Booting Linux on the ACL2 x86-ISA Model

Yahya Sohail and Warren A. Hunt, Jr.

Original Authors: Shilpi Goel, W. Hunt, Jr., Matt Kaufmann
UT Contributors: Cuong Chau, J Strother Moore
Additional UT Contributors: Soumava Ghoush, Keshav Kini, Robert Krug, Ben Selfridge, Rob Sumners, Nathan Wetzler
Centaur Technology Contributors: Anna Slobodova, Jared Davis, Sol Swords
Kestrel Contributor: Alessandro Coglio
Sponsors: Centaur Technology, DARPA, Intel, NSF

Computer Science Department
The University of Texas; Austin, TX (USA)
https://yahyasohail.com, https://cs.utexas.edu/∼hunt

October 10, 2024

Table of Contents

The ACL2 x86-ISA Specification Project Goals

What is an ISA? Why is the formal definition of ISA valuable?

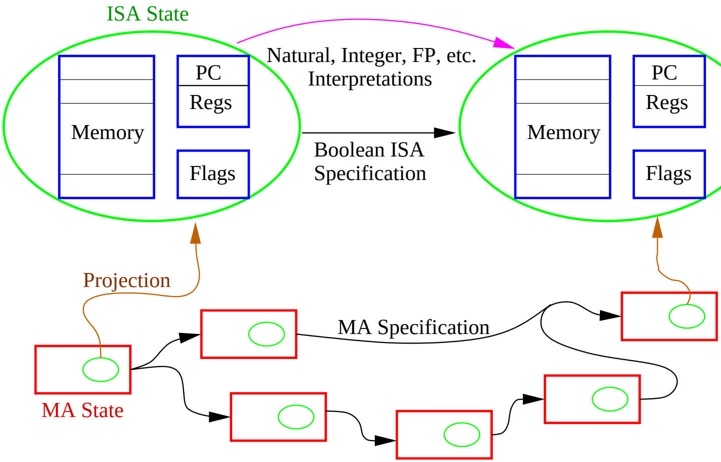
- It is the build-to specification for x86 implementations
- It is the compile-to specification for x86 programs
- It provides the contract between the software and hardware
- It documents the fundamental capabilities and limitations
- It provides the semantics for Linux, Windows, Excel, Emacs, ...

Abridged History

- In ∼2004, Hunt modeled the y86 ISA used in Bryant and O'Hallaron's architecture textbook
- Around 2009, Hunt created a simple x86-ISA model
- In 2012, Hunt and Kaufmann documented a more complete ACL2 x86-ISA model (UTCS Technical Report)
- In ∼2015, Goel's PhD work included adding x86-ISA instructions, supervisor mode, and memory management
- In ∼2017, Cuong Chau added floating-point support (SSE 1 and SSE 2 instructions)
- Later, Alessandro Coglio [Kestrel] and Goel added support for 32-bit instructions
- In 2023, Sohail added a timer, interrupts, console I/O, etc. so Linux could be booted, and run user programs

Serious Questions

- Have you ever written a C-language program?
- Have you ever compiled a C-language program?
- Have you ever run a C-language program?



x86-ISA Machine State Model

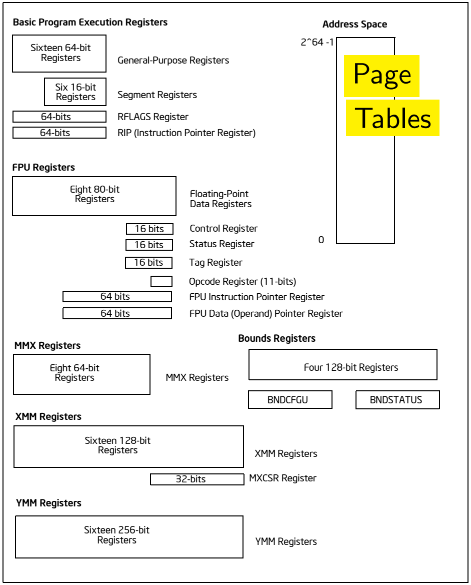


Diagram (copied from Intel's x86 Software Programmer's manual) summarizes the x86-ISA state.

- Memory contains most state
- Page tables memory resident

Each memory reference requires address translation

- Local/Global Descriptors
- Segment-register mapping
- 4-level address translation

Booting Linux

To boot Linux on top of our x86-ISA model:

- We extended our x86-ISA model with exceptions and interrupts
- Linux requires a timer and interrupt controller – which is not strictly part of the x86 ISA.
- We added a Linux console device

Contemporary BIOS/UEFIs include 16-bit real-mode code, which is how an x86 processor is configured when powered on.

We wrote a *bootloader* function in ACL2 that initializes our x86-ISA model state so Linux can boot – skipping the typical BIOS/UEFI and bootloader code.[1]

[1] Boot Linux.

Interrupts/Exceptions; Timer and Interrupt Controller

Interrupt Handling

- Prior to 2023, exceptions stopped our x86-ISA emulator
- Now, after each instruction
  - We check for a pending interrupt/exception
  - If observed, control is passed to the exception handler

Software Timer

- We removed Linux dependence on the Programmable Interval Timer (PIT) and Programmable Interrupt Controller (PIC)
- Our *timer* model posts an interrupt after some number of instructions have executed
- We wrote our own Linux device driver for interrupts

TTY (Console) I/O; Loading Linux into *RAM*

When emulating our x86-ISA model, we use our ACL2 TTY model:

- to connect a terminal to/from a TCP socket
- this allows us to provide *console I/O* for our x86-ISA model

We load a Linux OS binary into our x86-ISA *RAM*:

- we initialize the x86-ISA emulated registers,
- we write the initial **ramdisk** to *physical* memory, and
- we initialize the **zero-page** — a structure used to pass information from the bootloader to Linux.

Debugging our x86-ISA Model

- In principle, we've discussed everything necessary to boot Linux
- But, our x86-ISA model had several buggy instructions
- Tracking down flawed instructions was hard because the (often subtle) adverse effects may not manifest themselves until much later
- It's a bit like tracking down memory-pointer corruption bugs in C/C++ code, but harder...

Bug-Hunting Solution: Co-simulation

With a known good processor, we start the model and the processor in the same state and step them each one instruction at a time, comparing states, until you find where they differ.

- However, this approach is complicated by the fact that the x86 ISA doesn't fully specify all allowed behaviors
- That is, there can be differences in behavior between two different ISA-conforming implementations
- But our approach of comparing our model to what a real x86 processor does is how we *debug* our x86-ISA model.[2]

[2] Let's see how the boot is proceeding...

Our Co-Simulation Environment

- For our assumed-good x86 implementation, we use Linux's KVM APIs to create a hardware accelerated virtual machine
- This approach uses the x86 virtualization extensions, so most instructions are executed directly by a host, x86 processor
- We wrote Lisp code to dump the state of our x86-ISA model
- We step the VM
- We step our x86 model (the number of steps the KVM model advanced), and compare our x86 model state with what the KVM model produced.
- If they differ, we may have found a bug

Larger Traces

- Suppose we run the model for n instructions and then observe the effects of a bug (for example, a kernel panic)
- We can start our x86 model again and execute $\frac{n}{2}$ instructions, and then dump our model's state
- We can then load the that state into a VM and let the VM run
- If the VM also exhibits the buggy behavior, we conclude the bug was in the first $\frac{n}{2}$ instructions that were run on the model
- Otherwise, the bug will be found in the latter $\frac{n}{2}$ instructions
- Thus, we perform binary search to find the buggy instruction
- Once isolated, we fix the bug, and start all over again...

TLB – Improving Emulation Performance

After booting Linux successfully, we wanted more performance, so we implemented a TLB (translation look-aside buffer) cache.

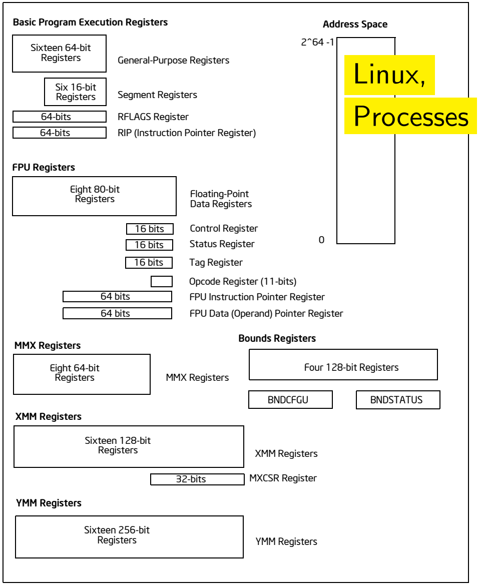
- Originally, we walked the page tables for every translation (and an *n*-byte access requires *n* translations)
- Processors cache translation information in a TLB
- We implemented *a* TLB conforming to the x86 documentation

Our TLB model is an ACL2 hashtable mapping:

- control register bits that effect address translation
- the access type (i.e. read, write, or execute), and
- the virtual page number

to a physical page number.

Linux, Programs and their Data



We have verified binary for:

- a *wc* (word count program)
- a *zero*-copy subroutine

The size of these proofs is substantial

- proof statements include the x86-ISA model,
- the specialization of the memory content, and
- we must process megabyte-sized proof terms.

Characteristics of our ACL2 x86-ISA Model

Model pieces:

- $\sim$ 165000 lines of code
- $\sim$ 48000 lines of documentation
- $\sim$ 3000 function definitions, $\sim$ 1100 lemmas
- Time to certify our x86-ISA model (on Intel i7-10700k laptop)
  - $\sim$ 1.8 hours to certify model with all its dependencies
  - $\sim$ 1.2 hours to certify model excluding dependencies
  - Both times include lemmas to reason about the behavior of the model and some verified x86-binary programs[3]

x86-ISA emulation performance

- Boots Linux in $\sim$10 minutes
- Emulates $\sim$1 million instructions per second

[3] Let's try it out!

Availability of the ACL2 x86-ISA Model

The `x86ISA` model is available as part of the ACL2 community books.

The community books can be obtained along with the ACL2 source from `https://github.com/acl2/acl2`. The model can be found in the `books/projects/x86isa` directory of the ACL2 source tree.

The model is also documented in ACL2's documentation system. You can find the documentation at `https://www.cs.utexas.edu/~moore/acl2/manuals/latest/?topic=ACL2____X86ISA`.

Future Work

Having a x86-ISA semantics enable:

- Boot other operating systems (e.g., FreeBSD, Windows)
- Verify operating-system routines (e.g., TCP/IP stack)
- Verify the correctness of user programs
- Model peripherals to support unmodified operating systems

Many programs *JIT* x86-ISA code; these binaries can be validated

The early, 64-bit x86 patents have expired. Maybe some group can produce a simple, verified x86 implementation.

Conclusion

The value of our x86-ISA specification includes:

- It is the build-to specification for x86 implementations
- It is the compile-to specification for x86 programs
- It provides the contract between the software and hardware
- It documents the fundamental capabilities and limitations
- It provides the semantics for Linux, Windows, Excel, Emacs, ...

x86-ISA instructions can be 15 bytes long – allowing (a max of) 1329227995784915872903807060280344576 instructions.

This *contract* is *executed* 10^9 times/second by 10^9 of processors.

Our social, financial, engineering, transportation, and government systems depend on this contract; we need mathematical precision to confirm the veracity of our hardware implementations and the correct operation of our x86-ISA binary programs.