Microcode Verification – Another Piece of the Microprocessor Verification Puzzle

Author's pre-print version.

Jared Davis, Anna Slobodova, and Sol Swords

Centaur Technology, Inc. {jared,anna,sswords}@centtech.com

Abstract. Despite significant progress in formal hardware verification in the past decade, little has been published on the verification of microcode. Microcode is the heart of every microprocessor and is one of the most complex parts of the design: it is tightly connected to the huge machine state, written in an assembly-like language that has no support for data or control structures, and has little documentation and changing semantics. At the same time it plays a crucial role in the way the processor works.
We describe the method of formal microcode verification we have developed for an x86-64 microprocessor designed at Centaur Technology. While the previous work on high and low level code verification is based on an unverified abstract machine model, our approach is tightly connected with our effort to verify the register-transfer level implementation of the hardware. The same microoperation specifications developed to verify implementation of the execution units are used to define operational semantics for the microcode verification.
While the techniques used in the described verification effort are not inherently new, to our knowledge, our effort is the first interconnection of hardware and microcode verification in context of an industrial size design. Both our hardware and microcode verifications are done within the same verification framework.

1 Introduction

Microprocessor design is a complex effort that takes hundreds of man-years. Verification of the microprocessor design remains the bottleneck of the design process. It consumes an increasing amount of resources and deploys more and more sophisticated methods including high-performance simulators and formal technology. There are many aspects to verifying the correctness of a microprocessor based system. In this paper, we will discuss only *functional verification*. Most of the papers about microprocessor verification are solely concerned with the verification of hardware. We will focus on the verification of *microcode* which is the heart and soul of a microprocessor.

While the external interface to a microprocessor is mandated by its Instruction Set Architecture (ISA), its internal behavior is governed by some processor-specific microarchitecture. For instance, contemporary x86 processors externally

support a wealth of instructions, sizes, and modes that take thousands of pages to describe (see Intel®64 and IA32 Architecture Software Developer Manuals). For performance reasons, implementations of modern x86 processors have a *frontend* that translates x86 instructions into simpler microoperations (uops), and a *backend* for executing these uops (Figure 1). Microcode (ucode) bridges the external world of Complex Instruction Set Computing and the internal world of Reduced Instruction Set Computing.

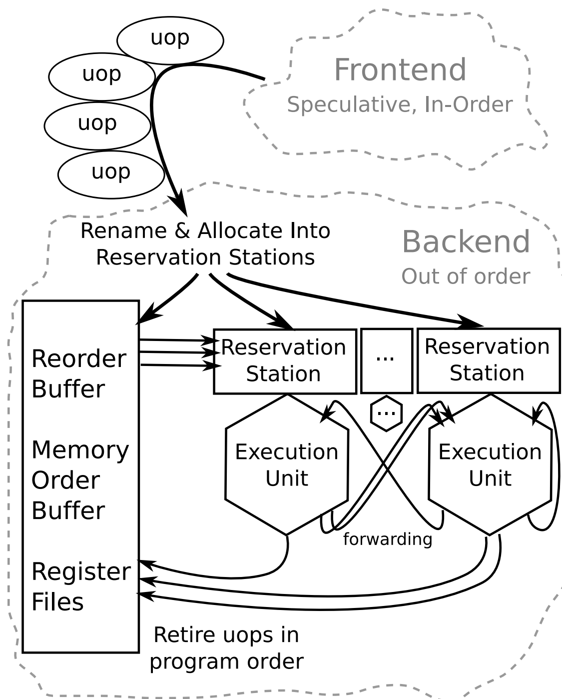

Fig. 1. Processor Backend: uops from the front-end are renamed, placed into the reorder buffer, and given to the reservation station of the appropriate execution unit. Each uop executes once its unit and operands become available. Results are forwarded among the execution units and also sent back to the reorder buffer, where they remain until retirement.

While simple and common x86 instructions are often translated into a single uop, more complex or obscure operations are implemented as ucode programs that are stored in a microcode ROM. Microcode programs are responsible for many complex features that a processor provides, e.g., they are used to implement transcendental functions, hardware virtualization, processor initialization, security features, and so on. Accordingly, their correctness is critical.

Unfortunately, there are many challenges to verifying microcode programs. Microcode verification can be seen as an instance of hardware/software co-verification, with all of the associated challenges. Whether using formal or testing-based methods, validation involves understanding both the micro-architecture and the microcode program, neither of which is easy.

Microcode is a very primitive, low-level language without even basic control constructs or data structures. At the same time, ucode programs are designed for efficiency rather than verification. During the design effort, not only are ucode programs frequently updated, but the very microcode language is extended with new operations and features. Even as the end of an effort nears and the hardware design is frozen, ucode programs continue to change—indeed, ucode patches become the preferred way of fixing bugs and adding or removing features.

The formal verification team at Centaur Technology applies formal methods to problems in various stages of the design process, including equivalence checking of transistor level and Register-Transfer Level (RTL) designs. In the area of the RTL verification, we have applied symbolic simulation supported by SAT-based and BDD-based technology to verify execution of individual microoperations in assigned execution units [1–3]. All of this verification has been carried out within the ACL2 system [4].

This paper presents our approach to formally verifying microcode routines for a new x86-64 processor in development at Centaur Technology. Our methods draw inspiration from the work of many published sources, but our work differs from each of these works in one or more aspects listed below:

i Our target is microcode – a language below the ISA level;

ii Our verification is done on an industrial scale design – an implementation of a fully x86-64 compatible microprocessor. In addition, it is done on a live project that undergoes continuous changes on the specification and implementation levels.

iii Our formal model of the microarchitecture is based on the specifications used in the RTL proofs. To our knowledge this is the first such interconnection of hardware and ucode verification done on a microarchitecture of such complexity.

Section 2 describes our formal ACL2 model of the processor's microarchitectural state and uop execution semantics. Our model can be run as a high-speed microcode simulator (around 250k uops/sec), and is also designed to achieve good reasoning performance in the theorem prover.

Section 3 gives a sketch of our approach to verifying microoperation sequences and loops, and how those can be composed to achieve correctness theorems about parts of code that constitutes subroutines. The sequential composition of the blocks is based on exploiting the power of the simplification engine within a theorem prover.

Section 4 describes the degree to which our abstract machine model has been proven to correspond to the actual hardware implementations. Parts of our model are contrived, but significant parts are directly based on specification functions

that have a mechanically proven correspondence to the Verilog modules of our processor.

Section 5 summarizes related work. Finally, Section 6 concludes the paper with comments about our future work.

2 Microcode Modeling

Microcode originates as a text program. Figure 2 shows an example of a microcode program.

```
clr_pram:
 MVIG.S64   g0, 0;         g0 = 0
clr_loop:
 STORE_PRAM g0, ADR, 0;    PRAM[ADR] = g0
 ADDIG.S64  ADR, ADR, 8;   ADR = ADR + 8
 NLOOPE.S64 g8, 1, ret;    g8--; if !g8 goto ret
 JMP_ALL    clr_loop;      goto clr_loop
ret:
 JLINK      ;              return
```

Fig. 2. A microcode routine that zeroes an area of PRAM memory. Here g0 and g8 are 64-bit registers. ADR is an alias to the 64-bit register g9. Labels like clr_pram, clr_loop and ret are resolved into ROM addresses by the assembler.

Figure 3 shows the relation between the model and hardware. An assembler translates microcode program into a binary image which is stored on the chip in a ROM. When executing microcode, the microtranslator unit fetches instructions from the ROM and translates them into backend uops that are then executed.

Our model consists of four parts described in more details below:

- **Microcode -** a constant representing the entire ROM image.
- **Microcode translator -** a function that maps ROM instructions into backend uops, plus a ucode sequencing instruction that determines the control flow after the uops' execution.
- **State -** a data structure representing the microarchitectural state.
- **Operational semantics** for backend uops, defining their effects on the state.

To build the microcode ROM, the source code files are collected and processed by a microcode assembler. The assembler converts each instruction into its binary encoding, producing a binary image that captures every ucode routine. When the processor is manufactured, this image is embedded into its ucode ROM. We extended the microcode assembler to produce, besides the ROM image and other debugging and statistical information it already generated, a Lisp/ACL2

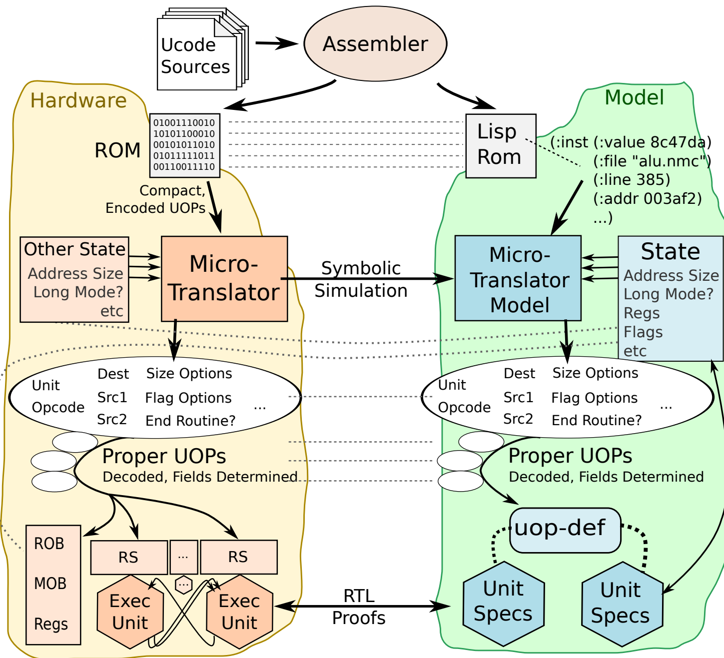

Fig. 3. Connections between our model and the processor implementation.

model of the ucode listing which can be conveniently loaded into our verification framework. This way, our model of the microcode program stays up-to-date as microcode routines or the assembler itself are changed.

The microcode translator functionality is derived by means of symbolic simulation from the RTL design of the microtranslator unit. A mapping from inputs representing a ROM instruction to outputs representing a sequence of backend uops and control flow information is represented by a set of (Boolean) formulas. For execution speed, we precompute a specialized version of these formulas for each instruction in the ROM and store them into a hash table. Some instructions' translations depend on the machine state (e.g. the current mode of operation), so the specialization does not always yield a final, concrete result, but only simpler formulas. In those cases, we finish the translation at execution time. To execute a single ROM instruction, we look up that instruction's specialized translation formula and evaluate that formula by substituting in any relevant state bits. The obtained sequence of uops and the control information is interpreted with respect to defined semantics which determines the next state, including the value of the program counter. Automatically deriving the micro-translator functionality from the actual RTL keeps this part of the model up-to-date.

The machine state is represented as a tuple containing essential machine variables such as the program counter, the stack, various sets of registers (e.g., the x86 general-purpose registers are in a field called *gregs* and the SSE media

registers are in *mregs*) and scratch memories. Parts of the state correspond to registers defined by the x86 architecture, but much of it is specific to the concrete micro-architecture of the project. A machine state may be a *running*, *halting*, or *divergent* state, where a divergent state is used to represent the result of a program that never terminates. These three types of states are distinguished by the value of the program counter; a natural number indicates a running state.

We describe the effect of each uop as a transformation of the state, in the usual interpreter semantics style. For instance, the effect of the `XADD` uop is modeled by the function

$$\textit{xadd-def}(\textit{uop}, s) \rightarrow s',$$

where s and s' are current and transformed states, resp., and uop represents a particular instance of a microoperation with all relevant information – operands, operand size, flag information, unit where the uop is executed, *etc.*:

dest	`g9`	src width	`64`	opcode	`XADD`
src1	`g8`	dest width	`64`	exec unit	`int`
src2	`8`	write flags?	`no`		

A function like *xadd-def* interprets this information, e.g., for the instruction above it would extract the value of `gregs[8]` and interpret it as 64-bit value, add it to `src2`, store the result in `gregs[9]`, and not update any flags.
Semantic functions for all types of uops are combined into a universal uop definition:

$$\mu\textit{op-def}(\textit{uop}, s) \stackrel{\text{def}}{=} \begin{cases} \textit{xadd-def}(\textit{uop}, s) & \text{if } \texttt{uop.type} \text{ is } \texttt{XADD} \\ \textit{xsub-def}(\textit{uop}, s) & \text{if } \texttt{uop.type} \text{ is } \texttt{XSUB} \\ \cdots & \cdots \\ \textit{error}(s) & \text{otherwise} \end{cases}$$

To model the execution of consecutive steps from a state s, we use an approach by Ray and Moore [5, 6]. We first define the function $\textit{run}(n, s)$ which returns the new state after executing n steps:

$$\textit{run}(n, s) \stackrel{\text{def}}{=} \begin{cases} s & \text{if } n = 0 \\ \textit{run}(n-1, \textit{step}(s)) & \text{otherwise} \end{cases}$$

Note that *step* executes one ROM instruction that can consist of several uops. Its definition and connection to our hardware verification proofs are explained in more detail in Section 4.

Finally, we can define $\textit{run}^* : s \rightarrow s$, which runs the machine until it enters a halting state. If the program does not terminate, then logically it returns the divergent state $\perp$ (whereas the actual execution of $\textit{run}^*$ would never terminate). This avoids the need to explicitly determine how many steps a program takes and allows us to pursue partial correctness results without proving termination (see *run-measure* in Section 3).

$$\textit{run}^*(s) \stackrel{\text{def}}{=} \begin{cases} s & \text{if } \textit{halting}(s) \\ \textit{run}^*(\textit{step}(s)) & \text{if } \neg\textit{divergent}(s) \\ \perp & \text{otherwise} \end{cases}$$

The next section will explain how we reason about this model. Then, in Section 4, we will explain the relationship between this model and the hardware design.

3 Microcode Verification

Before describing our verification methodology, we need to explain our verification objective. Unlike higher level languages, microcode does not have any nice control structures like for loops, while loops, if-then-else constructs, etc, and there is no such entity as a main program in microcode. It consists of a sequence of micro operations residing in a ROM. Figure 2 shows an example of a microcode snippet. Microoperations can move values between registers and to/from scratch memory, and can manipulate values by means of arithmetic and logical operations. Loops are implemented with conditional and unconditional jumps. Microcode is written for efficiency and would not please the eye of any programmer.

Our objective in microcode verification is to characterize the effect of executing microcode from an entry point to an exit point on the machine state. In order to do this, we incrementally verify blocks of code and compose the theorems into theorems about larger blocks. We have defined a macro *def-uc-block* that supports verification of a general block of code, and a macro *def-uc-loop* that supports reasoning about loops.

3.1 def-uc-block

We use the *def-uc-block* macro to specify blocks of straight-line code and to compose together previously defined blocks (including loops). The user specifies:

1. *start-pc* as an initial value for the program counter;
2. *block-precondition* as a state predicate;
3. *run-block*, a function that executes the machine model until the end of block is reached. This may be a simple application of the universal *run* function for a given number of steps, or as a combination of *run* and applications of previously defined blocks.
4. *block-specification* that describes the machine state after execution of this block (the post-state). This definition is in terms of updates to the start-state. As a consequence, those parts of the state that remain unchanged are left out of the description of the change. While we need to keep track of the changing values in some registers and memories, other parts of the state are used as temporary storage and become irrelevant for the final result. To avoid precisely characterizing these don't care values, we copy them from the actual post-state (produced by *run-block*) into the specification state, making their equivalence trivial. This is known as *wormhole abstraction* [7]).

For the code in Figure 2, the verification could start with the definition of *def-uc-block* with the arguments described in Figure 4.

name:	*clr_loop_last*
pc:	$\textit{get-label}(\textit{clr_loop})$
run:	$\textit{run}(4, s)$
precondition:	Let $\textit{adr} = s.\textit{gregs}[9], \textit{cnt} = s.\textit{gregs}[8]$ in $\textit{addr-ok}(\textit{adr}) \wedge (\textit{cnt} = 1) \wedge$ $\neg \textit{empty}(s.\textit{retstack})$

specification:	Let $\textit{adr} = s.\textit{gregs}[9], \textit{val} = s.\textit{gregs}[0]$ in {
	$s = \textit{pram-store}(\textit{val}, \textit{adr}, s);$
	$s.\textit{gregs}[8] = 0;$
	$s.\textit{gregs}[9] = \textit{adr} + 8;$
	return $\textit{stack-pop}(s);$
	}

Fig. 4. *def-uc-block* example (last run through the loop): *get-label* translates a label into initial value of the program counter. Running the block takes 4 steps. In the precondition, *addr-ok* identifies valid address to the PRAM memory. While g9 is used as a pointer to memory, g8 is a counter that controls the loop. The loop terminates when the counter clears. The specification describes the state update caused by the last run through the loop: the value of g0 is stored in the PRAM at address specified by g9, g8 is decremented, g9 is incremented by the size of the written entry, and the program counter is set to the value on the top of the return stack.

The expansion of the macro defines all the functions above and automatically proves some theorems about them. For instance, the *run-block* function has to satisfy following properties:

R1: $\textit{run}^*(\textit{run-block}(s)) = \textit{run}^*(s)$
applying $\textit{run}^*$ to the post-state brings us to the same state as applying $\textit{run}^*$ to the start state.

R2: $\textit{halting}(s) \Longrightarrow \textit{run-block}(s) = s$
run-block will not advance from a halting state.

R3: $\textit{divergent}(s) \Longrightarrow \textit{run-block}(s) = \bot$
Whenever we get into a divergent state, we converge into the $\bot$ state.

R4: $\neg \textit{divergent}(\textit{run}^*(s)) \Longrightarrow \neg \textit{divergent}(\textit{run-block}(s))$
If $\textit{run}^*$ terminates, then *run-block* terminates.

R5: *run-block* makes progress in the termination of $\textit{run}^*$:

$$\begin{array}{l} \neg \textit{halting}(s) \wedge \neg \textit{divergent}(\textit{run-block}(s)) \\ \Longrightarrow \\ \textit{run-measure}(\textit{run-block}(s)) < \textit{run-measure}(s) \end{array}$$

where *run-measure* is a non-executable function whose value is the minimum number of steps needed to bring the machine to a halting state, if that exists, and zero otherwise. It is defined as a Skolem witnessing function using the ACL2 feature *defchoose*.

R6: $\mathit{run\text{-}measure}(\mathit{run\text{-}block}(s)) \leq \mathit{run\text{-}measure}(s)$
A weaker monotonic condition.

For the *block-precondition* predicate we have an option to do a simple vacuity check. It exploits a symbolic simulator [8] that converts an ACL2 object that is defined over a finite domain into a symbolic object encoded as an And-Inverter Graph. Finding a state that satisfies the precondition is thus transformed into satisfiability of a Boolean formula, which we then translate into CNF and solve using an off-the-shelf SAT solver [9].

The main result of the *def-uc-block* expansion is the correctness theorem:

Theorem 1 (block-correct).

$$\begin{gathered} s.\mathit{pc} = \mathit{start\text{-}pc} \wedge \mathit{block\text{-}precondition}(s) \\ \Longrightarrow \\ \mathit{run\text{-}block}(s) = \mathit{block\text{-}specification}(s, \mathit{run\text{-}block}(s)) \end{gathered}$$

Theorem *block-correct* is the crucial point of the verification. We have two distinct methods for proving this theorem for each block.

- We can use bit-level symbolic execution [8], which computes a Boolean formula representing the correctness condition and attempts to solve it using a SAT solver. This is preferred for short blocks whose correctness proofs do not depend on much mathematics. This method is largely automatic (though it can be tuned with rules that determine how to process certain functions), and in many cases can either quickly prove the desired theorem or produce a counterexample showing a difference between the spec and the actual behavior of the routine. However, this method suffers from capacity limitations and is also difficult to debug in cases where a proof times out or otherwise fails.
- We can use ACL2's native proof engines, together with a litany of hints and rules that optimize its behavior on this sort of problem. E.g., we instruct the prover to only open the definition of *run* if the program counter of the state can be determined.

Both methods support composition of blocks, and both also support wormhole abstraction, obviating the need to specify and spend proof effort on don't-care fields of the state.

3.2 def-uc-loop

Although there is no explicit loop construct in the microcode, loops do appear in the code in various forms. Macro *def-uc-loop* supports their verification. Through the arguments to this macro, the user specifies:

1. *start-pc*, an initial value for the program counter
2. *loop-precondition*, a starting state predicate
3. *loop-specification*, the loop's effect on the machine state

4. *measure*, a term used in the proof of termination of the loop (e.g, value of a register that serves as a counter).
5. *done*, a condition (state predicate) that is satisfied upon entering the last execution of the loop.
6. *run-last*, run function for the execution of the last time through the loop.
7. *run-next*, run function for the execution of any but the last time trough the loop.

def-uc-loop also supports proving partial correctness for loops that may not terminate; in these cases, the measure may be omitted.

Execution of a *def-uc-loop* is usually preceded by two executions of *def-uc-block* that specify the effect of executing one round of the loop. In particular, the two cases describe the *run-last* block (executed under the precondition $done(s) \wedge$ $loop\text{-}precondition(s)$) and the *run-next* block (under the precondition $\neg done(s) \wedge$ $loop\text{-}precondition(s)$). Figure 5 shows an example of *def-uc-loop* arguments for the code on Figure 2.

Expansion of the macro defines a function *run-loop* that repeatedly executes *run-next* until the *done* condition first holds, then finishes by executing *run-last*. Properties R1–R6 are proved for *run-loop*, and the correctness theorem has exactly the same form as the correctness theorem for *def-uc-block*:

Theorem 2 (loop-correct).

$$s.pc = start\text{-}pc \wedge loop\text{-}precondition(s)$$
$$\Longrightarrow$$
$$run\text{-}loop(s) = loop\text{-}specification(s, run\text{-}loop(s))$$

This theorem is proved using induction defined by the scheme of the *run-loop* function and the two block-correct theorems for the *run-next* and *run-last* functions. The proof may be done either using ACL2's built-in proof engines or by applying our bit-level proof engine separately to the base case and induction step.

4 Hardware Connection

We would like our microcode model to be useful both for ad-hoc testing of microcode routines and for carrying out formal proofs of correctness about these routines. The closer the model is to the actual processor, the stronger the results of tests and proofs. Figure 3 shows the corresponding parts of our model and the processor implementation. Dark blue parts were proved to match dark orange parts of the hardware model.

We derive our instruction listing from the same microcode assembler that also produces the content of ROM, and we model the microcode decoder by effectively simulating the RTL of Micro-translator, as described in Section 2. Thus, our model of the translation from the text microcode into uops has a strong connection to the real design of the processor's frontend.

name:	*clr_loop*
pc:	*get-label*(*clr_loop*)
measure:	$s.gregs[8]$
done:	$s.gregs[8] = 1$
run-last:	*run-clr-loop-last*(s)
run-next:	*run-clr-loop-next*(s)

precondition: Let $adr = s.gregs[9], cnt = s.gregs[8]$ in
$addr\text{-}ok(adr \cdot (cnt - 1)) \wedge (cnt > 0) \wedge$
$\neg empty(s.retstack)$

specification: Let $adr = s.gregs[9], val = s.gregs[0],$
$cnt = s.gregs[8], idx = adr \div 8$ in {
$s = clr\text{-}pram\text{-}k(idx, idx + cnt, val, s);$
$s.gregs[8] = 0;$
$s.gregs[9] = adr + 8 \cdot cnt;$
return $stack\text{-}pop(s);$
}

Fig. 5. *def-uc-loop* example: The measure (the value of g8) will decrease with each run through the loop. The precondition assures that the last address to which we write is within a boundary and that the starting value of g8 is positive, assuring termination. The *run* function is composed from two previously defined run functions *(run-clr_loop_last s)* and *(run-clr_loop_but_last s)*. The specification describes the state upon termination of the loop: *clr-pram-k*(*start*, *end*, *val*, *s*) copies value from g0 throughout $PRAM[start : end - 1]$; g8 is set to 0; g9 is set to point at the address followed by the last written address; and the program counter is set to the value from the top of the return stack.

As for the backend, our model is a significant abstraction of the actual processor, which is depicted in Figure 1. For instance, we abstract away out-of-order execution of the micro operations. Consequently, things that appear very simple in our model, say, "get the current value in register `g0`," are actually quite complicated, involving, e.g., the register aliasing table, the reservation stations, forwarding, the reorder buffer, *etc.* This said, significant parts of our model *do* have a strong connection to the real hardware design. In previous work [1–3] we described how we have developed an RTL-level verification framework within ACL2, and used it to prove that our execution units for integer, media, and floating point instructions implement desired operations. This previous work means that, for many uops, we have a specification function, written in ACL2, that functionally matches the execution of the uop in a particular unit. Thanks to regression proofs, we can be quite confident that these specifications remain up-to-date.

Now we can sketch how the *step* function is defined. It takes a state of our abstract machine and returns the state of the machine after executing the ROM instruction pointed to by the current PC ($s.pc$). The extraction of the ROM instruction (*get-rom-inst*) is a simple lookup in the constant *ucode* – a list

representing the content of the ROM. It will then lookup the pre-computed formula for the result of running the micro-translator unit on this instruction. The hash table lookup returns a sequence of uops and additional sequencer information in the form of symbolic formulas (sym_uops, sym_seq). These formulas are pretty simple, depending on a few variables whose values can be extracted from the current state. *exec_uops* executes the sequence of uops by repeatedly applying $\mu op\text{-}def$ (see Section 2) one by one. $\mu op\text{-}def$ is directly connected to the proofs of hardware. In case of a conditional jump instruction, it also decides whether the branch is taken. The function *next-pc* will determine the value of the next PC which completes the execution of one instruction.

$$
\begin{aligned}
&step : s \rightarrow s' = \{\\
&\quad inst = get\text{-}rom\text{-}inst(s.pc, *ucode*)\\
&\quad (sym_uops, sym_seq) = lookup\text{-}uxlator(inst)\\
&\quad (uops, useq) = eval(s, sym_uops, sym_seq)\\
&\quad (branch_taken, s) = exec\text{-}uops(uops, s)\\
&\quad s.pc = next\text{-}pc(s, useq, branch_taken)\\
&\}
\end{aligned}
$$

It is important to note that our model is defined in extensible way. It allows us to relatively seamlessly move the boundary between the parts that are validated and those that are contrived.

5 Related Work

Our work builds upon countless ideas and advancements in microprocessor and machine code verification published over decades. Our contributions are in combining these advances and using them in an industrial setting, and in connecting methods for software and hardware verification under one unifying framework.

Operational semantics as a formalization of the meaning of programs was introduced in the 60s by McCarthy [10]. Early applications can be found in a technical report by van Wijngaarden *et al.* describing ALGOL68 [11]. Since then, structural operational semantics has been extensively used for mechanical verification of complex programs using various theorem provers: ACL2 and its predecessor NQTHM [12–14]; Isabelle/HOL [15]; and PVS [16]. Smith and Dill used operational semantics along with domain specific simplifications using a SAT solver and ACL2 for automatic equivalence checking of object code implementations of block ciphers [17]. More recent attempts to formalize operational semantics of complex ISAs come from Goel and Hunt [18] for x86, and Fox *et al.* for ARM [19]. All these papers model languages on the ISA level or above, and their operational semantics is not supported by any further verification. Wilding *et al.* [20] made use of the executability of ACL2 functions to validate their model by extensive testing against the hardware.

Many papers have discussed methodology for verifying the correctness of a microprocessor's microarchitecture with respect to its ISA [21, 22]. While these defined crucial concepts and methods for bridging the two different abstraction

levels, they did not go beyond theoretical models of small to moderate size. Even the more comprehensive machine verification project described by Hunt [23], which includes some simple microcode verification connected to top-down hardware verification, does not have the complexity and dimensions of industrial scale designs.

Some work has been published by researchers from Intel® on verification of backward compatibility of microcode [24, 25]. The idea is based on creating symbolic execution paths, storing them in a database and using them either for testing or for checking assertions. This work differs from ours in several aspects. First, it is not connected to hardware verification. Their operational semantics of microcode is defined through a translation to an intermediate language with predefined operational semantics. There is no direct connection of this semantics to what is actually implemented in the hardware. Second, their approach uses SAT/SMT, while we are using mostly theorem prover and symbolic simulator that is built-in and verified within the prover. Finally, the verification objectives of our work are very different: while we compare the effect of running a microcode routine to a fully or partially defined specification that can be written on a high abstraction level, their objective is to compare the behavior of two microcode routines for backward compatibility.

Since the beginning of the computing era, the correctness of programs has been on the minds of great computer scientists like Floyd [26], Hoare [27], Manna [28], *etc.* The first papers concerned with program verification were based on assertions, but at that point, researchers weren't equipped with high-level mechanized proof systems. Matthews *et al.* [29] merged the idea of operational semantics with assertion verification. Their work is closest to our verification approach. Both our approach and that of Matthews *et al.* decompose the program into blocks separated by cutpoints. The difference is that Matthews *et al.* use the inductive invariant approach: a set of cutpoint/assertion pairs is defined and the goal is to prove a global invariant of the form:

$$\forall i \ \ (pc(s) = cutpoint_i) \Rightarrow assertion_i(s).$$

In our approach, we characterize (fully or partially) the effect of running each block (a sequence of operations between two cutpoints) on the state, then sequentially compose blocks together to build up a characterization of the effect of a full microcode routine on the state. These two approaches have been shown to be logically equivalent (in sufficiently expressive logic) [5]. Wormhole abstraction has been introduced in [7]. The target of [7, 29] is a slightly higher level of machine code. The main difference to our work is that their operational semantics remains unverified.

Last year, a paper by Alex Horn, Michael Tautschig and others [30] demonstrated validation of firmware and hardware interfaces on several interesting examples. Their work differs from ours in several aspects: Their methods require both hardware and software to be described in the same language (C or SystemC); the hardware is much simpler than a microprocessor; and the main method used for verification is model-checking.

6 Conclusion and Future Work

We presented an approach to microcode verification that is tightly connected to ongoing hardware verification. Since our RTL and microcode proofs are done within the same system, we are able to reuse the functions specifying hardware behavior to model microoperations. While the microcode model is far from being completely verified, our methodology has the flexibility to move the boundary between proved and contrived parts of the model as we achieve more of its validation.

We tried our approach on several microcode routines. One routine, that was a representative of arithmetic operation routines, was a 54-instruction microcode routine that performs unsigned integer division of a 128-bit dividend by a 64-bit divisor, storing the 64-bit quotient and remainder. We proved the correctness of this routine using 11 *def-uc-block* forms. Of these, five specified the behavior of low-level code blocks, and the rest primarily composed these sub-blocks together according to the control flow.

Another example was a verification of one of the critical algorithms that run as a part of the machine bring-up process. The algorithm deals with *decompression* of strings. The processor reads a an input that is a concatenation of compressed strings of variable lengths and places it in scratch memory. The main routine runs in a loop where one round identifies the beginning of the next compressed substring and converts it back to uncompressed form. The beginning and the type of a compressed substring is identified by its header. The decompression algorithm is implemented in about 800 lines of code. Its logical structure contains several loops (including nested loops) and many subroutines. A handful of designated registers keep track of pointers to scratch memory, positions within strings, counters, and currently processed strings, and many more are used as temporary value holders. The correctness proof of the decompression uses about 50 *def-uc-block*/*def-uc-loop* structures. Since the algorithm and its implementation were frequently changing (adding new compression types, changing storage location, etc.), it was important to have automated support for the detection of these changes.

The main difference between our work and work done in academia is that we have to deal with a real contemporary industrial design that is not only very complex and sparsely documented, but also is constantly changing: both the hardware model and microcode are constantly updated. This requires automation to detect the changes and (wherever possible) make the relevant adjustments to proofs. Our automation includes regularly building the design model and running a regression suite for both hardware and microcode proofs.

Even though our microcode verification methodology is based on the powerful rewriting capabilities of the ACL2 theorem proving system [4], the approach could be applied using other rewriting systems as well.

In the future we would like to expand our verification effort to cover more of the critical microcode, e.g. in security-related areas. We also would like to pursue a more systematic approach to the verification of ISA instructions, connecting

our specifications to a formal model of x86 such as the one one developed by Goel and Hunt [18].

Acknowledgment

We would like to thank Warren Hunt for comments on the first draft of this paper.

References

1. Hunt Jr., W., Swords, S.: Centaur Technology media unit verification. In: Proceedings of the 21st International Conference on Computer Aided Verification (CAV). (2009) 353–367
2. Hunt Jr., W., Swords, S., Davis, J., Slobodova, A.: Use of Formal Verification at Centaur Technology. In Hardin, D., ed.: Design and Verification of Microprocessor Systems for High-Assurance Applications. Springer (2010) 65–88
3. Slobodova, A., Davis, J., Swords, S., Hunt Jr., W.: 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), Cambridge, UK, IEEE/ACM (July 2011) 89–97
4. Kaufmann, M., Moore, J.S., Boyer, R.S.: ACL2 version 6.1. `http://www.cs.utexas.edu/~moore/acl2/` (2013)
5. Ray, S., Moore, J.: Proofs styles in operational semantics. In: Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD). Volume 3312 of LNCS., Springer-Verlag (2004) 67–81
6. Moore, J.S.: Proving theorems about Java and the JVM with ACL2. Models, algebras and logic of engineering software (2003) 227–290
7. Hardin, D.S., Smith, E.W., Young, W.D.: A robust machine code proof framework for highly secure applications. In: Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and its Applications, ACM (2006) 11–20
8. Swords, S., Davis, J.: Bit-blasting ACL2 theorems. In: ACL2 '11. Volume 70 of Electronic Proceedings in Theoretical Computer Science. (2011) 84–102
9. Davis, J., Swords, S.: Verified AIG algorithms in ACL2. In: Proceedings of ACL2 Workshop. (2013)
10. McCarthy, J.: Towards a mathematical scioence of computation. In: Information Processing Congress. Volume 62., North-Holland (1962) 21–28
11. van Wijngaarden, A., Mailloux, B., Peck, J., Koster, C., Sintzoff, M., Lindsey, C., Meertens, L., R.G.Fisker: Revised report on the algorithmic language ALGOL 68 (1968)
12. Boyer, R., Moore, J.: Mechanized formal reasoning about programs and computing machines. Automated reasoning and its applications: essays in honor of Larry Woss (1996) 141–176
13. Greeve, D., Wilding, M., Hardin, D.: High-speed, analyzable simulators. In Kaufmann, M., Moore, J.S., Manolios, P., eds.: Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Publishers (2000) 89–106
14. Yu, Y.: Automated proofs of object code for a widely used microprocessor. PhD. Thesis (1992)

15. Strecker, M.: Formal verification of the Java compiler in Isabelle. In Voronkov, A., ed.: International Conference on Automated Deduction (CADE). Volume 2393 of LNCS., Springer (2002) 63–77
16. Hamon, G., Rushby, J.: An operational semantics for stateflow. In: Fundamentl Approaches to Software Engineering (FASE). Volume 2984 of LNCS. (2004) 229–243
17. Smith, E., Dill, D.: Automatic formal verification of Block Cipher implementations. In Cimatti, A., Jones, R., eds.: Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD), IEEE/ACM (2008) 45–51
18. Shilpi Goel and Warren Hunt Jr.: Automated code proofs on a formal model of the x86. In: Proceedings of the Fifth Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE). LNCS, Springer-Verlag (2013) To appear.
19. Fox, A., Myreen, M.: A trustworthy monadic formalization of the ARMv7 instruction set architecture. In: Proceedings of Federated Logic Conference (ITP). Volume 6172 of LNCS., Springer (2010)
20. Wilding, M., Greeve, D., Richards, R., Hardin, D.: Formal verification of partition management of the AAMP7G microprocessor. In Hardin, D., ed.: Design and Verification of Microprocessor Systems for High-Assurance Applications. Springer (2010) 175–192
21. Cyrluk, D.: Microprocessor verification in pvs. a methodology and simple example. `http://www.csl.sri.com/papers/csl-93-12/` (February 1994)
22. Sawada, J., Hunt Jr., W.: Verification of FM9801: An out-of-order microprocessor model with speculative execution, exceptions, and program-modifying capability. J. of Formal Methods in System Design **20(2)** (2002) 187–222
23. Hunt, Jr., W.A.: FM8501: A Verified Microprocessor. Volume 795 of LNCS. Springer-Verlag (1991)
24. Arons, T., Elster, E., Fix, L., Mador-Haim, S., Mishaeli, M., Shalev, J., Singerman, E., Tiemeyer, A., Vardi, M.Y., Zuck, L.D.: Formal verification of backward compatibility of microcode. In: Computer Aided Verification, Springer (2005) 185–198
25. Franzén, A., Cimatti, A., Nadel, A., Sebastiani, R., Shalev, J.: Applying SMT in symbolic execution of microcode. In: Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design (FMCAD), Austin, TX, FMCAD Inc (2010) 121–128
26. Floyd, R.: Assigning meanings to programs. In: Mathematical aspects of computer science, Proceeings of Symposia in applied mathematics. Volume XIX., American Mathematical Society (1967) 19–32
27. Hoare, C.: An axiomatic basis to computer programming. Communications of the ACM **12** (1969) 576–583
28. Manna, Z.: The correctness of programs. Journal of Computer and System Sciences **3** (1969) 119–127
29. Matthews, J., Moore, J.S., Ray, S., Vroon, D.: Verification Condition Generation Via Theorem Proving. In Hermann, M., Voronkov, A., eds.: Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR). Volume 4246 of LNCS., Phnom Penh, Cambodia, Springer (November 2006) 362–376
30. Horn, A., Tautschnig, M., Val, C., Liang, L., Mehlham, T., Grundy, J., Kroening, D.: Formal co-validation of low-level hardware/software interfaces. In Jobstman, B., Ray, S., eds.: Proceedings of the Formal Methods in Computer-Aided Design (FMCAD), ACM/IEEE (2013) 121–128