checker constructs the property’s *logic cone*. The cone contains the *state* and *combinatorial* variables that can potentially affect the correctness of the property, and it usually does not include the entire state space of the design. The number of state variables in the cone is related to the size of the reachable state space that must be explored. Thus, the primary consequence of an increase in state variables is an increase in memory

usage, which is the limiting factor for most model checking verification procedures. Because current microprocessors have a large number of state variables, there is a state space explosion problem for model checkers. The number of combinatorial variables in the cone does not affect memory usage, but it does affect the time required for verification, because the model checker must prove the correctness of a property for all possible values of the combinatorial variables. Because memory usage is far more likely to limit verifiability than run time, the number of combinatorial variables has less impact on verification effort than the number of state variables.

Different model checkers represent state spaces in different ways. Explicit model checkers, such as Murϕ [14], allocate memory for each reachable state encountered, leading to a steep increase in memory usage as the state space grows. In contrast, SMV is a symbolic model checker that allocates memory for collections of states that have similar characteristics. Transition relations express how a system moves from state to state, and SMV uses the transition relations to calculate the reachable state space. SMV uses ordered binary decision diagrams (OBDDs) to compactly represent the binary functions that encode transition relations. When using OBDDs, memory usage is directly related to the number of OBDD nodes and not the number of state variables. Thus OBDDs can lead to a significant memory savings, but they do not solve the state space explosion problem.

The state space explosion problem can be mitigated, to some extent, by exploiting symmetry [18, 23]. Symmetry can reduce the *effective number of states* that the model checker must traverse, because the model checker needs to evaluate only one state among a group of symmetric states. Symmetry can reduce the logic cone that affects a property that is being verified, and it exists in two basic forms:

- *Structural symmetry* occurs when multiple components are equivalent and can be permuted without changing functionality. For example, there can be structural symmetry across cores in a multicore processor or across ALUs within a core. There can be structural symmetry across the entries of a buffer. For a system with N structurally symmetric components, the number of effective states that must be traversed by the model checker can be reduced by a factor of N factorial, because all states resulting from permuting the N components are equivalent.
- *Data value symmetry* occurs when a variable's exact value does not matter. That is, we can choose any value from its range to represent this variable. An example of data value symmetry is a

word of memory traversing the memory hierarchy. Data value symmetry is powerful because it can take a 2^B state space for a B -bit datum and reduce it to a single value.

2.2. Theorem Proving

SMV's theorem prover, with significant help from its user, composes the model checking verifications of the individual components and properties. In an unrealistically simple scenario, the user specifies a single correctness property that must be proved for the entire system. For example, we could specify that a complicated implementation processor obeys a property which states that the result produced by the implementation processor is the same as the result produced by a simplistic, in-order, un-pipelined abstract processor. The theorem prover would then invoke the model checker to verify this property. In such a simplistic scenario, the theorem prover is actually unnecessary. The reason this scenario is implausible is that an all-encompassing property is likely to involve many if not all of the state variables in all of the components, and it is thus well beyond the ability of the model checker. Thus, the user must manually devise simpler and narrower "helper properties" that, once proven, can be used by the theorem prover to more easily verify more complicated, comprehensive properties. An example helper property might specify that the implementation processor fetches the same instructions as the abstract processor.

There are two major challenges. First, the user must develop a set of properties that is complete (i.e., a system that obeys all of these properties is correct) and individually tractable for the model checker. Second, the user must help in constructing the proof that the theorem prover will follow. Although it might appear that the theorem prover should be able to figure out the best way to prove each property, this process still requires user involvement. There are too many options for organizing the proof, and the theorem prover generally does not know enough about the system's semantics to choose wisely. This process requires the user to apply *temporal case splitting* [23], which we discuss next.

Temporal case splitting [23] splits the property verification into multiple *cases* in which a datum takes different paths through the system. By splitting the property verification into cases, we can often greatly reduce the state space for each case at the cost of having more cases to prove. For example, verifying three cases with 5, 10, and 20 state variables, respectively, is far preferable to verifying a single case with 35 state variables. The benefit is because the memory allocated can be exponential in the number of state variables, and 2^{35}

is so much larger than $2^5 + 2^{10} + 2^{20}$. As an example of case splitting, consider a processor that loads a block from memory, writes it into a register, and then later stores it back to memory. If addresses and registers are symmetric, then we have reduced the verification effort to just two distinct cases that must be proved: if the load and store are to the same address or to different addresses. Temporal case splitting is particularly effective when used in conjunction with symmetric variables.

2.3. Quantifying Verification Effort

Quantifying verification effort is not straightforward. If we were only using model checking, then we could consider the verification effort to be directly related to the number of state variables in the system. As the number of states increases, the memory consumption increases and eventually the model checker runs out of memory and hangs. One shortcoming of this metric is that the memory usage depends on the particular model checker and even the particular ordering of state variables that it uses for a particular OBDD. It is possible that Design A has more state variables than Design B, but the model checker just happens to choose a more fortuitous ordering of state variables for an OBDD in the model of Design A and thus uses less memory.

The verification effort required by SMV’s theorem prover layer is also difficult to quantify precisely, largely because it is a measure of human effort. Nevertheless, we found that it is a function of both the number of properties that must be verified and the number of cases per property. The user is responsible for defining all of the properties and, for each property, identifying all of the possible cases that must be proved. In particular, we have found that case splitting is a difficult process. For example, we have found that it is far more work to guide the theorem prover to verify a system with 2 properties each of which has 10 cases than it is to verify a system with 10 properties each of which has 2 cases. We use the number of properties and the number of cases per property as our metrics for theorem prover effort, even though they are imperfect.

Because of the difficulty in fairly comparing the automated effort of a model checker against the human effort to guide a theorem prover, we will use all of the previously mentioned metrics rather than attempt to coalesce them into one number.

Although we defined and applied the above described metrics while working with SMV, their use is not restricted to this particular tool. The numbers of states and OBDD nodes are indicative of (positively correlate to) memory consumption which is the limiting factor for model checking. The numbers of properties

and cases to be verified are also measures of general complexity for theorem provers.

2.4. A Simple Example

In this section, we present a simple example that shows the impact of considering symmetry during verification. This example is for purposes of illustration and not a novel contribution. We have developed a generic TLB-like cache whose symmetry can be exploited, i.e., all of the fully-associative TLB’s entries are equivalent and it can thus be indexed by a symmetric variable (called a *scalarset* in SMV). The abstract processor obtains the value of the physical address from the page table directly and then uses it to access the memory. The implementation processor indexes into the TLB to get the physical address (on a miss, it replaces a TLB entry with the address from the page table) and then uses this physical address to access the memory. In four experiments, we verify that the implementation system satisfies a single correctness property: the same data value is returned from memory for both the abstract and implementation model. In the first experiment, we exploit symmetry by indexing into the TLB with a scalarset. In the other three experiments we do not leverage symmetry, and we index the TLB with an (asymmetric) integer.

In Figure 2, we show the verification effort as a function of the size of the address and of the data word. We observe several phenomena. First and foremost, when using a scalarset index, we can verify the symmetric system with less effort than a trivially small system that does not exploit symmetry. Moreover, the verification of the symmetric system is parameterized (i.e., independent of the size of the address or data word). Second, the verification effort for the system that does not exploit symmetry explodes quickly—even with only 9-bit addresses and 3-bit data words, SMV’s model checker is unable to handle the state space explosion. If we want to verify this system with SMV, we need to off-load the verification effort from the model checker to the theorem prover. By adding one helper property, we can reduce the number of state variables from 321 to 297, and we end up with 41,982 OBDD nodes, which is tractable for the model checker. Third, we observe that the number of OBDD nodes is a highly non-linear function of the more fundamental number of state variables, as explained in Section 2.1., but it is correlated with the number of state variables. That is, if system A has more state variables than system B, then A will generally have more OBDD nodes than B.

Although this particular experiment does not prove that exploiting symmetry will always lead to such an

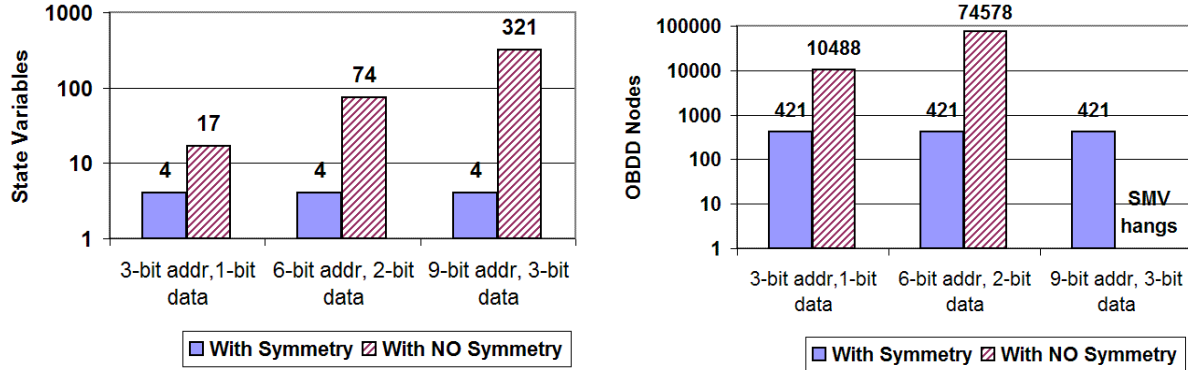


Figure 2. Verification Effort for Simple TLB. Note that y-axis is log scale.

enormous benefit, we have seen comparable benefits from symmetry in our experiments.

3. Guidelines for Easing Verification

In this section, we provide three concrete design guidelines for reducing verification effort. The first guideline is already fairly well-known, but it is worth repeating. The other two guidelines may be well-known to verifiers, but probably not to most architects; we, as computer architects, “derived” them during our experimentation. This list of guidelines is by no means exhaustive, but it covers several of the issues that arose in our efforts to design verifiable components.

In this paper, we do not explore the impact of these design decisions on performance or power or any other metric. The impact on these metrics and the importance of these metrics varies based on the processor model and application. Our goal here is to explore the impact of design decisions on verification effort, so that architects can weigh this impact against the other metrics they must consider when designing a processor.

Guideline #1. Try to construct the system such that it can be decomposed into small components that can be verified independently. This already well-known guideline requires that the interfaces between these small components are clean and well-defined. The benefit of adhering to this guideline is that it enables us to verify large systems with many components, by providing a better isolation of the verification effort required for each component. If we do not adhere to this guideline, then proving any property about the system may involve exploring the state space of the entire system, not just a small number of the system’s components directly related to the property, and we rapidly encounter the state explosion problem.

Guideline #2. Try to avoid performing operations on only part of an otherwise symmetric variable. No model checker of which we are aware, including SMV, can

completely preserve symmetry when a scalarset variable is broken into sub-components. An example of a component that violates this guideline is a set-associative cache that uses only part of an address as a tag. We examine the importance of this guideline in Sections 5.1. and 5.2..

Guideline #3. Try to limit the depth of pipelines and the amount of state propagated from one state to the next. Deeper pipelines with more state carried from one stage to the next lead to significant increases in the number of state variables and overall verification effort. We show the importance of this guideline in Section 5.3..

In the course of this research, we have also debunked several “guidelines” that we hypothesized would be useful but which we determined are not important to verification effort. These negative results are just as important to architects who seek to keep verifiability in mind during the design process.

Guideline #4 (Debunked). Try not to specialize a subset of a group of otherwise identical components. For example, if you have 4 ALUs, do not make one of them different from the others, or else they will not be symmetric. In Section 6.1., we demonstrate that adhering to this guideline does not reduce system verification effort.

Guideline #5 (Debunked). Try to avoid complicated operand bypass networks. In Section 6.2., we show that this guideline does not help verifiability.

4. Methodology

Processor Model. The abstract processor (specification of correctness) for *all* of the experiments discussed in this paper is the behavior of a trivially correct, single-cycle, in-order processor, with no cache or TLB. Its ISA includes ALU, Branch, Load, and Store instructions. For loads and stores, it first accesses the page table and then accesses main memory.

Table 1. Properties to Verify. Property names indicate what must be equal for implementation processor and abstract processor.

Fetched Instruction Properties
Program Counter, Next Program Counter, Instruction
Decoded Instruction Properties
Opcode, Source Registers, Destination Register, ALU Operands, Branch Operands
Memory Access Properties
Physical Address ^a , Load Value
Instruction Result Properties
ALU Output, Branch Target, Branch Condition

a. Implementation processor gets same physical address from TLB as abstract processor gets from page table

The implementation processor we are verifying has a 5-stage, in-order, scalar pipeline. It supports branch prediction, with a simple predict not-taken scheme. It has a TLB and a physically-addressed, single-level, write-allocate cache. This implementation core is fairly simple, which likely diminishes the impact of cache and TLB verifiability on overall system verifiability for two reasons. First, a more complicated implementation core would have more states and thus the complete system (i.e., with the TLB and cache) would have more states. Second, the implementation processor in this paper has a clean separation between the core and the memory subsystem, which enables each to be verified in isolation. For modern microprocessors without this separability, verifying each property might require exploring the combined state space of the core and memory subsystem. Both of these issues lead to more states that must be explored at once, and any incremental increases in verification effort for a cache or TLB would lead to larger absolute increases (potentially exponential) in verification effort.

Properties to Verify. In the process of verifying that the implementation processor is correct, we had to add helper properties to the definition of correctness (which is that the implementation processor is equivalent to the abstract processor). In Table 1, we list these helper properties.

5. Case Studies for Guidelines #2 and #3

In this section, we present case studies that illustrate the importance of Guideline #2 (Sections 5.1. and 5.2.) and Guideline #3 (Section 5.3.).

5.1. Cache and TLB Associativity

One of the basic design issues for a cache or TLB is choosing how set-associative it will be. The well-known engineering tradeoff is that increasing the associativity reduces the miss rate at the cost of being slower and more power-hungry. But how does the choice of associativity affect verifiability?

Intuitively, a fully-associative structure is the most symmetric, because all of its entries are equivalent. This symmetry reveals itself when we describe an address in a fully-associative cache. It consists of a tag and a block offset, both of which can be represented by scalarsets (SMV’s symmetric type of state variable). Note that we cannot treat the address as a single scalarset, because sometimes we operate on only part of an address (e.g., when comparing tags). A k -way set-associative structure has less symmetry than a fully-associative structure, because each address must be treated as a concatenation of three scalarsets instead of two. There are now scalarsets to represent the tag, index, and offset. Thus we have more state variables (and a larger state space) that the model checker must consider every time that it encounters an address. A direct-mapped (1-way) structure has the same symmetry as a k -way structure, because it still requires three scalarsets to represent an address.

To quantify the verifiability impact of associativity, we used SMV to verify processors with caches and TLBs with various associativities and a random replacement policy. For clarity, we only present data for three scenarios: fully-associative cache and TLB, k -way set-associative cache and TLB, and direct-mapped cache and TLB. Note that the size of each structure does not impact the verification effort for any property, because the number of sets is an unbounded scalarset.

In Table 2, we show the verification effort for all properties as a function of the cache associativity. We observe three interesting phenomena, and we use bold text to highlight the results for three of the most affected properties in the table. First, we conclude that our hypothesis was correct—in general, a fully-associative structure requires less verification effort than a set-associative structure. This result may not be relevant to most caches, because we cannot implement large fully-associative caches, but it may be important in other points of the design that require smaller cache-like buffers. Second, we see that the results for k -way set-associative are almost always equal to the results for direct-mapped. Also, for two properties, the number of cases is actually slightly greater for the k -way structures than for direct-mapped and fully-associative. For these properties (*Load Value* and *Physical Address*), we had to consider an additional case depending on which way of the set-

Table 2. Verifiability of properties as function of cache and TLB associativity. Shaded rows indicate properties whose verification efforts are unaffected by cache and TLB associativity.

Property	State Variables			OBDD Nodes			Cases		
	full/ full	k-way/ k-way	DM/ DM	full/ full	k-way/ k-way	DM/ DM	full/ full	k-way/ k-way	DM/ DM
Program Counter	16	16	16	7171	7340	7340	2	2	2
Next Program Counter	0	0	0	719	1757	1351	4	5	5
Instruction	8	9	9	898	1080	1158	3	4	4
Opcode	7	8	8	1050	2465	2465	2	3	3
Source Registers	7	8	8	1295	2500	2400	3	4	4
Destination Register	5	6	6	566	1207	1134	3	4	4
ALU Operands	29	29	29	26917	26917	26917	2	2	2
Branch Operands	26	26	26	20334	20334	20334	2	2	2
Load Value	38	41	41	75763	102658	136894	4	5	4
Physical Address	13	16	16	3477	11476	11224	3	4	3
ALU Output	28	36	36	10771	10731	10731	5	7	7
Branch Target	19	25	25	19874	36071	34984	5	7	7
Branch Condition	17	23	23	10144	10416	10416	6	9	9

associative structure is used. Fourth, we observe that the number of OBDDs is generally related to the number of state variables, but it is not a perfect correlation, for reasons discussed in Section 2.. In fact, there are some properties which have zero state variables, but still have a non-zero number of OBDD nodes. These situations occur when verifying the property involves combinational variables but no state variables.

We also evaluated the impact on verifiability of the cache and TLB replacement policy, and we compared random, Not Most Recently Used (nMRU), and Least Recently Used (LRU). We hypothesized that LRU would be hardest to verify due to dependencies imposed between otherwise symmetric ways in the buffers. However, the results (not shown due to space limitations) showed no significant verifiability differences. Random and nMRU required basically the same verification effort and LRU only marginally more.

5.2. L1 and L2 Cache Organizations

Microprocessors have different organizations for their L1 and L2 caches. The L2 is almost always significantly larger than the L1 caches, and it may have a different associativity or block size. These differences between the cache levels lead to differences in how addresses are interpreted and can thus decrease symmetry. In Figure 3, we illustrate how differences between

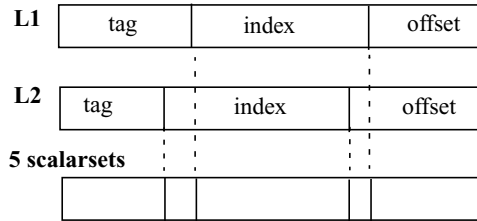


Figure 3. The problem with having different L1 and L2 organizations

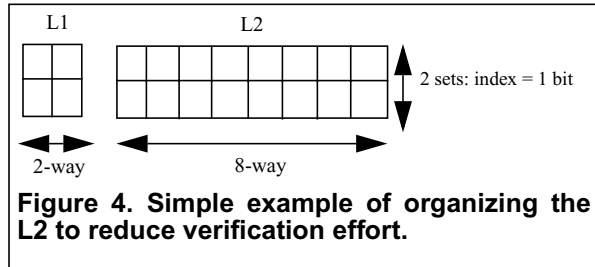
the L1 and L2 organizations can force SMV to use five scalarsets to represent each address.

One easy way to improve verifiability is to maintain the same block size between cache levels, instead of having larger block sizes in the L2 than in the L1. That would reduce the number of scalarsets per address from five to four. The tradeoff is that we lose the ability to exploit more spatial locality in the L2.

A perhaps less obvious way to improve verifiability is to set the L2's associativity to be K times more than the L1's associativity, where K is the ratio of the size of the L2 to the L1. By organizing the L2 in this fashion, we can then interpret addresses the same way for the L1 and the L2. That is, the same bits are used for the tag, index, and offset, as shown in Figure 4. In the figure, the L1 and L2 have the same block size, so they have the same number of block offset bits. The L1 is 2-way associative and has 4 entries total, so it has 2 sets. The L2 is 8-way associative and has 16 entries total, so it also has

Table 3. Verifiability of properties as function of pipeline depth. The shaded row indicates a property whose verification effort is unaffected by pipeline depth.

Property	State Variables			OBDD Nodes			Cases		
	depth 4	depth 5	depth 6	depth 4	depth 5	depth 6	depth 4	depth 5	depth 6
Program Counter	10	16	16	2488	7340	6878	2	2	2
Next Program Counter	0	0	0	452	1757	10006	5	5	5
Instruction	0	9	9	123	1080	10019	4	4	4
Opcode	2	8	8	144	2465	6737	3	3	3
Source Registers	0	8	8	123	2500	10085	4	4	4
Destination Register	0	6	6	123	1207	1532	4	4	4
ALU Operands	22	29	37	13224	26917	265099	2	2	2
Branch Operands	18	26	34	27493	20334	76720	2	2	2
Load Value	30	41	52	35157	102658	496682	5	5	5
Physical Address	12	16	24	8356	11476	11138	4	4	4
ALU Output	9	36	36	7326	10731	11176	7	7	7
Branch Target	0	25	25	580	36071	26573	7	7	7
Branch Condition	7	23	23	10000	10416	11884	9	9	9



2 sets. We chose the L2 to be 4 times more associative than the L1 because it has 4 times as many entries, and thus they have the same number of index bits.

With this approach, we reduce the number of scalar-sets per address to three. The tradeoff is that we have restricted the organization of the L2, and it may not be feasible to organize it this way. If the L1 is small and highly associative (e.g., 8KB, 4-way) and we want an L2 that is much larger (e.g., 2MB), then the L2 would have to be prohibitively associative (1K-way!).

We do not show experimental results for this case study, because it devolves to the same issue as in Section 5.1.: how many scalarsets are required to represent an address.

5.3. Pipeline Depth

Guideline #3 suggests keeping pipeline depths short, to avoid propagating more state through more stages. In this experiment, we vary the pipeline depth of our

implementation processor from four to six stages. The five-stage pipe is the classic design of F (fetch), D (decode), X (execute), M (memory), W (writeback). The 4-stage pipe is FD, X, M, W. The 6-stage pipe is the same as the 5-stage but with an extra M stage. The memory system is fixed in all experiments; the cache and TLB are set-associative with random replacement.

In Table 3, we show the results of verifying all of the properties, and we use bold text to highlight a subset of the properties that reveal the most significant trends. Almost all of the properties require more verification effort, in terms of the number of state variables (and OBDD nodes); some properties require more effort going from a depth of four to five but no change for a depth of six, while others increase steadily. Some of the properties show striking increases in the number of state variables, such as *ALU Output* and *Branch Target*. Interestingly, none of the properties requires more cases as a function of pipeline depth. This result may seem surprising at first, but it is intuitive when one considers that increasing the pipeline depth simply adds state that must be propagated. No additional paths must be added.

Although the absolute number of state variables for each property is well within the model checker’s capability, this number would be far greater for a more complicated processor with more components and interactions between them. This simple implementation processor is likely a “best-case” scenario, in that the verifiability of a more realistic processor model would

Table 4. Verification of “ALU Output” Property for Processor with Asymmetric ALUs

	State Variables	OBDD nodes	Cases
Processor 1: 1 type of ALU instruction, 1 type of ALU	28	10771	5
Processor 2: 3 types of ALU instructions, 1 type of ALU	30	10462	5
Processor 3: 3 types of ALU instructions, 3 types of ALUs	30	10462	5

likely be even more dependent on pipeline depth. For a more complicated processor, we would expect to need to offload model checker effort to theorem prover effort, and we would have far more properties and cases. We conclude that having multiple cores with shorter pipelines (e.g., Sun’s Niagara [20] or Piranha [4]) is beneficial for verifiability, as opposed to super-pipelined processors (e.g., Pentium4 [8]).

6. Debunking Guidelines #4 and #5

We now present case studies that debunk two misleading “guidelines.”

6.1. ALU Specialization

Guideline #4 recommended against specializing a subset of a group of components. As a specific example, we consider the specialization of ALUs. The Pentium4, for example, has different flavors of ALUs that can perform different functions [8]. This design decision, however, sacrifices symmetry. With specialized ALUs, we can no longer just choose *any* ALU from the set, and thus we cannot use a scalarset to choose which ALU an instruction should use.

We performed an experiment to quantify the impact of specializing the ALUs by comparing three implementation processors. Processor1 has one type of ALU instruction (e.g., ADD, OR, etc.) in its ISA and only one type of ALU. Processor2 has 3 different ALU instructions and only one type of ALU. Processor3 has 3 different ALU instructions and a different type of ALU for each type of ALU instruction.

In Table 4, we show the verification effort for each processor for just the *ALU Output* property, which is the most affected. We observe that the hypothesis behind Guideline #4 was wrong—the only difference between the processor verification efforts is due to Processor2 and Processor3 having more instruction types. These extra instruction types caused us to need one more bit to store each opcode, and this artifact caused the small difference in verification effort. The different types of ALUs had no impact. The reason for this result is that having different types of ALUs does not add any *state* variables. They add to the number of *combinational*

variables in the system, but it is the number of state variables that is the limiting factor. In retrospect, we perhaps should have realized that the presence of different ALUs adds no state, but we were led astray by forgetting that not all asymmetries matter. The lesson for architects is that specialization that breaks symmetry has a small impact on verification effort as long as the asymmetry does not involve state (i.e., is strictly combinational).

6.2. Operand Bypass Network

Guideline #5 suggested that we should expect operand and bypass networks to make verification more difficult. Architects have long considered bypassing to be a significant source of design complexity, and it would also appear to aggravate verification by adding to the number of paths that data could take through the pipeline.

We compared the verification of two implementation processors. Both have 5-stage pipelines with *k*-way set-associative caches and TLBs with random replacement policies. The difference is that Processor1 bypasses operand values between stages and Processor2 does not. The results, which initially surprised us, showed no differences in the number of state variables or case splits for any property. There were some extra combinational variables and some differences in OBDD nodes, probably due to rearrangements of variable ordering.

The insight for understanding this surprising result is that the amount of state needed to bypass operands in Processor1 is exactly the same as the amount of state needed to determine when to stall instructions waiting for operands in Processor2.

7. Related Work

There has been related work in the areas of design for verifiability, dynamic verification, and complexity-effective design.

Design for verifiability. Milne [25] first coined the phrase “design for verifiability,” and his paper mostly addressed how to insert latches into a design to simplify verification. In particular, the goal was to reduce the amount of asynchrony in a design. Martin [21] qualitatively compared the verification complexity of cache coherence protocols and made suggestions to ease veri-

fication effort. Marty et al. [22] compared the verification complexity of coherence protocols using the number of non-comment lines in the model description. Curzon and Leslie [12] re-designed a switching fabric to be more easily verified with a theorem prover. BlueSpec [13] is a high-level, object-oriented hardware description language that can be compiled to both RTL and a term rewriting system for automated verification. BlueSpec eliminates bugs due to translating an RTL design into the language used for verification, and it is orthogonal and complementary to our work.

Dynamic verification. DIVA [3] proposed using a small, provably-correct checker core to dynamically verify a large, complicated superscalar core. Their observation is that it is impossible to statically verify the superscalar core, but they can achieve the same result by verifying its execution at runtime. DIVA can dynamically verify all portions of the superscalar processor whose functionality is replicated in the checker core. By definition, dynamic verification cannot verify that a design is correct; rather, it verifies that a particular execution is correct. A dynamic verification approach like DIVA is fundamentally different from schemes that use homogeneous redundancy (e.g., redundant processors or threads) to tolerate faults, because identical components will share the same design bug and thus not be able to detect it. Some later research developed dynamic verification mechanisms for a multiprocessor’s memory system, via runtime checking that it obeys coherence [10, 27] or memory consistency [24].

Complexity-Effective Design. On a related but still different topic, there has been significant interest in complexity-effective designs, including a recently created workshop dedicated to this issue (Workshop on Complexity Effective Design). There are several definitions of complexity, most of which do not correspond directly to verification effort. Bazeghi et al. [5] recently developed the “ μ complexity” methodology for measuring and estimating processor design effort. Other research, most notably that of Palacharla and Smith [26], has provided guidelines for designing scalable superscalar cores, and they dubbed this work complexity-effective design. They consider a large, associative structure to be much more “complex” than a small, direct-mapped structure. However, the effort required to *verify* a structure depends quite a bit on the symmetry of its entries. If a structure is symmetric, then increasing its size and associativity do not increase the verification effort.

8. Conclusions

In this paper, we have argued that architects should explicitly consider verifiability when designing proces-

sors. The resources devoted to design verification are too expensive for us to continue to design processors without treating verifiability as a first-class design constraint, just like performance, power, temperature, reliability, etc. We have shown that subtle differences in a microarchitecture can lead to significant differences in verification effort. Looking into the future, we believe that the high costs of verification will motivate more radical redesigns of processors in order to ease the verification process. Tiled architectures, in particular, offer the potential for reduced verification effort, because the verification of each tile is the same, although we still must verify that the tiles communicate and synchronize correctly. Even when designing a tiled architecture, though, we believe that architects must still be careful to avoid adding features that complicate verification despite only offering limited benefits in other metrics.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant CCR-0309164, the National Aeronautics and Space Administration under grant NNG04GQ06G, an equipment donation from Intel Corporation, and a Duke Warren Faculty Scholarship. We thank Pradip Bose, Edmund Clarke, Chris Dwyer, Steven German, Alan Hu, Geert Janssen, Matt Kaufmann, Alvy Lebeck, Milo Martin, Albert Meixner, Costi Pistol, and Mark Tuttle for helpful discussions about this work.

References

- [1] Advanced Micro Devices. Revision Guide for AMD Athlon64 and AMD Opteron Processors. Publication 25759, Revision 3.59, Sept. 2006.
- [2] R. Alur et al. MOCHA: Modularity in Model Checking. In *Proceedings of the 10th International Conference on Computer Aided Verification*, pages 521–525, 1998.
- [3] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 196–207, Nov. 1999.
- [4] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzky, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 282–293, June 2000.
- [5] C. Bazeghi, F. J. Mesa-Martinez, and J. Renau. μ Complexity: Estimating Processor Design Effort. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, pages

- 209–218, Nov. 2005.
- [6] R. M. Bentley. Validating the Pentium 4 Microprocessor. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 493–498, July 2001.
- [7] M. Blum and H. Wasserman. Reflections on the Pentium Bug. *IEEE Transactions on Computers*, 45(4):385–393, Apr. 1996.
- [8] D. Boggs et al. The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology. *Intel Technology Journal*, 8(1), Feb. 2004.
- [9] P. Bose, D. H. Albonesi, and D. Marculescu. Guest Editors’ Introduction: Power and Complexity Aware Design. *IEEE Micro*, pages 8–11, Sept/Oct 2003.
- [10] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Dynamic Verification of Cache Coherence Protocols. In *Workshop on Memory Performance Issues*, June 2001.
- [11] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
- [12] P. Curzon and I. Leslie. Improving Hardware Designs Whilst Simplifying Their Proof. In *Proceedings of the 3rd Workshop on Designing Correct Circuits*, Sept. 1996.
- [13] N. Dave. Designing a Processor in Bluespec. Master’s thesis, Massachusetts Institute of Technology, Jan. 2005.
- [14] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol Verification as a Hardware Design Aid. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.
- [15] R. Hum. How to Boost Verification Productivity. *EE Times*, January 10 2005.
- [16] Intel Corporation. Intel Itanium Processor Specification Update. Order Number 249720-00, May 2003.
- [17] Intel Corporation. Intel Pentium 4 Processor Specification Update. Document Number 249199-065, June 2006.
- [18] C.-W. N. Ip. *State Reduction Methods for Automatic Formal Verification*. PhD thesis, Stanford University, Dec. 1996.
- [19] R. Jhala and K. L. McMillan. Microarchitecture Verification by Compositional Model Checking. In *Proceedings of the 13th International Conference on Computer Aided Verification*, pages 396–410, July 2001.
- [20] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way Multithreaded SPARC Processor. *IEEE Micro*, 25(2):21–29, Mar/Apr 2005.
- [21] M. M. K. Martin. Formal Verification and its Impact on the Snooping versus Directory Protocol Debate. In *Proceedings of the International Conference on Computer Design*, Oct. 2005.
- [22] M. R. Marty et al. Improving Multiple-CMP Systems Using Token Coherence. In *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture*, pages 328–339, Feb. 2005.
- [23] K. L. McMillan. Verification of Infinite State Systems by Compositional Model Checking. In *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 219–234, 1999.
- [24] A. Meixner and D. J. Sorin. Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2006.
- [25] G. J. Milne. Design for Verifiability. In *Proceedings of the Workshop on Hardware Specification, Verification, and Synthesis: Mathematical Aspects*, pages 1–13, 1989.
- [26] S. Palacharla and J. E. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.
- [27] D. J. Sorin, M. D. Hill, and D. A. Wood. Dynamic Verification of End-to-End Multiprocessor Invariants. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 281–290, June 2003.