Adding 32-bit Mode to the ACL2 Model of the x86 ISA

Alessandro Coglio
Kestrel Technology LLC, Palo Alto, CA (USA)
coglio@kestreltechnology.com

Shilpi Goel
Centaur Technology, Inc., Austin, TX (USA)
shilpi@centtech.com

The ACL2 model of the x86 Instruction Set Architecture was built for the 64-bit mode of operation of the processor. This paper reports on our work to extend the model with support for 32-bit mode, recounting the salient aspects of this activity and identifying the ones that required the most work.

1 Motivation and Contributions

A formal model of the ISA (Instruction Set Architecture) of the pervasive x86 processor family can be used to clarify and disambiguate its informal documentation [15], as well as support the verification and formal analysis of existing binary programs (including malware), the verified compilation of higher-level languages to machine code, the synthesis of correct-by-construction binary programs from specifications, and the verification that micro-architectures correctly implement the ISA. Furthermore, if the model is efficiently executable, it can be used as a high-assurance simulator, e.g. as a test oracle.

The ACL2 model of the x86 ISA [1, `:doc x86isa`] [11] is one such model. It includes a substantial number of instructions and has been used to verify several non-trivial binary programs. Its fast execution has enabled its validation against actual x86 processors on a large number of tests.

Prior to the work described in this paper, the model supported the simulation and analysis of 64-bit software only (which includes modern operating systems and applications), and was used to verify several 64-bit programs. We refer to this pre-existing model of the x86 ISA as the *64-bit model*.

However, legacy 32-bit software is still relevant. Many applications are still 32-bit, running alongside 64-bit applications in 64-bit operating systems. Most malware is 32-bit [16]; as part of a project to detect malware variants via semantic equivalence checking by symbolic execution, we were given a large corpus of known Windows malware for experimentation, and found indeed that it mainly contained 32-bit executables.[1] Verifying the 32-bit portion of a micro-architecture requires a model of the 32-bit ISA. Also, the aforementioned verified compilation of higher-level languages to machine code and synthesis of correct-by-construction binary programs from specifications, while presumably generally oriented towards 64-bit code, may occasionally need to target 32-bit code, e.g., to interoperate with legacy systems.

Thus, we worked on extending the 64-bit model with 32-bit support: all the non-floating-point instructions in the 64-bit model have been extended from 64 bits to 32 bits; furthermore, a few 32-bit-only instructions have been added. However, the 32-bit extensions have not been validated against actual x86 processors yet as thoroughly as the 64-bit portions of the model. Work on verifying 32-bit programs using this model has also started but is too preliminary to report. We refer to this extended model of the x86 ISA as the *extended model*.

This paper reports on our work to extend the 64-bit model. Section 2 provides some background on the x86 ISA and on the 64-bit model. Section 3 recounts the salient aspects of the 32-bit extensions to the model, including how they were carried out and which ones required the most work. Related and

[1] This statement refers to a project at Kestrel Technology.

Shilpi Goel and Matt Kaufmann (Eds.):
ACL2 Theorem Prover and its Applications (ACL2-2018)
EPTCS 280, 2018, pp. 77–94, doi:10.4204/EPTCS.280.6

future work are discussed in Section 4 and Section 5. The 64-bit model was developed by the second author [11], and the 32-bit extensions were developed by the first author (with some help and advice from the second author); this suggests that the model is extensible by third parties.

2 Background

We present a very brief overview of the x86 ISA that is relevant to this paper in Section 2.1. In Section 2.2, we describe some features of the 64-bit model of the x86 ISA.

2.1 x86 ISA: A Brief Overview

Intel's modern x86 processors offer various modes of operation [15, Volume 1, Section 3.1]. The IA-32 architecture supports 32-bit computing by giving access to 2^{32} bytes of memory, and it provides three modes:

1. *Real-address mode*, which is the x86 processor's mode upon power-up or reset.
2. *Protected mode*, which is informally referred to as the "32-bit mode"; this mode can also emulate the real-address mode if its *virtual-8086 mode* attribute is set.
3. *System management mode*, which is used to run firmware to obtain fine-grained control over system resources to perform operations like power management.

The Intel 64 architecture added the *IA-32e mode*, which has two sub-modes:

1. *64-bit mode*, which gives access to 2^{64} bytes of memory, and thus, allows the execution of 64-bit code.
2. *Compatibility mode*, which allows the execution of legacy 16- and 32-bit code inside a 64-bit operating system; in this sense, it is similar to 32-bit protected mode vis-à-vis application programs.

The 64-bit model of the x86 ISA, discussed briefly in Section 2.2, specified only the 64-bit sub-mode of IA-32e mode. In the subsequent sections, we focus on 32-bit protected mode and compatibility mode. Modeling the rest of the modes is a task for the future (see Section 5).

IA-32e and 32-bit protected modes are the most used operating modes of x86 processors today. A big difference between these modes is in their memory models — we first briefly discuss the memory organization on x86 machines. There are two main types of memory address spaces: *physical address space* and *linear address space*. The physical address space refers to the processor's main memory; it is the range of addresses that a processor can generate on its address bus. The linear address space refers to the processor's linear memory, which is an abstraction of the main memory (see paging below). Programs usually do not access the physical address space directly; instead, they access the linear address space.

Segmentation and *paging* are two main facilities on x86 machines that manage a program's view of the memory; see Figure 1. When using segmentation, programs traffic in *logical addresses*, which consist of a *segment selector* and an *effective address*. The x86 processors provide six (user-level) segment selector registers — `CS`, `SS`, `DS`, `ES`, `FS`, and `GS`. A segment selector points to a data structure, called a *segment descriptor*, that contains, among other information, the *segment base*, which specifies the starting address of the segment, and the *segment limit*, which specifies the size of the segment. As an optimization, the processor caches information from this data structure in the *hidden* part of these registers whenever a selector register is loaded. The segment base is added to the effective address to obtain the linear address. The upshot of this is that the linear address space is divided into a group of segments. An operating system can assign different sets of segments to different programs or even different segments to the same program (e.g., separate ones for code, data, and stack) to enforce protection and non-interference. It can

also assign the same set of segments to different processes to enable sharing and communication. The processor makes three main kinds of checks when using segmentation: (1) limit checks, which cause an exception if code attempts to access memory locations outside the segment; (2) type checks, which cause an exception if code uses a segment in some unintended manner (e.g., an attempt is made to write to a read-only data segment); and (3) privilege checks, which cause an exception if code attempts to access a segment which it does not have permission to access.

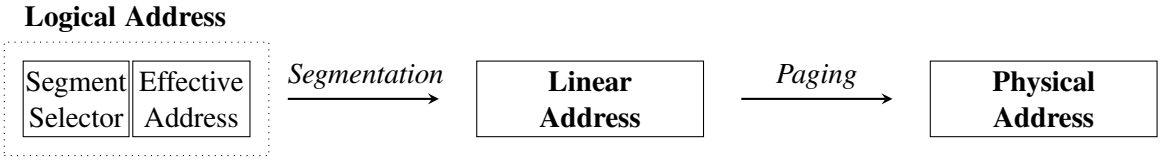

Figure 1: Memory Addresses

Paging is used to map linear addresses to physical addresses. It is conventional to define this mapping in such a manner that a larger linear address space can be simulated by a smaller physical address space. Each segment is divided into smaller pages, which can be resident either in the physical memory (if the page is in use) or on the disk (if the page is not currently in use). An operating system can swap pages back and forth from the physical memory and the disk; thus, paging supports a *virtual* memory environment. There are three kinds of paging offered by x86 processors: 32-bit (translates 32-bit linear addresses to up to 40-bit physical addresses), PAE (translates 32-bit linear addresses to up to 52-bit physical addresses), and 4-level (translates 48-bit linear addresses to up to 52-bit physical addresses).

In 64-bit mode, a logical address, an effective address, and a linear address for a memory location are usually the same because all segment bases, except those for two out of six segment selectors (i.e., `FS` and `GS`), are treated as zero. Moreover, limit checking and a certain kind of type checking are not performed in this mode. Thus, segmentation is effectively a legacy feature in 64-bit mode. However, in 32-bit and compatibility modes, segmentation is used in its full glory.

32-bit protected and IA-32e modes also differ with regard to paging. In 32-bit protected mode, 32-bit and PAE paging are used to translate 32-bit linear addresses. In both sub-modes of IA-32e mode, only 4-level paging is used. However, since 4-level paging translates only 48-bit linear addresses and IA-32e mode has 64-bit linear addresses, this mode has a notion of *canonical addresses*. Canonical addresses are 64-bit linear addresses whose bits 63 through 47 are identical — either all zeroes or all ones. Additionally, since compatibility mode traffics in 32-bit addresses, bits 47 through 32 of a 64-bit linear address are treated as zero. Any attempt to map a non-canonical linear address to a physical address in IA-32e mode will lead to an exception.

Of course, there are many other differences between 32-bit and 64-bit modes. Even instruction decoding differs in these modes. An x86 instruction has a variable length up to 15 bytes: the *opcode* bytes in the instruction determine which operation is to be performed; the other instruction bytes act as size modifiers, or specify the location of the operands, etc. 64-bit instructions can have additional bytes preceding an opcode, called the `REX` prefixes, which can modify the operation's sizes or operands' locations. In 32-bit mode, these bytes actually correspond to the increment/decrement (`INC`/`DEC`) opcodes. Task management is another big example — a task can consist of a program, a process, or an interrupt or exception handler, and it is executed and/or suspended by the processor as an atomic unit. Task management is essentially the software's responsibility in 64-bit mode, but it is handled by hardware in 32-bit protected mode. Generally speaking, 32-bit mode and compatibility sub-mode support many legacy features, and as such, they are less "streamlined" than the newer 64-bit mode.

We refer the reader to Intel's official documentation [15] for more information about the x86 ISA.

2.2 `x86isa`: A Model of the 64-bit Mode

Prior to this work, the `x86isa` library in the ACL2 community books contained the specification of a significant subset of 64-bit sub-mode of IA-32e mode. This 64-bit model follows the classical *interpreter style of operational semantics* [18, 19] — it can be described by the following main components:

- a machine state consisting of registers, memory, flags, etc.;
- instruction semantic functions that define the behavior of x86 instructions in terms of updates made to the machine state;
- a step function that fetches an x86 instruction from the memory, decodes it till the opcode byte(s) are read, and then executes it by calling a top-level opcode dispatch function — this dispatch function is basically a giant case statement that calls the appropriate instruction semantic function corresponding to the opcode; and
- a run function, which is the top-level interpreter that calls the step function iteratively.

The 64-bit model offers two views of x86 machines: the *system-level* view — when the `app-view` field in the machine state is `nil`, and the *application-level* view — when the `app-view` field is not `nil`. The system-level view is intended to specify the x86 ISA as accurately as possible. In this view, one can verify system programs that have access to system state, such as the memory management data structures. The application-level view can be used to simulate and verify application programs in the same sort of environment that programmers use to develop such programs. In this view, the operating assumption is that the underlying operating system and system features of the ISA behave as expected. For instance, unlike the system-level view, this view does not include the model of paging — it offers the linear memory abstraction instead of the physical memory. Also, the application-level view does not contain the specification of system instructions (which are present in the system-level view), e.g., instruction `LGDT`, which manipulates the segmentation data structures. However, the application-level view does specify 64-bit segmentation — i.e., `FS`/`GS`-based logical to linear address translation,[2] — but here it makes the following assumption: the hidden parts of these segment selectors are assumed to contain the correct segment base addresses. This assumption is reasonable because these hidden parts of the registers are automatically populated by the processor when the registers are loaded, and the instructions that load these registers are system-level instructions — thus, it is assumed that an operating system routine correctly loaded these registers. These views aim to mitigate the trade-off between verification utility and verification effort. Of course, if such assumptions that come with the application-level view's higher-level of abstraction are unacceptable to a user, he or she can operate in the system-level view for *all* programs.

This model has been used to do proofs about both application and system programs. Some examples of application programs verified using the model are a population count program, a word-count program, and a data-copy program. All of these programs except population count were verified by using the lemma libraries that are included in the `x86isa` library; the population count program was verified completely automatically using the `GL` library [26, 27]. The 64-bit model is also compatible with the Codewalker library [2]; Codewalker has been used to reason about small application programs like the x86 machine-code routine to compute the factorial of a number. An example of a system program verified using the `x86isa` library is zero-copy, which efficiently copies data from a source to a destination memory location by modifying the memory management data structures (i.e., using the *copy-on-write*

[2] Recall from Section 2.1 that for other segment selectors, the linear and logical addresses are the same in 64-bit mode.

technique).

The `x86isa` library uses ACL2 features like abstract stobjs [13] to provide both reasoning and execution efficiency. The simulation speed of the 64-bit model is around 300,000 instructions/second in the system-level view and 3 million instructions/second in the application-level view.[3] As a direct consequence, it has been possible to extensively validate the 64-bit model by running co-simulations against Intel's x86 machines, thereby increasing confidence in the model's accuracy. A goal of the `x86isa` library is to make the x86 ISA specification accessible to users, and as such, these books are extensively documented [1, `:doc x86isa`] — both from a user and a developer's point of view.

For more about the 64-bit model and its use in 64-bit code analysis, we refer an interested reader elsewhere — see [12] for a more detailed overview and [11, 14] for an in-depth report.

3 Extension of the Model

In this section, when we refer to code running in *32-bit mode*, we mean *application* software running in either 32-bit protected mode or compatibility sub-mode of IA-32e mode.

3.1 Challenges

Extending the model to 32-bit mode was not simply a matter of generalizing the sizes of the operands and addresses manipulated by the instructions. As explained in Section 2.1, 32-bit mode is more complicated than 64-bit mode, due to the legacy features that it provides.

In particular, in 32-bit mode, memory accesses are more complicated than in 64-bit mode: 32-bit mode uses segmentation, which is almost completely disabled in 64-bit mode. The 64-bit model made, throughout, the reasonable assumption that the effective addresses in instruction pointer, stack pointer, addressing mode base registers, etc. were also linear addresses — the only exception was when `FS` and `GS` segment selectors were used, in which case their bases were explicitly added to effective addresses to get the corresponding linear addresses. Effective/linear addresses were checked to be canonical as needed. This had to be generalized in the extended model: effective and linear addresses had to be differentiated, and segment bound checks had to be performed, before adding segment bases to effective addresses.

This required some changes in the interfaces between certain parts of the model, e.g., in the signature of the ACL2 functions to read and write memory. Generally, changes to interfaces and their semantics may break proofs whose restoration may require different or more general lemmas and invariants. As explained in Section 2.2, the 64-bit model was accompanied by several correctness proofs of non-trivial 64-bit programs — besides the termination, guard, and other proofs in the model proper. To keep the adaptation of all these proofs more manageable, the changes had to be carried out in incremental steps, to the extent possible. An overarching goal was to keep the whole build working at all times.

3.2 Mode Discrimination

Since the processor does certain things differently depending on whether the current mode is 64-bit or 32-bit, a predicate is needed to discriminate between the two modes in the model. The 64-bit model included a predicate `64-bit-modep` over the x86 state, defined to be always `t`, which was rarely called. This predicate was extended to distinguish between the two modes: if the `LMA` bit of the `IA32_EFER`

[3] All such measurements mentioned in this paper have been done on a machine with Intel Xeon E31280 CPU @ 3.50GHz with 32GB RAM.

register is set (i.e., the processor is in IA-32e mode), and the `L` bit of the `CS` register[4] is set (i.e., the current application is running in 64-bit sub-mode, not in compatibility 32-bit sub-mode), the predicate returns `t`; otherwise, the processor is in either protected 32-bit mode or compatibility 32-bit mode, and the predicate returns `nil`. Later, calls of this predicate were replaced by `(eql proc-mode *64-bit-mode*)`, where the value of the variable `proc-mode` is the current processor mode, read once from the x86 state at each execution step and passed to many of the model's functions; for simplicity, in the rest of the paper we still show calls of `64-bit-modep`.

In order to extend the instructions covered by the 64-bit model to 32-bit mode one by one, each branch of the top-level opcode-based dispatch was wrapped into a conditional of this form:

```
(if (64-bit-modep x86) <do-as-before> <throw-error>)
```

This let the model behave as before in 64-bit mode, and provided a clear indication, in concrete or symbolic execution, of instructions not yet extended to 32-bit mode. The addition of these conditionals required the addition of many `64-bit-modep` hypotheses to the existing theorems to keep all the proofs working (which rely on instructions not throwing errors in normal circumstances), a tedious but simple task; however, the extensions described in the subsequent sections required additional work to keep all the proofs working — see Section 3.10 for details.

3.3 Segmentation

Instructions manipulate operands in registers and memory, including stack operands and immediate operands. The register access functions in the 64-bit model actually needed no extension for 32-bit mode. However, as mentioned in Section 3.1, the memory access functions in the 64-bit model had to be extended to take segmentation into account. These memory access functions are used not only for operands, but also for fetching the bytes that form instructions (prefixes, opcodes, etc.). So adding segmentation was a prerequisite to extending any instruction. Segmentation was modeled in a form that is used uniformly in both 64-bit and 32-bit mode, as explained below.

The 64-bit model already included state components for the segment registers, including their hidden parts. A new function `segment-base-and-bound` was added to retrieve the base and bounds of a segment from (the hidden part of) the corresponding segment register in the x86 state. This function takes as arguments the machine state and (the identifying index of) a segment register and returns the base address, lower bound, and upper bounds of the segment as results via `mv`. More precisely, in 64-bit mode, the bases of the `FS` and `GS` segments are retrieved from model-specific registers that are physically mapped (in the processor) to the hidden base fields of the `FS` and `GS` registers, and that provide 64-bit linear addresses beyond the 32-bit linear addresses used in 32-bit mode. The bounds of a segment refer to effective addresses, which are offsets into the segment. The lower bound is 0 and the upper bound is the limit field in the segment register, unless the `E` bit in the segment register is set. This `E` bit indicates an expand-down segment (typically used for a stack): the lower bound is the limit field in the segment register, and the upper bound is either $2^{32}-1$ or $2^{16}-1$, depending on the `D/B` bit in the segment register, which indicates a 32-bit segment if set and a 16-bit segment if cleared. In 64-bit mode, since the bounds are ignored, `segment-base-and-bounds` just returns 0 as both bounds; it also just returns 0 as the base if the segment register is not `FS` or `GS`.

A new function `ea-to-la` was added to perform segment address translation; the name stands for 'effective address to linear address', chosen in analogy to the `la-to-pa` function that, in the 64-bit model,

[4] By expressions like 'bit x of the segment register y' we mean, more precisely, 'bit x of the segment descriptor whose selector is currently loaded in the segment register y'.

performed paging address translation, i.e., turned linear addresses into physical addresses.[5] `ea-to-la` takes as inputs the machine state and a logical address, which consists of an effective address and (the index of) a segment register; it returns as output a linear address, obtained by adding the effective address to the base of the segment. `ea-to-la` also returns as output a flag with error information if the effective address is outside the segment's bounds; error flag and linear address are returned via `mv`. The segment base and bounds are retrieved via the function `segment-base-and-bounds` described earlier. In 64-bit mode, there are no checks against the segment's bounds, but instead the resulting address is checked to be canonical; the resulting address differs from the input effective address only if the segment is `FS` or `GS`, whose base is added to the effective address.

The 64-bit model included top-level memory access functions to read and write unsigned and signed values of different sizes (bytes, words, double words, etc.). These were called `rmXX`, `rimXX`, `wmXX`, and `wimXX`, where `r` indicates 'read', `w` indicates 'write', `i` indicates 'integer' (vs. natural number, i.e., the functions with `i` read/write signed integers while the functions without `i` read/write unsigned integers), and `XX` consists of two digits indicating the bit size (`08`, `16`, `32`, etc.). There were also more general functions `rm-size`, `rim-size`, `wm-size`, and `wim-size`, which took the size in bytes as an additional input. These functions took linear addresses as inputs; they assumed that the base of the `FS` or `GS` segment had already been added when applicable, and that alignment checks had already been performed when needed.

New top-level memory access functions were added to read and write unsigned and signed values of different sizes, mirroring the aforementioned ones. They are called `rmeXX`, `rimeXX`, `wmeXX`, `wimeXX`, `rme-size`, `rime-size`, `wme-size`, and `wime-size`, where `e` stands for 'effective address'. These new functions take as inputs logical addresses (as effective addresses and segment registers)[6] instead of linear addresses, call `ea-to-la` to obtain linear addresses, perform alignment checks on the linear addresses if indicated by some additional inputs to these new functions, and then call the old 64-bit mode top-level memory access functions (which are no longer at the top level in the extended model). For clarity, the old functions were renamed to `rmlXX`, `rimlXX`, `wmlXX`, `wimlXX`, `rml-size`, `riml-size`, `wml-size`, and `wiml-size`, where `l` stands for 'linear address'.

The inputs of all these functions (new and old) include the machine state; the inputs of the write functions also include the value to write, while the inputs of the read functions also include a flag `:r` or `:x` to distinguish between reading data and fetching instructions. The read functions return an error flag, the value read, and the possibly updated machine state (see Footnote 9), via `mv`. The write functions return an error flag and the possibly updated machine state, via `mv`.

3.4 Paging

As explained in Section 2.1, there are three kinds of paging in the x86 processor. PAE and 32-bit paging are used only in 32-bit mode, and 4-level paging is used only in 64-bit mode. The 64-bit model included the latter, but not the former.[7]

As explained in Section 2.2, the 64-bit model included the specification of paging only in its system-level view, not in its application-level view. These views carry over to the extended model as well.

[5] `ea-to-la` actually translates a logical address to a linear address, but the name `la-to-la` would not have worked well. There is a sense in which the effective address portion of a logical address is more "important" than the segment selector, which provides just a "context" for the translation.

[6] A similar observation applies to these names, as the one made for the name of `ea-to-la` above.

[7] To be precise, a file in the 64-bit model contained a start towards modeling the two paging modes used in 32-bit mode, but this file's content was not used in the rest of the model.

However, the extended model supports 32-bit mode only in application-level view for now, that is, it only supports the execution and verification of 32-bit application software, not 32-bit system software. As such, we have deferred the addition of PAE and 32-bit paging to the model (see Section 5).

3.5 Instruction Fetching

The 64-bit model fetched instruction bytes by using the instruction pointer in the `RIP` register as the linear address passed to the top-level memory access functions `rm08` etc. mentioned in Section 3.3. The instruction pointer was incremented using ACL2's `+` operation, checking that the result was canonical after each increment.

In 32-bit mode, the instruction pointer is in the `EIP` or `IP` register, which refer to the low 32-bit or 16-bit portions of the `RIP` register, respectively. The `D` bit in the `CS` register selects the appropriate register: `EIP` if this bit is set and `IP` otherwise.

A new function `read-*ip` (where the `*` is meant to stand for `r`, `e`, or nothing) was added, to generalize the 64-bit mode reading of the instruction pointer from the `RIP` register. This new function takes as input the machine state and returns as output the instruction pointer. In 32-bit mode, the output is a 32-bit or 16-bit value, from the low bits of the `rip` component of the model of the x86 state.

A new function `add-to-*ip` was added to generalize the plain 64-bit increment of the instruction pointer and the subsequent canonical address check. This new function takes as inputs the machine state, an instruction pointer, and an integer amount to add (which may be negative); it returns as outputs the incremented instruction pointer and an error flag (`nil` if there is no error). In 32-bit mode, the input and output instruction pointers are 32-bit or 16-bit values, and the error flag is non-`nil` if the output instruction pointer is outside the limits of the `CS` register. Note that the input machine state is only used to check the `CS` limits, and that there is no output machine state: this function only increments instruction pointer values.

A new function `write-*ip` was added to store the final instruction pointer, after executing each instruction, into the `RIP`, `EIP`, or `IP` register. This function takes as inputs the machine state and the instruction pointer to store; it returns as output an updated machine state. In 32-bit mode, the instruction pointer is a 32-bit or 16-bit value that is stored in the low 32 or 16 bits of the `rip` component of the model of the x86 state.

The model of instruction fetching was modified to call the new functions `read-*ip` and `add-to-*ip` to manipulate the instruction pointer (while `write-*ip` is called by the models of the individual instructions), and to replace the calls to `rml08` (i.e., the renamed `rm08`; see Section 3.3) with calls to `rme08`. This change necessitated the largest effort, compared to the other changes to the 64-bit model, to keep all the proofs working: see Section 3.10 for details.

3.6 Stack Operations

Several instructions read from and write to the stack, via the stack pointer. The 64-bit model manipulated the stack pointer in a manner similar to the instruction pointer (see Section 3.5): the stack pointer was read from and written to the `RSP` register, and it was incremented and decremented via ACL2's `+` and `-` operations, checking that the result was canonical after each increment and decrement.

In 32-bit mode, the stack pointer is in the `ESP` or `SP` register, which are the low 32-bit or 16-bit portions of the `RSP` register, respectively. The `B` bit in the `SS` register selects the appropriate register: `ESP` if this bit is set and `SP` otherwise.

The extensions to the model for the stack pointer were similar to the extensions for the instruction pointer. New functions `read-*sp`, `add-to-*sp`, and `write-*sp` were added, quite analogous to the functions `read-*ip`, `add-to-*ip`, and `write-*ip` described in Section 3.5. Instead of `rip`, `read-*sp` and `write-*sp` read and write the `rsp` component of the model of the x86 state — only the low 32 or 16 bits, in 32-bit mode. Instead of the limits of the `CS` segment, `add-to-*sp` checks the new stack pointer value against the limits of the `SS` segment, taking into account the possibility of expand-down segments, which are typically used for the stack.

3.7 Addressing Modes

Other than stack and immediate operands, instructions can reference operands in memory via a variety of addressing modes. Each addressing mode involves a calculation of an effective address from one or more components, such as (the content of) a base register, an index to add to the base register, and a scaling factor with which to multiply the index before adding it to the base. The addressing mode to use is specified by the `ModR/M`, `SIB`, and displacement bytes of an instruction (not all of these bytes need to be present). There are 32-bit and 16-bit addressing modes [15, Tables 2.1 and 2.2]: the former apply to 64-bit mode, and to 32-bit mode when the address size is 32 bits; the latter apply to 32-bit mode when the address size is 16 bits. The address size is determined by the `D` bit of the `CS` register.

The 64-bit model included functions to decode the `ModR/M` and `SIB` bytes and to perform the effective address calculations for the 32-bit addressing modes, but not for the 16-bit addressing modes, which do not apply to 64-bit mode. In particular, the function `x86-effective-addr` performed the `ModR/M` and `SIB` decoding, and the effective address calculation, for the 32-bit addressing modes. Since the 16-bit addressing modes are fairly different from the 32-bit addressing modes, `x86-effective-addr` was renamed to `x86-effective-addr-32/64` (to convey that this is used for 32-bit and 64-bit addresses), a new function `x86-effective-addr-16` was added for the 16-bit addressing modes, and a new wrapper function `x86-effective-addr` was added that calls either `x86-effective-addr-32/64` or `x86-effective-addr-16`.

These functions have several inputs and several outputs. The inputs include the `ModR/M` and `SIB` bytes (0 if absent), the current instruction pointer (which is just past the opcode and, if present, the `ModR/M` and `SIB` bytes), and the machine state. The outputs are an error flag, the calculated effective address, the number of (displacement) bytes read (which is later added to the instruction pointer), and a possibly updated machine state. Based on the addressing mode, these functions may read the displacement bytes, advancing the instruction pointer; they call `rme08` to read these bytes, which may fail (see Section 3.3), in which case `x86-effective-addr` returns a non-`nil` error flag.

3.8 Operand Reading and Writing

Some instructions read and write their operands from and to memory or registers, based on some bits of the `ModR/M` byte — which, as explained in Section 3.7, is also used to determine the addressing mode when the operand is in the memory.

The 64-bit model included a function `x86-operand-from-modr/m-and-sib-bytes` that uniformly read an operand from memory or registers based on the `ModR/M` and `SIB` bytes. This function decoded the `ModR/M` byte, and, based on that, either read the operand value from a register, or from memory. In the latter case, it called `x86-effective-addr` to calculate the effective address, which it passed to `rml-size` (formerly `rm-size`) or its variants to read the value from that location; recall that in the 64-bit model an effective address was also a linear address. This function had several inputs and several outputs. The

inputs included the ModR/M and SIB bytes, the current instruction pointer (which is just past the opcode and, if present, the ModR/M and SIB bytes), and the machine state. The outputs included an error flag, the memory or register operand read, the calculated effective address (for a memory operand), the number of (displacement) bytes read (which is later added to the instruction pointer), and a possibly updated machine state. The calculated effective address is returned to avoid recalculating it when an instruction updates the operand, which is common (e.g., adding an immediate value to an operand in memory).

To uniformly write operands to memory or registers, the 64-bit model also included functions x86-operand-to-reg/mem and x86-operand-to-xmm/mem. These functions had several inputs, including the value to write, the effective/linear address (calculated externally, often by x86-operand-from-modr/m-and-sib-bytes as mentioned above), and the machine state. These functions returned an error flag and an updated machine state. The effective/linear address was passed to wml-size (formerly wm-size) or its variants to write the operand value to memory.

To support 32-bit mode, these operand read and write functions had to be generalized to traffic in effective addresses rather than linear addresses. In particular, the index of the segment register to use had to be an additional input. This required a change to the interface of these three functions, which are called in many places by the instruction models. Making this change in one shot would have required a lot of adjustments to instruction models and could have broken many proofs. Thus, the change was staged as follows: (1) introduce new versions of these functions, called x86-operand-from-modr/m-and-sib-bytes$, x86-operand-to-reg/mem$, and x86-operand-to-xmm/mem$, with the new interface and functionality; (2) separately modify the instruction models, one by one, to call the new functions instead of the old ones, repairing any failing proofs; (3) remove the old functions when they are no longer called; (4) rename the new functions to remove the ending $. The new functions to read and write operands call the new top-level memory functions rme-size and wme-size or their variants to access the operands in memory.

The segment register to pass to the new functions is determined using the default segment selection rules [15, Volume 1, Table 3-5] based on the kind of instruction and the addressing mode, and the presence of a segment-override prefix before the opcode. This determination is made by a new function select-segment-register that was added for this purpose. This function has several inputs, including the segment override prefix (0 if absent) and the machine state; this functions returns (the index of) a segment register as the only output.

3.9 Instructions

With all the extensions described above in place, extending the individual instructions' semantic functions to 32-bit mode was comparatively easy. This was also facilitated by the fact that the 64-bit model already supported different operand sizes for many of the core operations carried out by the instructions (e.g., the arithmetic and logic operations) leaving just the operand size determination logic of the instruction to be extended.

In particular, the existing function select-operand-size was extended to return the operand size in 32-bit mode, while returning the same result as before in 64-bit mode. The interface of this function was expanded to give it access to the D bit of the CS register, needed to determine the operand size in 32-bit mode: its inputs include the machine state and some of the decoded instruction prefixes; it returns the size as the only output. The instruction semantic functions that already called select-operand-size could then automatically extend their operand size determination to 32-bit mode.

A new function select-address-size was introduced to determine the address size in both 32-bit and 64-bit mode. Its inputs include the machine state and some of the decoded instruction prefixes; it

returns the size as the only output. The portions of the instruction semantic functions that determined the address size in 64-bit mode only were replaced with calls to this new function.

In the 64-bit model, the instruction semantic functions were reading immediate operands via `rm08` and related functions (renamed to `rml08` etc.) called on the instruction pointer (treated as a linear address), incrementing the instruction pointer via ACL2's `+`, and writing the final instruction pointer directly into the `rip` component of the x86 state. To support 32-bit mode as well, this was changed to read immediate operands via `rme08` and related functions called on the instruction pointer (treated as an effective address in the code segment), and to use the new functions described in Section 3.5 to increment and store the instruction pointer.

In the 64-bit model, the instruction semantic functions were reading and writing the stack pointer directly from and to the `rsp` component of the x86 state, calling `rm-size`, `wm-size`, and related functions (renamed to `rml-size`, `wml-size`, etc.) on the stack pointer (treated as a linear address) to read and write stack operands, and incrementing and decrementing the stack pointer via ACL2's `+` and `-`. To support 32-bit mode as well, this was changed to call `rme-size`, `wme-size`, and related functions on the stack pointer (treated as an effective address in the stack segment) to read and write stack operands, and to use the new functions described in Section 3.6 to read, write, increment, and decrement the stack pointer. The instruction semantic functions in the 64-bit model were performing alignment checks on the stack pointer as needed: this code was removed because alignment checks are performed inside `rme-size`, `wme-size`, and related functions, after effective addresses are translated to linear addresses, resulting in better factored code as a by-product.

Following the staged approach described in Section 3.8, calls to `x86-operand-from-modr/m-and-sib-bytes` etc. in the instruction semantic functions, were gradually redirected to call `x86-operand-from-modr/m-and-sib-bytes$` etc. instead. Since these new functions traffic in effective addresses instead of linear addresses, two additional adjustments had to be made to the instruction semantic functions. The first adjustment was to remove alignment checks performed on the linear addresses of the operands: alignment checks are already performed by (functions called by) the new functions, after translating the now effective addresses of the operands into linear addresses. The second adjustment was to remove the addition of the `FS` or `GS` segment base (when applicable) to the linear addresses of the operands: any segment base is added to the now effective addresses of the operands when they are translated into linear addresses, by (functions called by) the new functions. These adjustments resulted in better factored code as a by-product.

After each instruction semantic function was extended to 32-bit mode in the manner explained above, the top-level dispatch was adjusted to call the function not only in 64-bit mode, but also in 32-bit mode. Concretely, this was done by removing the wrapping conditional `(if (64-bit-modep x86) ...)` described in Section 3.2.

All the non-floating-point instructions of the 64-bit model have been extended to 32-bit mode, at least for the application-level view of execution.[8] The 34 floating-point instructions supported by the 64-bit model also need to be extended to 32-bit mode. We anticipate that this task will be straightforward because we have almost all the pieces necessary to extend the floating-point instructions. Specification functions from Russinoff's RTL library [3] are used to specify the core operations of these instructions, and these functions are already parameterized by size. As far as operand access and update is concerned: we have already extended memory access functions to work in 32-bit mode, but since the floating-point instructions use a different register set from the general-purpose instructions, we just need to extend

[8] The far variant of `JMP` does not handle 32-bit mode system segments yet, but these are only needed for the system-level view of execution. Nonetheless, we plan to add support for these soon.

register access functions to work in 32-bit mode.

3.10 Proof Adaptations

In extending the 64-bit ISA model to support 32-bit mode, so far we have discussed issues that were not especially unique to a formal model; for instance, extending a C emulator of the 64-bit x86 ISA to support the execution of 32-bit programs would likely require making similar changes. However, as discussed in Section 3.1, our aim was also to preserve the proofs done in the 64-bit model, including the proofs of the model's functions (e.g., guards) and the proofs of correctness of the 64-bit programs.

The proofs of 64-bit programs with the 64-bit model involved functions and expressions that were generalized for 32-bit mode as described in the previous subsections. Thus, theorems were added to rewrite the more general functions and expressions into the "old" ones under the `64-bit-modep` hypothesis. For instance, the following theorem was added, to rewrite the reading of the `RSP`, `ESP`, or `SP` register into just the reading of `RSP` (where `rgfi` reads the 64-bit value of the register specified as argument):

```
(implies (64-bit-modep x86)
         (equal (read-*sp x86) (rgfi *rsp* x86)))
```

Since the 64-bit proofs include the `64-bit-modep` hypothesis (added as described in Section 3.2), these rewrite rules fire and help reduce (sub)goals to the form they had in the 64-bit model prior to its extensions. Additional theorems had to be added in order to let the `64-bit-modep` hypothesis be relieved for updated x86 states, such as the following one, asserting that reading from a linear address does not change mode:[9]

```
(equal (64-bit-modep (mv-nth 2 (rml08 addr r-x x86)))
       (64-bit-modep x86))
```

All these theorems generally sufficed to keep the proofs of 64-bit application programs, i.e., in the application-level view.

Even though the extended model does not (yet) cover the system-level view, additional changes were needed to adapt the reasoning strategy to work for the proofs of some 64-bit system programs, as described below. Lemma libraries included in the 64-bit model for formally analyzing system programs contained utilities to reason about the memory management data structures — specifically, the paging structures. Every linear-to-physical address translation on the x86 ISA causes a traversal of entries in these paging structures,[10] which produces some side effects. The processor can set two bits in the paging entries during address translation: the *accessed* and *dirty* flags, which effectively *mark* the entries that govern the translation of a linear address. However, the mapping of that linear address to its corresponding physical address does not change as a result of these side-effect updates. This means that statements like the following are true, where `spec-function` is an x86 specification function that reads from or writes to memory, and `<hyp>` ensures that (1) `x86-1` and `x86-2` can differ only in the values of the accessed and dirty flags, and (2) `spec-function` traffics only in locations that are disjoint from the affected (i.e., marked) entries in the paging structures:

```
(implies <hyp> (equal (spec-function <args> x86-1) (spec-function <args> x86-2)))
```

The 64-bit model stored such theorems as congruence rules, as follows:

[9] Note that reading from memory may change the x86 state in general, by changing the 'accessed' flag in the paging data structures; this flag is discussed later in this section. `rml08` and similar functions return the possibly updated state as the third component of their multi-value result, accessed via `(mv-nth 2 ...)`.

[10] We ignore translation look-aside buffers and caches for this discussion.

```
(implies (xlate-equiv-memory x86-1 x86-2)
         (equal (spec-function-alt <args> x86-1)
                (spec-function-alt <args> x86-2)))
```

The function `spec-function-alt` is exactly the same as `spec-function` under one condition: it traffics only in locations that are disjoint from all the paging entries that are marked during address translation; if this condition is false, then `spec-function-alt` simply returns the input x86 state — unmodified. The relation `xlate-equiv-memory` says that two x86 states are equivalent if: (1) their paging structures are equal, modulo the accessed and dirty bits; and (2) all other memory locations are equal. This arrangement facilitates "conditional" congruence-based reasoning[11] — a rewrite rule transforms `spec-function` to `spec-function-alt` whenever applicable, and then these congruence rules can come into play. This strategy works well for system programs that do not explicitly modify the paging data structures. As a result of extending the model, we had to add `64-bit-modep` both to `xlate-equiv-memory` and to the conditions under which `spec-function-alt` is equal to `spec-function`.

We note that the top-level rewrite rules in the model are not mode-specific — an example is the opener lemma of the step function, whose hypotheses determine when to rewrite a call of the step function to the function that dispatches control to the appropriate instruction semantic function; basically, this lemma helps in "unwinding" the x86 interpreter during proofs. Thus, we share lemmas across different modes as much as possible. Rules about intermediate specification functions that do have a hypothesis like `(64-bit-modep x86)` come in useful during program verification when we need to be cognizant of the operating mode of a processor.

3.11 Performance

As discussed in Section 2.2, a high simulation speed is crucial for performing model validation via co-simulations against real machines. We first discuss how the simulation speed of 64-bit programs was adversely impacted by extending the model to support 32-bit programs and how we fixed this issue. Then we present performance-related information about the simulation of 32-bit programs.

3.11.1 64-bit Mode

Our initial attempt to extend the 64-bit model led to an unfortunate decrease in its simulation speed in the application-level view from around 3 million instructions/second to 1.9 million instructions/second.[12] Using Linux's Perf utility and ACL2's `profile` mechanism [1, `:doc profile`], we discovered that the predicate `64-bit-modep` was the main culprit — every call of this function allocated 16 bytes on the heap. Recall that this predicate reads values from fields in the x86 state (`IA32_EFER` and `CS` registers) and these fields usually contain large integers, i.e., *bignums*. Manipulating bignums is inefficient because they are stored on the heap and require special arithmetic functions. Contrast these bignums with *fixnum* integers, which can fit in the host machine's registers and whose computation can be done with built-in arithmetic instructions. For example, Lisp compilers can add two fixnum values using the add instruction of the host machine, but the addition of two bignum values is done by calling a Lisp function that is considerably slower than a machine instruction and that may also allocate memory on the heap.

Though `64-bit-modep` allocates just 16 bytes on the heap, it is called once at the beginning of the execution of *each* instruction to determine the processor's operating mode for that instruction. Therefore,

[11] Recall that a congruence rule cannot have anything other than a known equivalence relation as its only hypothesis.

[12] There was a similar decrease in speed in the system-level view, but for this discussion, we focus only on the application-level view.

the number of bytes allocated because of this function increases with the number of instructions to be executed. For instance, for a simple C function that computes the n-th Fibonacci number by implementing the Fibonacci recurrence relation, the number of 64-bit x86 instructions to be executed to compute `fib(30)` was 31,281,993. This program run allocated 500,511,936 bytes on the model, all because of the bignum computations done in `64-bit-modep`. Since we typically run a large number of instructions at once during co-simulations, this problem needed to be fixed.

We solved this issue by changing the definition of `64-bit-modep` — it is now an `mbe`, where our initial definition has been retained in the `:logic` part. For the `:exec` part, we used a function called `bignum-extract` from a pre-existing ACL2 community book [1, `:doc bignum-extract`]. Logically, `(bignum-extract x n)` returns the n-th 32-bit slice of an integer `x`. Therefore, in the `:exec` part of `64-bit-modep`, we simply extracted relevant 32-bit slices of the x86 fields using this function and read the required values from them. During execution, we used a raw Lisp replacement for `bignum-extract` (already provided in `std/bitsets/bignum-extract-opt.lisp`) that takes advantage of 64-bit CCL's implementation. CCL stores bignums in vectors of 32 bits, and extracting a 32-bit chunk of a bignum to operate on avoids expensive computations involving the *entire* bignum.

With this solution, the model's simulation speed for 64-bit programs went back to what it was at the beginning of this project (for both application-level and system-level views). We note that the number of bytes allocated on the heap to compute `fib(n)` is now independent of the number of instructions to be executed.

However, this solution does have two drawbacks:

1. The raw Lisp replacement for `bignum-extract` may not be efficient for Lisps other than CCL.
2. For efficient execution (not for reasoning), this solution requires a trust tag.

In the future, we plan to implement a different solution that will not suffer from the above drawbacks. We will take advantage of the x86 abstract stobj by defining the fields involved in the computation of `64-bit-modep` as multiple fixnum-sized fields in the concrete stobj, defining accessor and updater functions over the concrete stobj that operate on these fields collectively, and then proving that these concrete functions correspond to the currently existing accessor and updater functions defined over the abstract stobj. A benefit of this approach is that after the abstract stobj is defined, no other proofs would be affected.

3.11.2 32-bit Mode

The simulation speed of 32-bit programs in the application-level view is around 920,000 instructions/second — note that, for 32-bit mode, the system-level view has not been implemented yet. One reason for this slower speed (as compared to 64-bit mode) is that 32-bit programs often have to refer to segment descriptors over the course of their execution, and these operations usually involve bignums.

Our focus has been on the functionality of the extensions first rather than the performance, but in the future, we intend to speed up 32-bit mode using the same kind of approaches as those used in 64-bit mode. That being said, the current simulation speed of 32-bit programs has been more than adequate to run tests at this stage of development.

3.12 Documentation

As mentioned in Section 2.2, the 64-bit model was extensively documented. As the model was being extended to support 32-bit mode, the documentation was extended accordingly.

4 Related Work

The x86-CC and x86-TSO models of the x86 ISA in HOL [23, 25] are focused on concurrent memory access in multiprocessor systems. They support a few tens of instructions, defined via parameterized monadic "micro-code" operations that can be instantiated for both sequential and concurrent semantics. The models are accompanied by a tool to test the sequential semantics in HOL, and a tool to test the concurrent semantics in (unverified) OCaml.

The model of the x86 ISA in Coq developed for RockSalt [20] supports the verification of a binary analyzer that checks conformance to a sandboxing policy; however, the model is independent of this analyzer. It supports about 100 instructions in 32-bit mode only, and does not model paging tables. It uses two Domain-Specific Languages (DSLs) embedded in Coq to declaratively specify instruction decoding and to express instruction semantics in terms of "micro-code" operations. The model can simulate and validate (against real processors) about 50 instructions per second.

Degenbaev's model of the x86 ISA [9] supports both concurrent memory access in multiprocessor systems and a large number of instructions; in this work, the instruction semantics is modeled via a DSL. Baumann's related model of the x87 instruction set [6] supports floating-point instructions. These models are written in "pencil-and-paper", not in a theorem prover or other formal tool.

Models of the ISA of other processors have been developed in ACL2 and other theorem provers, e.g., [24, 10, 22] In particular, the model of the y86 ISA (an x86-like ISA) in ACL2 [4] is a precursor of our model. The modeling approaches for these other processors are similar to the ones for the x86 ISA models in ACL2 and other provers, but the ISAs of these other processors are less complicated than the x86.

Models of both intermediate-level and source-level languages have been also developed in ACL2 and other theorem provers, e.g., [17, 21]. There are some commonalities between the modeling approaches for these languages and the modeling approaches for processor ISAs, such as the suitability of an interpreter style of operational semantics. There are also many differences, e.g., processor ISAs typically have fewer data types and do not have to include specifications of operations like casting (e.g., converting integers to characters, using pointers as integers, etc.).

There exist several x86 emulators (e.g., Bochs, DOSBox, PCem, QEMU, Unicorn) that are written in more conventional programming languages (e.g., C++), not in the logical languages of theorem provers. The development of these emulators faces many of the same issues as the development of a formal model, such as the sheer complexity and the occasional ambiguity of the official documentation. However, the proof aspects are unique to formal models. In particular, extending or improving an emulator involves regression testing, while extending or improving a formal model may also involve proof adaptation.

5 Future Work

In this paper, we discussed our ongoing effort to extend the pre-existing 64-bit model of the x86 ISA to support Intel's 32-bit mode. In the short or medium term, we plan to add support for the following x86 ISA features:

- More instructions, prioritizing on the ones encountered in new programs to verify and on the ones that are more commonly used in general. Every new instruction will include both 64-bit and 32-bit mode support.
- The two kinds of paging used in 32-bit mode (see Section 3.4). This will enable the formal analysis of 32-bit system software, besides 32-bit application software.

- The remaining processor modes (real-address, virtual-8086, and system management), along with the instructions to switch between modes.
- The widely-used Intel's vector processing features (i.e., AVX, AVX2, AVX-512) that offer enhanced capabilities to perform data-intensive (integer and floating-point) computations efficiently. The x86 model already supports the decoding of these instructions, but this task will entail modifying some old instructions that now support these new features and adding many new instructions. A notable thing is that vector processing features differ between the 64-bit and 32-bit modes, thereby making this task a little more interesting.

In the short or medium term, we also plan to validate the 32-bit extensions against actual x86 processors, as was done for the 64-bit model.

A longer-term project is to support a concurrent semantics. At the very least, this will likely require some refactoring of the instruction semantic functions, which currently treat the execution of each instruction as an atomic event, because the x86 state is updated at the end of each instruction.

Another direction is to make some parts of the specification more declarative and concise via more macros to generate boilerplate code. Examples are the binding of various instruction bytes and fields (e.g., prefixes) to variables and common checks (e.g., many instructions throw a `#UD` exception if the `LOCK` prefix is used). A more ambitious endeavor is to raise the level of abstraction of some parts of the specification to the point of making them non-executable, and then use transformation tools like APT [5, 8] to generate efficient executable refinements of the specification along with correctness proofs checked by ACL2.

As briefly mentioned in Section 1, some work has started on 32-bit program verification, mainly aimed at detecting malware variants via semantic equivalence checking by symbolic execution. This work, performed by Eric Smith with some help about the model from the first author, will be carried forward in the near future.

Longer-term envisioned uses of the model include verified compilation from programming languages (e.g. C) to binaries, as well as binary program derivations by stepwise refinement using transformation tools like APT [5, 8], possibly on deeply embedded representations [7].

Acknowledgements

This work was partially supported by DARPA under Contract No. D16PC00178. Thanks to Eric Smith and James McDonald for providing feedback based on their use of the model for proofs, symbolic execution, and simulation.

References

[1] *ACL2 Theorem Prover and Community Books: User Manual*. Online; accessed: September 2018. `http://www.cs.utexas.edu/~moore/acl2/manuals/current/manual`.

[2] *ACL2 Books: Codewalker*. Online; accessed: September 2018. `https://github.com/acl2/acl2/tree/master/books/projects/codewalker`.

[3] *ACL2 Books:* `rtl/rel11` - *A Formal Theory of Register-Transfer Logic and Computer Arithmetic*. Online; accessed: September 2018. `https://github.com/acl2/acl2/tree/master/books/rtl/rel11`.

[4] *ACL2 Books: y86 Specifications*. Online; accessed: September 2018. `https://github.com/acl2/acl2/tree/master/books/models/y86`.

[5] *APT (Automated Program Transformations): Web page*. Online; accessed: September 2018. http://www.kestrel.edu/home/projects/apt.

[6] Christoph Baumann (2008): *Formal Specification of the x86 Floating-Point Instruction Set*. Master's thesis, Universität des Saarlandes. Online; accessed: September 2018. http://www-wjp.cs.uni-saarland.de/publikationen/Ba08.pdf.

[7] Alessandro Coglio (2014): *Pop-Refinement*. *Archive of Formal Proofs*. Formal proof development. http://afp.sf.net/entries/Pop_Refinement.shtml.

[8] Alessandro Coglio, Matt Kaufmann & Eric Smith (2017): *A Versatile, Sound Tool for Simplifying Definitions*. In: *Proc. 14th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2017)*, pp. 61–77, doi:10.4204/EPTCS.249.5.

[9] Ulan Degenbaev (2012): *Formal Specification of the x86 Instruction Set Architecture*. Ph.D. thesis, Universität des Saarlandes. Online; accessed: September 2018. http://rg-master.cs.uni-sb.de/publikationen/UD11.pdf.

[10] Anthony Fox & Magnus O. Myreen (2010): *A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture*. In Matt Kaufmann and Lawrence C. Paulson, editor: *Interactive Theorem Proving*, *Lecture Notes in Computer Science* 6172, Springer Berlin Heidelberg, pp. 243–258, doi:10.1007/978-3-642-14052-5_18.

[11] Shilpi Goel (2016): *Formal Verification of Application and System Programs Based on a Validated x86 ISA Model*. Ph.D. thesis, Department of Computer Science, The University of Texas at Austin. Online; accessed: September 2018. http://hdl.handle.net/2152/46437.

[12] Shilpi Goel (2017): *The x86isa Books: Features, Usage, and Future Plans*. In: *Proc. 14th International Workshop on the ACL2 Theorem Prover and its Applications (ACL2-2017)*, pp. 1–17, doi:10.4204/EPTCS.249.1.

[13] Shilpi Goel, Warren A. Hunt Jr. & Matt Kaufmann (2013): *Abstract Stobjs and Their Application to ISA Modeling*. In: *Proc. International Workshop on the ACL2 Theorem Prover and its Applications (ACL2-2013)*, pp. 54–69, doi:10.4204/EPTCS.114.5.

[14] Shilpi Goel, Warren A. Hunt, Jr. & Matt Kaufmann (2017): *Engineering a Formal, Executable x86 ISA Simulator for Software Verification*. In: *Provably Correct Systems*, Springer International Publishing, Cham, pp. 173–209, doi:10.1007/978-3-319-48628-4_8. Editors: Mike Hinchey, Jonathan P. Bowen, and Ernst-Rüdiger.

[15] Intel Corporation: *Intel 64 and IA-32 Architectures Software Developer's Manual*. Online; accessed: September 2018. Order Number: 325462-067US. (May, 2018). http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[16] John Leyden (2017): *64-bit malware threat may be itty-bitty now, but it's only set to grow*. *The Register*. Online; accessed: September 2018. https://www.theregister.co.uk/2017/05/24/64bit_malware.

[17] Hanbing Liu & J S. Moore (2004): *Java program verification via a JVM deep embedding in ACL2*. In: *International Conference on Theorem Proving in Higher Order Logics*, Springer, pp. 184–200, doi:10.1007/978-3-540-30142-4_14.

[18] John McCarthy (1964): *A Formal Description of a Subset of Algol*. In T. B. Steel, editor: *Formal Language Description Languages for Computer Programming, North Holland, 1966*, pp. 1–12.

[19] J S. Moore: *Mechanized Operational Semantics*. Online; accessed: September 2018. Lectures in the Marktoberdorf Summer School (August 5-16, 2008). http://www.cs.utexas.edu/users/moore/publications/talks/marktoberdorf-08/index.html.

[20] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan & Edward Gan (2012): *RockSalt: Better, Faster, Stronger SFI for the x86*. In: *Proc. of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, ACM, pp. 395–404, doi:10.1145/2254064.2254111.

[21] Michael Norrish (1998): *C formalised in HOL*. Ph.D. thesis, University of Cambridge. Online; accessed: September 2018.
https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-453.pdf.

[22] Alastair Reid (2016): *Trustworthy specifications of ARM ® v8-A and v8-M System level architecture*. In: *2016 Formal Methods in Computer-Aided Design (FMCAD)*, pp. 161–168, doi:10.1109/FMCAD.2016.7886675.

[23] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus Myreen & Jade Alglave (2009): *The Semantics of x86-CC Multiprocessor Machine Code*. In: *Proc. 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 379–391, doi:10.1145/1594834.1480929.

[24] Jun Sawada & Warren A. Hunt, Jr. (2002): *Verification of FM9801: An Out-of-Order Microprocessor Model with Speculative Execution, Exceptions, and Program-Modifying Capability*. *Formal Methods in Systems Design* 20(2), pp. 187–222, doi:10.1023/A:1014122630277.

[25] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli & Magnus O. Myreen (2010): *x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors*. *Communications of the ACM* 53(7), pp. 89–97, doi:10.1145/1785414.1785443.

[26] Sol Swords (2010): *A Verified Framework for Symbolic Execution in the ACL2 Theorem Prover*. Ph.D. thesis, Department of Computer Science, The University of Texas at Austin. Online; accessed: September 2018.
http://hdl.handle.net/2152/ETD-UT-2010-12-2210.

[27] Sol Swords & Jared Davis (2011): *Bit-Blasting ACL2 Theorems*. In: *Proc. 10th International Workshop on the ACL2 Theorem Prover and its Applications (ACL2-2011)*, pp. 84–102, doi:10.4204/EPTCS.70.7.