End-to-End Verification of ARM® Processors with ISA-Formal

Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes,
Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi

ARM Limited
110 Fulbourn Road
Cambridge, UK
first.last@arm.com

Abstract. Despite 20+ years of research on processor verification, it remains hard to use formal verification techniques in commercial processor development. There are two significant factors: scaling issues and return on investment. The *scaling issues* include the size of modern processor specifications, the size/complexity of processor designs, the size of design/verification teams and the (non)availability of enough formal verification experts. The *return on investment* issues include the need to start catching bugs early in development, the need to continue catching bugs throughout development, and the need to be able to reuse verification IP, tools and techniques across a wide range of design styles.

This paper describes how ARM has overcome these issues in our Instruction Set Architecture Formal Verification framework "ISA-Formal." This is an end-to-end framework to detect bugs in the datapath, pipeline control and forwarding/stall logic of processors. A key part of making the approach scale is use of a mechanical translation of ARM's Architecture Reference Manuals to Verilog allowing the use of commercial model-checkers. ISA-Formal has proven especially effective at finding micro-architecture specific bugs involving complex sequences of instructions.

An essential feature of our work is that it is able to scale all the way from simple 3-stage microcontrollers, through superscalar in-order processors up to out-of-order processors. We have applied this method to 8 different ARM processors spanning all stages of development up to release. In all processors, this has found bugs that would have been hard for conventional simulation-based verification to find and ISA-Formal is now a key part of ARM's formal verification strategy.

To the best of our knowledge, this is the most broadly applicable formal verification technique for verifying processor pipeline control in mainstream commercial use.

1 Introduction

Modern microprocessor designs apply many optimizations to improve performance: pipelining, forwarding, issuing multiple instructions per cycle, multiple independent pipelines, out-of-order instruction completion, out-of-order instruction issue, etc. All of these optimizations are supposed to be invisible to the

programmer in a uniprocessor context: the overall effect should be the same as executing instructions one at a time in program order. But each of these optimizations introduces corner cases that potentially change the behaviour and the different optimizations interact with each other in complex ways.

For example, in a pre-release version of one of ARM's dual-issue processors, there was a defect in the inter-pipeline forwarding control logic that resulted in an instruction reading its input value from the wrong place if the instruction was preceded by a conditional instruction whose condition did not hold (and whose results should therefore not be used as inputs). The shortest instruction sequence which could demonstrate this defect was 5 instructions long. The particular set of instructions that could trigger the defect was fairly narrow because it was necessary that the instructions used particular parts of the pipeline, and the instruction sequence had to be aligned such that the first of these instructions executed in pipeline 0.

For traditional simulation-based verification to detect this defect you would need a detailed understanding of the micro-architecture of that particular processor, of the corner cases caused by the forwarding paths and of the kinds of errors one is likely to make in implementing forwarding control logic. Creating such tests is not only hard and unreliable, but it is also expensive because the tests would be specific to the particular micro-architectural choices in a processor and different tests must be created for each processor.

This paper describes the "ISA-Formal" verification technique that we have developed at ARM for verifying that processors correctly implement the Instruction Set Architecture (ISA) part of the architecture specification. Our method uses bounded model checking to explore different sequences of instructions and was able to detect the above defect prior to release of the RTL to manufacturers.

The *effectiveness* of ISA-Formal is important to its adoption within ARM but it is not the most important requirement we had to satisfy in order to make formal verification a useful part of ARM's processor development flow. Before ISA-Formal could be deployed widely within ARM, we had make it work within the constraints of commercial processor development:

(1) Processor development takes a long time (2 years or more) and it is important to be able to be able to detect bugs at all stages of processor development. We have applied ISA-Formal all the way from incomplete designs that still contain bugs through to complete, heavily tested designs.

(2) Verifying a processor takes longer than design: the long tail of processor development is developing new tests for the processor and fixing any bugs. It is important that useful results can be obtained even in the early stages of verification — before the complete test infrastructure has been developed. ISA-Formal is able to find bugs involving instructions for which we do not have a specification; all we need is a specification of any instruction whose result could be affected by the bug.

(3) Verification teams work in parallel with design teams so it is important that verification teams are able to continue searching for new bugs even when there are multiple outstanding bugs waiting to be fixed. Some bugs can take

months to be fixed if they are not critical to immediate project milestones. ISA-Formal is able to work round known bugs in the processor.

(4) Any verification technique requires significant investment so reusability not only of the technique but also of the infrastructure is critical. We are able to reuse the tools across ARM v8-A/R (Application/Real-time) class and across v8-M (Microcontroller) class processors. The only part that needs to be customized for each processor is the Verilog abstraction function that extracts the effective architectural state from the micro-architectural state of a processor. This portability has been a great benefit while developing the technique because it allowed several processor teams to pool resources: one team worked on how to verify floating point instructions while another worked on branches and another worked on load-store instructions.

(5) Modern processor architectures and modern processors are large: the ARM v8-A ISA specification is over 2500 pages long, the v7-M ISA specification is over 600 pages long (almost half the length of the entire specification). It is important that verification techniques scale both in terms of human effort and computing resources. We have written a tool to automatically translate the source of the ARM Architecture Specifications to Verilog; and we split the verification task into thousands of small properties allowing effective use of large compute clusters.

We demonstrated these properties in three small-scale trials on different processors and have since refined and applied the technique on five further ARM processors: checking almost the complete instruction set architecture of these processors ranging from simple 3-stage microcontrollers up to sophisticated 64-bit out-of-order processors. ISA-Formal is now a key part of ARM's formal verification strategy.

We characterise our approach as "end-to-end verification" because it focusses on directly verifying the path from instruction decode through to instruction retire against the architectural specification in contrast to hierarchical or block-level verification which focusses on verifying individual blocks against micro-architectural specifications and then verifying that the composition of those blocks meets the overall specification.

ISA-Formal is strongly based on techniques developed in the academic community; our contribution is a description of the techniques needed to make it scale and of the challenges and solutions in creating a portable approach which can be applied in a commercial setting to a wide range of processor micro-architectures.

The remainder of this paper is structured as follows: Sect. 2 discusses related work; Sect. 3 illustrates the basic idea, demonstrating how ISA-Formal can be applied manually, to a single instruction and discusses the kinds of bugs it was able to discover in real processors; Sect. 4 describes how we scaled this idea up to handle full ISA specifications; Sect. 5 describes adaptations to handle a variety of different micro-architectural features; Sect. 6 reports on the results of applying this method to multiple processors; Sect. 7 concludes.

2 Related Work

Our work builds heavily on the pioneering work from the '90's such as Burch-Dill's automatic verification based on flushing refinements [5] and Srinivasan's verification based on completion refinements [19]. These and many other works used different notions of correctness of which Aagard et al. [2,1] give a useful taxonomy and establish conditions under which different notions of correctness are equivalent.

Our approach focusses on verifying RTL (Verilog) in contrast to work which verifies a high-level model of the microarchitecture design against a specification. For example, Lahiri et al. [14] verified the microarchitecture of the M*-core processor core (an early RISC-style architecture) and [13] verified the microarchitecture for an out-of-order processor through a series of successive refinements but neither verified against the RTL of an actual processor. In our experience, most errors are introduced while translating the microarchitecture design into RTL and during subsequent optimisation so verifying before RTL misses a lot of bugs. The challenge of verifying actual RTL is that it makes it hard to use abstraction techniques such as using uninterpreted functions because the actual RTL of an efficient processor tends not to have convenient blocks which match directly with parts of the original specification.

Many approaches to verifying pipeline control logic have used theorem proving techniques to tackle the difficult problems of handling pipeline forwarding and hazards in in-order processors [21,12] and, later, for out-of-order processors [8,16,7,9]. Theorem proving techniques are powerful and tend to suffer less machine-scaling issues than more automated techniques but their reliance on verification experts leads to severe human-scaling issues: it is hard to hire enough experts. We prefer to ride Moore's law and use more CPU-intensive but more automatable approaches.

There has been considerable commercial interest recently in formal verification of floating point units such as Kaivola et al. [10], KiranKumar [11] and Slobodova et. al [18]. This is impressive and important work but essentially orthogonal to our own: while it tackles the scaling issues that occur when verifying commercial processors, it focusses on individual blocks processing a single instruction with relatively simple input-output signals while our approach focusses on the entire pipeline and especially the control logic to handle interactions between instructions. We describe how we deal with verification of pipelines containing floating point units in Sect. 5.1.

3 Illustration: Hand-written properties

The basic approach to verification that we use in ISA-Formal is based on the above prior work. We start with the processor in a simple, well-defined state $uArch_0$ with no instructions in the pipeline. We then execute for a number of cycles where each cycle may issue an instruction. This serves to put the processor into a more complex state where hazards, forwarding, etc. can occur. And finally,

we execute an instruction I_n and test whether the instruction executes correctly. This is done by applying an abstraction function *abs* which extracts the architectural state of the processor immediately before I_n executes and immediately after I_n executes. We do not flush the pipe before or after I_n.

$$uArch_0 \xrightarrow{I_1} uArch_1 \dashrightarrow uArch_{n-1} \xrightarrow{I_n} uArch_n$$
$$\downarrow{abs} \qquad\qquad \downarrow{abs}$$
$$Arch_{n-1} \xrightarrow{I_n} Arch_n$$

A key part of making this scalable is that, instead of allowing the formal verification tool to choose any instruction for I_n, we enumerate all the instruction classes supported by the architecture and perform a separate check for each instruction class. Proving these simpler results is helpful early in processor development by making it easy to focus on checking the currently implemented instructions. Later in development, the pattern of failing instructions is a useful guide in localizing the fault: if all branch instructions are failing, there is no need to worry about bugs in the ALU. And as the size of the verification task scales up, splitting the verification task into many small properties lets us make more effective use of our verification cluster which is optimized for running many independent processes across hundreds of machines.

To make this more concrete, consider the task of checking an addition instruction in the classic 5-stage pipeline illustrated in Fig. 1. This consists of 5 pipeline stages responsible for instruction fetch (IF), decode (ID), execute (EX), memory access (MEM) and writeback of results (WB). Values are read from the register file at the ID/EX boundary and results are written to the register file at the MEM/WB boundary. Forwarding paths (aka bypass logic) are used to reduce the number of stalls by allowing the result of one instruction to be used as an input to the ALU if required by the next instruction. Conventionally, most of the control signals from decode and those that control the pipeline and forwarding paths are not shown — although that is where many of the most difficult bugs lie. We use this simple microarchitecture to explain the technique, Sect. 5 discusses how we adapt the approach to handle more realistic microarchitectures including dual issue, out-of-order retire and register renaming.
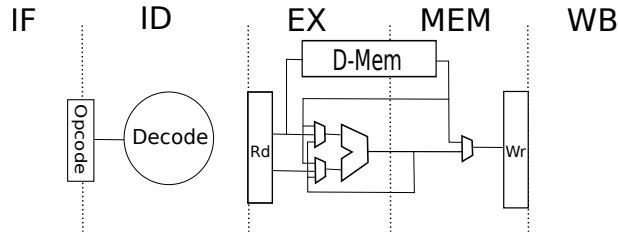

Fig. 1. A 5-stage processor pipeline, with forwarding paths, omitting I-Fetch

Our first challenge is to implement the abstraction function *abs* which is responsible for converting the micro-architectural state of the processor into an architectural state. To verify an addition instruction, the function *abs* must extract the current values of the integer registers.

Many simple processors commit their results in order in a single pipeline stage. This means that, at the beginning of the cycle where the add instruction commits, the micro-architectural register file should contain the same values as the architectural register file *before* the add executes and, at the end of the cycle, the micro-architectural register file should contain the same values as the architectural register file *after* the add executes. We can therefore obtain the state *before* by reading the state at the end of the writeback stage and the state *after* by reading from the end of the Mem stage.

The other part of the input state of the processor that we require is the opcode of the current instruction. The opcode is normally discarded shortly after instruction decode and is not available at the point where an instruction commits. We therefore need to implement a "pipeline follower" which copies the opcode from one stage to the next and implements the same pipeline stall/flush logic as the datapath. This is similar to the introduction of "ghost state" in Lahiri et al. [13]. The followers and abstraction logic for the pre/post-states are illustrated in Fig. 2
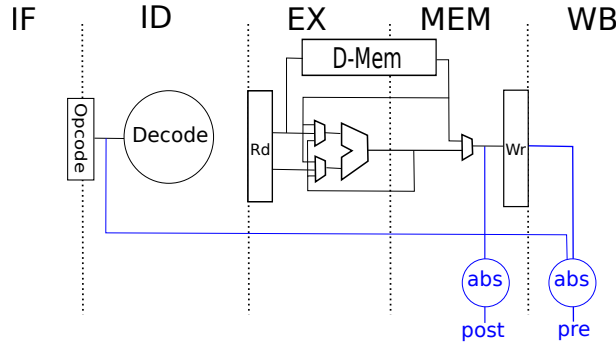

Fig. 2. A 5-stage processor pipeline with state abstraction and follower

Of course, modern ARM processors are considerably more challenging than a simple 5-stage pipeline: Sect. 5 describes the variations on the above approach required to apply ISA-Formal in practice.

Our second challenge is to create a specification of the addition instruction. For any individual instruction, the specification can often be written as a short piece of purely combinational logic. For example, ARM's 16-bit encoding of the instruction "ADD Rd, Rn, Rm" has opcode `0b0001100 | Rm<<6 | Rn << 3 | Rd` and adds the contents of registers Rn and Rm and writes the result to register Rd.

This can be implemented by the following System-Verilog.

```
assign ADD_retiring = (pre.opcode & 16'b1111_1110_0000_0000)
                      == 16'b0001_1000_0000_0000;
assign ADD_result   = pre.R[pre.opcode[8:6]]
                      + pre.R[pre.opcode[5:3]];
assign ADD_Rd       = pre.opcode[2:0];
```

To complete the example, we add assertions that the abstracted result matches the result of the specification when retiring an add instruction.

```
assert property (@(posedge clk) disable iff (~reset_n)
    ADD_retiring |-> (ADD_result == post.R[ADD_Rd]));
```

The above specification is remarkably simple so it is worth examining what kinds of defect this specification could catch.

Decode Errors Most obviously, this specification would detect any error in instruction decoding. But many decode errors are also caught by other verification methods such as directed or random testing so, at first sight, this does not seem especially useful. However, the instruction decoder is responsible not just for determining how to execute the current instruction but also for setting signals that determine whether it is safe to apply optimizations involving later instructions. A property like the above found a decoder bug involving one such signal that determined whether two adjacent instructions could be fused into a single micro-op: the signal was being incorrectly set for one instruction. This defect had been missed despite extensive testing of the processor: there were tests to ensure that the optimization did happen but testing is ill-suited to checking that it never happens in any other circumstance.

Datapath Errors An error in a datapath would be caught by this kind of check although, in practice, many errors of this kind are caught by other verification methods already in use.

Interactions between instructions Most usefully, and unlike methods based on Burch-Dill flushing, this specification will detect errors caused by interactions between instructions such as errors in the forwarding logic that can supply inputs to this instruction. The example given in the introduction of a sequence of five instructions which triggered an error in the forwarding control logic was detected by a hand-written property like the above. Bugs like this are significantly more important to catch because the forwarding paths vary from one processor to another, the control logic is difficult to get right and the errors are hard to catch by conventional tests.

We currently use bounded model checking which verifies that a sequence of n instructions does not go wrong but to show that any sequence does not go wrong, we would need to find invariants about the processor and use those to get unbounded proofs. Going further, in order to complete ISA verification, we would need to verify that instructions are not lost, duplicated or reordered (we have done this for some processors) and, to complete verification of the core, we would need to verify exception taking mechanisms, the instruction fetch unit and the memory management unit.

4 Generating Verification IP with Architecture Explorer

The main challenge in applying the above approach to a full processor is one of scaling. The ARM v8-M architecture has 384 instruction encodings and the instruction set part of the architecture specification is over 600 pages long [4]; and the ARM v8-A/R architecture has 1280 instruction encodings and is over 2500 pages long [3]. Some of the encodings explicitly disallow using certain registers as sources or destinations to the instructions, many of the instructions are conditional and there are a variety of other complications and corner cases. In addition, changes are regularly added to the architecture specification. All these reasons make the prospect of writing, testing and maintaining a Verilog specification like that shown above unattractive.

Over the last 5 years we have developed tools which transform ARM's official Architecture Reference Manuals into executable specifications of the v8-A/R and v8-M architectures [17]. A key part of making this specification useful was to test it thoroughly before using the specification to verify anything else. In many ways, this is like Fox and Myreen's testing of their ARM ISA specification [6] except that we were able to use ARM's internal architecture conformance testsuite (that is normally used to test processors) to test the specifications with billions of instructions that probe each instruction's corner cases.

The core of this specification is ARM's Architecture Specification Language (ASL) that grew out of the pseudocode used in earlier versions of the architecture reference manuals. At a high level, ASL is an indentation-sensitive, imperative, strongly typed, first-order language with type inference, exceptions, enumerations, arrays, records, and no pointers. All integers in ASL are unbounded and there is direct support for N-bit bitstrings and functions are allowed to be polymorphic in the width of a bitstring. For example, memory read returns a value of type `bits(8*size)` where `size` is constrained to be 1, 2, 4 or 8.

The task of scaling the ISA-Formal approach up to handle the full instruction sets with all their complexities is therefore one of translating the rich, expressive ASL language to combinational System-Verilog using the synthesizable subset of Verilog that is accepted by commercial Verilog model checkers. The challenge in doing this is that synthesizable Verilog is intended to describe hardware and imposes several limitations upon us; (1) Verilog integers are finite and the bitwidth is a part of the type; (2) Combinational Verilog is normally written in a declarative style with no assignments or control flow and few function calls; (3) Synthesizable Verilog does not support unbounded for-loops or while-loops; (4) Synthesizable Verilog does not support exceptions; (5) The width of bitstrings in Verilog must always be a manifest constant and there is no form of polymorphism over bitwidths of functions.

We were able to overcome the first four issues using relatively conventional compiler techniques. (1) We use a global flow-insensitive value range analysis to compute the required width of most integer variables and use a large, but safe bound for any integers with unknown range. (2) Verilog includes a rarely used procedural subset which most of the language can be translated into. (3) User-supplied bounds on loops can be used to unroll all loops. (4) A whole-program

transformation which adds additional flags and control flow to make exception and return-related control flow explicit.

The most challenging problem was dealing with bitstring polymorphism. Virtually all polymorphism was caused by instructions which could operate on data of different widths such as 8, 16, 32 or 64-bit load instructions. This observation enabled us to eliminate almost all polymorphism by automatically specializing such instruction encodings to create a separate instruction for each data width and then to use alternate passes of constant propagatation and a "monomorphization" pass which identifies calls to polymorphic functions where the bitwidth is a manifest constant and replaces the call with a call to a monomorphic instance of the polymorphic function. The remaining polymorphism is handled by a set of ad-hoc transforms in the Verilog backend.

5 Applying ISA-Formal to CPUs

In practice, few processors are as simple as the 5-stage pipeline sketched in Fig. 1 and we have had to develop a number of techniques in writing abstraction functions to deal with complex functional units, out-of-order retire, dual issue pipelines, instruction fusion, and register renaming.

5.1 Complex Functional Units

For the most part, our end-to-end approach to verification works: commercial model checkers are able to handle the complexity of most components without assistance. However, for complex functional units such as floating point and the memory system we choose to use other more scalable verification techniques such as the end-to-end memory-system verification technique described by Stewart et al. [20]. This modular approach lets ISA-Formal verification focus on control logic and forwarding paths that controls, feeds and is fed by these complex units.

In order to make ISA-Formal modular, we partition the specification on function call boundaries into different parts "Instruction Set Architecture (ISA)," "Floating Point," "Exception," "Address Translation," etc. and only generate Verilog for the "ISA" part. Any functions on the interfaces to other partitions are written by hand and many are just a few lines long: returning some component of the result of the pre-state or changing some component of the post-state.

On the interfaces, we adopt a variety of approaches to filling the resulting gaps in the generated Verilog using interface properties, subset behaviour checking and abstract functions. In general, these approaches will prevent us from detecting bugs in some parts of the processor using ISA-Formal. We tackle this by tracking which parts of the processor are not being checked by ISA-Formal and ensuring that an alternative verification technique is used on those parts.

Interface properties For some components such as the memory system, we were already creating interface specifications which were sufficiently strong that we could use the interface specification instead of the memory system. This only

required us to convert the architectural view of the memory system to the micro-architectural view by translating requests/responses between representations.

Subset behaviour checking For components such as floating point units, a specification of the full behaviour would still be too complex to use in verification but is quite simple if we restrict ourselves to a subset of the full behaviour. For example, if we restrict the inputs to $\pm\{0, 1, \infty, S\text{-}NaN, Q\text{-}NaN\}$ then it is easy to create specifications of all the FP instructions for this subset and perform some verification. Obviously, this would not be sufficient to detect errors in the floating point unit itself, but this subset gives enough different values that errors in the control and forwarding logic can be detected.

We could use SystemVerilog assumptions to restrict inputs to the chosen set of inputs, but this would restrict all of the checks that ISA-Formal performs on instructions: whether the instruction sets condition flags, raises an exception, accesses memory, which registers are written, etc. Instead, we add an additional signal indicating whether the inputs are in the supported subset and use that signal only to restrict checks of the values written to floating point registers.

Abstract Functions The final option is to use the processor as an oracle. That is, we add logic to track the inputs and outputs from some functional unit and then use the output value if the inputs of a function in the architectural specification match the actual inputs of a functional unit in the processor. Since we are choosing to trust the behaviour of that unit, this cannot detect errors in the unit but it can detect errors in the surrounding control and forwarding logic.

5.2   Out of order completion

In an in-order core, all instructions retire strictly in-order, but some slower instructions may complete out of order. Retiring a load (say) after the memory protection check but before the data returns from the memory system allows independent instructions to continue without waiting for the access to complete. Such optimizations are important to verify because they introduce difficult corner cases in the design such as ensuring that the result of the load is written back even if the processor takes an exception.

The difficulty in verifying out-of-order completing instructions is that it is hard to construct the post-state: by the time that the load instruction completes, some of the instructions issued *after* load will also have completed. This is further complicated because some load instructions may be split into multiple micro-ops which complete independently.

Our solution to this is to take a snapshot of the pre-state when the load instruction *retires*. As each micro-op for the instruction under test completes, the snapshot is updated with the change. Finally, when the last micro-op completes, the final post-state is available and the instruction can be checked against the architecture specification.

5.3 Dual issue pipelines

Dual issue pipelines decode and execute two consecutive instructions in parallel. To handle dual issue pipelines, we add a further abstraction function to extract the intermediate state between execution of the two instructions. Our initial approach to checking these was to create two copies of the combinational logic implementing the specification: one copy for each pipeline. This worked but consumed a lot of memory and would scale badly for 3 or more-issue processors so, instead, we use a single copy of the specification and insert multiplexors to select which pre/post state is used with the specification.

The most serious problem encountered occurs if the second instruction can suppress part of the behaviour of the first instruction. For example, if both instructions modify the carry flag, then the final value written will be the result of the second instruction. In this case, the carry flag value from the first instruction may not be available at the writeback stage and we need to identify the correct signal to use and add a pipeline follower to propagate the value down to the point of serialization. Any error in choice of signal is detected when that signal is used as part of the pre-state of the second instruction.

5.4 Instruction Fusion

A high-performance processor might wish to fuse commonly occuring pairs of consecutive instructions into a single instruction. For example Malik et al. [15] describes a processor that detects sequences of dependent ALU instructions such as

```
SUB R4, R1, R2   ; R4 := R1 - R2
ADD R4, R4, R3   ; R4 := R4 + R3
```

and fuses them into a single macro-operation that reads three inputs from the register file and performs two add/subtract operations.

Optimizations of this kind raise a potential problem in sequences where the results of the first instruction are overwritten by the second instruction because the processor may not calculate the post-state of the first instruction or the pre-state of the second instruction.

Our solution is to add additional verification logic to calculate the missing intermediate state. The correctness of this logic is verified when checking that all uses of the SUB instruction (i.e., the first instruction of the pair) is correct and that justifies use of the result when checking that the SUB/ADD fused pair (i.e., the first/second instruction pair) gives the correct overall result.

5.5 Register Renaming

Processors with out-of-order instruction issue differ significantly from processors with in-order issue because they speculatively execute instructions past branch instructions. To allow them to recover from mis-speculation, they use a register rename table that maps architectural registers such as "X0" to one of a large

pool of physical registers. As instructions are decoded, source registers are "renamed" using this table; free physical registers are allocated and the rename table is updated with mappings from destination register names to these physical registers. Instructions typically execute as soon as their input dependencies are satisfied but, to preserve the illusion that instructions execute in program order, a reorder buffer (ROB) only commits instructions in program order.

Despite the added complexity of speculative execution, register renaming and reorder buffers, it is actually simpler to apply ISA-Formal to out-of-order processors because they have a single clearly identified point of serialization implemented in the reorder buffer. In contrast, in-order processors have a variety of different mechanisms to support a limited degree of out-of-order execution such as varying pipeline length or supporting out-of-order completion of slow instructions and these different mechanisms are scattered across the processor.

5.6   Debugging Abstraction Functions

From the above, it should be apparent that creating the abstraction code remains a difficult task and involves a lot of work with the CPU designers to get right. While debugging these abstraction functions, we have found that it is useful to start by using hand-written properties like those described in Sect. 3 for instructions that touch the major parts of the processor. For example, a data-processing instruction, a load, a store, a floating point move, etc.

It is significantly easier to debug the abstraction function using hand-written specifications than using a mechanical translation from the specification. Once we have debugged the abstraction functions, we switch to using the machine-generated specifications exclusively, and rarely look at the generated code.

5.7   Handling Known Problems

One of the major difficulties we experienced before developing ISA-Formal was that formal verification tools would report variations on the same defect over and over again. This was a problem early in development when we might know that part of the processor was missing or incomplete; and it is a problem at any stage that once the bug report has been filed, the verification team wants to focus on finding other problems until the bug has been dealt with.

A critical technique for handling known problems is to maintain a list of assumptions corresponding to each individual bug or feature. As each bug is fixed, we remove the corresponding assumption and confirm that the bug has been fixed. Using assumptions is a simple technique but it greatly increases our ability to use formal verification to detect errors early in development and it very effectively decouples processor design from verification allowing the tasks to proceed in parallel.

6 Results

This section describes the results of applying ISA-Formal in three small-scale trials and five full-scale uses. These eight trials and uses cover the full lifetime of ARM processor developments; they cover both application processor targetted at mobile phones, etc. and microcontrollers targetted at embedded uses; and they cover micro-architectures ranging from 3-stage, in-order pipelines through dual-issue, in-order pipelines to out-of-order pipelines.

6.1 ARM's Development Phases

ARM's development process involves four stages of roughly equal length: Develop and Test (D&T), Alpha, Beta and Access. The goal of each stage is to create a basic pipeline design in D&T; make it feature complete by the end of Alpha; improve power, performance and area through Beta; and to improve confidence in the design through the access period where the design is made available to the lead partners for that processor for evaluation and feedback. Testing steadily increases throughout this process and each stage applies roughly an order of magnitude more testing than the previous stage.

6.2 Small-scale Trials

We carried out three small-scale trials on processors that were already in the access phase to demonstrate the ability of ISA-Formal to detect defects that were hard to detect by other means. These trials consisted of developing hand-written properties like those described in Sect. 3 and demonstrated the ability to detect defects that had been found by other means as well as new defects.

The defect described in the introduction is an example of a bug we detected during this trial process. The trigger sequence of the defect is conditional execution of instructions executing in two pipeline stages with a combination of taken and not-taken instructions. In a 2-pipeline design, the size of the smallest trigger sequence is 5 instructions: one to set up the condition, two (one per pipe) to generate values that might be forwarded, and two (one per pipe) to consume forwarded values. (There are several variations on that basic pattern.) Using traditional simulation-based verification, patterns like this would have to be tested on all combinations of instructions that have forwarding paths between them in that particular micro-architecture and each processor will have a different set of forwarding paths. There are many, many sequences of instructions like this to be tested so defects of this form are typically only found during soak-testing during the Access phase. Using ISA-Formal, we created hand-written properties for one or two instructions corresponding to each major unit in the datapath (the ALU, shifter, multiplier, etc.), we created abstraction functions for each of the two pipelines, and, since we left the opcode received from the fetch unit unconstrained, the commercial bounded-model-checker explored sequences of instructions up to some bound. We ran about a dozen properties through the model checker and after two minutes proof time detected the failing trigger sequence.
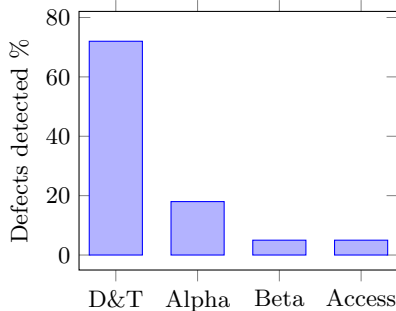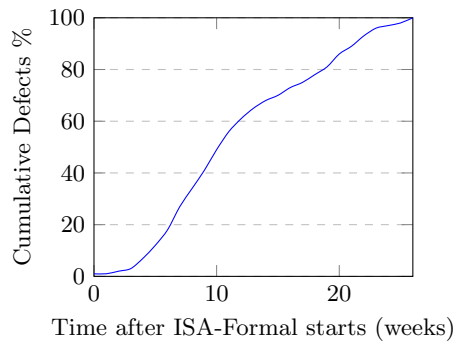
Fig. 3. Defect Detection by Phase

Fig. 4. Defect Detection by Time

The same experience was repeated on all three processors: bugs were found with relatively little effort with the bulk of the work being done by junior engineers supervised by formal experts and with input from the microarchitects. The consistent combination of low human effort and low machine effort was an important part of demonstrating that ISA-Formal could detect difficult defects that, at best, would have been caught only during the Access phase.

6.3   Production Usage

Based on the success of the small-scale trials, ARM decided to adopt ISA-Formal as part of the formal verification strategy on five processors that were in earlier stages in their development: three in D&T, one in Alpha and one in Access. This work used the tool described in Sect. 4 to generate Verilog for all instructions directly from ARM's official Architecture Reference Manuals allowing engineers to focus on developing abstraction functions and testing the processor.

Defects have been found in all five processors with the distribution roughly in proportion to the effort invested in that processor. The small-scale trials had demonstrated that ISA-Formal can detect difficult to detect defects late in processor development; the production usage demonstrated that ISA-Formal is effective at detecting defects in earlier phases of development. Fig. 3 and Fig. 4 show the distribution of confirmed, distinct defects detected using ISA-Formal by phase and by time. Fig. 3 shows that ISA-Formal is capable of catching many defects early in development (overcoming the problem of being able to find many distinct defects in parallel with development) and that it is capable of finding defects late in development even after extensive testing by other methods. Fig. 4 shows that ISA-Formal is able to start detecting defects in just a few weeks work and continues to find bugs as processors are developed.

We also found that ISA-Formal was able to detect issues affecting all areas of the instruction set: FP/SIMD, Memory, Branches, Integer, Exceptions and System instructions (e.g., memory fence instructions) Table 5 shows the distribution of bugs found by ISA-Formal by the area of the processor affected (combining

FP/SIMD	25%
Memory	21%
Branch	21%
Integer	18%
Exception	8%
System	7%

Fig. 5. Defect Detection by Area

Processor	Lines of code
#1	2400
#2	2250
#3	4600
#4	1000
#5	2500

Fig. 6. Size of Verification Code

results for all processors). (The "Integer" category includes both integer datapath instructions and basic pipeline control issues — it is often hard to separate the two since integer instructions are so fundamental to a processor.)

It is encouraging to note that the two largest sources of detected bugs were FP/SIMD instructions and memory instructions. As Sect. 5.1 explains, we do not test the FPU or the memory subsystem but, despite this, we are still able to test and find defects in the forwarding, pipeline control and register logic connected to these units.

The effort of creating, testing and debugging the machine-readable specification and a tool to translate it to Verilog is considerable but can be shared across multiple processors and can be used for other purposes within the company (e.g., documentation, testing of architecture extensions, etc.). The primary cost of implementing ISA-Formal on a new processor is the effort required to implement the pipeline follower and abstraction function on each processor. As a rough indication of the effort required, Table 6 shows the number of lines of code required for each (anonymized) processor. Most processors need around 2,500 lines of support code: a fairly modest cost. The outliers are processor #4 which has not yet added a follower for floating point registers and processor #3 which is a more complex processor than the other four.

Beyond the bug numbers, we found that applying ISA-Formal early in the development was capable of finding bugs that would not normally be caught until much later. For example, very early in development of an out-of-order processor, ISA-Formal found a bug that occurred when all the free registers in the physical register pool were in use. This was found before the processor could even execute load-store instructions so we would not normally be catching such bugs that early.

7 Conclusions

Two barriers to widespread industry adoption of formal verification techniques to check processors are scaling and return on investment issues. The end-to-end approach to verification that we adopt tackles both issues: it allows machine-generation of verification IP from the architecture specification, it allows engineers to detect bugs that affect actual instruction sequences very early in deployment, and it encourages creation of reusable tools, techniques and IP that can be used across an unusually wide range of micro-architectural styles.

This paper describes the steps needed to turn the basic idea into a scalable, reusable technique: automation, dealing with a range of different microarchitectural design techniques, and initial bringup issues. We have applied this method to 8 different ARM processors spanning all stages of development up to release. In all processors, this has found bugs that would have been hard for conventional simulation-based methods to find and ISA-Formal is now a key part of ARM's formal verification strategy.

To the best of our knowledge, this is the most broadly applicable formal verification technique for verifying processor pipeline control in mainstream commercial use.

References

1. Aagaard, M., Cook, B., Day, N.A., Jones, R.B.: A framework for microprocessor correctness statements. In: Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods. pp. 433–448. CHARME '01, Springer-Verlag, London, UK, UK (2001), http://dl.acm.org/citation.cfm?id=646705.702043
2. Aagaard, M.D., Jones, R.B., Melham, T.F., Oleary, J.W., Seger, C.J.H.: A methodology for large-scale hardware verification. In: Formal Methods in Computer-Aided Design. pp. 300–319. Springer (2000)
3. ARM Ltd: ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile). ARM Ltd (2013)
4. ARM Ltd: (In Preparation) ARM Architecture Reference Manual (ARMv8, for ARMv8-M architecture profile). ARM Ltd (2016)
5. Burch, J.R., Dill, D.L.: Automatic verification of pipelined microprocessor control. In: Proceedings of the 6th International Conference on Computer Aided Verification. pp. 68–80. CAV '94, Springer-Verlag, London, UK (1994), http://dl.acm.org/citation.cfm?id=647763.735662
6. Fox, A.C.J., Myreen, M.O.: A trustworthy monadic formalization of the ARMv7 instruction set architecture. In: Kaufmann, M., Paulson, L.C. (eds.) Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6172, pp. 243–258. Springer (2010), http://dx.doi.org/10.1007/978-3-642-14052-5_18
7. Higgins, J.T., Aagaard, M.D.: Simplifying design and verification for structural hazards and datapaths in pipelined circuits. In: Proceedings of the High-Level Design Validation and Test Workshop, 2004. Ninth IEEE International. pp. 31–36. HLDVT '04, IEEE Computer Society, Washington, DC, USA (2004), http://dx.doi.org/10.1109/HLDVT.2004.1431229
8. Hunt, Jr., W.A., Sawada, J.: Verifying the FM9801 microarchitecture. IEEE Micro 19(3), 47–55 (May 1999), http://dx.doi.org/10.1109/40.768503
9. Jhalal, R., McMillan, K.L.: Microarchitecture verification by compositional model checking. In: Berry, G., Comon, H., Finkel, A. (eds.) Computer Aided Verification, Lecture Notes in Computer Science, vol. 2102, pp. 396–410. Springer Berlin Heidelberg (2001), http://dx.doi.org/10.1007/3-540-44585-4_40
10. Kaivola, R., Ghughal, R., Narasimhan, N., Telfer, A., Whittemore, J., Pandav, S., Slobodová, A., Taylor, C., Frolov, V., Reeber, E., Naik, A.: Replacing testing

with formal verification in Intel Core i7 processor execution engine validation. In: Proceedings of the 21st International Conference on Computer Aided Verification. pp. 414–429. CAV '09, Springer-Verlag, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-02658-4_32

11. KiranKumar, V., Gupta, A., Ghughal, R.: Symbolic trajectory evaluation: The primary validation vehicle for next generation Intel processor graphics FPU. In: Formal Methods in Computer-Aided Design (FMCAD), 2012. pp. 149–156. IEEE (2012)

12. Kroening, D., Paul, W., Mueller, S.: Proving the correctness of pipelined micro-architectures. In: Waldschmidt, K., Grimm, C. (eds.) Proc. of ITG/GI/GMM-Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen". pp. 89–98. VDE Verlag (2000)

13. Lahiri, S.K., Bryant, R.E.: Deductive verification of advanced out-of-order microprocessors. In: Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings. pp. 341–353 (2003), http://dx.doi.org/10.1007/978-3-540-45069-6_33

14. Lahiri, S.K., Pixley, C., Albin, K.: Experience with term level modeling and verification of the M*CORE^{TM} microprocessor core. In: Proceedings of the Sixth IEEE International High-Level Design Validation and Test Workshop 2001, Monterey, California, USA, November 7-9, 2001. pp. 109–114 (2001), http://dx.doi.org/10.1109/HLDVT.2001.972816

15. Malik, N., Eickemeyer, R.J., Vassiliadis, S.: Interlock collapsing ALU for increased instruction-level parallelism. In: Proceedings of the 25th Annual International Symposium on Microarchitecture. pp. 149–157. MICRO 25, IEEE Computer Society Press, Los Alamitos, CA, USA (1992), http://dl.acm.org/citation.cfm?id=144953.145794

16. McMillan, K.L.: Verification of an implementation of Tomasulo's algorithm by compositional model checking. In: Proceedings of the 10th International Conference on Computer Aided Verification. pp. 110–121. CAV '98, Springer-Verlag, London, UK, UK (1998), http://dl.acm.org/citation.cfm?id=647767.733764

17. Reid, A.: Creating trustworthy specifications of ARM v8-A and v8-M system level architecture. In preparation (2016)

18. Slobodová, A., Davis, J., Swords, S., Hunt Jr, W.: A flexible formal verification framework for industrial scale validation. In: Formal Methods and Models for Codesign (MEMOCODE), 2011 9th IEEE/ACM International Conference on. pp. 89–97. IEEE (2011)

19. Srinivasan, S.K.: Automatic refinement checking of pipelines with out-of-order execution. IEEE Transactions on Computers 59(8), 1138–1144 (2010)

20. Stewart, D., Gilday, D., Nevill, D., Roberts, T.: Processor memory system verification using DOGReL: a language for specifying end-to-end properties. In: International Workshop on Design and Implementation of Formal Tools and Systems. DIFTS '14 (2014)

21. Windley, P.J.: Formal modeling and verification of microprocessors. Computers, IEEE Transactions on 44(1), 54–72 (1995)