Using x86isa for Microcode Verification

Shilpi Goel Rob Sumners
Centaur Technology, Inc.
7600-C N. Capital of Texas Hwy, Suite 300
Austin, TX 78731
{shilpi,rsumners}@centtech.com

1 Introduction

Microcode verification is a crucial part of ascertaining whether a given microarchitectural design correctly implements an instruction set architecture. The prerequisites of microcode verification are a formal model of ISA-defined instructions, a formal model of micro-operations (*uops*), and a proof methodology that relates the latter to the former. In this paper, we present a report on our preliminary work towards constructing a framework to verify the microcode of x86 processors designed at Centaur Technology. Our objective is to verify the implementation of every x86 ISA-level instruction — i.e., to prove that the sequence of uops generated by the processor for an instruction correctly implements that instruction.

The x86 ISA is one of the most complex architectures in widespread use today; it is described by Intel® manuals [6] in ∼5000 pages consisting mostly of English prose and some pseudocode. The x86 ISA is also extended regularly — another Intel® manual [7] describes new instructions that are intended to be supported by future processors. As such, writing *and* maintaining an accurate x86 ISA specification is in itself a substantial undertaking. We use the x86isa model [8, 3] as the x86 ISA specification. For this project, we redesigned parts of x86isa, especially the opcode maps' representation, so that it is easily extensible. We describe relevant aspects of x86isa in Section 2.

Our microcode model has been derived from Centaur's SystemVerilog RTL definitions. As in previous work [5], we use the same specifications of uops for microcode verification that are used for verifying the execution units. In this work, we also verify instruction decoding itself; i.e., whether the instruction bytes are decoded correctly and if they trigger decode-time exceptions when expected (in accordance with x86isa). We describe relevant aspects of this process in Section 3.

A pictorial overview of our verification effort is given in Figure 1. All of our work is done using the ACL2 theorem proving system [1]. We emphasize that though we have used our framework to verify some microcode at Centaur, our project is in its early days still. Our verification focus is currently on the processor's front-end and the execution units — we do not account for register mapping, reordering of uops, caches, etc.

2 x86isa: Instruction Listings

The Intel® manuals include *opcode maps*, which list the instructions supported by their x86 processors in a tabular format. Though useful to see certain kinds of information (e.g., which opcode corresponds to which instruction mnemonic), these maps do not contain all information relevant to instruction decoding (e.g., which CPUID features must be supported for an instruction to be valid and thus, not throw an "undefined operation" exception). Moreover, these maps are somewhat out-of-date — for instance, as of this writing, they do not list AVX-512 instructions.

Submitted to:
SpISA 2019

© Goel and Sumners
This work is licensed under the
Creative Commons Attribution License.

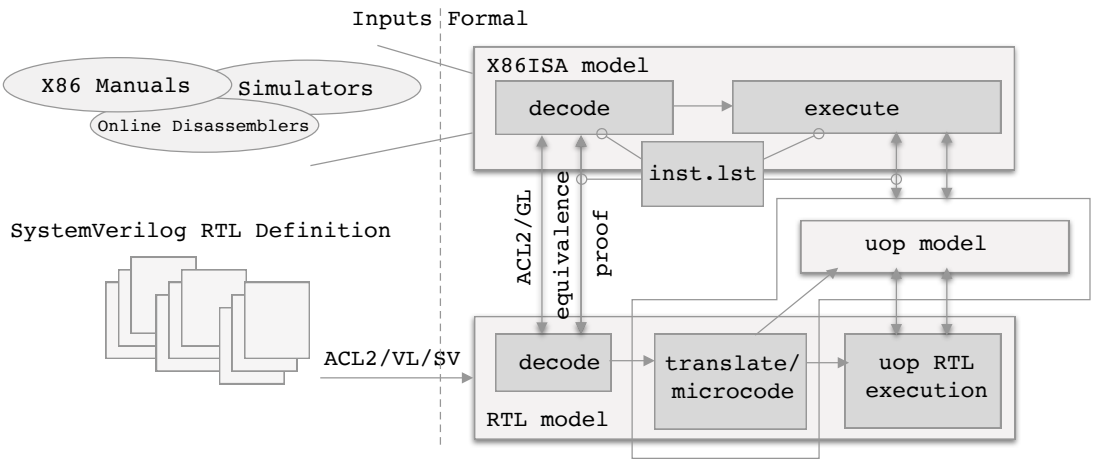

Figure 1: Overview: x86 Instruction Verification at Centaur Technology

In x86isa, we enhance the basic idea behind these opcode maps by defining data structures in ACL2, called inst.lst, that contain all the information needed to decode x86 instructions. An inst.lst is a list of instructions, where each instruction is essentially defined as a product type consisting of the following:

1. **Mnemonic:** can either be a string (e.g., "ADD") or a keyword (e.g., :2-BYTE-ESCAPE).
2. **Opcode:** is a product type that describes an instruction variant of the mnemonic. It contains the prefixes (including REX, VEX, and EVEX), opcode bytes, mode of operation, ModR/M byte (used for operand addressing and sometimes as an opcode extension), and CPUID feature flags.
3. **Operands:** is a product type that describes the encoding of each operand of the instruction (e.g., the first operand is an XMM register indexed by the reg field of the ModR/M byte).
4. **Semantic Function:** lists the name of the ACL2 function that implements this instruction variant.
5. **Exceptions:** list the conditions for detecting certain decode-time exceptions (e.g., throw an undefined operation exception if the lock prefix is used but unsupported by an instruction).

We obtained the initial version of inst.lst by parsing the tables in the description pages of each instruction in the Intel® manuals (Chapters 3-5, Vol. 2). The x86isa model essentially revolves around inst.lst. We automatically generate many functions from the shape of these structures as well as the actual data in them — e.g., functions to perform instruction decoding (including checking for exceptional behavior) and to dispatch control to the appropriate semantic functions. This makes the x86 ISA specification easily extensible. To support the decoding of new instructions and their variants in x86isa, one simply has to add appropriate entries to inst.lst, and to support their (concrete/symbolic) execution, one has to list the appropriate semantic function in those entries. We note that the semantic functions that we use internally at Centaur for both unit-level and microcode proofs are different, though similar in spirit, from those listed in inst.lst in the public-facing model of x86isa — this is because our internal functions contain some proprietary information. More information about x86isa, such as its program verification capabilities, can be found elsewhere [8].

3 Design Verification

An x86 microprocessor is a complex hardware design; as such, in order to make the formal verification of the processor tractable, we focus on proving that the processor executes a single x86 instruction

correctly from a clean or flushed state. This preface of starting from a "clean" state affords the ability to view the execution of the processor as a single function (implemented over many clock cycles) which maps the relevant components of the initial state to modified components of a final state. This limited scope significantly reduces the complexity of verification by limiting the need to specify invariants on the processor state. Nevertheless, given the complexity of x86 processors and the x86 ISA itself, even this limited proof goal is still an extensive task.

The single-instruction correctness statement is expressed as the correctness of executing the uop sequence produced by decoding and translating (including microcode execution) a sequence of bytes defining a single x86 instruction. The length and complexity of uop sequences resulting from translation can vary from very small simple sequences for integer operations between registers to long complex sequences arising from the execution of microcode programs for more involved instructions. The uops are then scheduled and carried out within execution blocks which implement the intended functions of the uops. Our stated primary interest of verifying microcode is akin to program verification operating on a uop model – a significant task in and of itself. This core work is then extended to x86 processor verification by proving that byte sequences are decoded correctly and that the uops themselves are implemented correctly in the execution units.

In order to prove the eventual goal that the RTL implementation function (generated using existing VL and SV packages [2] from source SystemVerilog) is consistent with the x86isa model, we use ACL2 itself to reduce the problem into proving the correctness of component lemmas. For each of these component lemmas, we then use the GL package in ACL2 [9] to translate the ACL2 lemma to a propositional formula which is then bit-blasted using combinations of heuristics and algorithms including SAT, BDDs, and AIG rewriting. Reducing our top-level goal to lemmas which can be automatically discharged via bit-blasting is a complicated task for which we leverage the x86isa model and the uop model. The reduction of our top-level goal is essentially comprised of the following steps:

1. **Decode:**
   - *Inputs:* byte sequence from instruction memory and current x86 configuration state.
   - *Outputs:* either a well-formed x86 instruction (correlated to inst.lst) or an exception.
   - *Proof Goal:* prove that the x86isa and RTL decode functions are consistent.
   - *Decomposition:* reduction by cases due to parsing of the byte sequence and further cases due to instruction prefixes and configuration for exception generation drawn from inst.lst.
2. **Translation/Microcode:**
   - *Inputs:* well-formed x86 instruction (correlated to inst.lst) and current x86 configuration state.
   - *Outputs:* a sequence of uops implementing the instruction.
   - *Proof Goal:* prove that the execution of the generated uop sequence starting from an initial register/memory/configuration state will result in a state consistent with instruction execution.
   - *Decomposition:* primary reduction per instruction case defined in inst.lst structure with secondary reductions required in splitting up longer microcode programs and special input/data cases for more complex instructions.
3. **Micro-operation Execution:**
   - *Inputs:* uop instruction to execute and data from current x86 register/memory/configuration state.
   - *Outputs:* resulting output from execution of the uop.
   - *Proof Goal:* prove that the RTL execution units produce a computational result consistent with the uop model specification.
   - *Decomposition:* primary reduction per uop instruction supported by uop model with secondary reductions on input data cases pertinent to the implementation of the uop.

The x86isa model (and especially the inst.lst structure) and the uop model primarily provide specifications to prove against the hardware design. In addition, the definition of these models are leveraged throughout the proof decomposition process to organize, generate, and collect lower-level component proofs into the larger composite results. Even further, these models are used to help simplify the inputs required from users (e.g. defining constraints for legal instantiations of an instruction given just the mnemonic and configuration mode).

4 Conclusion

In the past, the x86isa model has been used to formally verify some 64-bit application and system machine-code programs [8]. More recently, it has been extended to support the analysis of 32-bit application programs [4]. This paper constitutes the first report of x86isa being used for hardware verification.

Prior work regarding microcode verification [5] at Centaur focused on uops in the microcode ROM. In this project, in addition to reasoning about instruction decoding, we take a principled approach towards accounting for uops emitted directly by the decode unit as well.

This paper reports on preliminary work done to verify microcode at Centaur. We are already using our framework to prove the correctness of Centaur's implementation of x86 ISA-level instructions. We plan to extend our framework by automating the following main tasks: inferring coverage information for these proofs and discharging commonly occurring proof obligations during symbolic simulation.

References

[1] *ACL2 Home Page*. Online; accessed: July 2019. http://www.cs.utexas.edu/users/moore/acl2.

[2] *Centaur's Hardware Verification Libraries in the ACL2 Community Books*. Online; accessed: July 2019. https://github.com/acl2/acl2/tree/master/books/centaur.

[3] x86isa *Library in the ACL2 Community Books*. Online; accessed: July 2019. https://github.com/acl2/acl2/tree/master/books/projects/x86isa.

[4] Alessandro Coglio & Shilpi Goel (2018): *Adding 32-bit Mode to the ACL2 Model of the x86 ISA*. In: Proceedings of the 15th International Workshop on the *ACL2 Theorem Prover and Its Applications, ACL2 2018*, Austin, Texas, USA, November 5-6, 2018, *EPTCS* 280, pp. 77–94, doi:10.4204/EPTCS.280.6.

[5] Jared Davis, Anna Slobodova & Sol Swords (2014): *Microcode Verification – Another Piece of the Microprocessor Verification Puzzle*. In Gerwin Klein & Ruben Gamboa, editors: *Interactive Theorem Proving*, Springer International Publishing, Cham, pp. 1–16.

[6] Intel Corporation: *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*. Online. Order Number: 325462-070US. (May, 2019). https://software.intel.com/en-us/articles/intel-sdm.

[7] Intel Corporation: *Intel® Architecture Instruction Set Extensions Programming Reference*. Online. Order Number: 319433-037. (May, 2019). https://software.intel.com/en-us/articles/intel-sdm.

[8] Shilpi Goel (2016): *Formal Verification of Application and System Programs Based on a Validated x86 ISA Model*. Ph.D. thesis, Department of Computer Science, The University of Texas at Austin.

[9] Sol Swords (2010): *A Verified Framework for Symbolic Execution in the ACL2 Theorem Prover*. Ph.D. thesis, Department of Computer Science, The University of Texas at Austin. Available at http://hdl.handle.net/2152/ETD-UT-2010-12-2210.