Formal Verification of Pentium®4 Components with Symbolic Simulation and Inductive Invariants

Roope Kaivola

Intel Corporation, JF4-451, 2111 NE 25th Avenue, Hillsboro, OR 97124, USA

Abstract. We describe a practical methodology for large-scale formal verification of control-intensive industrial circuits. It combines symbolic simulation with human-generated inductive invariants, and a proof tool for verifying implications between constraint lists. The approach has emerged from extensive experiences in the formal verification of key parts of the Intel IA-32 Pentium®4 microprocessor designs. We discuss it in the context of two case studies: Pentium 4 register renaming mechanism and BUS recycle logic.

1 Introduction

It is easy to explain why formal verification of microprocessors is hard. A state-of-the-art microprocessor may have millions of state elements, whereas a state-of-the-art formal verification engine providing complete coverage can usually handle a few hundred significant state elements. This basic technology problem stems from the computational complexity of many of the algorithms used for formal verification. At the same time, the strong guarantees of correctness given by verification would be particularly valuable in the domain of microprocessors. The products are used in circumstances where their reliability is crucial, and just the financial cost of correcting problems can be very high.

In an industrial product development project formal verification is a tool, one among others, and it has to compete with other validation methods such as traditional testing and simulation. We believe quite strongly that in this setting the greatest value of formal verification comes from its ability to provide complete coverage, finding subtle design problems that testing may have missed, and yielding strong evidence about the absence of any further problems. This is naturally not the only usage model for formal methods: more lightweight approaches providing partial coverage such as partial model exploration, bounded model checking, capturing design intent by assertions etc. can also provide value to a project. Nevertheless, systematic testing and simulation often provide reasonable partial coverage already, and in our opinion the most compelling argument for the adoption of formal verification is that it can provide guarantees of correctness that testing cannot.

Formal verification has been pursued in various forms for roughly a decade now in Intel [19]. Usually the goal of a verification effort is to relate a detailed register transfer level circuit model to a clear and abstract specification. In some areas, most notably in arithmetic hardware, verification methods have reached sufficient maturity that they can now be routinely applied (for discussion see [14]). The work still requires a great deal of human expertise, but tasks that used to take months can be carried out in days, and the work can be planned ahead with reasonable confidence. In other areas the track record

is mixed. Traditional temporal logic model checkers perform well for local properties, but quickly run out of steam when the size of the sub-circuit relevant for the property grows. Decompositions and assume-guarantee reasoning can alleviate this problem, but the help they provide tends to gradually diminish as the circuits grow larger, as well.

Large control intensive circuits have proved to be particularly resistant to formal verification. They contain enough state elements that even after applying common reduction strategies, such as data independence or symmetry reductions, the system is far too large for traditional model-checking. The state elements tend to be tightly interconnected, and natural decompositions often do not exist, or lead to a replication of the circuit structure in the specification. Circuit optimizations often take advantage of various restrictions that are expected to hold throughout the execution of the circuit, and in order to prove correct behaviour, one needs to first establish the validity of these restrictions. If a restriction depends on the correct behaviour of the circuit in a global level, local low level properties become contingent on correctness in a global level. In effect, either everything in the circuit works, or nothing does.

In the current paper we discuss a methodology that has allowed us to tackle these large verification problems with far greater success than before. Our starting point is a simple, concrete and computationally undemanding approach: We try to mechanically verify inductive invariants written by a human verifier, and conformance to a non-deterministic high-level model. The approach avoids any explicit automated computation of a fixed point. The concreteness of the computation steps in the approach allows a user to locate and analyze computational problems and devise a strategy around them when a tool fails because of capacity issues. This is a very common scenario in practice, and in our experience one of the key issues regarding the practical usability of a verification tool. On a philosophical level, we approach verification in much the same way as program construction, by emphasizing the role of the human verifier over automation.

Our methodology has gradually emerged over several years of work on large verification tasks. In the current paper we report on two cases: BUS recycle logic and a register renaming mechanism, both from a Pentium® 4 design. The BUS recycle mechanism contains about 3000 state elements, and with the methods presented here, the verification is a fairly straightforward task. The logic covered in the verification of the register renaming mechanism involves about 15000 state elements in the circuit and an environment model. The case is to our knowledge one of the largest and most complex circuit verification efforts in the field to date.

We consider the main contributions of the current paper to be the empirical observation that the methodology is an efficient strategy for practical verification, and the collection of heuristics and technical innovations used in our implication verification tool. The methodology scales smoothly to circuits with tens of thousands of state elements, and allows us to relate low-level RTL circuit models to algorithmically clear and concise high-level descriptions. Building the verification method on the intuitively very tangible idea of an invariant allows us to communicate the work easily to designers, and to draw on their insights in the work. A somewhat surprising observation in the work was just how difficult the computational task still is. Encountering this kind of complexity in a verification strategy that is heavily user guided leads us to believe that the chances of success for fully automatic methods on similar tasks are negligible.

2 Methodology Overview

2.1 Background

Let us assume that we have a circuit ckt and want to verify that a simple safety property I_{spec} holds throughout the execution of the circuit, under some external assumptions I_{ext} . The circuit models we use are effectively gate-level descriptions of the circuit functionality. They are mechanically translated from the RTL code used in the development project, linked to the actual silicon via schematics.

Let us write $Nodes$ for the set of node or signal names of the circuit, and define the *signature* Sig and the set of *valuations* Val of the circuit by $Sig \equiv_{df} Nodes \times int$ and $Val \equiv_{df} Sig \rightarrow bool$. We call the elements of Sig *timed signals*. Intuitively they are references to circuit nodes at specific times. The combinational logic and the state elements of the circuit naturally generate a set of *runs* of the circuit, $Runs \subseteq Val$, the definition of which we do not formalize here. Our circuit models do not have distinguished initial states, which means that the set $Runs$ is closed under suffixes. A circuit can be powered up in any state, and we use an explicit initialization sequence to drive it to a sufficiently well-defined state. The initialization sequence can be described by a collection of timed signals, a valuation assigning values to elements of these signals, and an initialization end time t_{init} . We write $Iruns$ for the set of all initialized runs of the circuit.

We formulate the specification I_{spec} and the external assumptions I_{ext} as implicitly conjuncted sets of *constraints*. Intuitively a constraint is a property characterizing a set of runs of the circuit, and formally we define the set of constraints $Constr$ by $Constr \equiv_{df} Val \rightarrow bool$. If $C \subseteq Constr$ and $v \in Val$, we define $C(v) \equiv_{df} \bigwedge_{c \in C} c(v)$. We also define a *next step* operation N for timed signals $(s, t) \in Sig$ by $N(s, t) \equiv_{df} (s, t + 1)$ and the notion extends naturally to signatures, valuations, constraints and constraint sets. We consider an invariant property I_{spec} to be valid over a circuit iff it is valid for all time points after the end of the initialization sequence for all initialized runs of the circuit.

More generally, we want to verify the conformance of the circuit behaviour against a non-deterministic high-level model (HLM). In this case, a specification consists of an HLM and a relation between RTL states and HLM states. The components of an HLM state form its signature Sig_{HLM} . We write Val_{HLM} for the set of valuations mapping each $s \in Sig_{HLM}$ to an element of an appropriate type, and describe the behaviour of the HLM by a set of initial state predicates $init_{HLM} \subseteq Val_{HLM} \rightarrow bool$ and a set of transition predicates $trans_{HLM} \subseteq Val_{HLM} \times Val_{HLM} \rightarrow bool$. Both sets are considered implicitly conjuncted. The relation between RTL and HLM states is described by an abstraction map $abs \in Val \rightarrow Val_{HLM}$. We consider the circuit behaviour to conform to the HLM iff for all initialized runs $v \in Iruns$,

- the RTL state at the end of the initialization sequence maps to an HLM state satisfying all the all the initial state predicates in $init_{HLM}$, and
- for all points n after the end of the initialization sequence, the RTL transition from point n to $n + 1$ maps to an HLM transition satisfying all the HLM transition predicates in $trans_{HLM}$.

2.2 Symbolic Simulation

Symbolic simulation is based on traditional notions of digital circuit simulation. In conventional symbolic simulation, the value of a signal is either a constant (T or F) or a symbolic expression representing the conditions under which the signal is T. To perform symbolic simulation on circuit RTL models, we use the technique of *symbolic trajectory evaluation (STE)* [20]. Trajectory evaluation extends the normal Boolean logic to a quaternary logic, with the value X denoting lack of information, i.e. the signal could be either T or F, and the value $\top$ denoting contradictory information, and carries out circuit simulation with quaternary values. In the current work we use STE to symbolically simulate the circuit and trace the values of relevant signals. In this context we can view STE as a routine that is given a circuit ckt , an antecedent signature and valuation $sig_{ant} \subseteq Sig$, $v_{ant} \in Val$, and a signature of interest $sig_{intr} \subseteq Sig$, and which produces a valuation $v_{STE} \in Val$ such that:

$$\forall v \in Runs. (\forall s \in sig_{ant}. v(s) = v_{ant}(s)) \Rightarrow (\forall s \in sig_{intr}. v(s) = v_{STE}(s)) \quad (1)$$

Technically our verification work is carried out in the Forte verification framework, built on top of the Voss system [10]. The interface language to Forte is FL, a strongly-typed functional language in the ML family [18]. It includes binary decision diagrams as first-class objects and symbolic trajectory evaluation as a built-in function. In writing the specification constraints I_{spec} and I_{ext} , we use all the facilities of the programming language FL. When describing an HLM, we use an abstract record-like FL data-type to characterize its signature and the intended types of the state components.

2.3 Inductive Verification

A simple way to verify the safety property I_{spec} is to strengthen it to an inductive invariant I_{ind} . Using a symbolic circuit simulator, the base and inductive steps can then be carried out as follows:

- symbolically simulate the circuit model from an unconstrained state, driving the circuit initialization sequence on the inputs of the circuit, and check that I_{ind} holds in the state after circuit initialization,
- symbolically simulate the circuit model from an unconstrained state for a single cycle, and check that if I_{ind} holds in the start state of the simulation, then it also holds in the end state, assuming that I_{ext} holds

In more detail, this approach involves the following verification steps:

- A** Determine the sub-circuit of interest, i.e. determine signatures sig_{inv} and sig_{ext} so that sig_{inv} contains all the timed signals referenced in I_{spec} and abs , and for every $s' \in N(sig_{inv})$, the value of s' in the circuit simulation is a function of timed signals in sig_{inv} and sig_{ext} . Define the signature of interest for all subsequent STE runs as the union of the two sets.
- B** Determine a set of constraints $I_{ind} \subseteq Constr$ such that $I_{spec} \subseteq I_{ind}$.
- C** Symbolically simulate the circuit with STE using the initialization signature and valuation as antecedent. Write $v_{STEinit}$ for the valuation computed by STE after the initialization sequence.

- D** Verify that after initialization, the inductive invariant I_{ind} holds, and the circuit state maps to an initial HLM state, i.e. that $I_{ind}(v_{STEinit})$ and $init_{HLM}(abs(v_{STEinit}))$ hold
- E** Symbolically simulate the circuit with STE using antecedent signature sig_{intr} and an antecedent valuation function that assigns a fresh symbolic variable to every element of sig_{intr} . Write v_{STE} for the valuation computed by STE.
- F** Verify that the inductive invariant remains true, and the circuit transition maps to a legal HLM transition:

$$(I_{ind} \cup I_{ext})(v_{STE}) \Rightarrow N(I_{ind})(v_{STE}) \quad (2)$$

$$(I_{ind} \cup I_{ext})(v_{STE}) \Rightarrow trans_{HLM}(abs(v_{STE}), abs(N(v_{STE}))) \quad (3)$$

Much of step A can be automated by causal fan-in analysis of signals, but human intuition is needed to determine which signals to include in the inductive set sig_{inv} and which to consider external inputs sig_{ext} in the scope of the proof. Step B is obviously the most labour-intensive task, and requires great ingenuity and detailed understanding of the circuit's behaviour. Typically a verifier builds I_{ind} incrementally by adding constraints to I_{spec} until the resulting set is inductive. Steps C and E are carried out automatically by STE, with a moderate amount of user intervention required to create a reasonable BDD variable ordering and to avoid simulating unnecessary parts of the circuit. Step D is usually trivial. Step F is the most computation intensive one. It is carried out with the implication verification tool discussed next. It is easy to see from equation 1 and the disjointness of the symbolic variables used in step E, that the verification goals for invariant and HLM conformance verification follow from the steps above. It is also easy to see that the method is complete in a theoretical sense: Since circuits are finite, one can in principle write down a constraint characterizing its reachable state space, and verify every valid I_{spec} with the help of this constraint.

3 Implication Verification

In verification of the inductive step, we need to determine the implication between two sets of constraints: Given $A, G \subseteq Constr$ and a valuation $v \in Val$, determine whether $A(v) \Rightarrow G(v)$, i.e. whether $\bigwedge_{a \in A} a(v) \Rightarrow \bigwedge_{g \in G} g(v)$. The problem is non-trivial, given that for some of our cases the sets have tens of thousands of elements, and the timed signals map to relatively complex BDD's in the right side of the implication.

For small instances, one can solve the problem by just computing $\mathcal{A} = \bigwedge_{a \in A} a(v)$, $\mathcal{G} = \bigwedge_{g \in G} g(v)$, and the implication $\mathcal{A} \Rightarrow \mathcal{G}$. In our setting, the naive solution is feasible only when the constraint sets have less than fifty elements. A simple improvement is to consider each goal separately, to compute $\mathcal{A} \Rightarrow g(v)$ for each $g \in G$. In our experience this strategy works when the constraint sets have up to a few hundred elements. To move forward we developed techniques which improve the obvious strategy primarily along two axes. First, instead of using all the assumptions $a \in A$, we can pick a selection of them and use the selected constraints incrementally in a particular order. Secondly, we can avoid the computation of complex BDD's by considering an exhaustive collection of special cases instead of a single general case, and by applying parametric substitutions.

Let us discuss assumption selection first. It is quite likely that some assumptions in A are more useful than others for verifying a goal $g(v)$. In practice we have found that a simple and useful strategy is to look for overlap in the support of BDD variables, and to pick assumptions sharing at least one variable with the goal. This heuristic is incomplete, but in our experience two or three iterations of assumption selection using the rule, each followed by the application of the selected assumptions to the goal, are very likely to pick all relevant assumptions. If we are unable to verify the goal at this stage, it is better to give up and give the user an opportunity to analyze the problem, rather than continue applying more assumptions, which will quickly cause a BDD blowup.

In which order should assumptions be applied? A heuristic we have found consistently useful is to look for assumptions that are as specific to the goal as possible. For example, if a goal talks about a particular data element in a table, assumptions about the same data element are preferable to assumptions about some other data element. To automate the heuristic, we statically classify BDD variables into a number of increasingly specific buckets, for example to reset/control/pointer/data variables, and prioritize assumptions based on the bucket the variables shared with the goal belong to. The more specific the bucket with the shared variables, the earlier the assumption should be used.

Selection, ordering and other similar heuristics can be fully automated, and they allow verification of many goals without user intervention. Nevertheless, the larger the assumption set is, the easier it is for a mechanism to pick useless assumptions, leading to BDD blow-ups. The verifier also often knows roughly what assumptions would be likely to contribute to a goal. After all, the verifier has written the constraints and typically has a reason for believing why they hold. Therefore it is beneficial to allow the verifier to customize a strategy for a goal. The degree of customization may vary. In some instances, the user may just want to guide the heuristics. In others, the user may want to completely control the strategy down to the level of an individual assumption.

Our second area for complexity reduction, i.e. case splitting and parametric substitutions, are needed because we encounter goals for which BDD's are incomputable within the limits of the current machines. For example, the BDD for a goal about table elements in a range determined by control bits and pointers reflects both the circuit logic for all the relevant entities and the calculation of the constraint itself. Case splitting also allows us to use different verification strategies for different cases. For example, the reason an invariant holds might depend on an operating mode, and one would like to write different verification strategies for different modes.

Parametric substitutions are a well-understood technique [13] with direct library support in Forte. The essence of the parametric representation is to encode a Boolean predicate P as a vector of Boolean functions whose range is exactly the set of truth assignments satisfying P . Then, if one wants to verify a statement of the type $P \Rightarrow Q$, one can instead verify the statement Q' obtained from Q by replacing all occurrences of variables of P in Q by the corresponding parametric representations. This allows us to evaluate a goal and required assumptions only in scenarios where the parameterized restriction holds, never evaluating the constraints in full generality.

To apply the various heuristics and human-guided techniques, a user needs a flexible way to direct the verification of a goal. We formulate the task as a specialized tableau system refining sequents $A \vdash_v G$, intuitively “ A implies goal G under valuation v ”, where

concr	$\frac{A \vdash_v g}{A \vdash_v [] \triangleright g(v)}$
choose (<i>sel</i>)	$\frac{A \vdash_v [] \triangleright g}{A \setminus \text{elems}(A_s) \vdash_v A_s \triangleright g} \quad \begin{array}{l} \text{where } A_s = \text{sel}(A, v, g) \\ \text{and } \text{elems}(A_s) \subseteq A \end{array}$
apply	$\frac{A \vdash_v [a_1, a_2, \dots, a_n] \triangleright g}{A \vdash_v [a_2, \dots, a_n] \triangleright (a_1(v) \Rightarrow g)}$
split (<i>S</i>)	$\frac{A \vdash_v g}{A \cup \{c_1\} \vdash_v g \dots A \cup \{c_n\} \vdash_v g} \quad \begin{array}{l} \text{where} \\ A \vdash_v (c_1 \vee' \dots \vee' c_n) \end{array} \quad \{c_1, \dots, c_n\} = S \subseteq \text{Constr}$
psplit (<i>S</i>)	$\frac{A \vdash_v g}{A \vdash_{v_1} g \dots A \vdash_{v_n} g} \quad \begin{array}{l} \text{where } \{c_1, \dots, c_n\} = S \subseteq \text{Constr} \\ \text{and } v_i = \lambda x. \text{param}(c_i(v))(v(x)) \end{array}$

where $[]$ is the empty list, $\text{sel}(A, v, g)$ is a function returning a list of constraints in A , $\text{elems}(A)$ the set of elements of list A , $a \vee' b \triangleq_{df} \lambda x. a(x) \vee b(x)$, and $\text{param}(c, b)$ for a function that computes a parametric substitution corresponding to $c \in \text{bool}$ and applies the substitution to $b \in \text{bool}$.

Fig. 1. Implication verification tableau rules

$A \subseteq \text{Constr}$ and $v \in \text{Val}$, and the right side G is either an uninstantiated goal $g \in \text{Constr}$, or an instantiated goal $[a_1, \dots, a_n] \triangleright g$, where $a_1, \dots, a_n \in \text{Constr}$ and $g \in \text{bool}$. The tableau rules are listed in Figure 1. We consider a leaf of a tableau *good*, if it is an instantiated sequent where the goal is true, i.e. of the type $A \vdash_v A_s \triangleright T$. It is easy to see that if there exists a tableau with root $A \vdash_v g$ and only good leaves, then $\bigwedge_{a \in A} a(v) \Rightarrow g(v)$.

To verify a goal, the user describes a strategy for building a tableau. Basic strategy steps describe case splits and selection functions for the *choose* rule. They can be combined with looping, sequencing and stop constructs. Trusted code then builds a tableau on the basis of the strategy, using the *apply* and *concr* rules automatically. The process provides constant feedback: which strategies are used, which assumptions are applied, what the BDD sizes are, etc. This allows the user to quickly focus on a problem, when one occurs. The code also stores aside intermediate results to facilitate counterexample analysis later on. In our experience, this approach to user interaction leads to a good combination of human guidance and automation. Many goals can be verified completely automatically with a heuristic default strategy, and user guidance can be added incrementally to overcome complexity barriers in more complex goals.

The tableau rules have evolved during a significant amount of practical work, and although they may look peculiar at first sight, they reflect key empirical learnings from that work. For example, we consciously keep the *choose* and *apply* rules separate, although a more elegant system could be obtained by replacing them with a single rule picking an assumption from A and applying it. The reason for this is that the selection process usually involves iteration over elements of A , which becomes a bottleneck if done too often, and so in practice it is better to select and order a collection of assumptions at once. Similarly, distinguishing uninstantiated and instantiated goals reflects the need to postpone the time when BDD evaluation happens.

4 BUS Recycle Logic

Recycle logic forms a key component of the Bus cluster in Pentium 4, the logical portion of the processor enabling communication with other components of a computer system (for an overview of Pentium 4 micro-architecture, see [11]). The Bus cluster communicates with the processor's Memory Execution cluster, which processes data loads and stores and instruction requests from the execution units, and passes requests to the Bus cluster as needed. The Bus cluster includes the Bus sequencing unit (BSU) which is the centralized transaction manager to handle all the transactions that communicate between the core and the External Bus Controller (EBC). The BSU consists of an arbiter unit and two queues: the Bus L1 Queue (BSL1Q) to track requests through the Level 1 cache pipelines and the Bus Sequencing Queue (BSQ) to manage transactions that need to go to the EBC or the Programmable Interrupt Controller.

All requests travel through the arbiter unit. If a request is granted, Level 1 Cache and the BSL1Q both receive it. If the Level 1 Cache satisfies the request, BSL1Q can drop the request after allocating it. Otherwise the request stays in the BSL1Q for some time and then moves to the BSQ when the BSQ has all the resources to accept it. For every granted cacheable request, the BUS recycle logic initiates an address match against all outstanding requests residing in the BSL1Q and the BSQ queues. If the incoming request has a conflict with an existing request in the queues or in the pipeline ahead of it, it is recycled and the issuing agent will need to reissue the request. For simplicity we can assume that there are two types of requests: READ and WRITE requests.

The recycle logic is intended to guarantee that no two requests with conflicts should reside in the BSL1Q or the BSQ. This is essential to avoid data corruption and to maintain the cache coherency. One such conflict is between two cacheable READ requests: no two cacheable READ requests with the same address should reside in the BSL1Q and BSQ. Let v_i , r_i and $addr_i$ stand for the signals containing the valid bit, cacheable read bit and the address vector in BSQ. We consider all the signals at the same moment, without relative timing offsets, and write just v_i for the timed signal $(v_i, 0)$. Let i and j be indices pointing to different entries in the BSQ, and define the constraint $R(i, j)$ as the function mapping a valuation v to:

$$v(v_i) \wedge v(r_i) \wedge v(v_j) \wedge v(r_j) \Rightarrow v(A_i) \neq v(A_j)$$

The specification I_{spec} consists of constraints $R(i, j)$ over all pairs of different entries in BSQ, similar constraints comparing all pairs of different BSL1Q entries, and a set of constraints comparing pairs with one element in BSQ and the other in BSL1Q.

Showing that I_{spec} is satisfied in the base case, after a global reset, is trivial, as reset clears all the valid bits. However, in general we have found it valuable to run the base case step early, since it is usually a computationally cheap and quick sanity check.

The inductive step fails immediately for I_{spec} for several reasons:

- I_{spec} does not contain any information on what needs to happen when a new request comes into the pipeline and its address matches with the addresses of existing entries in the BSL1Q and the BSQ.
- The recycle logic powers down when there is no request in the pipeline and no valid entry in the Bus queues, but I_{spec} contains no information about the power-up logic.

- I_{spec} does not capture any conditions involving requests moving from BSL1Q to BSQ. It also does not contain any information on the request-acknowledgement protocol between BSL1Q and BSQ.
- Address match computation is done over several cycles, and I_{spec} does not capture the relation of already computed partial information to the actual addresses.

We strengthened I_{spec} by adding constraints which capture the conditions mentioned above along with several others, and arrived at I_{ind} . This was an iterative process and required low level circuit knowledge. At each step of the iteration, new BDD variables were introduced and placed in the existing variable ordering based on the knowledge of the circuit. For example, variables related to the power-up logic were placed highest, those related to control logic next, and variables related to the address bits were interleaved and placed at the bottom. This provided a good initial order which was occasionally fine tuned. We reached an inductive I_{ind} after some fifty iterations of adding new constraints based on debugging the failure of the previous attempt.

The logic we verified consists of about 3000 state elements and is clearly beyond the capacity of model-checkers which compute reachable states of the circuit. The entire verification task took about three person months. The BDD complexity was not very high, peaking at about 10M BDD nodes, and the peak memory consumption during the entire verification session did not exceed 1M. The recycle logic had undergone intensive simulation-based validation and our verification did not uncover new bugs. However, we artificially introduced some high quality bugs found earlier into the design and were able to reproduce them with ease.

5 Register Renaming

The Intel NetBurst® micro-architecture of the Pentium 4 processor contains an out-of-order execution engine. Before reaching the engine, architecturally visible instructions have been translated to micro-operations (μ ops) in the Front End of the processor, and these μ ops have been sent to the out-of-order engine by the Micro-Sequencer (MS). The out-of-order engine consists of the Allocation, Renaming, and Scheduling functions. This part of the machine re-orders μ ops to allow them to execute as quickly as their input operands are ready. It can have up to 126 μ ops in flight at a time.

The register renaming logic renames the logical IA-32 registers such as EAX onto the processors 128-entry physical register file. This allows the small, 8-entry, architecturally defined IA-32 register file to be dynamically expanded to use the 128 physical registers in the Register File (RF) to remove false conflicts between μ ops. The renaming logic remembers the most current version of each register in the Register Alias Table (RAT) so that a new μ op coming down the pipeline can translate its logical sources (lsrsrcs) to physical sources (psrsrcs).

The Allocator logic allocates many of the machine resources needed by each μ op, and sends μ ops to scheduling and execution. During allocation, a sequence number is assigned to each μ op, indicating its relative age. The sequence number points to an entry in the Reorder Buffer (ROB) array, which tracks the completion status of the μ op. The Allocator allocates one of the 128 physical registers for the physical destination data (pdst) of the μ op. The Register File (RF) entry is allocated from a separate list

of available registers, known as the Trash Heap (TH), not sequentially like the ROB entries. The Allocator maintains a data structure called the Allocation Free List (ALF), effectively a patch list that keeps track of the binding of logical to physical destinations done at μ op allocation. Both ROB and ALF have 126 entries. The lists are filled in a round-robin fashion, with head and tail pointers. The head pointer points to the index of next μ op to be allocated, and the tail pointer to the index of the next μ op to be retired.

When the Allocator receives a new μ op with logical sources and destination, it grabs a free physical register from the Trash Heap (TH), updating TH in the process, associates the new pdst with the ldst of the μ op in RAT, stores the ldst and old and new pdst in ALF at head pointer location, moves head pointer to next location, and sends the μ op to scheduling with the psrscs and pdst. The μ ops are retired in-order by the ROB. When a μ op retires, the Allocator returns the old pdst of the retired μ op into Trash Heap.

Events are also detected and signalled by ROB at μ op retirement. When an event occurs, we need to restore RAT back to where it was when the eventing μ op was allocated. This is done on the basis of the information in the ALF: effectively one needs to undo the changes to RAT recorded in ALF, and to return the new pdsts of all the younger μ ops to the Trash Heap. In the same way, when a branch misprediction occurs, the RAT needs to be restored to the state where it was when the mispredicting μ op was allocated. For branch misprediction recovery, the Allocator interacts with the Jump Execution Unit (JEU) and the Checker-Replay unit (CRU), and maintains a data structure called Branch Tracking Buffer (BTB) keeping track of all branch μ ops in the machine, and whether they are correctly resolved.

The renaming protocol has several sources of complexity. For example, up to three μ ops are allocated or retired at a time, there can be multiple active branch mispredictions in different stages at the same time, and after a misprediction recovery, allocation starts without waiting for all the previous μ ops to retire. The RTL implementation of the protocol is highly optimized and consists of several tens of thousands of code lines. It also has a number of implementation-specific complications. For example, instead of one ALF head pointer, there are eight different versions of it, all with their own control logic, and in total, there are over forty pointers to ALF. Many one-step state updates of the abstract level also spread over multiple cycles in the implementation.

It is easy to come up with various expected properties of the protocol, e.g. that two logical registers should never be mapped to the same physical register in the RAT. However, to understand precisely what is expected of the register renaming logic and how it interacts with scheduling and retirement, we wanted to describe the out-of-order mechanism as a whole. We formalized a simple in-order execution model, a non-deterministic high-level model (HLM) of the out-of-order engine with register renaming, and sketched down an argument showing that the out-of-order model should produce the same results as the simple model, when one considers the stream of retiring μ ops. While both models were written down precisely, the argument connecting the two was not done with full rigour. The primary goal was to guarantee that our specification of the renaming protocol was based on a complete picture.

The actual target of our verification effort was to establish that the behaviours produced by the Allocator, including ALF, TH, BTB and RAT, are consistent with a high level model of the rename logic. The HLM has about 200 lines of code describing the

```

lettype mhs_t =
  MARBLE_HLM_STATE
  { h :: bv_widx_t }           // ALF head pointer
  { btb :: ( int -> bool ) }   // BTB array
  { brgoodv :: ( int -> bool ) } // input from CRU:: brgood valid
  { brgood :: ( int -> bv_widx_t ) } // input from CRU:: brgood uop
  ...
let head_trans_cond ( s, s' ) =
  ( do_alloc s ) => ( s':>h = s:>h +% 3 ) | ( s':>h = s:>h )
;
let do_good_idx i s =
  Exists_list [ gi | CHANNELS ] .
    ( s:>brgoodv gi ) AND ( widx2idx ( s:>brgood gi ) '= i )
;
let btb_trans_cond i ( s, s' ) =
  s':>btb i =
    ( ( do_alloc_idx i s )
      => ( s:>uopjmp ( idx2bank i ) )
      | ( ( do_good_idx i s ) => F | ( s:>btb i ) )
    );
...

```

Fig. 2. Parts of register renaming HLM

```

let btb_abs n2v t i =
  let alloc213      = n2v allocateokm213h t in
  let hr            = wrow2row ( head_th_old_wrow TH0 n2v t ) in
  let alloc213_i    = alloc213 AND ( hr '= idx2row i ) in
  let goodbrseq271 c = ptr2idx_bv ( V n2v ( goodbrseqnumCm271h c ) t ) in
  let goodbr271_i c = ( n2v ( crcgoodbrCm271h c ) t ) AND ( goodbrseq271 c '= i ) in
  // Is there a stage 213 write to this index?
  alloc213_i
  => // If so, the real values are waiting to be written into the FLM
    ( btb_alloc_213 n2v t i )
  | ( // Is there a stage 271 clear to this index?
    // If so, the real value is F, else the real values are in the BTB
    ( Exists_list [ c | CHANNELS ] . goodbr271_i c )
    => F | ( btb_raw n2v t i )
  );
...

```

Fig. 3. Parts of RTL-to-HLM abstraction mapping

```

let head_consistent_213_cond n2v t =
  let aok = n2v allocateokm213h t in
  let uopv u = n2v ( aluopvUm213h u ) t in
  aok = ( Exists_list [ u | UOPS ] . uopv u )
;
let mpred_m_igjeseq_btb_cons_cond ch d idx n2v t =
  let mbrclear = mjmp_clear_ch_d ch d n2v t in
  let igjeseq = widx2idx ( igjeseq_widx ( JUMP_BRCLEAR_DELAY + d ) ch n2v t ) in
  let btb_bit = btb_entry_idx TH0 n2v t in
  ( mbrclear AND ( igjeseq '= idx ) )
  ==>
  ( btb_bit = T );
...

```

Fig. 4. Some consistency invariants

transition relation. The HLM state is defined as an FL abstract data-type. Parts of the HLM are shown in Figure 2. They specify that the head pointer either stays put or moves by three steps, based on whether new μ ops are allocated, and that the BTB bit is set for branch μ ops at allocation and cleared based on an indication from CRU. In the verification, the HLM is compared against a full circuit model, combined with an abstract environment circuit. Figure 3 shows a part of the RTL-to-HLM abstraction mapping for a BTB entry. Since the RTL takes multiple cycles for a single-step HLM update, the abstraction function looks at intermediate values when the RTL is in the middle of an update.

We started the verification by determining the sets of timed signals for the invariant sig_{inv} and the proof input boundary sig_{ext} . Together the sets have about 15000 signals. We determined them initially manually, later with automated circuit traversal routines. The advantage of the manual process is that it forces the user to gain a detailed understanding of the RTL behaviour early. The work took several months. After the determination of signals, the simulation steps were quite easy and could be done in a matter of days. We used a relatively coarse static BDD ordering, with reset signals on top, followed by other control signals, interleaved pointer signal vectors and interleaved data signal vectors. The largest individual BDD's in STE simulation have only about 30k nodes, and the complete simulation uses less than 10M BDD nodes.

By far the hardest part of the verification was the determination of a strong enough inductive invariant I_{ind} . In the end, the invariant contains just over 20000 constraints, instantiated from 390 templates. The primary method for deriving the invariant was counterexample analysis. Constraints were added incrementally to prevent circuit behaviour that either failed to conform with the HLM or violated parts of the invariant introduced earlier. The precise content of the added constraints was based on human understanding of the intended circuit behaviour, aided by design annotations and by observing simulation behaviour. During the work, the verifier would typically add a collection of interdependent invariants related to some particular aspect of circuit functionality, e.g. allocation logic, at a time. In the intermediate stages of the work the tentative invariant would have quite accurate description of some functional aspects of circuit behaviour, but leave the yet uncovered aspects unconstrained. Figure 4 contains a few example constraints: a simple relation between some allocation control signals, and a requirement that the BTB bit corresponding to a mispredicted branch μ op must be set. The hardest systematic problem in the invariant determination was that some basic properties depend on complex global protocols, which makes it hard to build the invariants incrementally from bottom up. Another common theme is the don't-care space for an invariant. Typically it is easy to see that a certain relation is needed, but determining precisely when it is needed and when it holds can be challenging. The large majority of the invariants are quite natural. Exceptions are constraints that are needed for proper behaviour but hold unintentionally, such as implicit synchronization between different parts of design. In the process of invariant determination, the capability for rapid experimentation and the quick feedback provided by the Forte toolset were highly valuable.

For verification of the invariants we used the full arsenal of techniques discussed in Section 3. Most control invariants were verified automatically, and combinations of user-guided and heuristic strategies were used for pointer and data invariants. Case

splitting with parametric substitutions was particularly useful for invariants that relate a pointer and the data it points to, as it allowed us to consider each possible value in the pointer range separately. The largest individual BDD's in the verification had about 20M nodes, and the total usage peaked at about 100M nodes.

The verification took about two person-years of work, part of which was attributable to methodology development. Prior to the current effort, several person-years of work had gone into an attempt to verify the protocol using traditional temporal logic model checking and assume-guarantee reasoning. While achieving partial results, the effort looked unlikely to converge, which led to the adoption of the current approach. During the work we found two subtle bugs in the design. They had escaped dynamic testing and found their way into silicon, but did not cause functional failures in the current design as they were masked in the context of the whole processor. However, they constituted 'time bombs' that could have gone off on future proliferation products.

6 Conclusion

We have discussed a practical methodology which has allowed us to extend the scope of full formal verification to systems that are several magnitudes beyond the limits of current tools based on traditional temporal logic model checking [8]. In our experience the approach is widely applicable. In addition to the cases here, we have used it e.g. to verify bypass and cache behaviour. The approach is based on well-known techniques: BDD's [5] and STE [10]. In a sense, in building our strategy on human-produced invariants, we are going back to basics in verification methods [6, 9, 17].

The verification is computationally manageable, but requires a great deal of human guidance, and a detailed understanding of the design. On the other hand, in our experience any formal verification on a large design requires both of these, so the current approach is no different. Furthermore, the approach does automate a large portion of the task. We believe that completely user-guided verification, such as pure theorem proving, would be infeasible on designs of the size we are dealing with.

Another approach to induction-based verification is to use SAT instead of BDD's [21]. We carried out some experiments, and for many goals, the implication verification could be done automatically with common SAT engines, but for others, the engines failed to reach a conclusion. This leads us to believe that replacing BDD's with SAT is a plausible, maybe in some respects superior approach, but it will also require the creation of a methodology and a tool interface to allow a human to flexibly guide the tool around computational problems, analogous to the role of BDD tableaux used here. Fully automated SAT-checking can of course be used very effectively as a model exploration method on large designs [3, 7], but used in this way, it does not provide the kind of full coverage that in our opinion is one of the compelling advantages of formal verification.

There is a large body of work examining automated discovery of invariants [1, 2, 4, 12, 22]. Here we have concentrated on a simpler, nearly orthogonal task: given a collection of invariants, how to verify them. An automatic method for invariant discovery can of course help and is compatible with our approach, but does not remove the need for verification. Our work also has a number of similarities with formal equivalence verification [15, 16], but the distance between RTL and HLM is larger here.

Finally, given the amount of human effort required for full formal verification, does it make sense to pursue it at all? We believe it does for two reasons. First, full formal verification can provide far higher guarantees of correctness than any competing method. Secondly, it has been our experience that once a robust, practical solution to a verification problem has been found, the effort needed for similar future verification tasks falls dramatically.

Acknowledgements. I would like to thank Khurram Sajid for joint work on the BUS protocol, the numerous participants of the earlier register renaming verification efforts and Robert Jones for many useful discussions, and Tom Schubert for the opportunity to carry out this work.

References

1. S. Bensalem and Y. Lakhnech. Automatic generation of invariants. *Form. Methods Syst. Des.*, 15(1):75–92, 1999.
2. S. Bensalem, Y. Lakhnech, and S. Owre. InVeST: A tool for the verification of invariants. In *CAV '98, Proceedings*, pages 505–510. Springer-Verlag, 1998.
3. P. Bjesse, T. Leonard, and A. Mokkedem. Finding bugs in an alpha microprocessor using satisfiability solvers. In *CAV '01, Proceedings*, pages 454–464. Springer-Verlag, 2001.
4. N. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theor. Comput. Sci.*, 173(1):49–87, 1997.
5. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Comp.*, C-35(8):677–691, Aug. 1986.
6. R. Burstall. Program proving as hand simulation with a little induction. In *Information Processing*, pages 308 – 312. North Holland Publishing Company, aug 1974.
7. F. Copt, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi. Benefits of bounded model checking at an industrial setting. In *CAV '01*, pages 436–453. Springer.
8. J. Edmund M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 1999.
9. R. W. Floyd. Assigning meaning to programs. In *Proceedings of Symposium in Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967.
10. S. Hazelhurst and C.-J. H. Seger. Symbolic trajectory evaluation. In T. Kropf, editor, *Formal Hardware Verification*, chapter 1, pages 3–78. Springer Verlag; New York, 1997.
11. Hinton, G., Sager, D., Upton, M., Boggs, D., Carmean, D, Kyker, A. and Roussel, P. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, Q1, Feb. 2001.
12. R. Jeffords and C. Heitmeyer. Automatic generation of state invariants from requirements specifications. In *SIGSOFT '98/FSE-6, Proceedings*, pages 56–69. ACM Press, 1998.
13. R. B. Jones. *Symbolic Simulation Methods for Industrial Formal Verification*. Kluwer Academic Publishers, 2002.
14. R. Kaivola and K. Kohatsu. Proof engineering in the large: formal verification of Pentium 4 floating-point divider. *Int'l J. on Software Tools for Technology Transfer*, 4:323–334, 2003.
15. Z. Khasidashvili, J. Moondanos, D. Kaiss, and Z. Hanna. An enhanced cut-points algorithm in formal equivalence verification. In *HLDVT '01*, page 171. IEEE, 2001.
16. H. H. Kwak, I.-H. Moon, J. H. Kukula, and T. R. Shiple. Combinational equivalence checking through function transformation. In *ICCAD '02*, pages 526–533. ACM, 2002.
17. Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag New York, Inc., 1995.
18. L. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996.

19. T. Schubert. High level formal verification of next-generation microprocessors. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 1–6. ACM Press, 2003.
20. C.-J. H. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–189, Mar. 1995.
21. M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In *FMCAD '00*, pages 108–125. Springer-Verlag, 2000.
22. A. Tiwari, H. Rueß, H. Säidi, and N. Shankar. A technique for invariant generation. In *TACAS 2001*, pages 113–127. Springer-Verlag, 2001.