

Verifying x86 Instruction Implementations

Shilpi Goel

Anna Slobodova

Rob Sumners

Sol Swords

Centaur Technology, Inc., Austin, TX. USA

{shilpi,anna,rsumners,sswords}@centtech.com

Abstract

Verification of modern microprocessors is a complex task that requires a substantial allocation of resources. Despite significant progress in formal verification, the goal of complete verification of an industrial design has not been achieved. In this paper, we describe a current contribution of formal methods to the validation of modern x86 microprocessors at Centaur Technology. We focus on proving correctness of instruction implementations, which includes the decoding of an instruction, its translation into a sequence of micro-operations, any subsequent execution of traps to microcode ROM, and the implementation of these micro-operations in execution units. All these tasks are performed within one verification framework, which includes a theorem prover, a verified symbolic simulator, and SAT solvers. We describe the work of defining the needed formal models for both the architecture and micro-architecture in this framework, as well as tools for decomposing the requisite properties into smaller lemmas which can be automatically checked. We additionally cover the advantages and limitations of our approach. To our knowledge, there are no similar results in the verification of implementations of an x86 microprocessor.

1 Introduction

The capacity of formal verification tools and techniques has improved significantly in recent decades. The capabilities of efficient solvers for decidable logic problems have extended the umbrella of tasks that can be discharged automatically with proper setup. In addition, greater computational capacity with faster machines and more parallel and distributed processing have given more leverage to formal tools.

Unfortunately, the scale and complexity of microprocessor designs have kept ahead of the ability to formally verify their correctness. Modern microprocessors support more complex and extensive instruction sets, more threads and cores, and more aggressive optimizations to reduce latency and increase parallelism. In addition, as microprocessors develop into systems-on-a-chip, the potential for further optimizations and complexity only grows. Indeed, the best approach for applying formal verification resources in microprocessor verification is divide-and-conquer. Splitting an unmanageable task into smaller ones not only limits the scope in which verification is carried out, but also allows for tackling different parts of the design with different methods. It is a big advantage when all those different methods can be implemented within the same formalism, ensuring consistent composition of intermediate results.

Our work presented in this paper describes attempts to prove correctness of instruction implementations for an industrial x86 processor. This task includes verification of the Register-Transfer Level (RTL) design implementation of instruction decode, translation, and execution units as well as associated microcode. Verification of some of these aspects of an x86 instruction execution have been previously reported[22, 13, 28, 24, 14, 38]. However, those efforts considered those steps in isolation (one at a time) and none of them incorporated all of them within one verification framework. Compared to our previous work [19, 14, 17, 33, 32], we have:

- improved verification of decoding and translation by supporting the proof of a more general form of an instruction;
- added verification of microcode generated as a result of instruction decoding and increased automation of these proofs;
- expanded the verification coverage by considering *all* micro-operations dispatched to execution units and unifying the scope of proofs to the same top-level execution module.

The largest remaining gaps in our correctness proofs lie in the verification of memory access and scheduling of micro-operations.

We use the ACL2 theorem prover [2] and built-in verified proof routines [37, 35] to model and verify a target RTL implementation of instruction execution. All specifications are written in ACL2—our x86 ISA specification [31, 16] as well as our proprietary micro-architectural model. Leveraging publicly available ACL2 libraries (many of which were developed by our team), we translate the SystemVerilog design into a formal model within ACL2.

This paper is organized as follows. Section 2 will cover preliminary topics of x86 instruction set architecture as well as micro-architecture. We present an overview of our approach and the properties we target in Section 3. Our methodology is explained using examples that describe the verification of two x86 instructions. In Sections 4 and 5, we present detailed example instruction proofs and, in particular, we cover microcode definitions and verification. We present additional notes on how we generate proofs, and perform proof exploration and debugging in Section 6. We discuss related work in Section 7 and conclude the paper in Section 8.

2 Preliminaries

The primary goal of formal verification efforts at Centaur Technology is to prove that microprocessor designs operate in accordance with their specifications. The primary component of the specification for these microprocessors is compliance with the x86 instruction set architecture. We discuss instruction set architectures, x86 in particular, and the common aspects of microprocessor implementations of complex instruction sets like x86.

2.1 x86 Instruction Set Architecture

Microprocessors execute programs, which are defined as a sequence of instructions. These instructions result from the parsing or decoding of byte sequences fetched from memory. Each instruction performs operations on data loaded from either registers or memory, and stores the results in either registers or memory. An *Instruction Set Architecture* (or ISA) specifies how byte sequences decode into legal instructions and how the resulting instructions operate on the current programmer-visible state. An *ISA model* formally defines the semantics of an ISA. It generally consists of a type definition for the programmer-visible state and an update function, which takes the current state and returns the updated state after executing an instruction.

The x86[20] family of instruction set architectures covers a wide range of microprocessors built by several companies, including Centaur Technology, over the last several decades. The x86 family began as a 16-bit architecture and was incrementally extended to support wider data paths and an ever-increasing library of operations. We use “the x86 ISA” to mean the current iteration of x86, which supports legacy 16-bit and 32-bit modes, as well as modern 64-bit modes and vector instructions supporting up to 512-bit operations.

Figure 1 presents the state type and update function of the x86 ISA model, which we call `x86isa` [8], that is used as the specification in our project. The state of the `x86isa` consists of an instruction pointer (IP), general-purpose registers, and a model of memory. In addition, the `x86isa` state includes a bank of configuration registers controlling modes of operation, additional registers and memory tables to define address mappings, larger registers for vector operations, and registers which store side results and effects from instruction operation. Even relatively simple x86 integer instructions can be complicated, with conditions on how to map address accesses to memory and proper updating of flags denoting edge conditions in instruction operation.

The `x86isa` update function `x86-model-step` (see Figure 1) is a composition of four component functions. The function `x86-fetch-code` pulls bytes from memory at the current instruction pointer. The function `x86-decode` takes the fetched bytes and current configuration, and returns an exception `dx` and the instruction `instr` to execute. `dx` is relevant when a byte sequence is ill-formed or when the instruction is illegal to execute in the current state. The instruction `instr` is relevant when there is no exception. The function `x86-exec` takes the valid instruction `instr` and returns its computed results `rslts` and exceptions `ex`, if any occur during the computation. The results `rslts` include output data as well as side effects of the computation. For instance, many instructions will update flags to denote boundary conditions that have been reached—e.g.,

a zero flag is hit on a *decrement* instruction if the result is zero; this can be checked as a condition for, say, branching out of a loop. Finally, the function `x86-update` updates the x86 state either with the results of the computation or the effects of a triggered exception, and this state is then returned from the update function.

```

type x86-model-state is
  tuple ⟨IP, config, registers, memory⟩

func x86-model-step (state) is
  bytes      = x86-fetch-code(state.IP, state.memory)
  ⟨dx, instr⟩ = x86-decode(bytes, state.config)
  ⟨ex, rslts⟩ = x86-exec(instr, state)
  next-state = x86-update(dx, ex, rslts, state)
  return next-state

```

Figure 1: x86 ISA Model. Functions in bold represent specifications which we use for proofs about the microprocessor design. Details are in Figure 9.

The decoding and execution of even a single x86 instruction is complex. To begin with, instructions can be encoded in as few as 1 byte and as many as 15 bytes. Figure 2 presents the pieces of an x86 instruction, which consists of a variable number of bytes for instruction prefixes (i.e., legacy prefix overrides, and REX, VEX, and EVEX prefix bytes), up to 3 bytes for the instruction opcode, a variable number of bytes that specify which registers or memory addresses will be sources and destinations of the instruction (i.e., the ModR/M byte, SIB byte, and displacement bytes), and up to 8 bytes defining immediate data used in the instruction.

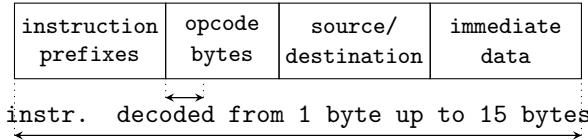


Figure 2: Decode of x86 instr. from byte sequence

With the extensive set of operations performed by x86 instructions, and the various modes and configurations in which these operations are performed, a complete formal x86 ISA model is a significant undertaking in and of itself [31]. We derive our specification for x86 instruction decode and execution from existing work [31, 17], but with several necessary modifications and extensions. We recognize around 3400 x86 instruction *variants* in our formal models. An instruction with the same mnemonic can have different variants—e.g., one variant can have a register as the destination operand and another can have a memory location.

2.2 Microprocessor Organization

Microprocessors which implement complex instruction architectures like x86 generally do not directly execute instructions, but instead, translate them into sequences of simpler operations, termed *micro-operations* or *uops*. These uops in effect define a new internal instruction architecture within the microprocessor, termed *micro-architecture*. The micro-architecture is optimized to support pipelining and the scheduling and mapping of operations onto execution units, which carry out the computations of these uops. Note that different variants of the same instruction may internally correspond to different uops.

Figure 3 presents a standard generalized architecture of an x86 microprocessor. Program code to execute is loaded from memory through a cache hierarchy attempting to optimize for locality. The instructions are then decoded from byte sequences into instruction structures (unless exceptions are detected). Each instruction is then translated via the XLATE and UCODE blocks into a uop sequence which implements that instruction. This translation is complex in and of itself; for simpler instructions, a table is consulted

and a fixed sequence of uops is generated. For more complicated instructions, a *prelude* sequence of uops will lead to a trap into *microcode* that has support for conditional branching and looping controlled by UCODE. Microcode is generally stored in an on-chip ROM. We term the composition of these sources of uops a *micro-op program* or *micro-program*. The uops which result from the stepping of the micro-program are then scheduled (and possibly reordered) for dispatch in EXEC. The source data for the uops are either coming from registers or memory loads, and results either being written to registers or stored to memory.

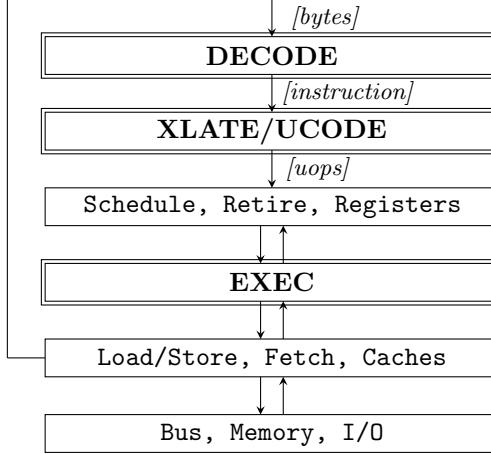


Figure 3: Microporcessor Organization

2.3 Microcode Model

In a fashion similar to the formal definition of `x86isa`, we define a formal model of the micro-architecture in ACL2 that we term the *microcode model* or *ucode model* (see Figure 4). This model is defined by its state and an effect of the execution of a uop to the state.

The uop format in the ucode model consists of an opcode, source and destination register identifiers, and immediate data. There are additional fields in the uop which are used to help optimize the scheduling and execution of the uops, which we elide here. Note that examples of uops and micro-programs will be presented in Sections 4 and 5.

The ucode model state includes a program counter (*PC*), several sets of registers which implement the x86 ISA registers and flags, as well as configuration and micro-architecture specific registers used by the uops. The *PC* is a structure comprised of a prelude sequence of uops, a trap address into a fixed microcode ROM, and a set of additional side parameters to further refine the generated uops. The ucode state also includes a configuration object and a simple memory which extend the programmer-visible state in `x86isa`.

The ucode model also defines a state update function `ucode-model-step` outlined in Figure 4. The function `ucode-get-uop` constructs the next uop defined by the *PC* structure (either as a next uop in the prelude or retrieved from ROM if *PC* is a rom-address). The function `ucode-fetch-data` pulls the data from registers and memory needed for the uop computation. The function `ucode-exec` defines the computation of the given uop on the input data and returns the results of this computation (which can also include any execute-time exceptions). The function `ucode-next-pc` computes the next value for *PC* by either stepping through the prelude or updating the rom-address, depending on whether a branch was taken or not. A special address is used to signal a halt or completed micro-program. Finally, the `ucode-update-state` function stores all results from the uop execution in appropriate locations in registers and memory, as well as updates *PC* to *new-PC*.

The `ucode-exec` function defines the operational semantics of every uop supported in the micro-architecture. These uops include simple integer operations, more complex floating-point and vector operations, memory loads and stores, branching and jumps, along with many more. The full specification of the uops is beyond the scope of this paper but we cover a small selection of uops in some detail in Sections 4 and 5.

```

type ucode-model-PC is
  tuple ⟨prelude, rom-address, side-params⟩

type ucode-model-state is
  tuple ⟨PC, config, registers, memory⟩

func ucode-model-step (ustate) is
  uop    = ucode-get-uop(ustate.PC, ustate.config)
  data   = ucode-fetch-data(uop, ustate)
  results = ucode-exec(uop, data, ustate.config)
  new-PC= ucode-next-pc(ustate.PC, results)
  next-ust= ucode-update-state(results, new-PC, ustate)
  return next-ust

```

NOTE: The functions in bold are either verified against RTL blocks or use symbolic simulation of RTL blocks in their definition.

Figure 4: Ucode Model

The definition and use of the ucode model is critical to our approach. Where the `x86isa` model provides the specification for microprocessor verification, the ucode model provides a critical point of decomposition in the verification effort. The `ucode-exec` function is the specification for verifying the EXEC block of the microprocessor design. In contrast, `ucode-get-uop` and `ucode-next-pc` are defined by symbolic execution of the corresponding UCODE and XLATE RTL blocks. The composition of `ucode-model-step` applied to the sequence of uops generated by DECODE and XLATE and uops stored in ROM is then verified against the `x86-exec` function from `x86isa`. We detail more of this in Section 3.

2.4 Introduction to ACL2 and Supporting Tools

All of our work is done using the ACL2 theorem prover [2] with an interface to external SAT solvers. ACL2 is a theorem prover supporting proofs in an untyped first-order logic, with some support for higher-order logic styles of definition. ACL2 is written in and runs in Common Lisp environments, where compliant functions defined in ACL2 have compiled Common Lisp counterparts supporting efficient evaluation. In addition, ACL2 supports the intrinsic capability of defining functions in Common Lisp that generate other definitions and logical events. This support of macros in ACL2 is critical to our approach since we use them to generate and prove requisite lemmas from large data structures that codify x86 instructions and uop sequences, and automate ucode proofs. Section 6 will cover more of these uses of model definitions in automating proofs and building supporting tools.

In addition, we use many existing tools and libraries written in ACL2. We use the `VL` toolset [7, 4] to parse and translate microprocessor RTL definitions written in SystemVerilog into syntax trees. The `SV` library [5, 3] takes these syntax trees from `VL` and produces semantic next-state functions for signals in the design. Built on top of `SV` is a multi-cycle extraction tool `SVTV`[6]. `SVTV` supports the generation of function definitions in ACL2 that correspond to applying inputs to the design, stepping the design some specified number of clock cycles, and extracting relevant outputs along the way. For the design blocks we target, we have defined `SVTVs` that capture the effects of decoding a single instruction and executing a single instruction or uop. This allows us to view these blocks as the corresponding ACL2 function definitions. We use the names `SV-DECODE`, `SV-XLATE`, `SV-UCODE`, and `SV-EXEC` to represent the ACL2 input to output mapping functions derived from these `SVTVs`, and refer to them as the *SV design functions* as a group.

The `GL` tool [37, 36] is a verified prover of ACL2 theorems on finite domains. `GL` uses symbolic simulation of ACL2 function definitions to reduce finite ACL2 theorems into propositional logic formulas. These propositional logic formulas are then either proven with BDDs, or simplified using AIG algorithms and transmitted

to a SAT solver to check if they are true or produce a counterexample. The BDD and AIG algorithms are written in ACL2 and proven correct. We use external SAT solvers but the proofs from the SAT solvers are checked for validity. We make extensive use of GL, from proving the necessary correspondence between the SV design functions and the corresponding x86 and uop model functions, to defining tools for exploration of possible proofs and generation of constraints.

3 Overview

Our focus is verifying the correct operation of the RTL-level definitions implementing the DECODE, XLATE, UCODE, and EXEC blocks of the microprocessor. The primary function of these blocks in the microprocessor is the correct decode and execution of x86 instructions via correct uop execution and sequencing—these are the highlighted blocks from Figure 3 and correspond to the bolded functions in the x86 and ucode models. Our goal is to prove the correctness of a single instruction invocation through these RTL blocks from a generalized legal state with assumptions ensuring no interference or impedance to the execution of the instruction. Importantly, the focus on a single instruction invocation allows us to reduce the scope of the SV design functions we need for proofs, and this in turn dramatically reduces the times spent in GL for symbolic simulation or SAT solving. Also, the single instruction focus reduces the need to specify invariants, which in turn reduces the fragility of the proofs to changes in the RTL.

As stated before, we define all functions and prove all theorems in the ACL2 theorem prover. Our goal is to prove that the SV design functions correspond to `x86-isa` and `ucode` model functions. Our correctness statements for each of these blocks are defined in Figure 5, while Figure 6 depicts how these pieces fit together for checking single-instruction correctness.

```

theorem decode-correctness is
   $\forall (bytes, cfg):$ 
    get-instr(SV-DECODE(map-decode(bytes, cfg))) =
    x86-decode(bytes, cfg)

theorem xlate/ucode-correctness is
   $\forall (instr, state):$ 
    run-xlate/ucode(instr, state) =
    x86-exec(instr, state)

theorem exec-correctness is
   $\forall (uop, data, cfg):$ 
    get-results(SV-EXEC(map-exec(uop, data, cfg))) =
    ucode-exec(uop, data, cfg)

```

Figure 5: Correctness Statements for Target Blocks

The mapping functions `map-decode` and `map-exec` referenced in Figure 5 build input signal bindings for the SV design functions. The extraction functions `get-instr` and `get-results` pull out specification-level results from the output signals of the SV design functions. These `map` and `get` functions establish the correspondence of runs/steps in the design and in the model. Although the proofs for `decode-correctness` and `exec-correctness` are considerable (especially `exec-correctness`, which we will discuss in more detail in Section 3.1), their statements are fairly straightforward correspondence theorems. The property of `xlate/ucode-correctness` is less direct. The main reason for this is that the specification for the XLATE and UCODE design blocks is complex—XLATE and UCODE produce and run a micro-program to completion, the final results of which correspond to the results of `x86-exec`.

Before considering the definition of `run-xlate/ucode`, we first return to the `ucode` model and the functions which define `ucode-model-step` (see Figure 4). The functions `ucode-fetch-data` and `ucode-update-state` involve accessing and updating register banks and memory, and correspond to blocks of the microprocessor we do not target. As such, we simply define models of these behaviors as functions in ACL2. We also define the function `ucode-exec` as a model of the EXEC block of the microprocessor, but for each uop, we

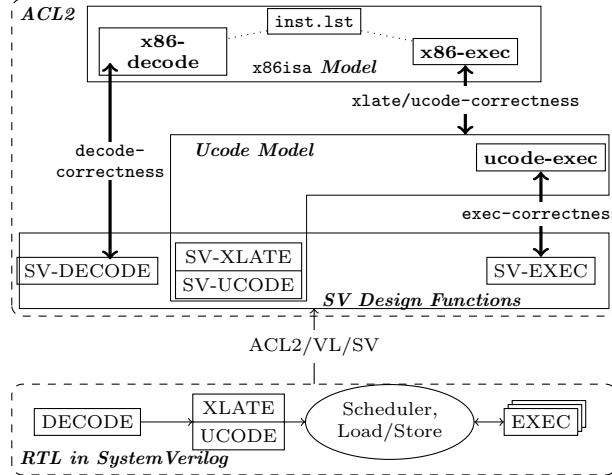


Figure 6: Top-level Correctness Decomposition

prove that the EXEC block is consistent with ucode-exec in **exec-correctness**. For the definitions of the functions ucode-get-uop and ucode-next-pc, we use the SV-UCODE design function directly. Specifically, we split the SV-UCODE design function into SV-UCODE-READ, which is responsible for creating uops in the design for a given PC, and SV-UCODE-STEP, which determines the next rom-address, depending on branching. These ucode-model functions are defined in Figure 7.

```

func ucode-get-uop (PC, config) is
  if in-prelude(PC) then first-prelude(PC)
  else
    get-uop(SV-UCODE-READ(map-uc-read=(PC, config)))

func ucode-next-pc(PC, results) is
  if in-prelude(PC) then remove-first-prelude(PC)
  else
    get-pc(SV-UCODE-STEP(map-uc-step=(PC, results)))

```

Figure 7: Ucode Model Using SV-UCODE functions

The definition of the function run-xlate/ucode is then provided in Figure 8. **run-xlate/ucode** first calls SV-XLATE (with map and get functions to transfer from specification objects to design signals and back) to generate the initial pc for ucode execution, and then calls **run-ucode-model**, which executes **ucode-model-step** until a halt address is reached and the instruction results are extracted.

Our direct use of SV-UCODE and SV-XLATE design functions in defining the ucode model **run-xlate/ucode** has the significant benefit of not needing to define and maintain a separate specification for these blocks. This is important because the operation and even interfaces of these blocks (and especially microcode ROM itself) are fairly complex and can change rapidly during design iteration. This approach does introduce some challenges when we attempt proofs. In particular, in order to automate proofs of **xlate/uicode-correctness**, we will need to have the successive expansions of the **ucode-model-step** function not cause a significant explosion in either the symbolic simulation of GL or the subsequent calls to SAT solvers. We assist this process by identifying criteria that are required of earlier steps in GL processing to keep subsequent steps from being too expensive. One particularly important criteria is that we have sufficient constraints on the instruction input to **run-xlate/ucode** to ensure that the (symbolic) PC produced by SV-XLATE has a sequence of fixed uop opcodes (i.e., data, source, destination fields may be symbolic, but the actual uop operations are fixed). We implement these criteria as prechecks before attempting to submit a **xlate/uicode-correctness** lemma instance.

We finally tie together the target specifications from Figure 5 to state our single-instruction correctness

```

func run-ucode-model (ustate) is
  if is-halted(ustate) then ustate
  else
    run-ucode-model(ucode-model-step(ustate))

func run-xlate/ucode(instr, state) is
  cfg = state.config
  pc = get-init-pc(SV-XLATE(map-xlate(instr, cfg)))
  ustate = make-ucode-state(pc, state)
  ustate' = run-ucode-model(ustate)
  results = extract-instr-results(ustate')
  return results

```

Figure 8: `run-xlate/ucode` function

goal in Figure 9. The function `run-xlate/ucode-RTL` is the same as `run-xlate/ucode`, except that `ucode-exec` is replaced with the RTL equivalent `get-results(SV-EXEC(map-exec(uop, data, cfg)))` from `exec-correctness`. We state (informally) that `single-instruction-correctness` captures the requirements for verifying correct instruction operation in the target RTL blocks while making reasonable (unverified) assumptions on the other RTL blocks in the microprocessor.

```

func x86-step(bytes, state) is
  ⟨dx, instr⟩ = x86-decode(bytes, state.config)
  ⟨ex, rslts⟩ = x86-exec(instr, state)
  return ⟨dx, ex, rslts⟩

func rtl-step(bytes, state) is
  cfg = state.config
  ⟨dx, instr⟩ = get-instr(SV-DECODE(map-decode(bytes, cfg)))
  ⟨ex, rslts⟩ = run-xlate/ucode-RTL(instr, state)
  return ⟨dx, ex, rslts⟩

theorem single-instruction-correctness is
   $\forall (\text{bytes}, \text{state}):$ 
    x86-step(bytes, state) = rtl-step(bytes, state)

```

Figure 9: Single-Instruction Execution Specification

3.1 Verification of Uops: exec-correctness

Formal models of microprocessors usually include an effect of one step of the machine on the state. The granularity of the step depends on the objective of verification. In our case, the smallest step we consider is an execution of one uop. In most published work, operational semantics is created by authors of the model and validated (or not) by testing. Our work differs from that approach in the fact that our operational semantics of uops provably reflects behavior of the underlying design, with the exception of uops which implement loads and stores—we only model memory accesses. In particular, we prove the property called `exec-correctness` (see Figure 5), i.e., each uop that can be dispatched to an execution unit will produce the effects described by our uop specification functions. These same uop specifications are also used in proving `xlate/ucode-correctness` — this allows us to replace a uop’s RTL implementation with these specifications while doing microcode proofs. Figure 6 shows the central position these specification functions occupy in our project.

The formal verification of `exec-correctness` is an ongoing effort at Centaur Technology that has been developed over several years. This work is usually done early on in the design cycle for catching functional

bugs in the execution units as soon as possible. To illustrate the scale of this effort, we proved `exec-correctness` for about 600 uops for our last project. In these proofs, we allow multiple uops in the pipeline, and only assume that the scheduler worked as intended, i.e., did not schedule uops that would collide while writing results. For instance, a uop with a 4-cycle latency cannot be scheduled to start on the same unit one cycle after a uop with a 3-cycle latency was dispatched to the unit.

3.1.1 Uop Specifications

Uops are dispatched by a scheduler to individual execution units along with all the information that the unit may need to process them—e.g., opcode, source and destination sizes, operands, immediate values, etc. At this point, the facts of where the operands came from and where the result is heading are abstracted away. The core of operational semantics for every uop is a function that operates on data types defined in the ISA: integers (signed and unsigned), packed integers, floating point numbers, bit vectors, etc. The results are again those data types and (where appropriate) bit vectors representing exceptions.

In Figure 10, we present the specification of a very simple integer uop, logical AND, with flag computations elided for presentation’s sake. We will see this uop again in Section 4 (specifically, in Table 2) later.

```
func and-spec(input) is
  ⟨srcsize, dstsizeget-sizes(input)
  ⟨operand1, operand2get-operands(input)
  arg1 = truncate(srcsize, operand1)
  arg2 = truncate(srcsize, operand2)
  result = truncate(dstsize, arg1 & arg2)
return result
```

Figure 10: `and-spec`: Specification Function for logical AND

3.1.2 Uop Correctness Proofs

At Centaur Technology, we have been doing formal verification of execution units—those that perform logical, integer, SIMD integer, and floating-point operations—for over a decade [19, 9, 14]. Formal methods have proven to be very effective in the validation of data-path intensive designs. With the size of operands in the x86 ISA growing to 512 bits, targeted or random simulations have become inadequate to cover arithmetic and cryptographic units. At the same time, increasing capacity of SAT solvers and evolving symbolic simulation methods make the verification of these units mostly automatic. About ten percent of the uops and operations still require human assistance, e.g., various flavors of square-root and divide, and floating-point add, multiply, and fused multiply-add.

All uop proofs are done within the scope of the EXEC hardware block, which includes execution units for all types of operations as sub-modules. The benefit of working with the entire EXEC block instead of a specific sub-module for uop verification is that the hardware interface of EXEC changes a lot less frequently than that of the sub-modules, which allows us to write our specification functions in a uniform way. The formal model of EXEC is rebuilt automatically every time the hardware design changes, and thus, the resulting SV design functions are always up-to-date.

The SV design functions corresponding to EXEC are amenable to various ACL2 proof techniques, like symbolic simulation and propositional logic checks using GL. Since almost all uops have fixed latency (i.e., the computation finishes in a fixed number of clock cycles), symbolic simulation is our main tool to prove their correctness. GL symbolically executes the given module, extracts results and flags, and compares them to the corresponding specification function. In most cases, this happens without any user intervention. If a function is too complex to be processed by GL directly, the user can split it into cases (e.g., for floating-point adders), or SV-EXEC can be decomposed into several runs of the sub-units (e.g., for multipliers). For both these cases, the decomposition is proven correct using ACL2. For cases like dividers that have variable latency, more sophisticated proof techniques are needed—e.g., the user has to guide ACL2 by discovering and proving inductive invariants.

We present what **exec-correctness** looks like for a simple uop, logical AND, in Figure 11 where **and-spec** is the specification of AND in **ucode-exec**.

theorem and-exec-correctness is

```
 $\forall (input):$ 
get-results(SV-EXEC(map-exec(AND, input))) =
and-spec(input)
```

Figure 11: Logical AND Correctness Theorem

As a part of establishing **exec-correctness**, our proof scripts automatically generate documentation in HTML about the proof's details and coverage. This way, the documentation is always in sync with the proofs, and accessible to hardware designers.

4 Illustrative Example: Verifying SHRD

In this section, we describe in detail how we verify the implementation of the instruction **SHRD**, which stands for “shift-right double”. In the following section (Section 5), we briefly describe **VPSHRDQ**, which is another instruction that performs “shift-right double” but on *packed data*, to illustrate how we handle more complicated instructions.

SHRD, along with its shift-left counterpart **SHLD**, was introduced to aid bit string operations, and is supported on all Intel processors since the 80386. These instructions manipulate general-purpose x86 registers, memory (depending on the variant used), and the rflags register.

4.1 Instruction Specification

The **SHRD** instruction takes three operands:

```
SHRD <destination>, <source>, <shiftAmt>
```

The x86 ISA provides two variants of this instruction: in one, the **shiftAmt** is an 8-bit immediate and in the other, it is the 8-bit wide CL register. For both of these variants, the **destination** can either be a register or memory operand, the source must be a register, and the **destination** and **source** must be of the same size: either 16, 32, or 64 bits.

This instruction behaves as follows: the **destination** is shifted right by a value indicated by **shiftAmt** (which is masked appropriately, as dictated by the instruction size) and the resulting empty bit positions are filled with bits shifted in from the **source** (least-significant bit first). **SHRD** can also affect flags; though we do specify and verify flag computations, we omit flag-related discussions for the sake of brevity. The specification function of **SHRD** from **x86-exec** in our **x86isa** model is called **shrd-spec**. This function describes the core operation of **SHRD**, without dealing with machine aspects like operand addressing and fetching.

In this paper, we focus on the following **SHRD** variant:

```
variant: SHRD RCX, RDX, <imm8>
bytes: 0x48 0xF0 0xAC 0xD1 <imm8>
```

That is, it is a 64-bit **SHRD** instruction whose destination is a register and which takes an immediate byte as the **shiftAmt** operand. See Figure 12 which shows a concrete run of this variant.

| <i>— Initial Values —</i> | |
|------------------------------|------------------------------|
| RDX := 0x1122_3344_5566_7788 | RCX := 0x0123_4567_89AB_CDEF |
| <i>— Final Values —</i> | |
| RDX := 0x1122_3344_5566_7788 | RCX := 0x7788_0123_4567_89AB |

Figure 12: SHRD RCX, RDX, 16

4.2 Microcode Implementation

On one of Centaur Technology’s processor designs, the variant of SHRD we are interested in has two prelude uops, followed by a trap to a routine in the microcode ROM labeled `ent_shrdEvGv_64reg`. The microcode ROM contains a compact representation of the uops in order to conserve space. As such, there is an additional step required to obtain the uops corresponding to a ROM instruction—the *microsequencer* hardware block (the part of UCODE defining SV-UCODE-READ) is responsible for translating this compact representation to a uop sequence. The microprogram corresponding to this variant is presented in Tables 1 and 2; uops that compute only the flag values are elided. In the second column of these tables, we show the contents of the relevant state components for the concrete run shown in Figure 12.

Table 1: SHRD: Prelude Uops

| Uop | Concrete Run & Description |
|----------------------------------------------------------|-------------------------------------------------------------------------------------|
| <code>MOVSX G2, RCX</code> (SSZ: 64; DSZ: 64) | $G2 \leftarrow 0x0123_4567_89AB_CDEF$ <i>Move RCX to internal register G2</i> |
| <code>MOVZX G3, <imm8></code> (SSZ: 8; DSZ: 64) | $G3 \leftarrow 16$ <i>Move immediate to internal register G3</i> |

Table 2: SHRD: Uops in `ent_shrdEvGv_64reg`

| | |
|------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| <code>AND G3, G3, 63</code> (SSZ: 8; DSZ: 64) | $G3 \leftarrow 16$ <i>Mask immediate operand</i> |
| <code>MOV G10, -1</code> (SSZ: 64; DSZ: 64) | $G10 \leftarrow 0xFFFF_FFFF_FFFF_FFFF$ <i>Move -1 to internal register G10</i> |
| <code>JE G3, 0, ent_nop</code> (SSZ: 16; DSZ: 16) | No jump taken <i>Jump to routine ent_nop if G3 == 0</i> |
| <code>SUB G5, 0, G3</code> (SSZ: 32; DSZ: 32) | $G5 \leftarrow 0xFFFF_FFF0$; $ZF \leftarrow 0$ <i>Store -G3 in internal register G5; clear the zero flag because result is non-zero</i> |
| <code>SHR<ZF> G10, G10,</code> G5 (SSZ: 64; DSZ: 64) | $G10 \leftarrow 0xFFFF$ <i>Shift G10 right by (G5 & 63) if ZF == 0</i> |
| <code>AND<ZF> G10, G10, 0</code> (SSZ: 64; DSZ: 64) | $G10 \leftarrow 0xFFFF$ <i>Set G10 to 0 if ZF == 1</i> |
| <code>AND G6, RDX, G10</code> (SSZ: 64; DSZ: 64) | $G6 \leftarrow 0x7788$ <i>Store (RDX & G10) in internal register G6</i> |
| <code>SHR G7, G2, G3</code> (SSZ: 64; DSZ: 64) | $G7 \leftarrow 0x0000_0123_4567_89AB$ <i>Store (G2 >> G3) in G7</i> |
| <code>SHL G2, G7, G3</code> (SSZ: 64; DSZ: 64) | $G2 \leftarrow 0x0123_4567_89AB_0000$ <i>Store (G7 << G3) in G2</i> |
| <code>OR G2, G2, G6</code> (SSZ: 64; DSZ: 64) | $G2 \leftarrow 0x0123_4567_89AB_7788$ <i>Store (G2 G6) in G2</i> |
| <code>ROR G7, G2, G3</code> (SSZ: 64; DSZ: 64) | $G7 \leftarrow 0x7788_0123_4567_89AB$ <i>Rotate G2 right by G3 and store result in G7</i> |
| <code>OR RCX, G7, G7</code> (SSZ: 64; DSZ: 64) | $RCX \leftarrow 0x7788_0123_4567_89AB$ <i>Store the result of G7 G7 in RCX</i> |

Most ucode programs are written somewhat differently from software programs that would implement the same behavior. For instance, in Table 2, the final result is moved from the internal register `G7` to `RCX` using a *logical or* operation, instead of a more “natural” *move* operation. Such choices are deliberate—microcode programmers carefully pick uops that either have a lower latency or that reduce code and data dependencies to aid re-ordering. The code is also written in such a way so it can be re-used for several variants of an instruction, e.g., with different sizes of immediate field. Also, adding new uops is non-trivial, so pre-existing uops have to be creatively used to implement new instructions. The upshot is that all of this definitely needs functional verification!

```

Mnemonic: SHRD   Opcode: 0x0F_AC
Operands: OP1 := [ModR/M.r/m GPR MEM]
          OP2 := [ModR/M.reg GPR]
          OP3 := [IMM8]
Exceptions: #UD: if LOCK prefix used
             #GP(0): if the memory address is non-canonical
             ...

```

Figure 13: Entry for SHRD in `inst.1st`

4.3 Correctness Proof

As described in Section 2, establishing an instruction’s correctness involves proving that the DECODE, XLATE/UCODE, and EXEC hardware operate correctly. We discuss the first two for our `SHRD` example below; a general discussion of EXEC verification is in Section 3.1. First, we cover a critical data structure we use in both the `decode-correctness` and `xlate/ucode-correctness` proofs.

4.3.1 `inst.1st` Data Structure

A key part of our framework is `inst.1st` [32], which is a data structure that we defined in the `x86isa` model that contains a listing of x86 instructions supported (or slated to be supported) by x86 processors, including all the information needed to decode and dispatch these instructions. The entry for `SHRD` is depicted in Figure 13. `Operands` describes the three operands of `SHRD`; for instance, the first operand is obtained from the reg field of the ModR/M byte of `SHRD`’s byte sequence¹, and it can either be a general-purpose register or a memory operand. `Exceptions` lists the exceptions that `SHRD` can throw, either at decode- or execute-time.

4.3.2 DECODE

The proof of `decode-correctness` for any instruction (including `SHRD`) entails showing that the incoming bytes are parsed and mapped to an appropriate and valid instruction data structure or the correct exception is returned. The `x86-decode` specification function defines these behaviors and we split `decode-correctness` into checking these two pieces.

For the parsing of the incoming bytes, `x86-decode` pulls apart the byte sequence to determine the proper components and fills in the instruction data structure accordingly (assuming that the maximum limit of 15 bytes is not exceeded). The `inst.1st` structure is consulted during this parsing to determine if certain bytes (e.g., the ModR/M or immediate data) are expected.

In addition to byte parsing, we also check if exceptions are thrown when the instruction is illegal. An exception may be thrown for a variety of reasons, from simple parsing violations (e.g., more than 15 bytes) to instructions which are illegal in the current state or configuration (e.g., an instruction which requires a certain priority level). Beyond some basic parsing exceptions, the exception checks are defined in the `inst.1st` structure. The `x86-decode` function filters out these exceptions to only include exceptions which are checked at decode time, as some exceptions are checked during UCODE, or during EXEC, or in a block which we do not currently target (e.g., page faults). For `SHRD`, only one exception specifier from `inst.1st` makes it through this filter: the exception which fires if a LOCK prefix exists. Additionally, we generate and perform a check which ensures that any byte sequence that does not map to any entry in `inst.1st` will lead to an illegal instruction exception being thrown.

After case splitting on opcode value and a few internal parameters of the DECODE block, the resulting generated lemmas go through processing of GL and SAT solving in a range of 5 to 10 seconds. This case split produces a few thousand cases to check across all opcodes, but these can be checked in parallel across multiple machines.

¹Though not explicitly listed here, we also account for *field extensions*. For instance, the REX.R bit (if present) is used along with the 3-bit ModR/M.reg field when 4 bits are needed for operand addressing.

4.3.3 XLATE/UCODE

After verifying that our hardware throws decode-time exceptions for illegal byte sequences of `SHRD` when the `x86isa` model mandates it should (and no such exceptions in all other cases), we need to prove that legal byte sequences of `SHRD` perform the intended operation on our hardware. To this end, we have developed a framework that enables us to pick and automatically populate the candidate instruction for verification. By using SV-DECODE, we ensure that this population is done with provably legal values (concrete and/or symbolic) corresponding to this candidate—that is, these values are known to not cause any decode-time exceptions. These values are then passed to XLATE/UCODE, and thus, we obtain a verification target for these blocks that is consistent with the candidate instruction. This process is described below using `SHRD`, our running example.

We provide the following input to this framework in order to pick the relevant variant of `SHRD`:

```
Mnemonic: SHRD    Opcode: 0x0F_AC
Variant:   Size := 64; OP1 := GPR
Mode:      64-bit mode
Indices:   OP1 := RCX; OP2 := RDX
Symbolic:  OP3
```

The first line is used to find the appropriate entry for `SHRD` in `inst.lst`, which gives us information about the arity and kinds of operands of this instruction. The second line lets us pick the right variant by specifying the operation width, 64, and that the first operand should be a register, not a memory location. The third line picks the machine configuration, and the fourth line picks the registers for the first two operands—note that the registers’ indices are fixed, not their contents. The last line instructs the framework to pick a symbolic value for the third operand, i.e., the immediate byte that specifies the shift amount. All of this information is used to partially populate the instruction-related data structures corresponding to those used in the RTL. We then submit the following conjecture to be bit-blasted in GL using a SAT decision procedure:

When SV-DECODE is given these partially-filled instruction structures as input, an exception is detected.

If this conjecture is indeed a theorem, then likely there is some contradiction in the way the framework is instructed to pick the variant and one would need to revisit that. Otherwise, GL/SAT will produce a counterexample²; i.e., concrete assignments to variables in the partially-filled instruction structures that do *not* lead to any exceptions. This gives us any assignments that were missed during the partial population of the structures. We then generalize this counterexample to ensure that we symbolize those parts of the instruction variant that we care about (e.g., the immediate operand for this `SHRD` variant). In this manner, we obtain appropriately populated instruction structures that are consistent with our candidate `SHRD` variant. Typically, this entire process takes around 10 seconds, even for instructions with a more complicated encoding, like AVX512 instructions.

These legal structures are then passed through our ACL2 model of the XLATE and UCODE blocks (SV-XLATE and SV-UCODE), which gives us the micro-op program—that is, the prelude uops (i.e., uops generated by the translator) and if there is a trap to the microcode ROM, the address of the ucode routine. As mentioned in Section 3, it is important for the uops’ opcodes to be fixed, though they can contain symbolic data (like the immediate byte in our `SHRD` example), and even source/destination indices (which are fixed to `RCX` and `RDX` in this example).

We then attempt to prove that the single-instruction correctness property holds for all relevant executions of this micro-op program. That is, the effects produced by the function `shrd-spec` on the ISA-visible components of the ucode state are the same as those produced by the implementation (i.e., uops’ execution), provided that the arguments of `shrd-spec` correspond to the instruction’s operands. These kinds of proofs can be done by techniques like the clock functions and step-wise invariants approaches [27, 29, 30], and decompilation-into-logic [25, 1], all of which could employ either GL/SAT’s automatic bit-blasting and/or ACL2 rewriting. The central idea though is symbolic simulation of these uops on our ucode model. For the prelude uops, the ucode model’s update function directly dispatches control to the appropriate semantic functions. For the ucode routine, the update function dispatches control to our microsequencer model, which

²Partial population of the instruction structure makes the SAT problem tractable.

translates the ROM representation into uops; the resulting uops are then simulated in a manner similar to the prelude uops.

It is important to mention that being able to get the prelude uops and the trap address automatically in this manner enables us to keep up with the constantly-changing RTL. For non-major “everyday” changes in the RTL (e.g., if the ROM addresses change or if the uops use different internal registers), our proofs usually work without any intervention on our part.

Note that a single microcode routine is often used to implement many different variants of the same (or a different but similar) instruction. For instance, `ent_shrdEvGv_64reg` is also used to implement the SHRD variant which uses the CL register for `shiftAmt`. Effort duplication in single-instruction verification can be avoided by proving the sub-routine correct once for all the variants that use it. However, there is some merit to not decomposing the proof in this manner; such a decomposition would involve specifying the preconditions for the routine’s correctness, which may change if the RTL design changes. Therefore, the decision to do proof decomposition can be taken on a case-by-case basis—we typically do not perform decomposition for instructions whose proofs can be automated efficiently using bit-blasting procedures, and we typically perform decomposition for instructions whose microcode routines need some interactive theorem proving.

5 AVX512 Example: Verifying VPSHRDQ

The `VPSHRDQ` instruction is an AVX512 instruction intended to be a part of future Intel processors [21] (as of this writing). It operates on XMM/YMM/ZMM registers and memory (depending on the variant used), and does not affect any flags.

5.1 Instruction Specification

Intel’s AVX512 instructions are their most recent SIMD (Single Instruction on Multiple Data) instructions—i.e., the same operation is performed on multiple data elements. `VPSHRDQ` operates on 64-bit elements, and therefore, can be thought of as the vector version of the 64-bit `SHRD`. However, `VPSHRDQ` is a much more complicated instruction than `SHRD`, in part due to its more complex instruction encoding and exception checking, and a wider data path. Additionally, AVX512 instructions have some features that are not supported by general-purpose instructions. For instance, by default, all data elements in the operands are processed, but one can specify an *opmask register* that allows their conditional processing. That is, operation for data element n is performed only if bit n of the opmask register is set. If this bit is not set, then either the n -th element of the destination retains its original value (*merge-masking*; default) or it is zeroed out (*zero-masking*), based on the masking mode specified in the instruction.

The `VPSHRDQ` instruction takes four operands:

```
VPSHRDQ <destination>[<Opmsk>][<MaskingMode>],  
        <source1>, <source2>, <ShiftAmt>
```

Depending on the `<Opmsk>`, the shift-right double operation is performed on 64-bit elements in `<source1>` and `<source2>` (akin to the first two operands of `SHRD`), and the result, depending on `<MaskingMode>`, is either merge- or zero-masked, and stored in the `<destination>` operand.

We focus on the 512-bit variant of `VPSHRDQ` that has ZMM registers as the first three operands:

```
variant: VPSHRDQ ZMM1<OpmskReg><MaskingMode>,  
           ZMM2, ZMM3, <imm8>  
bytes: 0x62 0xF3 0xED 0b<MaskingMode>1001<OpmskReg>  
       0x73 0xCB <imm8>
```

Like in `SHRD`, the immediate byte is symbolic, and additionally, we symbolize the opmask register and the masking mode. This allows us to reason about multiple invocations of the same instruction variant in one proof—of course, we still require that this variant translates to a fixed set of uop operations, for reasons previously described in Section 3.

5.2 Microcode Implementation

On one of our processors, the 512-bit version of **VPSHRDQ** is implemented by performing the requisite operations separately on the low and high 256 bits of the data and then combining the results. This variant has five prelude uops and then it traps to a microcode ROM routine labeled **avx_double_shift_or_q**; the microprogram is listed in Tables 3 and 4. Note that though **VPSHRDQ** has fewer uops than **SHRD**, the behavior of each of its uops is significantly more complicated than those of **SHRD**.

Table 3: **VPSHRDQ: Prelude Uops**

| Uop | Description |
|--------------------------------------------------|---------------------------------------------------------------------------------------------------|
| DLSHFTCNT T26, <IMM8> (SSZ: 256, DSZ: 256) | <i>Store the left shift count (for operand ZMM3) in internal register T26</i> |
| PSRLQ T27, ZMM2L, <IMM8> (SSZ: 256, DSZ: 256) | <i>Store packed-right-shift result for the low 256 bits of ZMM2 in internal register T27</i> |
| PSRLQ T29, ZMM2H, <IMM8> (SSZ: 256, DSZ: 256) | <i>Store the packed-right-shift result for the high 256 bits of ZMM2 in internal register T29</i> |
| PSLLVQ T28, ZMM3L, T26 (SSZ: 256, DSZ: 256) | <i>Store the packed-left-shift result for the low 256 bits of ZMM3 in internal register T28</i> |
| PSLLVQ T30, ZMM3H, T26 (SSZ: 256, DSZ: 256) | <i>Store the packed-left-shift result for the high 256 bits of ZMM3 in internal register T30</i> |

Table 4: **VPSHRDQ: Uops in avx_double_shift_or_q**

| | |
|----------------------------------------------------------------------|-----------------------------------------------------------|
| PORQ ZMM1L, T27, T28, <MaskMode>, <Opmsk> (SSZ: 256, DSZ: 256) | <i>Store (T27 packed-or T28) in low 256 bits of ZMM1</i> |
| PORQ ZMM1H, T29, T30, <MaskMode>, <Opmsk> (SSZ: 256, DSZ: 256) | <i>Store (T29 packed-or T30) in high 256 bits of ZMM1</i> |

5.3 Correctness Proof

Our verification approach, previously illustrated using **SHRD**, works well even for complex AVX512 instructions like **VPSHRDQ**. The verification of **VPSHRDQ**'s XLATE/UCODE part is very similar to that for **SHRD** (Section 4.3.3), so we omit a description here. However, the **decode-correctness** proofs of **SHRD** and **VPSHRDQ** do differ (primarily in exception checking). Another difference between the proofs of **SHRD** and **VPSHRDQ** lies in the unit-level verification of the uops themselves (Section 3.1)—since **VPSHRDQ**'s uops are more complicated to implement in the EXEC RTL, the **exec-correctness** proofs were also more involved.

5.3.1 DECODE

The **decode-correctness** proof for **VPSHRDQ** is again broken into two pieces: correctness of the parsing of the incoming bytes and the exceptions thrown. The check that parsing is correct is generated and checked in the same way as it is for **SHRD**—the incoming bytes are split based on which prefixes, opcode bytes, and postfix bytes are included in the instruction. The generated checks cover all variants of **VPSHRDQ**.

Similar to **SHRD**, the exception checks for **VPSHRDQ** are also derived from the **inst.1st** entry for **VPSHRDQ**. But, in the case of EVEX instructions, the exception checking is more extensive. There are several EVEX “type-checks” ensuring that proper values are set for the EVEX instruction fields. For example, **VPSHRDQ** uses the **<Opmsk>** and the **<MaskMode>**, but this means it cannot have an **<Opmsk>** of 0 (which represents no mask) with the **<MaskMode>** set to zero-masking. The “types” of the instructions are defined in the **inst.1st** structure, and the appropriate checks are codified in **x86-decode**. The resulting lemma proofs in GL and SAT are still quickly proven per case within a few seconds.

6 Debugging and Automation

All of our formal models—RTL, ucode, and `x86isa`—and specification functions are directly executable in ACL2. This affords us the capability to run concrete tests in the exact same framework in which we do our proofs. This is useful to not only validate our models efficiently, but also to run preliminary tests before we embark on verifying the implementation of an instruction. If any of our proofs fail, we can examine the failure by executing the offending case on our models immediately—this way, we can quickly weed out any potential failures due to, say, an incomplete specification. Additionally, if a proof succeeded under certain conditions but not others, we can provide useful feedback to the logic designers by giving them concrete ISA-level test cases that are representative of successful and unsuccessful runs. In case of EXEC, we can generate waveforms that exhibit behaviors that are interesting to the logic designers.

As already discussed in Section 3.1.2, we have been automating unit-level uop proofs at Centaur Technology for over a decade now. Proving `exec-correctness` for most of the uops is completely automatic now using GL. For `decode-correctness` proofs, we automatically generate a set of lemmas which cover `decode-correctness` for each possible opcode value (there are roughly a thousand legal opcode values). We check that all ill-formed incoming byte sequences cause an exception and all well-formed byte sequences will emit a valid opcode. For each opcode, we generate checks that correspond to each entry in the `inst.1st` data structure. We also generate a check that for any valid opcode, if no such entry matches the incoming byte sequences, then an exception is generated.

An example of the benefits of automation that are afforded to us by our use of ACL2 and GL was discussed earlier in this paper—we need to have a fixed uop opcode sequence in the generated PC descriptor from SV-XLATE. This property is one of several that we have diagnosed as a prerequisite for ensuring that the subsequent GL checks will be efficient. In our framework, we generate a partially-symbolic PC for a given instruction specification from the user. We not only check that the uop opcode sequence is fixed in the generated PC resulting from SV-XLATE, but we also generate a check to ensure that no other uop opcode sequence is possible for the given instruction specification. In this case, we use a counterexample from a GL check to partially fill-in a PC value and then generate a second GL check to ensure that this PC is not over-constrained for subsequent proof attempts.

The scale of our problem necessitated the development of utilities that automate many of the tasks common to every instruction; ACL2’s macros are indispensable in this respect. We discussed an example of this in Section 4.3.3, where we described our framework that picks the right candidate instruction and automatically populates legal instruction structures. Another example is the use of macros to generate lemmas for decoding ROM instructions during ucode proofs. Even the generation of most of our proof statements is automated. For instance, we have utilities that generate the `xlate/ucode-correctness` statement for AVX512 instructions. Many instruction implementations are amenable to automated verification using GL/SAT bit-blasting (e.g., those that have no loops in their microcode routines). Automating the proof of `xlate/ucode-correctness` involves careful orchestration of the rewrite mechanism by using different theories in different phases of the proof.

7 Related Work

Formal verification of microprocessors is an active area which has a long history (since the early 1970s), both in academia and industry. There is an extensive list of previous efforts in microprocessor verification (in x86 or otherwise), but most are not directly applicable here, given the scope of what we address in both the scale of industrial processor design and/or the particular requirements of implementing the x86 ISA. As such, we mention only those works that have goals and/or approaches that are similar to ours and that focus on industrial processors.

The most recent report of mainstream commercial use of formal analysis techniques to verify processor designs described how pipeline control verification was done using bounded model checking on ARM processors [26]; the formal verification framework here could also make use of data-path verification results obtained using other formal techniques. This work is impressive, but both its focus and approach are different from ours. There has been a significant amount of work done to verify a processor’s core execution units [11, 24, 14]. Symbolic simulation has been used to analyze microcode at Intel since 2003 [10, 18, 15]. Their formal tool, MicroFormal, is based on SAT/SMT, and was used to verify assertions and perform

backward compatibility checks of microcode. Work has also been done to verify whether an x86 instruction decoder marks the boundaries of instructions correctly [12, 13, 22]; however, this work did not account for checking decode-time exceptions. Further, the x86 ISA is far more extensive and complex now.

In contrast to these approaches, we extended our prior work on the execution unit [19, 9] as well as microcode [17] verification, and along with proofs about instruction decoding and composition of uops, are able to prove instruction implementations correct. Notably, the operational semantics of uops provably reflects behavior of the RTL design. Other complex parts of processors have also been formally verified; e.g., register renaming and BUS recycle logic [23], memory systems [34], etc. Though we do not focus on such hardware blocks yet, we plan to tackle them in the future.

8 Conclusions

In this paper, we presented our progress towards comprehensive formal verification of x86 processors designed at Centaur Technology. We built upon our previous work on unit-level uop verification, enhanced it, developed a framework to verify the DECODE, XLATE/UCODE, and EXEC blocks, and composed all these critical pieces together to prove instruction implementations correct.

The approach presented in this paper is capable of finding bugs in the decode, translate, microsequencer, and execution units, in addition to microcode programs and the assembler that translates ROM representations to uops. Though we have not been working on this project for long, we have already found some bugs that could not have been found by unit-level uop verification alone. For example, we found instances where uops had “don’t-care” as a source when they should have had the concrete value 0. We also found a case when one of many variants of an AVX512 instruction did not take the opmask register into account. Another example was that some AVX512 instructions did not raise a decode-time exception when expected (e.g., some instructions throw an exception if the non-default masking mode is chosen). There are bugs that our approach can indeed miss—e.g., in the memory system (load/store uops, caches), scheduler (operand dependency and forwarding), register mapping, pipelining (interactions among instructions), etc.

Our approach affords us the ability to adopt a divide-and-conquer strategy for verifying instruction implementations. Unit-level uop verification and DECODE proofs can be done independently of each other. Similarly, we do not need to have the instruction and/or uop specifications beforehand—as soon as we add an instruction’s information to `inst.1st`, we can start verifying the DECODE part. All these proofs use the same formalisms; thus, we avoid any potential inconsistencies that may arise due to translation between different tools/representations. Moreover, it is straightforward to compose the individual lemmas into a top-level theorem.

A key advantage of our approach is that we do not need to write specifications for (or even understand) how legal instruction bytes are mapped to instruction structures, which are then translated to uops. Also, our approach is immune to even major changes in the microcode ROM representation format, and consequently, we are not affected by changes in the microcode ROM assembler. This is facilitated by being able to simulate the relevant hardware blocks by using our formal models of the SystemVerilog designs. Also, our uop specifications are validated not only when we prove correspondence with EXEC, but also when we check that the micro-programs using these uops correspond to the x86 instruction specifications.

All of our work is done using the ACL2 theorem proving system (with SAT used for bit-blasting), and we contribute to hardware verification libraries that are available publicly. Being able to analyze commercial x86 designs using tools that are available freely reinforces our view that formal verification of hardware in the industry is possible without needing to purchase expensive license-only tools. That being said, our approach does not preclude the use of such commercial tools; for instance, one can use off-the-shelf Boolean reasoning engines as backends for GL.

The presented effort is a work in progress. While we are close to achieving complete verification of execution units and the decoder, we have done single-instruction correctness proofs of only a small subset of the 3400 x86 instruction variants in this described framework. We have a lot of plans for future work, both for short and long term. As far as verifying instruction implementations is concerned, we are working on adding automation for tasks like proof coverage analysis. We are also planning to extend our verification scope by focusing on other parts of the processor, like the memory system and scheduler.

References

- [1] ACL2 Books: Codewalker. Online, Accessed: December 2019.
<https://github.com/acl2/acl2/tree/master/books/projects/codewalker>.
- [2] ACL2 Home Page. Online, Accessed: December 2019.
<http://www.cs.utexas.edu/users/moore/acl2>.
- [3] Documentation of SV: A Hardware Verification Library. Online, Accessed: December 2019.
http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2___SV.
- [4] Documentation of VL Verilog Toolkit. Online, Accessed: December 2019.
http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2___VL.
- [5] SV: A Hardware Verification Library. Online, Accessed: December 2019.
<https://github.com/acl2/acl2/tree/master/books/centaur/sv>.
- [6] SVTV: A Structure for Simulation Pattern of a Hardware Design. Online, Accessed: December 2019.
http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2___DEFSVTV.
- [7] VL Verilog Toolkit. Online, Accessed: December 2019.
<https://github.com/acl2/acl2/tree/master/books/centaur/vl>.
- [8] x86isa Library in the ACL2 Community Books. Online, Accessed: December 2019.
<https://github.com/acl2/acl2/tree/master/books/projects/x86isa>.
- [9] W. A. Hunt Jr., S. Swords, J. Davis, and A. Slobodova. Use of Formal Verification at Centaur Technology. In D. Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 65–88. Springer, 2010.
- [10] M. D. Aagaard. A hazards-based correctness statement for pipelined circuits. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 66–80. Springer, 2003.
- [11] M. D. Aagaard, R. B. Jones, T. F. Melham, J. W. O’Leary, and C.-J. H. Seger. A methodology for large-scale hardware verification. In *International Conference on Formal Methods in Computer-Aided Design*, pages 300–319. Springer, 2000.
- [12] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. Combining theorem proving and trajectory evaluation in an industrial environment. In *Proceedings of the 35th Annual Design Automation Conference, DAC ’98*, pages 538–541, New York, NY, USA, 1998. ACM.
- [13] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. Formal verification using parametric representations of boolean constraints. In *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*, pages 402–407. IEEE, 1999.
- [14] Anna Slobodova, Jared Davis, Sol Swords, and Warren A. Hunt Jr. A flexible formal verification framework for industrial scale validation. In *Proceedings of the 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 89–97, Cambridge, UK, July 2011. IEEE/ACM.
- [15] T. Arons, E. Elster, L. Fix, S. Mador-Haim, M. Mishaeli, J. Shalev, E. Singerman, A. Tiemeyer, M. Y. Vardi, and L. D. Zuck. Formal verification of backward compatibility of microcode. In *International Conference on Computer Aided Verification*, pages 185–198. Springer, 2005.
- [16] A. Coglio and S. Goel. 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, volume 280 of *EPTCS*, pages 77–94, 2018.

- [17] J. Davis, A. Slobodova, and S. Swords. Microcode verification – another piece of the microprocessor verification puzzle. In *ITP '14: Proceedings of Interactive Theorem Proving*, pages 1–16. Springer, LNCS 8558, 2014.
- [18] A. Franzén, A. Cimatti, A. Nadel, R. Sebastiani, and J. Shalev. Applying smt in symbolic execution of microcode. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, pages 121–128, Austin, TX, 2010. FMCAD Inc.
- [19] W. A. Hunt Jr. and S. Swords. Centaur Technology media unit verification. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV)*, pages 353–367, 2009.
- [20] 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, May, 2019. Order Number: 325462-070US. <https://software.intel.com/en-us/articles/intel-sdm>
- [21] Intel Corporation. Intel® Architecture Instruction Set Extensions Programming Reference, May, 2019. Order Number: 319433-037. <https://software.intel.com/en-us/articles/intel-sdm>
- [22] R. B. Jones. *Symbolic simulation methods for industrial formal verification*. Springer Science & Business Media, 2002.
- [23] R. Kaivola. Formal verification of pentium® 4 components with symbolic simulation and inductive invariants. In *International Conference on Computer Aided Verification*, pages 170–184. Springer, 2005.
- [24] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodova, C. Taylor, V. Frolov, E. Reeber, and A. Naik. Replacing testing with formal verification in intel® core™ i7 processor execution engine validation. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification*, pages 414–429, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- [25] M. O. Myreen, M. Gordon, and K. Slind. Machine-Code Verification for Multiple Architectures - An Application of Decompilation into Logic. In *Formal Methods in Computer-Aided Design, 2008. FMCAD '08*, pages 1–8, Nov 2008.
- [26] A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, E. Shepherd, P. Vrabel, and A. Zaidi. End-to-End Verification of Arm Processors with Isa-Formal. In S. Chaudhuri and A. Farzan, editors, *Proceedings of the 2016 International Conference on Computer Aided Verification (CAV'16)*, volume 9780 of *Lecture Notes in Computer Science*, pages 42–58. Springer Verlag, July 2016.
- [27] Robert S. Boyer and J S. Moore. Mechanized Formal Reasoning About Programs And Computing Machines. *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*, pages 147–176, 1996.
- [28] D. M. Russinoff. *Formal Verification of Floating-Point Hardware Design: A Mathematical Approach*. Springer, 2019.
- [29] Sandip Ray and J S. Moore. Proof Styles in Operational Semantics. In A. J. Hu and A. K. Martin, editors, *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2004)*, volume 3312 of *LNCS*, pages 67–81, Austin, TX, Nov. 2004. Springer-Verlag.
- [30] Sandip Ray, Warren A. Hunt, Jr., John Matthews, and J S. Moore. A Mechanical Analysis of Program Verification Strategies. *Journal of Automated Reasoning*, 40(4):245–269, May 2008.
- [31] Shilpi Goel. *Formal Verification of Application and System Programs Based on a Validated x86 ISA Model*. PhD thesis, Department of Computer Science, The University of Texas at Austin, 2016.
- [32] Shilpi Goel and Rob Sumners. Using x86isa for Microcode Verification. In *SpISA 2019: Workshop on Instruction Set Architecture Specification*, 2019.

- [33] A. Slobodova. Pragmatic approach to formal verification. In *SAT '15: Proceedings of Theory And Applications Of Satisfiability Testing*, pages IX–XI. Springer, LNCS 9340, 2015.
- [34] D. Stewart, D. Gilday, D. Nevill, and T. Roberts. 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)*, 2014.
- [35] S. Swords. Term-level reasoning in support of bit-blasting. In A. Slobodova and W. A. Hunt, Jr., editors, *Proceedings 14th International Workshop on the ACL2 Theorem Prover and its Applications*, Austin, Texas, USA, May 22-23, 2017, volume 249 of *Electronic Proceedings in Theoretical Computer Science*, pages 95–111. Open Publishing Association, 2017.
- [36] S. Swords and J. Davis. Bit-blasting acl2 theorems. In D. Hardin and J. Schmaltz, editors, *Proceedings 10th International Workshop on the ACL2 Theorem Prover and its Applications*, Austin, Texas, USA, November 3-4, 2011, volume 70 of *Electronic Proceedings in Theoretical Computer Science*, pages 84–102. Open Publishing Association, 2011.
- [37] S. O. Swords. *A Verified Framework for Symbolic Execution in the ACL2 Theorem Prover*. PhD thesis, University of Texas at Austin, December 2010. <http://hdl.handle.net/2152/ETD-UT-2010-12-2210>.
- [38] Warren A. Hunt, Jr., Matt Kaufmann, J S. Moore, and Anna Slobodova. Industrial hardware and software verification with ACL2. In *Verified Trustworthy Software Systems*, volume 375. The Royal Society, 2017. (Article Number 20150399).