Replacing Testing with Formal Verification in Intel® Core™ i7 Processor Execution Engine Validation

Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, and Armaghan Naik

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

Abstract. Formal verification of arithmetic datapaths has been part of the established methodology for most Intel processor designs over the last years, usually in the role of supplementing more traditional coverage oriented testing activities. For the recent Intel® Core™ i7 design we took a step further and used formal verification as the primary validation vehicle for the core execution cluster, the component responsible for the functional behaviour of all microinstructions. We applied symbolic simulation based formal verification techniques for full datapath, control and state validation for the cluster, and dropped coverage driven testing entirely. The project, involving some twenty person years of verification work, is one of the most ambitious formal verification efforts in the hardware industry to date. Our experiences show that under the right circumstances, full formal verification of a design component is a feasible, industrially viable and competitive validation approach.

1 Introduction

Most Intel processors launched over the last ten years have contained formally verified components. This is hardly surprising, as their reliability is crucial, and the cost of correcting problems can be very high. Formal verification has been applied to a range of design components or features: low-level protocols, register renaming, arithmetic units, microarchitecture descriptions etc. [19,4]. In an industrial product development setting, formal verification is a tool, one among others, and it competes with traditional testing and simulation. Usually testing can produce initial results much faster than formal verification, and in our view the value of formal verification primarily comes from its ability to cover every possible behaviour. In most of the cases where formal verification has been applied, its role has been that of a supplementary verification method on top of a full-fledged simulation based dynamic validation effort.

The single most sustained formal verification effort has been made in the area of arithmetic, in particular floating point datapaths. In this area verification methods have reached sufficient maturity that they have now been routinely applied for a series of design projects [17,3,13,21,6], and expanded to cover the full datapath functionality of the Execution Cluster EXE, a top-level component of a core responsible for the functional behaviour of all microinstructions. In the current paper we discuss further expansion of this work on Intel® Core™ i7 design [1]. For this project, we used formal verification as the primary validation vehicle for the execution cluster, including full datapath, control and state validation, and dropped most usual RTL simulation and all coverage driven simulation validation for the cluster. To give a flavour of the magnitude of the work, Intel Core i7 design implements over 2700 distinct microinstructions.

Most of the particular verification techniques we applied are already documented in literature [3,12,13,15], and our goal here is to discuss the overall programme and the factors that allowed us to be succesful in mostly replacing testing by verification. We believe that the effort is an important step forward in the industrial application of formal verification, At the time of writing this paper, Intel Core i7 is Intel's flagship, top of the line processor. We used formal verification as the main pre-silicon validation vehicle for a large, crucial component of the design in the actual development project, providing results that were competitive with traditional testing-based methods in timeliness and validation cost, and at least comparable if not superior in quality - the execution cluster had the lowest number of issues escaping to silicon for any cluster of the design.

Our methodology has gradually emerged over several years of work on large verification tasks. On a philosophical level, we approach verification as program construction, by emphasizing the role of the human verifier over automation. Technically most of our work is based on symbolic simulation. This works particularly well for self-contained pipelines, such as processor execution units. In fact, the vast majority of execution units inside an Intel microprocessor can be completely verified with direct symbolic simulation. Direct symbolic simulation does not fare quite so well when the amount of interdependence between pipelines increase, as is typically the case in verification of control logic. To extend symbolic simulation to such feedback-intensive verification problems, we use inductive invariants written by a human verifier. The concreteness of the computation steps in the approach allows a verifier to locate and analyze computational problems and devise a strategy around them if and when capacity issues arise. 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. Building the verification method on the intuitively tangible ideas of symbolic simulation and invariants also allows us to communicate the work easily to designers, and to draw on their insights.

In the rest of the paper, we will first look briefly at Intel IA-32 processor structure, the execution cluster, Intel Core i7 design, and a typical processor design and validation flow. We will then outline execution cluster verification in past projects and Intel Core i7, and discuss the challenges, advantages and drawbacks of applying formal verification in a live development project. In Section 7, we will touch on the basic technologies enabling our work, and in Section 8 describe different aspects of the Intel Core i7 execution cluster verification effort: datapath, control and state verification.

2 Intel IA-32 Processor Structure

Intel IA-32 processor architecture has evolved gradually over the years. Typically a new IA-32 design project is intended to maintain functional backwards compatibility with the earlier designs while providing improvements along different axes: collections of new instructions (e.g. MMX™, SSE, SSE2 etc.), new capabilities (e.g. 64-bit address support, vPro™ technology), improved performance (clock frequency, throughput, power), or design adjustments to meet side conditions set by a new manufacturing process. Components from earlier designs are often reused in later projects, especially for proliferations but also for entirely new microarchitectures like NetBurst® or Core™. A typical single-core IA-32 processor consists of the following major design components called clusters [9]:

- The *front end cluster FEC* fetches and decodes architectural instructions, translates them to microinstructions and computes branch predictions. For example, a simple instruction with a memory source operand typically maps to a memory load microinstruction and a computation microinstruction, and a complex instruction, such as FCOS (Cosine) to a microprogram.
- The *out-of-order cluster OOO* receives streams of microinstructions from the front end, keeps track of dependencies between them, schedules ready-to-execute microinstructions for execution, takes care of branch misprediction and event recovery, retires completed instructions and updates architectural state.
- The *execution cluster EXE* carries out data computations for all microinstructions. The EXE cluster usually also performs memory address calculations and determines and signals branch mispredictions.
- The *memory cluster MEC* interacts with the front end and execution clusters when they need memory accesses, contains first levels of caches and interfaces with the external environment of the processor, e.g. the main memory or external bus.

In a multi-core design like Intel Core i7 [1], a single processor contains several cores and logic for communication and arbitration between different cores and the environment of the processor such as memory or external bus. Some of the logic that would exist in MEC in a single core design may be pushed to the logic outside the cores in a multi-core design. A register-transfer level (RTL) description of a cluster in System Verilog usually contains a few hundred thousand lines of code. While not a physical entity like the above, microcode is also a major design component, the complexity of which is comparable to that of the clusters.

In this paper we discuss the validation of the execution cluster EXE of the Intel Core i7 design. This cluster consists of the following units:

- The *integer execution unit IEU* contains logic for plain integer and miscellaneous other operations (e.g. control register access)
- The *SIMD integer unit SIU* (single instruction multiple data) contains logic for packed integer operations (MMX and SSE)
- The *floating point unit FPU* implements plain and packed floating point operations such as FDIV, FMUL, FADD etc.
- The *address generation unit AGU* performs address calculations and access checks for memory accesses as well as miscellaneous memory-related operations.
- The *jump execution unit JEU* implements jump operations and determines and signals branch mispredictions.
- The *memory interface unit MIU* receives load data from and passes store data to memory cluster, maintains store forwarding buffers, performs various datatype conversions, and takes care of data bypassing

Intel Core i7 EXE cluster implements over 2700 distinct microinstructions, and supports simultaneous multi-threading (SMT), which allows two independent threads to run simultaneously on the core.

3 Intel Processor Design and Validation Flow

A processor design project implementing a new microarchitecture typically involves a team of over 500 engineers over a time-line of 2 to 3 years [5]. The pre-silicon development effort can be divided into two roughly equal stages:

- *front end development*, which focuses on architecture, RTL and functionality, and
- *execution stage*, which focuses more on timing and physical design.

Pre-silicon development culminates in *tape-out*, the moment the design database is considered healthy enough to be sent to a fabrication plant for the first samples to be produced. After tape-out usually about 9-12 months of post-silicon development work is required to obtain a production-quality design [5].

Organizationally validation forms a separate organization within the product development team. Validation is an ongoing activity throughout the development effort. Pre-silicon validation starts at the same time as the design, and often design and validation are racing to add new design features and the infrastructure to validate them. Pre-silicon validation goes through three stages:

- *Design exercise* checks for basic functionality of design with 'easy' stimulus
- *Stress testing* checks for corner-case functionality by selections of 'hard' stimuli
- *Coverage testing* attempts to hit coverage goals through biased random stimulus

The goal of the pre-silicon validation effort is to tape out a product that is healthy enough to enable post-silicon development and validation. Typically the first silicon is able to boot an operating system and run at least some meaningful software content.

The standard approach for pre-silicon validation is register-transfer level (RTL) simulation. These dynamic validation activities take place at two levels of granularity:

- *Cluster simulation* concentrates on validating each cluster in isolation, in the context of a cluster test environment (CTE). CTE is a test bench approximating the interface of the rest of the design towards the cluster under test. Cluster simulation provides better controllability and is significantly faster than full chip simulation. It is the primary vehicle for pre-silicon validation.
- *Full chip simulation* targets the validation of the entire design. It is especially useful in analysis of multi-cluster protocols, and essentially checks whether the design faithfully implements the IA-32 architecture.

Both cluster and full chip simulation compare the observed behaviour of the design against a reference model. For full-chip simulation the reference model is an architecture level IA-32 simulator. For each cluster, the reference model is purpose-built - for the execution cluster EXE the reference model consists of a microcode simulator, which specifies the intended behaviour of each microinstruction, and a collection of ad-hoc checkers. The register-transfer level description is connected downwards to schematics with formal equivalence verification (FEV) tools.

The main workhorse of the pre-silicon dynamic validation effort is coverage-driven testing. Essentially in this testing methodology validators enumerate all different interesting scenarios they can think of for the design under test, and attempt to hit all of these by random or biased random stimulus, with the implicit intention that in the process they also hit most of the interesting scenarios they did not happen to think of. The methodology is very powerful in practice and results in remarkably clean designs, as long as interesting scenarios are carefully identified and the temptation to manually select the stimulus to hit a particular case avoided. However, in many cases no amount of testing provides certainty: a single dyadic extended-precision (80-bit) floating point instruction has 2^{160} possible source data combinations. Hitting coverage goals can also be very hard – usually for any given coverage goal, there is a point well below 100% after which additional coverage becomes exceedingly hard to gain.

After tape-out, post-silicon validation tries to identify any functional issues pre-silicon validation may have missed. It also serves at the ultimate reality check: not all electrical and physical phenomena can always be modelled accurately beforehand, and the actual behaviour of the circuit may diverge from the logical description. Testing the actual silicon instead of simulation makes any debug work much harder, as we lose visibility of and control over internal state. However, the silicon is much faster than any simulation. For example, a typical Core i7 pre-silicon full-chip simulation runs at 2-3Hz, whereas the launch frequency of the processor was 2.66GHz. This means that the total number of all pre-silicon simulation cycles on a large server farm amounts to no more than a few minutes of run time on a single actual processor.

4 Execution Cluster Verification – A Retrospective

While exploratory formal verification initiatives are carried out by members of Intel research laboratories, in the product development context formal verification is commonly done by a separate dedicated group within the validation team of the product development organization. In most projects, formal verification is viewed as a complementary activity alleviating possible shortcomings of dynamic validation in select target areas, with dynamic validation forming the backbone of the validation effort.

In a project, there is a spectrum of different usage models for formal verification. It can be used for a one-off effort for establishing the basic soundness of some particular feature, as a periodical activity validating and re-validating a design after major modifications, or as a part of regular regression suite preventing the introduction of faulty code into the design database. Most register transfer level verification work is done on models that are automatically compiled from RTL source code, essentially at gate level. The code, written by circuit designers, is usually highly optimized using knowledge about expected operating constraints, often to the degree that it is “almost wrong”. Formal verification has often little influence over design style or decisions.

The single most sustained formal verification programme within Intel has been carried out in the area of floating-point and arithmetic datapath verification. In fact, most Intel processors over the last ten years have had formally verified floating-point datapaths. The work discussed in the current paper builds on and extends this body of work.

The earliest concerted floating-point verification effort in Intel was carried out on the Pentium® Pro design [17]. The goal of the effort was full formal verification of all floating-point datapaths of the design, and it was done using the forte toolset, symbolic trajectory evaluation (STE) and a word-level model checking tool. The verification effort was essentially a one-off research project, establishing the basic feasibility of such verification of an industrial-scale design. After the original project, the verification code was regressed to verify the changes in proliferation projects.

The first processor design where formal verification of floating point and other arithmetic datapaths was done within the product development organization was the Pentium® 4 project. The effort was also based on the forte toolset and symbolic trajectory evaluation, but used mainstream BDD manipulation techniques, decompositions and theorem proving instead of word-level tools. Some of the verification efforts on the more complex operations, such as division and multiplication, are reported in [12,13,15] The verification code was maintained throughout the original Pentium 4 project and all its proliferations with regular, though occasionally infrequent regressions, and many parts of the code have lived on in other projects.

The Pentium 4 EXE formal verification effort consolidated some of our verification methodology and introduced a number of technical improvements. For the first time, formal verification was carried out in the level of the entire EXE cluster, instead of individual units. This turns out to be important in practice, as clusters are the lowest level of granularity for which there are clean, well-documented interfaces and on which dynamic validation works, which facilitates comparisons and reviews between formal and dynamic validation content. During the Pentium 4 project we also started to validate assumptions used in formal verification against test traces, by translating FV assumptions to DV checkers, first manually and later automatically.

The Intel® Core™ 2 processor design project extended the range of formal verification from floating-point and other high-complexity, high-risk datapaths to almost all datapaths in the EXE cluster [6]. This project was also the first to combine formal verification and dynamic validation efforts in a single team, and reduce the amount of testing in areas covered by the formal approach. As with Pentium 4, the effort was based on forte toolset and standard BDD manipulation utilities, and the verification code was maintained and regressed throughout the project and for proliferations.

5 Execution Cluster Verification for Intel Core i7

During the early stages of Intel Core i7 development, EXE cluster validation was separated into two teams, formal verification and dynamic validation, according to traditional lines. However, during the front end development stage, reflecting the success of the previous EXE cluster formal verification initiatives, a decision was reached to wind down the dynamic validation activities and reduce cluster test environment (CTE) development. Considering the usual stages of validation, dynamic validation was used extensively for the early design exercise, a moderate amount of stress testing was done, and coverage driven testing was dropped entirely. In certain areas where verification work was delayed, dynamic validation was used as a temporary back-up olution.

To account for the more prominent role of formal verification, the scope of the Intel Core i7 EXE formal verification effort was expanded to include full control, state and bypass verification in addition to the already standard datapath verification - see Section 8 for more discussion. We spent significant effort in software engineering the verification code in a way that would help code maintenance over a live, continuously evolving design and ease reusability for future projects. At the time of the design tape-out, all datapath and control verification was completed. Some bypass work and most of state verification took place after initial tape-out, prior to the design reaching production quality. At any given time, five to eight persons were working on the project, and the total amount of work was about twenty person years. The size of the team was comparable to a typical testing-based cluster validation team, perhaps slightly on the high side.

To provide timely feedback and to prevent bugs from slipping into the continuously changing design, existing verification code was routinely run on all new design models. Nightly regression runs consisting of a representative selection of verification sessions were responsible for catching most of the design issues, and weekly regressions carrying out a complete re-verification the rest. The computational effort to carry out the regressions was a fraction of the amount of cycles needed for a usual simulation-based regression suite. To check the soundness of external assumptions used in the formal verification activity, all assumptions were automatically translated to checkers that were piggybacking on full-chip RTL simulation test runs. This was an extremely useful activity and resulted in many assumption refinements.

The verification effort found a variety of issues in different aspects of the design. However, the success of validation is often measured not by what it finds but by what it misses, and it is probably instructive to look at the issues the verification effort failed to find. In the end, we missed three bugs that escaped to silicon and needed to be fixed prior to achieving production quality:

– When writeback of 64-bit MMX data from SIU is bypassed to a floating point store, an event was signaled incorrectly for negative infinity data.
– When two simultaneous broadcasts of cs.l bit (“Code Segment is in 64-bit mode”) happen on different threads, one of the broadcast values ends up being stored in a register for both threads, and the broadcast value for the other thread is dropped.
– The EXE cluster produces the #GP (General Protection) fault instead of #SS (Stack Segment) fault for descriptor loads that cross the canonical boundary.

The first of these escaped due to an incomplete formal specification for the floating point store consuming the bypassed data. In its original form it was treating the eventing condition as a don’t-care. The second problem was caused by formal verification work not having been done yet on the failing piece of logic before tape-out. In the third case, the EXE cluster correctly implemented a micro-architectural protocol between EXE and MEC, but the protocol itself failed to yield the expected architectural result. While not zero, this was the lowest number of bugs escaping to silicon for any cluster.

Furthermore, during the pre-silicon stage two other issues went undetected by the formal verification effort, but were caught in full chip testing:

– Packed floating point precision flag was incorrectly raised for precise unmasked underflows.
– Certain floating point constant ROM reads corrupted the result flags of a subsequent floating point compare operation.

The first of these was caused by both the RTL code and the formal specification inheriting material from an earlier project, and missing an intended design change between the projects, and the second by the fact that the control verification work that would have identified the unintended interference between the operations had not been done yet at the time. To summarize, out of the five misses, three could be attributed to an incorrect formal specification, and two to formal verification work not being completed early enough. The positive side of this is that there were no issues that would have fallen through the cracks because of failures in our methodology.

6 Formal Verification Value Proposition

The conventional wisdom about formal verification in industrial context is easy to spell out. Simulation yields partial results quickly and progresses reliably in a linear fashion, although reaching full coverage is very hard, and completeness unattainable. Formal verification, on the other hand, while in principle holding the promise of completeness, is in practice woefully capacity constrained and either slow or downright unable to produce meaningful results. Although a caricature, we feel this view is not altogether unjustified. To better understand the barriers of more wide-spread application of formal verification in industry, at least from an Intel perspective, let us look briefly at some possible application models for formal verification:

- FV may be applied to the fundamental algorithms,
- FV may be applied as an extra layer of protection,
- FV may be mixed with dynamic simulation on the same design, or
- FV may replace simulation as the primary validation approach.

In the first usage model, formal and dynamic validation do not directly overlap. Usually, dynamic validation cannot start until an implementation has been coded, and validation of the underlying algorithms is done only by inspection and reviews. Recent forays into such early microarchitecture validation in Intel [4] have been very encouraging.

As discussed above, much of Intel's formal verification work has historically followed the second usage model, where formal verification is done on top of a full dynamic validation effort. There are several pragmatic problems in this approach. First, if dynamic validation is done diligently, it will find most of the bugs, and thereby get most of the credit. Secondly, the few remaining bugs are likely to be in extreme corners of the design, and formal verification will look at these only if a very thorough and costly effort is made to cover all aspects of the design. This means that doing a little formal verification will not find any new issues, and doing a thorough effort only a few, in both cases leading to a perceived low return on investment. The areas where projects have routinely chosen to do formal verification have then been limited to those where an uncaught problem would be so visible and costly that the extra effort of doing formal verification can be justified. As a positive exception, SAT-based bounded model checking has been very successfully used as a bug-hunting tool in targeted areas.

The third usage model, mixing formal and dynamic techniques on validating a single design area, sounds appealing at face value. However, the following fundamental problem makes it hard to offset the dynamic validation effort by formal verification. The coverage-based validation paradigm is based on the identification of all interesting aspects of the design and the sets of interesting cases for all these aspects, with the expectation that once we gain coverage for most of these, we will have also exercised those aspects of the design we failed to account for in the process. Now, even if some aspects of the design are formally verified, we will still need to identify and gain coverage for them in order to give us confidence that we are also reaching the unaccounted parts of the design. Therefore, formal verification gives little or no reduction in simulation effort. A practical implication of this problem is also that it is hard to gradually offset dynamic validation by formal verification.

Consider then the fourth usage model, replacing dynamic simulation by formal verification. In our view, in to be successful, formal verification needs to

– work at the same level of design granularity as simulation
– address all the aspects of the design simulation does,
– relate to the surrounding simulation-based collateral, and
– provide timely feedback about the changing design.

The timeliness aspect merits some more discussion. A common complaint regarding formal verification in project context is “FV is usually late”. In many situations this is caused by the FV computational complexity problem simply being too hard. If verifiers need to first solve a research problem, it is little wonder that they are unable to produce quick feedback or put together meaningful schedules. In our opinion, the key aspect in alleviating this problem is a collection of “FV recipes”, tried-and-tested strategies for solving certain classes of problems. These allow the verifiers either to identify a computational strategy, or to flag a verification task as being of unknown complexity. A difficulty in producing timely feedback may also come just from lack of collateral. In our context, dynamic validation is usually able to draw on material from earlier projects. If formal verification needs to implement comparable material from scratch, it starts off with a handicap. While there may be good reasons for slow progress of FV work, we cannot see that it would step beyond a secondary role without timely results.

There were a number of factors enabling the effort discussed in the current paper. First of all, the choice of symbolic simulation as the primitive verification approach allowed us to work on cluster-size design objects. Secondly, symbolic simulation directly supported main datapath verification tasks, and combined with inductive invariants allowed us to address control and state verification, as well. Thirdly, the background of previous execution cluster verification projects had given us a wealth of experience, existing verification recipes and directly re-usable code, allowing us to progress quickly with many verification tasks. Fourthly, through the translation of FV assumption to simulation checkers, we were also able to relate our work to the existing dynamic validation collateral and get feedback from it.

The issues our validation effort missed can be traced back to failures to meet the requirements above. Two bugs were related to timeliness - our control and bypass verification efforts were running late, as we were hashing out methodology and doing the actual verification at the same time. The other three bugs were cases of validation against an incorrect specification. Mechanical checking of specifications against the simulation reference model would have likely identified at least two of these.

To be an effective verification engineer in our environment, one needs to have both an understanding of the design and the FV technologies. In our experience, the two systematically hardest problems the team members faced were the analysis of BDD behaviour in a computation, and the identification of hardware invariants – both of these skills are still something of a black art.

Looking forward, replicating the work reported here for subsequent projects has been considerably easier. The methodology and infrastructure for the complete effort is in place, the verification code is engineered to support reuse and be robust in the face of design changes, and a variety of strategies and guidelines have been obtained from practical experience. We are already seeing very effective early validation results from future projects - our hope is that the methodology will allow us to arrive at a logically clean design faster than before, and this way allow for faster design convergence.

We believe that the programme discussed in the current paper shows that in areas where a verifier can concentrate on verification, instead of solving verification research problems, the effort to carry out formal verification is comparable to thorough coverage-based validation. Such an effort is not easy, and existing verification collateral, in the form of verifier experience, reusable code or verification recipes, is likely to be needed to enable timely results. In areas where these circumstances exist, the choice of whether to do or not to do formal verification is in the end a risk tolerance question. On the one hand, formal verification can provide complete design coverage, on the other a formal verification based validation programme is going to involve more unknowns than a traditional testing based one.

7 Technical Framework

Technically our verification work is carried out in the Forte verification framework, originally built on top of the Voss system [8]. The interface language to Forte is *reFLect*, a lazy, strongly-typed functional language in the ML family [18]. Most of our verification code is written in reFLect: specifications, whether they are functional specifications or relational constraints, verification facilities, analysis routines etc. The execution of a verification task in our framework amounts to the evaluation of a reFLect program. Let us next briefly touch on a collection of aspects of the framework that have been key enablers for our work: symbolic evaluation of terms, symbolic trajectory evaluation, weakening techniques, parametric substitutions, relational STE and reflection.

Binary decision diagrams are first-class objects in the forte framework. In fact, in the reFLect language the type Bool includes not just the constants T and F, but arbitrary BDD's. For verification purposes, a very important feature of the language is that it allows *symbolic evaluation* of objects containing BDD's. For example, consider the following code:

```
let a = variable "a"; let b = variable "b"; let c = variable "c";
let moo = a => [ F, F, b ] | [ F, T, c ];
```

where 'variable' is a function that generates a BDD variable, $x \Rightarrow y|z$ means if-then-else, and $[\ldots]$ is used to build tuples. When evaluated, 'moo' yields $[F, \neg a, a \wedge b \vee \neg a \wedge c]$. The symbolic evaluation capability allows us to use arbitrary reFLect code when writing specifications and then use the evaluation mechanism of the language for determining satisfaction of a specification for all possible assignments to symbolic variables.

The forte framework directly supports *symbolic simulation* on circuit models through symbolic trajectory evaluation (STE) [20] as a built-in function. Symbolic simulation is based on traditional notions of digital circuit simulation, but the value of a signal in simulation can either by a constant (T or F) or a symbolic expression representing the conditions under which the signal is T. 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. It carries out circuit simulation with quaternary values starting from a maximally unconstrained start state. In the current work we use STE to symbolically simulate the circuit and trace the values of relevant signals. A single STE simulation is best viewed as an over-approximation of the class of all actual Boolean traces of the circuit agreeing with the stimuli driving the STE simulation: If we manage to verify a Boolean property on the STE simulation, we can deduce that it also holds for all circuit traces.

The computational effort required for an STE simulation is often reduced by the technique of *weakening*, in which the simulated value of a given circuit node at a given time is replaced with the undefined value X. In explicit weakening, the user manually defines the weakening points, and in dynamic weakening any BDD that is larger than a user given threshold is automatically replaced with the undefined value X. We also use more sophisticated automated techniques using causal fan-in information from a circuit trace to determine weakening points. The automated weakening techniques solve most circuit simulation capacity problems without need for human intervention.

Parametric substitution is a technique for reducing symbolic evaluation complexity when we are interested in the result of the evaluation only under a given set of constraints [10]. It is an algorithm that takes a Boolean condition A, and computes a substitution list $v/B = [(v_1,B_1),(v_2,B_2),\ldots]$, which associates each variable v_i occurring in A with a BDD B_i on a set of fresh variables such that the range of the functions B_i is exactly the range of assignments to v_i's that satisfy A. For example, if we want to evaluate an implication $A \Rightarrow C$, we can compute the substitution v/B, apply it to C, and check whether $C' = T$ for the resulting C'. In general, parametric substitution allow us to evaluate a term only in scenarios where the parametrized constraint holds: If C contains a subterm $A => D|E$, in C' the corresponding term will be $T => D'|E'$, and we will never need to evaluate E'. We often use parametric substitutions together with case splitting to bring down the complexity of a problem C by decomposing it into a number of cases $A_1,\ldots,A_n$, and then consider each case A_i separately, parameterizing A_i to ease the computation of C.

In the work discussed in the current paper, we access STE through a layer called *relational STE* or rSTE, a package built around STE to support relational specifications. Effectively, rSTE is a tool allowing us to check whether one list of constraints ("the input constraints"), implies another list of constraints ("the output constraints") over all traces of the circuit. Most common computational complexity reduction techniques, including weakening, parametric substitution etc. are made easily accessible to the user as rSTE options. It also provides sophisticated debug support, breakpoints etc. to enable users to quickly focus on usual verification problems. For the verification of the implication between input and output constraints, we use the tool discussed in [11].

Finally, the *reflection* mechanism allows terms in the reFLect language to be used as objects in the language itself [7]. We use it for sanity check traversals, and to reason about proof decompositions in the theorem prover Goaled [16].

8 Intel Core i7 EXE Verification Effort

8.1 High-Level EXE Model

The structure of our verification work is motivated by a generic abstract model of the execution cluster EXE. This model has a set of *abstract microinstructions*. These may be executed in EXE through a number of *schedule ports*. An instruction scheduled on a particular port will execute in an *execution unit*, and write the result back through a *writeback port* after a *latency*, which depends on the instruction. The set of implemented instructions, the collection of schedule and writeback ports and execution units and the mapping of instructions to ports, execution units and latencies is left open at the level of the generic model and is fixed by every individual design. The abstract cluster model has a number of *state components*, which an instruction may read or update synchronously. Our abstract EXE model does not model memory or caches, allowing us to avoid all the related intricacies. In a real design, EXE accesses memory through the memory cluster MEC, but for the purposes of EXE verification, it suffices for us to model the interactions at the EXE-MEC interface only: the load and store addresses and store data EXE sends to, and the load data EXE receives from MEC.

Much of our work on building the Cluster Verification Environment (CVE) for EXE is proof engineering [13] and software engineering to create a standard, uniform methodology for writing specifications and carrying out verification tasks. The aim of the effort is to support reuse and code maintenance over a constantly changing design, and separate common and project-specific parts to allow shared code to be written only once. We use reFLect user-defined record-like datatypes to enforce structure. For example, all functional micro-instruction specifications are mappings from an abstract type of source data to an abstract type of writeback data. The CVE collects all verification code to a single common directory structure.

8.2 Datapath Verification

Datapath verification is the most important part of our programme. Much of the effort is related to formally specifying the intended behaviour of the over 2700 individual microinstructions. On the one hand, this is a formidable task - the informal specifications of IA-32 architectural instructions take two volumes [2]. On the other, the existence of written specifications as a starting point was very helpful. The formal specifications taking most effort were typically those for auxiliary micro-architectural operations without direct architectural counterparts, which were often lacking in precise documentation. The largest systematic problem in microinstruction specification was the determination of don't-care spaces. Often the architectural specification might explicitly leave an aspect undefined, but the de-facto micro-architectural specification required for backwards compatibility would be stricter. Particular complications included:

- IEEE floating point arithmetic with micro-architectural variations in FPU
- the microarchitecture mixing with the architecture complicates the IA-32 memory addressing mechanism in AGU, already non-trivial due to its gradual evolution over a long history

- mispredicted branch signalling and recovery protocol interacts with the jump operation datapath in JEU
- mispredicted branch and event recovery protocols interact with control register operations datapath in IEU
- some of MIU functionality is intermingled with the bypass network, which made it hard to determine clean verification task boundaries

In the time-frame of the Core i7 work, the formal specifications of the micro-operations were essentially stand-alone, and some of the missed bugs were caused by incorrect specifications. To reduce this risk, we are currently examining ways to more closely link the formal specifications with the C++ specifications used by dynamic validation.

We verified separately for each micro-operation *op* and for each port *p* on which the operation *op* may be scheduled that the following holds for all circuit traces:

$$D(p,op) \wedge DE(p,op) \wedge DI(p,op) \Rightarrow (wb(p) = spec(op,src(p)))$$

Where *D*, *DE* and *DI* are sets of basic constraints, environment constraints and internal constraints for operation *op* on port *p*, respectively, *src* and *wb* refer to the source data and write-back results in the circuit trace, and *spec* is the formal specification for the operation *op*. In reality the specification target is not strict equality, but may include partially undefined don't-care results etc. In the actual verification work we would routinely refine the specification and discover missing environment assumptions, augmenting *DE* and *DI*, until either the verification succeeded, or we had identified a design issue. We also translated all environment assumptions *DE* and *DI* to simulation checkers that were routinely run on full-chip simulation traces. This activity was extremely useful in weeding out incorrect or overly restrictive assumptions.

Most micro-operations could be verified with direct symbolic simulation using a reasonably straightforward variable ordering, with the following exceptions:

- The FADD family of operations uses a case split and parametric substitution strategy, as discussed in [3,21].
- The FMUL, IMUL and PMUL families of multiplication operations need a sequential decomposition as in [15,21].
- The FDIV, FSQRT and IDIV families of operations need an iterative sequential decomposition as in [12,13].
- Intel Core i7 added a collection of SSE4 instructions for accelerated string and text processing, e.g. for faster XML parsing. The verification of these require advanced parametric substitution strategies and sequential decomposition.
- The PSADBW and MPSADBW operations (sum of absolute differences) require a case split and parametric substitution strategy.
- Most AGU operations require the use of symbolic indexing to deal with complexity caused by segment register file (SRF) reads.

For a self-contained example of a related datapath verification task, see [22].

8.3 Control and Bypass Verification

The datapath verification above uses collections of internal constraints *DI* as assumptions. The first goal of the control verification is to establish these, by strengthening them to an inductive invariant I, and showing for every operation op and relevant schedule port p:

$$D(p,op) \wedge DE(p,op) \wedge I \wedge IE \Rightarrow DI(o,op)$$

Here D and DE are sets of basic and external constraints for op, I is the global control invariant for the design, and IE is an external control invariant on circuit inputs. The second goal is then to show that I indeed is an inductive invariant, i.e. that I holds at the end of the circuit initialization sequence and is inductive. The techniques used to establish this are discussed in [11,14].

The global control invariant for the Core i7 EXE cluster contains roughly 800 individual components. This large number reflects the aggressive clock gating the design uses to conserve power. Many internal circuit restrictions that would automatically propagate from an existing environment assumption were all circuit clocks toggling continuously will need to be established via an inductive invariant when we do not know whether all the relevant clocks will toggle or not.

In the datapath verification, we usually sample source data at the inputs of the execution unit where the microinstruction is executed. The task of the *bypass verification* is then to show that the data at the execution unit inputs is either the properly bypassed result data of an earlier operation, or the value received by the cluster from a register file read, depending on the bypass control inputs. The primary challenges in the bypass verification were the identification of the relevant micro-architectural invariants, and the management of the sheer number of different possible bypass scenarios.

8.4 State Verification

The abstract EXE model has a number of state components. The primary register file does not reside inside the cluster in Core i7, but there is a variety of auxiliary registers, for example the segment register file, floating point control word file etc. In CVE, each state component is annotated with a specification of which instructions are allowed to read or update it, and under which constraints. The verification of an update to a state component by an instruction is done in the context of the datapath proof for that instruction. The inverse, verification of the claim that the value of a state component does not change when there is no updating instruction is done as a separate verification task we call *state stability proof*. The validation of these typically involves new control invariant clauses added to the global invariant I.

In the abstract EXE model, all access to state data is synchronous. All instructions reading a state component do so at a fixed offset relative to their start time, and if they update the state components, the update takes place exactly one cycle after the read so that any subsequent operation reading the component will get the updated value. In reality, an operation often takes several cycles longer to update a state component than to read it. However, to guarantee consistency, we require that when an update happens, no reader should access the state component before the value is actually updated. We call the verification of this restriction the *state sequential consistency proof*. Usually the restriction follows from external scheduling constraints. The necessity of articulating these constraints explicitly to carry out the sequential consistency proof has the upside that the constraints can then be translated to simulation checkers and used to identify scheduling violations, for example due to insufficient synchronization in microcode.

Acknowledgments

We would like to thank Tom Schubert, Blair Milburn and Bob Bentley for the opportunity to carry out this work, Jim Grundy and Eran Talmor for reFLect support and enhancements throughout the project, John O'Leary for Goaled support, Brad Kelly and Evgeny Semenov for their CTE work, and Michael Bair and Kyle Dirks for their healthy scepticism towards the initiative.

References

1. First the Tick, Now the Tock: Next Generation Intel® Microarchitecture (Nehalem) Intel Corp., `http://www.intel.com/technology/architecture-silicon/next-gen/whitepaper.pdf`
2. IA-32 Intel® Architecture Software Developer's Manual, Vol. 2A and 2B. Intel Corp.
3. Aagaard, M.D., Jones, R.B., Melhan, T.F., O'Leary, J.W., Seger, C.-J.H.: A methodology for large-scale hardware verification. In: Johnson, S.D., Hunt Jr., W.A. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 263–282. Springer, Heidelberg (2000)
4. Beers, R.: Pre-RTL formal verification: an Intel experience. In: DAC 2008: Proc. of the 45th annual conf. on Design automation, pp. 806–811. ACM, New York (2008)
5. Bentley, B.: Validating a modern microprocessor. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 2–4. Springer, Heidelberg (2005)
6. Flaisher, A., Gluska, A., Singerman, E.: Case study: Integrating FV and DV in the verification of the Intel Core™2 Duo microprocessor. In: FMCAD, Formal Methods in Computer-Aided Design, pp. 192–195 (2007)
7. Grundy, J., Melhan, T., O'Leary, J.: A reflective functional language for hardware design and theorem proving. Journal of Functional Programming 16(2), 157–196 (2006)
8. Hazelhurst, S., Seger, C.-J.H.: Symbolic trajectory evaluation. In: Kropf, T. (ed.) Formal Hardware Verification. LNCS, vol. 1287, pp. 3–78. Springer, Heidelberg (1997)
9. Hinton, G., Sager, D., Upton, M., Boggs, D., Carmean, D., Kyker, A., Roussel, P.: The microarchitecture of the Pentium® 4 processor. Intel. Technology Journal Q1 (February 2001)
10. Jones, R.B.: Symbolic Simulation Methods for Industrial Formal Verification. Kluwer Academic Publishers, Dordrecht (2002)
11. Kaivola, R.: Formal verification of Pentium® 4 components with symbolic simulation and inductive invariants. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 170–184. Springer, Heidelberg (2005)
12. Kaivola, R., Aagaard, M.D.: Divider circuit verification with model checking and theorem proving. In: Aagaard, M.D., Harrison, J. (eds.) TPHOLs 2000. LNCS, vol. 1869, pp. 338–355. Springer, Heidelberg (2000)
13. Kaivola, R., Kohatsu, K.: 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)
14. Kaivola, R., Naik, A.: Formal verification of high-level conformance with symbolic simulation. In: HLDVT, High-Level Design Validation and Test, pp. 153–159 (2005)
15. Kaivola, R., Narasimhan, N.: Formal verification of the Pentium® 4 floating-point multiplier. In: DATE, Design, Automation and Test in Europe, pp. 20–27 (2002)
16. O'Leary, J.: Using a reflective functional language for hardware verification and theorem proving. In: Third Workshop on Applied Semantics (APPSEM 2005), September 12–15, 2005, pp. 12–15 (2005)

17. O'Leary, J.W., Zhao, X., Gerth, R., Seger, C.-J.H.: Formally verifying IEEE compliance of floating-point hardware. Intel. Technology Journal Q1 (Feburary 1999)
18. Paulson, L.: ML for the Working Programmer. Cambridge University Press, Cambridge (1996)
19. Schubert, T.: High level formal verification of next-generation microprocessors. In: DAC 2003: Proceedings of the 40th conference on Design automation, pp. 1–6. ACM Press, New York (2003)
20. Seger, C.-J.H., Bryant, R.E.: Formal verification by symbolic evaluation of partially-ordered trajectories. Formal Methods in System Design 6(2), 147–189 (1995)
21. Slobodova, A.: Challenges for formal verification in industrial setting. In: Brim, L., Haverkort, B.R., Leucker, M., van de Pol, J. (eds.) FMICS 2006 and PDMC 2006. LNCS, vol. 4346, pp. 1–22. Springer, Heidelberg (2007)
22. Slobodova, A.: Formal verification of hardware support for advanced encryption standard. In: FMCAD, Formal Methods in Computer-Aided Design, pp. 61–64 (2008)