The x86isa Books: Features, Usage, and Future Plans

ACL2-2017

Shilpi Goel
shilpi@centtech.com
x86 Machine-Code Verification

Reasoning about x86 machine-code programs is hard...
...unfortunately, it can be necessary at times.

• Formal, executable model of the x86 instruction-set architecture
  - 64-bit mode
  - Uniprocessor
• Framework to reason about x86-64 machine-code programs
x86 Machine-Code Verification

Reasoning about x86 machine-code programs is hard...
...unfortunately, it can be necessary at times.

- Formal, executable model of the x86 instruction-set architecture
  - 64-bit mode
  - Uniprocessor
- Framework to reason about x86-64 machine-code programs

This talk is about x86isa’s:
- current capabilities
- implementation
- future directions
What Can I Do with x86isa?

Use as an x86 instruction-set simulator for concrete program runs

- *Monitor program runs à la GNU Debugger (GDB)*

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

*Sources: GDB and Pin Websites*
What Can I Do with x86isa?

Use as an x86 instruction-set simulator for concrete program runs

- Monitor program runs à la GNU Debugger (GDB)

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

- Dynamically instrument programs à la Intel’s Pin

As a dynamic binary instrumentation tool, instrumentation is performed at run time on the compiled binary files. Thus, it requires no recompiling of source code and can support instrumenting programs that dynamically generate code.

Sources: GDB and Pin Websites
What Can I Do with x86 isa?

Use as a framework to reason about x86 programs

- Both user- and system- mode programs
  - System calls
  - Memory management
    - Paging
    - Segmentation
What Can I Do with x86isa?

Use as a framework to reason about x86 programs

- Both user- and system-mode programs
  - System calls
  - Memory management
    - Paging
    - Segmentation

- Kinds of formal analysis
  - Functional correctness
  - Detect dependence on undefined behavior
  - Determine bounds on resource consumption
  - Security properties
  - ...

Diagram:
- User-mode programs
- Standard libraries
- Application programs
- System call procedures
- Operating system
- System-mode programs
- x86 CPU
Specification?

~3400 pages

Intel® 64 and IA-32 Architectures
Software Developer’s Manual

NOTE: This document contains all seven
Volumes when evaluating your design needs.

~3000 pages

Intel® 64 and IA-32 Architectures
Software Developer’s Manual

Combined Volumes:
1, 2A, 2B, 2C, 3A, 3B and 3C

AMD64 Technology

AMD64 Architecture
Programmer’s Manual

Running tests on x86 machines

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int num, rotate_by, old_cf;
    int res, cf;

    // Set CF.
    int stc
    // Set EAX = 0.
    int mov $0, %%eax\n    // Set EBX = 0.
    int mov $0, %%ebx\n    // Set ECX = 0.
    int mov $0, %%ecx\n    // Set CL = rotate_by.
    int mov %4, %%ecx\n    // Set EDX = old_cf = 1.
    int mov %3, %%edx\n    // Set EAX = num.
    int mov %2, %%eax\n    // Rotate AL by CL.
    int rcl %%cl, %%al\n    // Set EBX = old_cf if CF = 1.
    int cmovb %%%edx, %%ebx\n    // Otherwise, EBX = 0.
    int mov %%%eax, %0\n    // Set res = EAX.
    int mov %%%ebx, %1\n    // Set cf = EBX.

    printf("=g\"",&(res), ":=g\"",&(cf)
    printf("=g\"",&(num), ":=g\"",&(old_cf), ":=g\"",&(rotate_by)
    printf("=g\"",&(rax), ":=g\"",&(rbx), ":=g\"",&(rcx), ":=g\"",&(rdx));

    return 0;
}
```

```c
int main(int argc, char *argv[])
{
    int num, rotate_by, old_cf;
    int res, cf;

    // Set CF.
    int stc
    // Set EAX = 0.
    int mov $0, %%eax\n    // Set EBX = 0.
    int mov $0, %%ebx\n    // Set ECX = 0.
    int mov $0, %%ecx\n    // Set CL = rotate_by.
    int mov %4, %%ecx\n    // Set EDX = old_cf = 1.
    int mov %3, %%edx\n    // Set EAX = num.
    int mov %2, %%eax\n    // Rotate AL by CL.
    int rcl %%cl, %%al\n    // Set EBX = old_cf if CF = 1.
    int cmovb %%%edx, %%ebx\n    // Otherwise, EBX = 0.
    int mov %%%eax, %0\n    // Set res = EAX.
    int mov %%%ebx, %1\n    // Set cf = EBX.

    printf("=g\"",&(res), ":=g\"",&(cf)
    printf("=g\"",&(num), ":=g\"",&(old_cf), ":=g\"",&(rotate_by)
    printf("=g\"",&(rax), ":=g\"",&(rbx), ":=g\"",&(rcx), ":=g\"",&(rdx));

    return 0;
}
```
Co-Simulations for Model Validation

Co-simulations

State-by-State

Diff

ACL2 printing
functions

GDB scripts, Pin

x86isa
in ACL2

x86

GCC/LLVM

C

Program Opcodes

Implemented?

No

Implement missing opcodes

Yes

Binary Program Loader in ACL2
x86isa: Design Goals

**Accuracy**
Reliable program analysis

**Execution Efficiency**
Aid in co-simulations
Up to 3.3 mil. ins/sec

**Usability**
Balance verification effort and verification utility

**Reasoning Efficiency**
Reduce user effort
x86isa: Design Goals

**Accuracy**
Reliable program analysis

**Execution Efficiency**
Aid in co-simulations
Up to 3.3 mil. ins/sec

**Usability**
Balance verification effort and verification utility

**Reasoning Efficiency**
Reduce user effort
x86isa: Design Goals

**Accuracy**
Reliable program analysis

**Execution Efficiency**
Aid in co-simulations
Up to 3.3 mil. ins/sec

**Usability**
Balance verification effort and verification utility

**Reasoning Efficiency**
Reduce user effort

abstract stobjs, guards, type declarations, mbe...
x86isa: Design Goals

**Accuracy**
Reliable program analysis

**Execution Efficiency**
Aid in co-simulations
Up to 3.3 mil. ins/sec

**Usability**
Balance verification effort and verification utility

**Reasoning Efficiency**
Reduce user effort

abstract stobjs, guards, type declarations, mbe...
x86isa: Design Goals

**Accuracy**
Reliable program analysis

**Execution Efficiency**
Aid in co-simulations
Up to 3.3 mil. ins/sec

**Usability**
Balance verification effort and verification utility

**Reasoning Efficiency**
Reduce user effort

modes of operation

abstract stobjs, guards, type declarations, mbe...
Development Style

Interpreter-style of Operational Semantics

- *model-specific fields*
  - *ms*
  - *u/s*

- *regs*

- *byte-addressable mem*

- *flags*

---

- **x86 State**
  - *x86 read*
  - *x86 write*

- **interface to the x86 state**
  - *rb*
  - *wb*

- **read bytes**
  - *rb*

- **write bytes**
  - *wb*

---

- **ADD**
- **SUB**
- **MUL**
- *...*
- **MOV**
- **PUSH**
- **POP**

- **(step x86)**
  - fetch, decode, & execute one instruction

- **(run n x86)**
2.1.1 Global and Local Descriptor Tables

When operating in protected mode, all memory accesses pass through either the global descriptor table (GDT) or an optional local descriptor table (LDT) as shown in Figure 2-1. These tables contain entries called segment descriptors. Segment descriptors provide the base address of segments as well as access rights, type, and usage information.

Each segment descriptor has an associated segment selector. A segment selector provides the software that uses it with an index into the GDT or LDT (the offset of its associated segment descriptor), a global/local flag (determines whether the selector points to the GDT or the LDT), and access rights information.

**Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode**

**Figure 3-2. 64-Bit Mode Execution Environment**

**Source:** Intel Manuals
x86 State: Some Model-Specific Fields

- **MS: Model State**
  - If a model-related error occurs (e.g., an unimplemented opcode is encountered), this field is populated with an appropriate error message.
  - The model is expected to reflect the real machine’s state only if this field is empty.
x86 State: Some Model-Specific Fields

• **MS: Model State**
  - If a model-related error occurs (e.g., an unimplemented opcode is encountered), this field is populated with an appropriate error message.
  - The model is expected to reflect the real machine’s state only if this field is empty.

• **U/S: User/System**
  - Switches the mode of operation of the x86 model:
    ‣ *User-level mode*
    ‣ *System-level mode*
x86 State: Some Model-Specific Fields

• **MS: Model State**
  - If a model-related error occurs (e.g., an unimplemented opcode is encountered), this field is populated with an appropriate error message.
  - The model is expected to reflect the real machine’s state only if this field is empty.

• **U/S: User/System**
  - Switches the mode of operation of the x86 model:
    ‣ *User-level mode*
    ‣ *System-level mode*

• **ENV: Environment**
  - Specifies an external environment
  - Also includes an *oracle* that is instrumental in modeling non-deterministic, undefined, and random behaviors
Rationale for Different Modes of Operation

• To prove an x86 program correct, one would need to prove the correctness of the supporting operating system code as well.

• E.g., statically compiling a Hello World C program generates an executable file of size ~0.8MB!
  - `printf` is a standard C library function that ultimately relies on the `write` system call provided by the OS.

```c
#include <stdio.h>
int main() {
  printf("Hello, world!\n");
  return 0;
}
```

• A user may wish to assume that these underlying OS services are correct.
## Modes of Operation

<table>
<thead>
<tr>
<th>User-level Mode</th>
<th>System-level Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verification of application programs</td>
<td>Verification of system programs</td>
</tr>
<tr>
<td>Assumptions about correctness of certain OS operations (e.g., system calls)</td>
<td>No such assumptions</td>
</tr>
<tr>
<td>Linear memory address space*</td>
<td>Physical memory address space (includes specification of paging)</td>
</tr>
</tbody>
</table>

* Linear memory ($2^{64}$ bytes) is an OS-provided abstraction of the physical memory. 64-bit programs cannot access physical memory directly.
Interface to the x86 State

• \texttt{xr} and \texttt{xw}:
  - Accessor and updater for all x86 state components, except memory

• \texttt{rb} and \texttt{wb}:
  - Accessor and updater for linear memory
    - User-level mode:
      - Memory field specifies the linear memory.
      - These functions directly access this field.
    - System-level mode:
      - Memory field specifies the physical memory.
      - These functions first convert linear addresses to physical addresses (\textit{paging}), and then use them to access the memory field.
Instruction Semantic Functions

- 413 x86 instruction opcodes are specified [::doc implemented-opcodes]
- Some instructions like SYSCALL and SYSRET are implemented differently depending on the mode of operation.

```
... MOV %rax, 0
SYSCALL
MOV %rbx, %rax
... save user state
restore user state
```

User-level Mode

System-level Mode
x86 Machine-Code Proofs

- *Symbolic simulation* is central to program verification.
  - Control the *unwinding* of the x86 interpreter during code proofs.
  - For all those times proofs fail, see [:doc debugging-code-proofs].
  - And also, **Codewalker** works in the user-level mode.
x86 Machine-Code Proofs

- **Symbolic simulation** is central to program verification.
  - Control the *unwinding* of the x86 interpreter during code proofs.
  - For all those times proofs fail, see [:doc debugging-code-proofs].
  - And also, **Codewalker** works in the user-level mode.

- Examples of code proofs included in x86isa:
  - *Straight-line, computationally-intensive application program*
    - bit-twiddling popcount
  - *Application program with loops and system calls*
    - word-count
  - *System program that modifies the linear memory abstraction*
    - zero-copy (copy-on-write technique)
Possible Future Directions: x86 ISA Modeling

- **Exceptions and Interrupts**
  - *Already implemented:*
    - relevant system registers
    - memory-resident data structures (descriptor tables)
    - detection of exception-causing conditions (e.g., #DE)
      - halt the program execution upon encountering these conditions
  - *TODO:*
    - Detection of interrupts
      - Consult an oracle at every instruction boundary?
    - Use the descriptor tables to locate the appropriate exception- and interrupt-handling procedures in the memory
Possible Future Directions: x86 ISA Modeling

- **Caches and Multiprocessing** [long-term project]
  - Model caches, translation-look aside buffers, store buffers
  - Specify how memory reads & writes are resolved by multiple processors
  - Reason about cache coherence, etc.
Possible Future Directions: x86 ISA Modeling

- **Caches and Multiprocessing** [long-term project]
  - Model caches, translation-look aside buffers, store buffers
  - Specify how memory reads & writes are resolved by multiple processors
  - Reason about cache coherence, etc.

- **Simulate a stripped-down version of a mainstream OS**
  - An OS is tightly intertwined with low-level x86 system features
    - Difficult to separate OS-specific behavior from x86 behavior...
  - Ideally, run co-simulations against a “bare” x86 processor
    - Difficult to work with such a machine...
  - Simulating an OS is a way to validate x86 isa’s system-level mode
    - Another would be to co-simulate against QEMU (for instance)
Possible Future Directions: Program Analysis

- **Codewalker + x86isa** [almost there...]
  - *Already implemented:* Codewalker can be used in the user-level mode
  - *TODO:* Support for analysis in the system-level mode
Possible Future Directions: Program Analysis

- **Codewalker + x86isa** [almost there...]
  - *Already implemented:* Codewalker can be used in the user-level mode
  - *TODO:* Support for analysis in the system-level mode

- **Automated Precondition Discovery** [long-term project]
  - Difficult to discover the preconditions under which the program behaves as expected
  - Suggest hypotheses that are candidates to be top-level preconditions
    - Observe why some rules fail to fire when expected
    - Obtain conditions that would make those rules applicable
    - Generalize these conditions
    - Avoid suggestions that lead to contradictory or unsatisfiable preconditions
Other Possible Applications

**Firmware Verification**
formally specify software/hardware interfaces

**Micro-architecture Verification**
x86 ISA model can serve as a build-to specification
Thanks!

[Documentation]

x86isa in the ACL2+Community Books Manual

Thanks!
x86isa

Formal Specification

- A formal, executable x86 ISA model (64-bit mode)

Instruction-Set Simulator

- Executable file readers and loaders (ELF/Mach-O)
- A GDB-like mode for dynamic instrumentation of machine code
- Examples of program execution and debugging

Code Proof Libraries

- Helper libraries to reason about x86 machine code
- Proofs of various properties of some machine-code programs

Manual

- Documentation