Applications of Hierarchical Verification in Model Checking

Robert Beers, Rajnish Ghughal, and Mark Aagaard

Performance Microprocessor Division, Intel Corporation,
RA2-401, 5200 NE Elam Young Parkway, Hillsboro, OR 97124, USA.

Abstract. The LTL model checker that we use provides sound decomposition mechanisms within a purely model checking environment. We have exploited these mechanisms to successfully verify a wide spectrum of large and complex circuits. This paper describes a variety of the decomposition techniques that we have used as part of a large industrial formal verification effort on the Intel Pentium® 4 (Willamette) processor.

1 Introduction

One of the characteristics that distinguishes industrial formal verification from that done in academia is that industry often works on large, complex circuits described at the register transfer level (RTL). In comparison, academic work usually verifies either small RTL circuits or high-level abstractions of complex circuits. Academia remains the primary driving force in the development of new verification algorithms, and many of these advances have been succesfuly transfered from academia to industry [2,4,6,7,9,12,15,16, 18]. However, improving the strategies for applying these algorithms in industry requires exposure to circuits of a size and complexity that are rarely available to academics.

An important task in continuing the spread of formal verification in industry is to document and promulgate techniques for applying formal verification tools. In this paper we describe a variety of decomposition strategies that we have used as part of the formal verification project for the Intel Pentium® 4 (Willamette) processor. This paper focuses on strategies taken from verifying two different implementations of queues and a floating-point adder.

The verification tool we use is an LTL model checker developed at Intel that supports a variety of abstraction and decomposition techniques [14,15]. Our strategies should be generally applicable to most model-checking-based verification tools. In theory, the techniques are also applicable for theorem-proving. But, because the capacity limitations of model checking and theorem proving are so very different, a strategy that is effective in reducing the size of a model checking task might not be the best way to reduce the size of the problem for theorem proving.

2 Overview

In this section we give a high-level view of LTL model checking and provide some intuition about how the model checker works. The focus of this paper is on the application

T. Margaria and T. Melham (Eds.): CHARME 2001, LNCS 2144, pp. 40–57, 2001.
© Springer-Verlag Berlin Heidelberg 2001

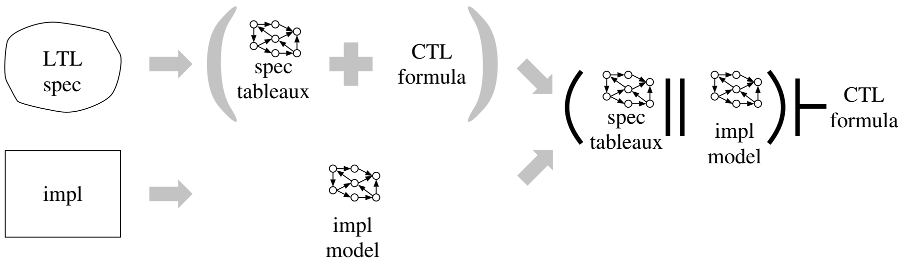

Fig. 1. Verification flow

of decomposition techniques, thus we concentrate on the usage of the model checker rather than the tool itself.

A high-level view of model checking for a linear temporal logic (LTL) is shown in Fig. 1. LTL model checking is done by converting the specification (which is an LTL formula) to a tableaux and a CTL formula. The tableaux is an automaton that recognizes traces that satisfy the LTL formula. The CTL formula checks that the tableaux never fails. The verification run computes the product machine of the tableaux and the implementation and checks that the product machine satisfies the CTL formula [8].

Because specifications are converted into automata, and model checking verifies automata against temporal formulas, specifications can themselves be verified against higher-level specifications. Figure 2 shows an implementation that is verified against a middle-level specification, which, in turn, is verified against a high-level specification.

We now step through a simple example of hierarchical decomposition, using the circuit `pri` shown in Fig. 3. The circuit `pri` takes three inputs (`i0`, `i1`, and `i2`) and outputs a `1` on the highest priority output line line whose input was a `1`. We model

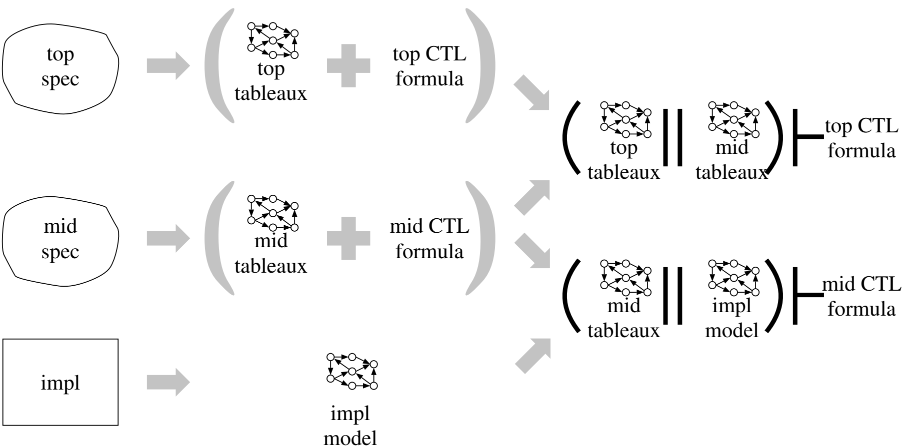

Fig. 2. Hierarchical verification flow

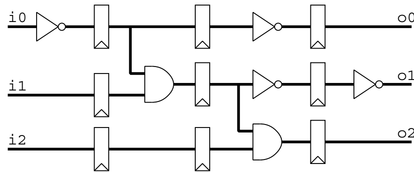

Fig. 3. Example circuit: `pri`

combinational logic as having zero delay and flip flops as being unit delay. We draw flip flops as rectangles with small triangles at the bottom.

Theorem 1 shows an example property that we will verify about `pri`. The property says that, for `pri`, if the output `o0` is `1`, then `o2` will be `0`. For the sake of illustration, we will pretend that verifying Theorem 1 is beyond the capacity of a model checker, so we decompose the problem. We carry out the verification in three steps by verifying Lemmas 1–3.

Theorem 1. $\mathtt{pri} \models (\mathtt{o0} \Longrightarrow \neg\mathtt{o2})$

Lemma 1. $\mathtt{pri} \models (\mathtt{o0} \Longrightarrow \neg\mathtt{o1})$

Lemma 2. $\mathtt{pri} \models (\neg\mathtt{o1} \Longrightarrow \neg\mathtt{o2})$

Lemma 3. $(\mathtt{Lemma1}, \mathtt{Lemma2}) \models (\mathtt{o0} \Longrightarrow \neg\mathtt{o2})$

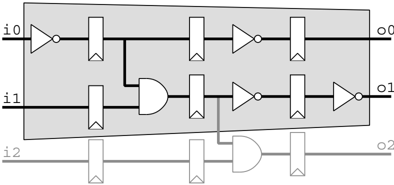

Fig. 4. Cone of influence reduction for Lemma 1

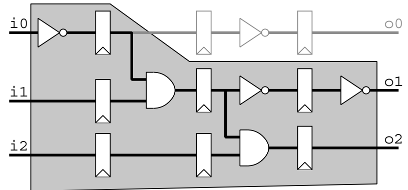

Fig. 5. Cone of influence reduction for Lemma 2

By decomposing the verification, we end up with multiple verification runs, but each one is smaller than Theorem 1, the original run (Table 1). The first two runs (Lemmas 1 and 2) are smaller than Theorem 1, because, using *cone of influence* reduction, Lemmas 1 and 2 do not need to look at the entire circuit. The last run, Lemma 3, glues together the earlier results to verify the top-level specification. When verifying Lemma 3, we do not need the circuit at all, and so its verification is very small and easy to run.

One advantage of this style of hierarchical verification is that it provides some theorem-proving-like capabilities in a purely model checking environment. Two disadvantages are that the approach is still subject to the expressiveness limitations and

Table 1. Summary of verification of `pri`

property	latches	inputs	total variables
Theorem 1	9	3	12
Lemma 1	6	2	8
Lemma 2	7	3	10
Lemma 3	0	3	3

some of the capacity limits of BDDs and that it does not support reasoning about the entire chain of reasoning. That is, we use Lemma 3 to verify the right-hand-side of Theorem 1 but we cannot explicitly prove Theorem 1, instead we use a database of specifications to track chains of dependencies.

3 Parallel Ready Queue

This section describes part of the verification of a "parallel ready queue" (`prqueue`) that operates in parallel on k channels (Fig. 6). Packets arrive, one at a time, at any of the `arrive` signals. A packet departs the queue if it is the oldest ready packet. Packet i becomes ready when `ready`$[i]$ is `1`. Once a packet is ready, it remains ready until it has departed. Packets can become ready to leave the queue in any order. Hence, it is possible for a packet to leave the queue before an older packet if the older packet is not yet ready.

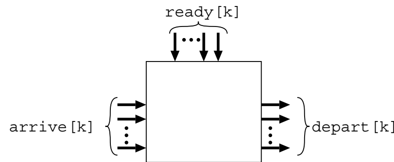

Fig. 6. Parallel queue circuit (`prqueue`)

There are a number of environmental constraints that are required for the circuit to work correctly:

1. A packet will not arrive on a channel if the queue currently contains a packet in that channel.
2. At most one packet will arrive at any time.
3. If a packet arrives on a channel, the corresponding ready signal will eventually be set to `1`.
4. A channel will not be marked as ready if it does not contain a packet.

Note, the environment has the freedom to simultaneously mark multiple packets as ready.

The top level specification (Theorem 2) covers both liveness and safety.

Theorem 2. *PriQueue Specification*
1. *A packet that arrives on* `arrive[i]` *will eventually depart on* `depart[i]`.
2. *If a packet departs, it is the oldest ready packet in the queue.*

3.1 Implementation

We briefly describe the high-level algorithm for the implementation of `prqueue`. There are many sources of complexity, e.g., datapath power-saving control logic and various gated clock circuits, which we ignore for the sake of simplicity in the following description. The implementation of `prqueue` uses a two-dimensional $k \times k$ scoreboard (`scb`) to maintain the relative age of packets and an array of k `rdy` signals to remember if a channel is ready. Figure 7 shows the scoreboard and ready array for a queue with four channels.

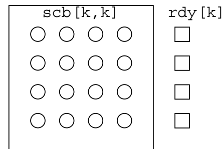

Fig. 7. Scoreboard in parallel ready queue

Initially, all of the elements in the scoreboard and ready array are set to `0`. When a packet arrives on `arrive[i]`, all of the scoreboard elements in column `i` are set to `0` and all of the elements in row `i` are set to `1`. The intuition behind this algorithm lies in two observations:

- The scoreboard row for the youngest (most recently arrived) packet is the row with the most `1`s, while the oldest packet has the row with the most `0`s.
- The packet for row `i` is older than the packet for row `j` if and only if $\mathtt{scb}[\mathtt{i},\mathtt{j}] = \mathtt{1}$.

The diagonal elements do not contain any semantically useful information. When $\mathtt{ready}[i]$ is set to `1`, packet i will be marked as ready to leave the queue. If at least one of the packets is marked as ready, the scoreboard values are examined to determine the relative age of all ready packets and the oldest among them departs the queue. The process of packets arriving, being marked as ready, and departing is illustrated in Fig. 8.

As is all too common when working with high-performance circuits, the specification and high-level description of the implementation appear deceivingly straightforward. This circuit has been optimized for area and performance and the actual implementation is much more complex than the high-level description. For example, the logic for $\mathtt{ready}[i]$ signal involves computation of multiple conditions which all must be satisfied before the packet is ready to depart the scoreboard. We have ignored many such details of implementation for the purpose of the simplicity of the description. The actual circuit verified consists of hundreds of latches with significantly larger values for k.

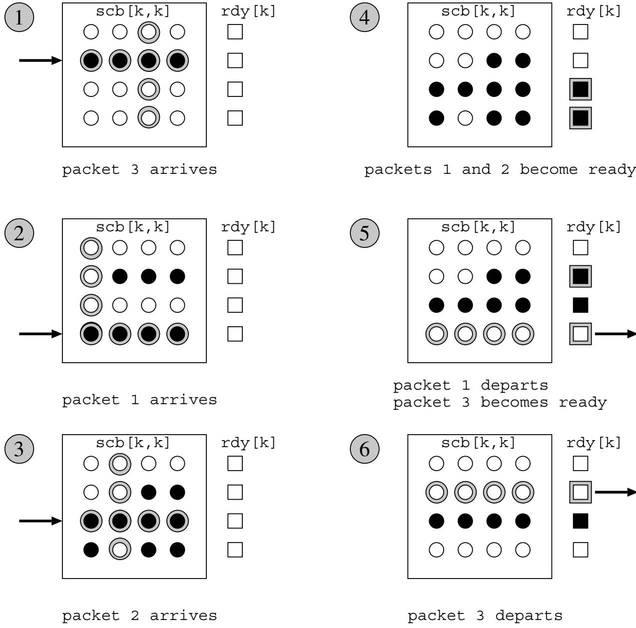

Fig. 8. Scoreboard in parallel ready queue

3.2 Verification

Theorem 3 is one of the major theorems used in verifying the circuit against the top-level specification 2.

Theorem 3 says: *if a packet is ready, then a packet will depart*. The signal $\texttt{depValid}[i]$ is the "valid bit" for $\texttt{depart}[i]$; it says that the packet on row i is departing. There are some environmental assumptions such as issues with initialization that we do not encode in this specification for the sake of simplicity.

Theorem 3 (willDepart).
$(\exists i.\ \texttt{rdy}[i]) \implies (\exists i.\ \texttt{depValid}[i])$

Theorem 3 is a safety property of the implementation. The logic expression $\exists i.\ \texttt{rdy}[i]$ stands for the boolean expression $\texttt{rdy}[0] \vee \texttt{rdy}[1] \vee ...\texttt{rdy}[k]$. Theorem 3 is dependent upon the fanin cone for all k^2 entries in the scoreboard. The fanin cone of the scoreboard consists of hundreds of latches. This amount of circuitry and the temporal nature of information stored in the scoreboard array puts it well beyond the capacity of model checking for values of k found in realistic designs.

To carry out the verification, we decomposed the task into two parts. First, we verified that the scoreboard and ready array impart a transitive and anti-symmetric relationship that describes the relative ages of packets (Lemmas 4 and 5). Second, we verified that these properties imply Theorem 3.

Lemma 4 (Transitivity $i\ j\ k$).
$\texttt{rdy}[i] \wedge \texttt{rdy}[j] \wedge \texttt{rdy}[k] \wedge \texttt{scb}[i,j] \wedge \texttt{scb}[j,k] \implies \texttt{scb}[i,k]$

Lemma 4 says that if packet i is older than packet j and packet j is older than packet k, then packet i is older than packet k. (Recall that packet i is older than packet j if $\texttt{scb}[i,j] = 1$). The benefit of Lemma 4 is that we reduce the size of the circuit needed from k^2 scoreboard entries to just three entries. The cost of this reduction is that we had to verify the lemma for all possible combinations of scoreboard indices such that $i \neq j \neq k$, which resulted in $k \times (k-1) \times (k-2)$ verification runs

Lemma 5 (AntiSym i j).
$\texttt{rdy}[i] \wedge \texttt{rdy}[j] \implies \texttt{scb}[i,j] = \neg\texttt{scb}[j,i]$

Lemma 5 (anti-symmetry) says that if packet i is older than packet j, then packet j cannot be older than packet i (and vice versa.) It is easy to see why this lemma is true by observing that whenever a packet arrives the corresponding column values are set to `0` while the corresponding row values are set to `1`. The anti-symmetry lemma relates only two values of the scoreboard entries. We verified Lemma 5 for all possible combinations of i and j (such that $i \neq j$), which resulted in $k \times (k-1)$ verification runs.

After verifying Lemmas 4 and 5, we verified Theorem 3 by removing the fanin cone of the scoreboard array while assuming the two lemmas. Removing the fanin cone of the scoreboard array is essentially an abstraction which preserves its pertinent properties using transitivity and anti-symmetry lemmas. This particular abstraction preserves the truth of the safety property under consideration. The transitivity and anti-symmetry lemmas provide just enough restriction on the scoreboard values to establish a relative order of arrival among packets that are ready to exit. Note that the transitivity relationship between three scoreboard entries is sufficient to enforce proper values in the scoreboard array even when more than three packets are ready to depart.

The major sources of complexity in verifying Theorem 3 were:

- The large fanin cone of the scoreboard array, which builds up complex temporal information.
- The size of the scoreboard array: $k \times k$ entries.

We were able to verify Theorem 3 without this hierarchical decomposition for very small values of k. However, the verification task becomes increasingly more difficult for larger values of k. The hierarchical verification approach overcomes the model checking complexity for two reasons:

- The transitivity and anti-symmetry lemmas relate three and two values of the scoreboard entries. The verification complexity of these lemmas is relatively small and is independent of the size of the scoreboard.
- For the verification of Theorem 3, we were able to remove the fanin cone of the entire scoreboard. We enforce proper values in scoreboard entries by enforcing the transitivity and anti-symmetry safety properties which themselves are not of a temporal nature. However, these lemmas together provide enough information to build an implicit order of arrival of packets that allows the implementation to pick the eldest packet.

4 Floating-Point Adder

We briefly preview the basics of binary floating-point numbers [11] before diving into some of the decompositions used in verifying a floating point adder. A floating-point number is represented as a triple (s, e, m) where, s is a sign bit, e is a bit-vector representing the exponent, and m is a bit-vector representing the mantissa. We have verified the sign, mantisa and exponent for both *true addition* and *true subtraction* (the signs of the operands are the same or differ), but consider only the verification of the mantisa for true addition in this paper.

The real number represented by the triple (s, e, m) is:

$$(-1)^{\hat{s}} \times 2^{\hat{e}-bias} \times \hat{m} \times 2^{-n_m+1}$$

where $\hat{x}$ is the unsigned integer encoding by the bit vector x and *bias* is a format-dependent *exponent bias*. The mantissa has an implicit leading 1, and so it represents the value $1.m$, which is in the range $[1, 2)$.

4.1 Implementation

A simple algorithm to calculate the result mantissa for floating-point addition is shown in Fig. 9.

input: two normal floating-point numbers $f_1 = (s_1, e_1, m_1)$ and $f_2 = (s_2, e_2, m_2)$

expdiff	$:= \lvert\hat{e_1} - \hat{e_2}\rvert$;	absolute diff of exps
expcmp	$:= \hat{e_1} > \hat{e_2}$;	$\hat{f_1}$ is the larger number
bigman	:= if expcmp then $1.m_1$ else $1.m_2$;	mant of larger number
smallman	:= if expcmp then $1.m_2$ else $1.m_1$;	mant of smaller number
alignman	:= shift_right(smallman, expdiff);	align small mant
addman	:= bigman + alignman;	add mantissas
resultman	:= round(addman);	round result

Fig. 9. Simple floating-point true addition algorithm

A simplified implementation for a floating-point true addition circuit `fpadder` is shown in Fig. 10. It follows the basic structure of the algorithm in Fig. 9. The actual implementation of the adder that we verified is much more complex than this simple depiction. For example, the actual implementation has more than just one adder and more than one parallel datapaths for the sake of performance.

4.2 Verification

We use a reference model similar to the algorithm depicted in Fig. 9 as the specification for floating-point addition. Theorem 4 is our top-level theorem for the mantissa.

Theorem 4 (Mantissa).
f_1 and f_2 are normal $\Longrightarrow$ (spec_resultman = impl_resultman)

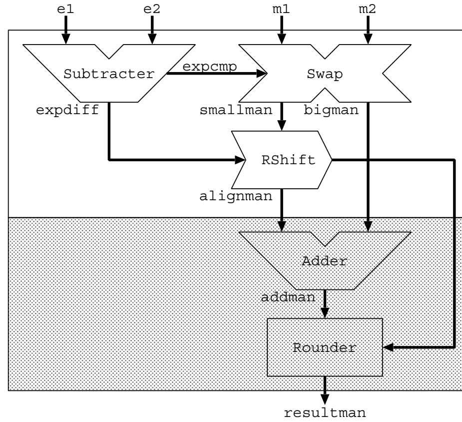

Fig. 10. Simplified circuit for floating-point true addition : `fpadder`

For the purpose of the verification, we essentially removed all state elements by *unlatching* the circuit thus arriving at a combinational model of the circuit. We used our model checker, instead of an equivalence checker, to verify the above property on the combinational model as only the model-checking environment provides support for hierarchical decomposition. Also, for the purpose of simplicity we do not include environmental assumptions in the description of Theorem 4.

The main challenge we faced in verifying Theorem 4 was BDD blow-up due to the combined presence of a shifter and adder. As has been reported by others [2,5], good variable orderings exist for fixed differences of exponents; however, the BDD size for the combination of variable shifting and addition is exponential with respect to the width of the datapath. Note that both the specification and implementation contain this combination, so we could not represent either the complete implementation or the complete specification with BDDs. Our strategy was to do structural decomposition on *both* the specification and implementation.

Lemma 6 (Shiftermatch).
f_1 and f_2 are normal $\Longrightarrow$
$(\texttt{spec_alignman} = \texttt{impl_alignman}) \wedge$
$(\texttt{spec_bigman} = \texttt{impl_bigman}) \wedge$
$\textit{good_properties}(\texttt{spec_alignman}, \texttt{spec_bigman})$

We decomposed the specification and implementation at the output of the shifter at the `alignman`, `bigman` and `sticky` signals. This decomposed the verification into two steps: from the inputs of the circuit through the shifter, and the adder and the rounder shown in Fig. 10. The two steps are shown in different shades. Through judicious use of cone of influence reductions and other optimizations to both the specification and implementation, we were able to verify the adder and the rounder without needing to include the shifter.

For the first half of the circuit, we verified that the output of the shifter in the specification is equivalent to that in the implementation. Our initial specification for

the rounder was that for equal inputs, the specification and the implementation return equal outputs. However, we discovered that the implementation was optimized such that it returned correct results only for legal combinations of exponent differences and pre-rounded mantissas.

We then began a trial and error effort to derive a set of properties about the difference between the exponents and the pre-rounded mantissa. We eventually identified close to a dozen properties that, taken together, sufficiently constrained the inputs to the rounder such that the implementation matched the specification. Some of the properties are implementation dependent and some are of a more general nature. A few examples of implementation independent properties are:

- The leftmost exponent-difference number of bits of `spec_alignman` must be all $0s$.
- For exponent differences larger than a specific value, the `sticky` bit and the disjunction of a fixed implementation-dependent number of least significant bits in `spec_alignman` must be 1.
- The `expdiff+1`th leftmost bit of `spec_alignman` must be 1.

We modified the original lemma for the first half of the circuit to include these properties. We were able to verify Theorem 4 after removing both the shifters and enforcing the lemmas strengthened with good properties for the shifter and sticky bit.

The above approach illustrates how similar hierarchies present in the implementation and specification can be effectively used to decompose a verification task into smaller and more manageable sub-tasks. We have observed that this technique is particularly applicable when verifying an implementation against a reference model.

In [5], the authors present an approach for verification of floating-point adders based on word-level SMV and multiplicative power HDDs (*PHDDs). In their approach, the specifications of FP adders are divided into hundreds of implementation-independent sub-specifications based on case analysis using the exponent difference. They introduced a technique called short-circuiting to handle the complexity issues arising due to ordering problems.

Our approach uses a model-checker based on BDDs as opposed to word-level model-checker or *PHDDs. Our approach involves a small number of decompositions, as opposed to hundreds of cases. However in our approach, the decomposition specifications are implementation-dependent. We believe that for different implementations one can use the same basic approach towards decomposition. The $good_properties$ will require the user to incorporate implementation-dependent details. Also, the hierarchical decomposition separates the shifter and the adder and hence does not require any special techniques such as short-circuiting to handle the verification complexity due to conflicting orders requirement.

5 Memory Arrays

Modern microprocessors contain dozens of first-in first-out (FIFO) queues. Certainly, the functional correctness of the processors depends upon the correct behavior of every FIFO in their designs. Due to the evolution of processors (further pipelining, wider datapaths,

and so on), verifying just one typical FIFO can be a challenging task, especially for large FIFOs that are implemented as memory-arrays for area/performance optimizations. Verification of such FIFOs requires decomposing the problem into invariants about the control and datapath implementations. Our work does not refute this notion. Yet, although the proof structure of every FIFO we examined always required many invariants at various level of abstraction, we were able to leverage commonalities in the decomposition paths and methods, even for FIFOs being used in entirely different manners and being verified by different individuals.

5.1 Implementations of Memory Arrays

Memory arrays used as FIFOs are implemented as the array combined with a set of finite state machines that control a write (tail) pointer and a read (head) pointer (Fig. 11). Each pointer advances downward with its respective operation, incrementing by the number of elements written to or read from the array, and "wrapping" to the first (top) element when leaving the last (bottom) element. These pointers may take various forms depending upon the physical layout of the memory—linear arrays require only addresses whereas two-dimensional arrays may require an address and an offset into it, which may be implemented as slices of a single address.

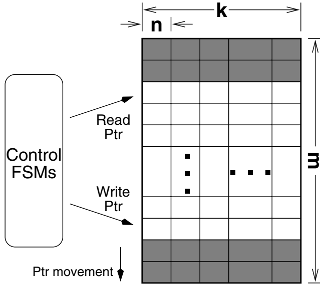

Fig. 11. Memory array circuit

Optimizations for area and performance usually lead to two-dimensional arrays rather than one-dimensional arrays. We use the term *slot* as the storage elements necessary for one queue element, and *row* as a group of slots associated with one address. Thus, an array consists of m rows with k slots per row containing n bits of data.

In one of the arrays we verified, all operations (writes and reads) access an entire row at a time. Other arrays allowed *spanning* writes and/or *partial* reads and writes (see Fig. 12). A spanning operation accesses slots in adjacent rows at the same time. A partial operation accesses fewer than k slots, and may or may not be spanning. In a partial access, the offset always changes but the row pointer may not, and vice versa for a spanning operation.

Optimizations in the control FSMs also lead to simplifications that tend to be problematic for verification and decomposition efforts. For example, the FSMs rarely, if ever,

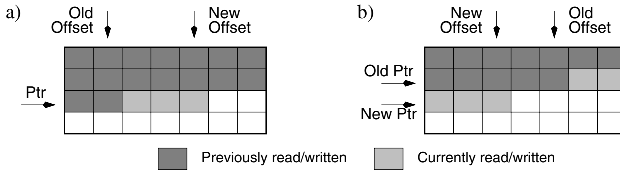

Fig. 12. a) Partial operation, b) Spanning (and partial) operation

take into consideration the position of both pointers. Thus, when examining only the read pointer logic there is nothing to prevent the read pointer from passing the write pointer (an unacceptable behavior). In the absence of constraint properties, only by including the logic of both pointers *and* the datapath could we prevent this from occurring. This detail alone nearly yields memory-array verification intractable for model checkers.

5.2 Overview of Memory Array Verification

Every FIFO in a processor has one or more correctness invariants that the design must satisfy. Verifying most top-level invariants requires reasoning about the behavior of the entire control and datapath. One array we worked with contained over 2100 variables in the necessary logic. Therefore, decomposition was a necessity.

Our technique for verifying memory-array invariants is summarized as follows:

1. Divide the memory array into symmetric, tool-manageable *slices* (e.g., slice equals one or more slots per row, or slice equals one or more rows.)
2. Define a set of properties applicable to every slice.
3. Define properties about the control FSMs that are as abstract as possible (for cone reduction) yet strong enough to guarantee the top-level invariant.
4. Verify the top-level invariants using the properties from 2 and 3.
5. For each slice, verify that if the properties about the control FSMs and the slice hold in a state, then the slice properties will hold in the next state.
6. For each property about the control FSMs, verify that if all properties about the control FSMS hold in a state, then the property will hold in the next state.
7. For any slice or control FSM property that is still intractable with the tools, repeat the decomposition procedure.

We identified a set of five invariants upon the read and write pointers that were used in nearly every memory array abstraction:

- The write pointer changes only when a write occurs.
- The read pointer changes only when a read occurs.
- If a row pointer changes, it increments by one (or more, depending upon the arrays input and output widths) and wraps at the maximum row.
- The read pointer will not pass the write pointer (begin reading bogus data)
- The write pointer will not pass the read pointer (overwrite existing data)

Arrays that allow partial and spanning operations contain offsets indicating which slots in a row are being accessed. Interactions between the offset circuitry and the memory array elements require additional invariants:

- If a read (write) occurs and the row pointer does not change, then the offset increments.
- If a read (write) occurs and the row pointer changes, then the offset decrements or stays the same.

With modern processor designs, reducing the complex nature of the circuitry controlling most FIFOs requires several hierarchical steps just to get to the abstractions above. These levels of the proof trees deal with implementation details for each particular FIFO.

The (deeply) pipelined nature of the array's datapath and control often requires temporally shifting the effects of these properties in order to substantially reduce the variable counts in "higher" proofs. In other words, using assumptions upon signals in different pipeline stages incurs a penalty if the signals must propagate to further pipestages. It is better to use existing assumptions, which cover the "wrong" pipestages, to prove similar specifications about the propagated signals, especially as the level of abstraction (and freedom given to the model checker) increases. FSM and datapath sizes and design complexities often required temporally shifting the same property to several different times to align with the various FIFO processing stages.

5.3 Array Examples

Here we present two examples of the memory-array verifications we carried out.

Our first example is a "write-once" invariant. This property requires that every slot is written to only once (to prevent destruction) until the read pointer passes it. The FIFO allowed for spanning and partial writes. Thus, every slot on every row has its own write enable line that is calculated from the write pointer, next offset, and global write enable.

For this invariant we sliced the array by row so that the final proofs involved m *predicate variables*. The predicate variables were defined as sticky bits that asserted if the write enable bits on the row behaved incorrectly. In this "write-once" context, incorrect behavior of the write enables includes the following:

1. The row is not being written to but one or more of the write enables asserted.
2. During a write to the row, a write enable for a slot behind the write pointer (and offset) and ahead of the read pointer (and offset) was asserted.

Note that there are other aberrant behaviors of the write enables but they fall outside the scope of the top-level invariant. They could be included in the predicate variable definitions without affecting the correctness of the final proof. However, needlessly extending the definitions runs the risk of undoing the necessary cone reductions.

The complexity in this verification comes from the inclusion of the read pointer in the predicate definitions. As mentioned before, the control FSMs generally do not take into account the behavior or values of the other pointers or FSMs. This FIFO's environmental requirement is that it will never receive more items than it can hold; otherwise, the write pointer could pass the read pointer. Even with this constraint we were forced to fully

abstract the read and write pointers before attempting to prove that the predicate bits would never assert.

The last step was to sufficiently abstract the behavior of the predicate variables so they could all be used in the top-level proof. The following invariants were sufficient:

1. Initially, the predicate bit is not asserted.
2. If a row is not being written, its predicate bit does not change.
3. If a row is being written, its predicate bit does not transition from 0 to 1.

These invariants reduce the behavior of the predicate bits to a relation with the write pointer. Together they clearly satisfy the top-level specification (no predicate bit will ever become asserted, thus all write enables behaved as expected). While they may appear simple, the size of the memory array and the capacity limitations of model checking deemed them necessary.

Our second example was our first full-scale attempt at verifying an invariant upon an entire memory array. In this FIFO, control information needed by the read pointer's FSMs is written into the array by the write FSMs. The FIFO's correctness depends upon it being able to write valid and correct control data into the array and then safely and accurately read the data out.

Ideally we would have liked to have been able to prove the correctness of the read FSMs while allowing the control information to freely be any value of the model checker's choosing. However, it quickly became apparent that circuit optimizations required constraining the control data to a "*consistent*" set of legal values. Because rigorous verification work does not particularly enjoy the idea of correct data appearing for free, we were forced to show that control information read from the array always belonged to this consistent set. Fortunately, the idea of a valid set of bit values lends itself well to the notion of predicate variables.

The number of bits in the control data set required slicing the array twice: once by row, and then by slot per row. The slot-wise predicate bits indicated whether the slot's current control data was consistent. The row-wise bits, of which there were actually two sets, examined the predicate bits for the entire row while considering the write pointer and offset (or the read pointer and offset). Because the control information flows through the entire FIFO, we had to prove consistency in the following progression through the array's datapath:

- At the inputs to the memory array, for every slot that will be written, the control data is consistent.
- For each array row, if the write pointer moves onto or stays on the row, control data in each slot behind the (new) write offset is consistent.
- For each row, if the write pointer moves off the row, the entire row is consistent.
- For each row, if the write pointer neither moves onto nor moves off of the row, the consistency of each slot's control data does not change.
- For each row, if the read pointer is on the row, and if the write pointer is on the row, then all slots behind the write offset are consistent.
- For each row, if only the read pointer is on the row, all slots are consistent.
- At the memory array's output muxes, when a read occurs, the *active* output slots contain consistent control data, where *active* is defined by the write and read pointers

and offsets (i.e., if pointers on same row, active is between the offsets; otherwise active is above read offset.)

The last step establishes that we will always received consistent control data from the memory array.

As with the FIFO of the write-once invariant, we spent a significant effort reducing the read and write FSM logic to the minimum required behaviors and proving these behaviors at various stages in the pipeline (to align with the FIFO stages described above).

6 Conclusion

Related work in decomposition and model checking can be grouped into three large categories: sound combinations of model checking and theorem proving, ad-hoc combinations of different tools, and hard-coding inference rules into a model checker.

Sound and effective combinations of theorem proving and model checking has been a goal for almost ten years. Joyce and Seger experimented with using trajectory evaluation as a decision procedure for the HOL proof system [13]. They concluded that for hardware verification, most of the user's interaction is with the model checker, not the proof system. Consequently, using a model checker as a decision procedure in a proof system does not result in an effective hardware verification environment. The PVS proof system has included support for BDDs and mu-calculus model checking [17] and, over time, has received improved support for debugging. Aagaard *et al* have described extensions to the Voss verification system that includes a lightweight theorem proving tool tightly connected to the Voss model checking environment [3,1]. Gordon is experimenting with techniques to provide a low-level and tight connection between BDDs and theorem proving in the HOL proof system [10].

There are two advantages to combining theorem proving and model checking over a pure model checking based approach, such as we have used. First, general purpose logics provide greater expressability than specification languages tailored to model checking. Second, the sound integegration of model checking and theorem proving allows more rigorous results. The principal advantage of the approach outlined here was pragmatic: it enabled us to achieve a more significant result with less effort. Hierarchical model checking allowed an existing group of model checking users to extend the size and quality of their verifications with relatively minor costs in education.

Because finding an effective and sound combination of theorem proving and model checking has been so difficult, a variety of ad hoc combinations have been used to achieve effective solutions at the expense of mechanically guaranteed soundness. Representative of these efforts is separate work by Camilleri and Jang *et al*. Camilleri [4] has used both theorem-proving and model-checking tools in verifying properties of a cache-coherency algorithm. Jang *et al* [12] used CTL model checking to verify a collection of 76 properties about an embedded microcontroller and informal arguments to convince themselves that their collection of properties were sufficient to claim that they verified their high-level specification.

In an effort to gain some of the capacity improvements provided by a proof system without sacrificing the automation of model checking, McMillan has added inference

rules for refinement and decomposition to the Cadence Berkely Labs SMV (CBL SMV) model checker [16]. Eiríksson has used CBL SMV to refine a high-level model of a protocol circuit to a pipelined implementation [9].

The model checker we use shares characteristics of ad hoc combinations of techniques and of adding theorem proving capabilities to a model checker. In essence, the model checker provides us with a decision procedure for propositional logic. This allows us to stay within a purely model checking environment and prove propositional formulae that can be solved by BDDs. However, because we cannot reason about the syntax of formulas, we cannot do Modus Ponens reasoning on arbitrary chains of formulas. Hence, the complexity of our decompositions is limited by the size of BDDs that we can build and we use a separate database facility to ensure the integritity of the overall decomposition.

Although we do not use a theorem prover, we have found that experience with theorem proving can be useful in finding effective decomposition strategies. The challenge is to find *sound* and *effective* decomposition techniques. Assuring the soundness of a decomposition technique benefits from a solid grounding in mathematical logic. Mathematical expertise can also be helpful by providing background knowledge of a menu of decomposition techniques to choose from (e.g., temporal induction, structural induction, data abstraction, etc.). In most situations, many sound decomposition techniques are applicable, but most will not be helpful in mitigating the capacity limitations of model checking. Picking an effective decomposition technique requires knowledge of both the circuit being verified and the model checker being used. We often found that slight modifications to specifications (e.g. the ordering of temporal or Boolean operators) dramatically affect the memory usage or runtime of a verification. Over time, our group developed heuristics for writing specifications so as to extract the maximum capacity from the model checker.

Decomposition techniques similar to the ones that we have described have become standard practice within our group. For this paper we selected illustrative examples to give some insights into how we use decomposition and why we have found it to be successful. Table 2 shows some statistics of a representative sample of the verifications that have used these techniques.

Table 2. Circuit sizes and verification results

Circuit	total latches	average number of latches per decomposition	number of verification runs	maximum memory usage	total run time	
t-mem	903	9	70	830MB	27h	
f-r-stall	210	160	10	150MB	23h	
rt-array	500	311	6	110MB	14h	
rq-array	2100	140	300	120MB	100h	Section 5
all-br	1500	175	200	600MB	170h	Section 5
m-buf-com	200	140	20	250MB	20h	
fpadd	2000	400	100	1200MB	2h	Section 4
fpsub	2500	500	100	1800MB	7h	

The formal verification effort for the Intel Pentium® 4 processor relied on both formal verification and conventional simulation-based validation. The goal, which in fact was achieved, was that simulation would catch most of the errors and that formal verification would locate pernicious, hard-to-find errors that had gone undetected by simulation.

Acknowledgments. We are indebted to Kent Smith for suggesting the decomposition strategy for the parallel ready queue and many other helpful discussions. We also would like to thank Bob Brennan for the opportunity to perform this work on circuits from Intel microprocessors and Ravi Bulusu for many discussions and helpful comments on the floating-point adder verification.

References

1. M. D. Aagaard, R. B. Jones, K. R. Kohatsu, R. Kaivola, and C.-J. H. Seger. Formal verification of iterative algorithms in microprocessors. In *DAC*, June 2000.
2. M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. Formal verification using parametric representations of Boolean constraints. In *DAC*, July 1999. (Short paper).
3. M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. Lifted-fl: A pragmatic implementation of combined model checking and theorem proving. In L. Thery, editor, *Theorem Proving in Higher Order Logics*, pages 323–340. Springer Verlag; New York, Sept. 1999.
4. A. Camilleri. A hybrid approach to verifying liveness in a symmetric multi-processor. In E. L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics*, pages 49–68. Springer Verlag; New York, Sept. 1997.
5. Y.-A. Chen and R. Bryant. Verification of floating-point adders. In A. J. Hu and M. Y. Vardi, editors, *CAV*, pages 488–499, July 1998.
6. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. on Prog. Lang. and Systems*, 16(5):1512–1542, Sept. 1994.
7. E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *LICS*, pages 353–362, 1989.
8. E. M. J. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press; Cambridge, MA, 1999.
9. A. P. Eiríkson. The formal design of 1m-gate ASICs. In P. Windley and G. Gopalakrishnan, editors, *Formal Methods in CAD*, pages 49–63. Springer Verlag; New York, Nov. 1998.
10. M. Gordon. Programming combinations of deduction and BDD-based symbolic calculation. Technical Report 480, Cambridge Comp. Lab, 1999.
11. IEEE. *IEEE Standard for binary floating-point arithmetic*. ANSI/IEEE Std 754-1985, 1985.
12. J.-Y. Jang, S. Qadeer, M. Kaufmann, and C. Pixley. Formal verification of FIRE: A case study. In *DAC*, pages 173–177, June 1997.
13. J. Joyce and C.-J. Seger. Linking BDD based symbolic evaluation to interactive theorem proving. In *DAC*, June 1993.
14. G. Kamhi, L. Fix, and O. Weissberg. Automatic datapath extraction for efficient usage of hdds. In O. Grumberg, editor, *CAV*, pages 95–106. Springer Verlag; New York, 1997.
15. S. Mador-Haim and L. Fix. Input elimination and abstraction in model checking. In P. Windley and G. Gopalakrishnan, editors, *Formal Methods in CAD*, pages 304–320. Springer Verlag; New York, Nov. 1998.
16. K. McMillan. Minimalist proof assistants: Interactions of technology and methodology in formal system level verification. In G. C. Gopalakrishnan and P. J. Windley, editors, *Formal Methods in CAD*, page 1. Springer Verlag; New York, Nov. 1998.

17. S. P. Miller and M. Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *Workshop on Industrial-Strength Formal Specification Techniques*, Apr. 1995.
18. Y. Xu, E. Cerny, A. Silburt, A. Coady, Y. Liu, and P. Pownall. Practical application of formal verification techniques on a frame mux/demux chip from Nortel Semiconductors. In L. Pierre and T. Kropf, editors, *CHARME*, pages 110–124. Springer Verlag; New York, Oct. 1999.