

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.

Branch: master ▼

[acl2](#) / [books](#) / [projects](#) / [x86isa](#) /

- 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.

Branch: master ▼

[acl2](#) / [books](#) / [projects](#) / [x86isa](#) /

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

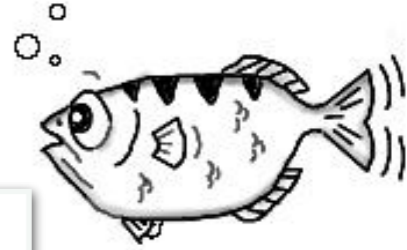
This talk is about `x86isa`'s:

- 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)*



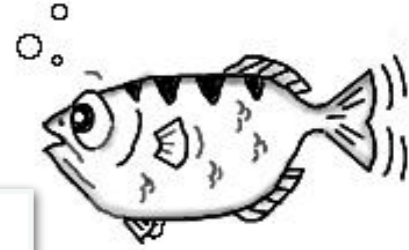
GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- 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.

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*

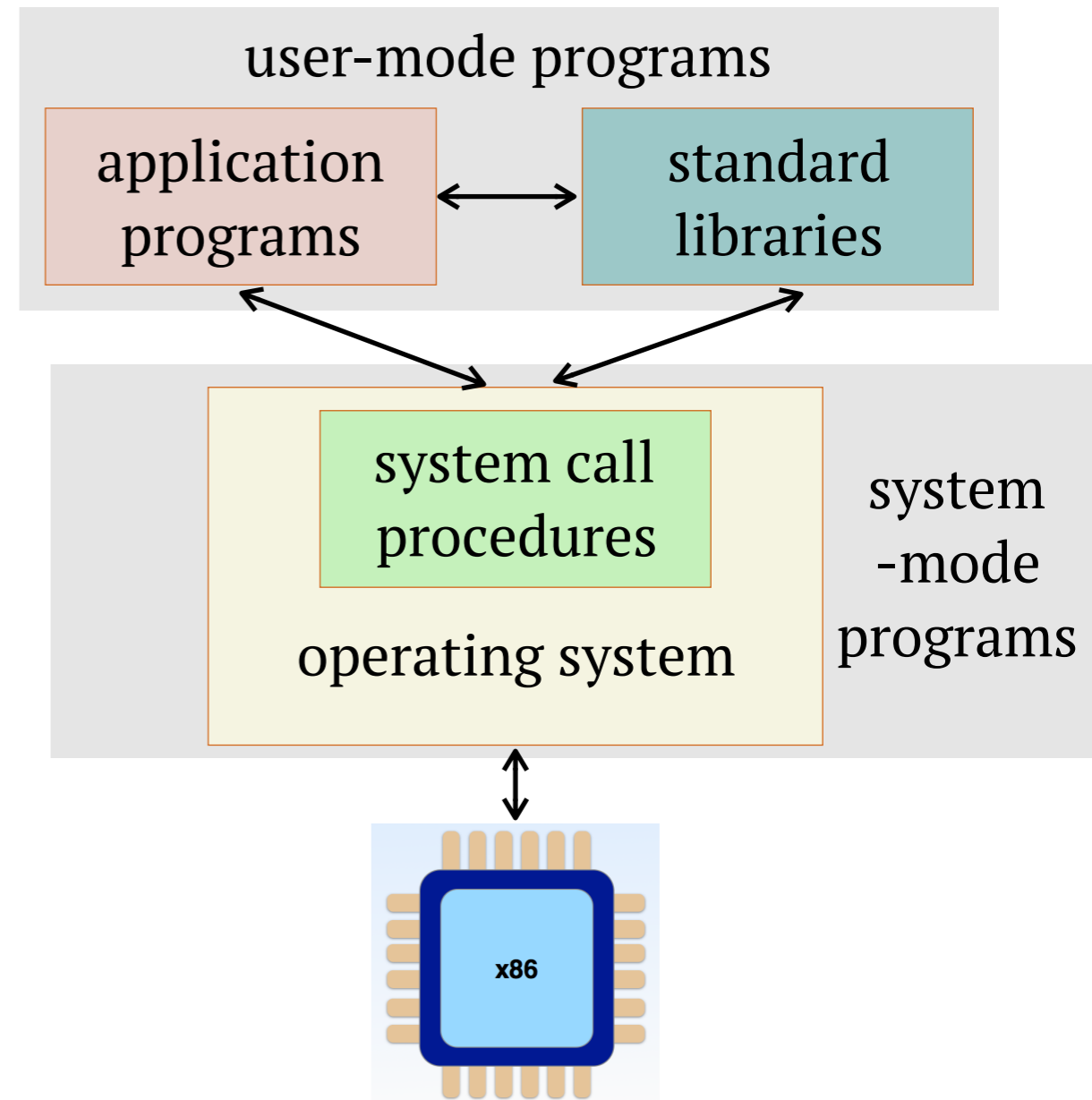


As a dynamic binary instrumentation tool, instrumentation is performed at run time on the compiled binary files. Thus, it requires no recompiling of source code and can support instrumenting programs that dynamically generate code.

What Can I Do with x86isa?

Use as a framework to reason about x86 programs

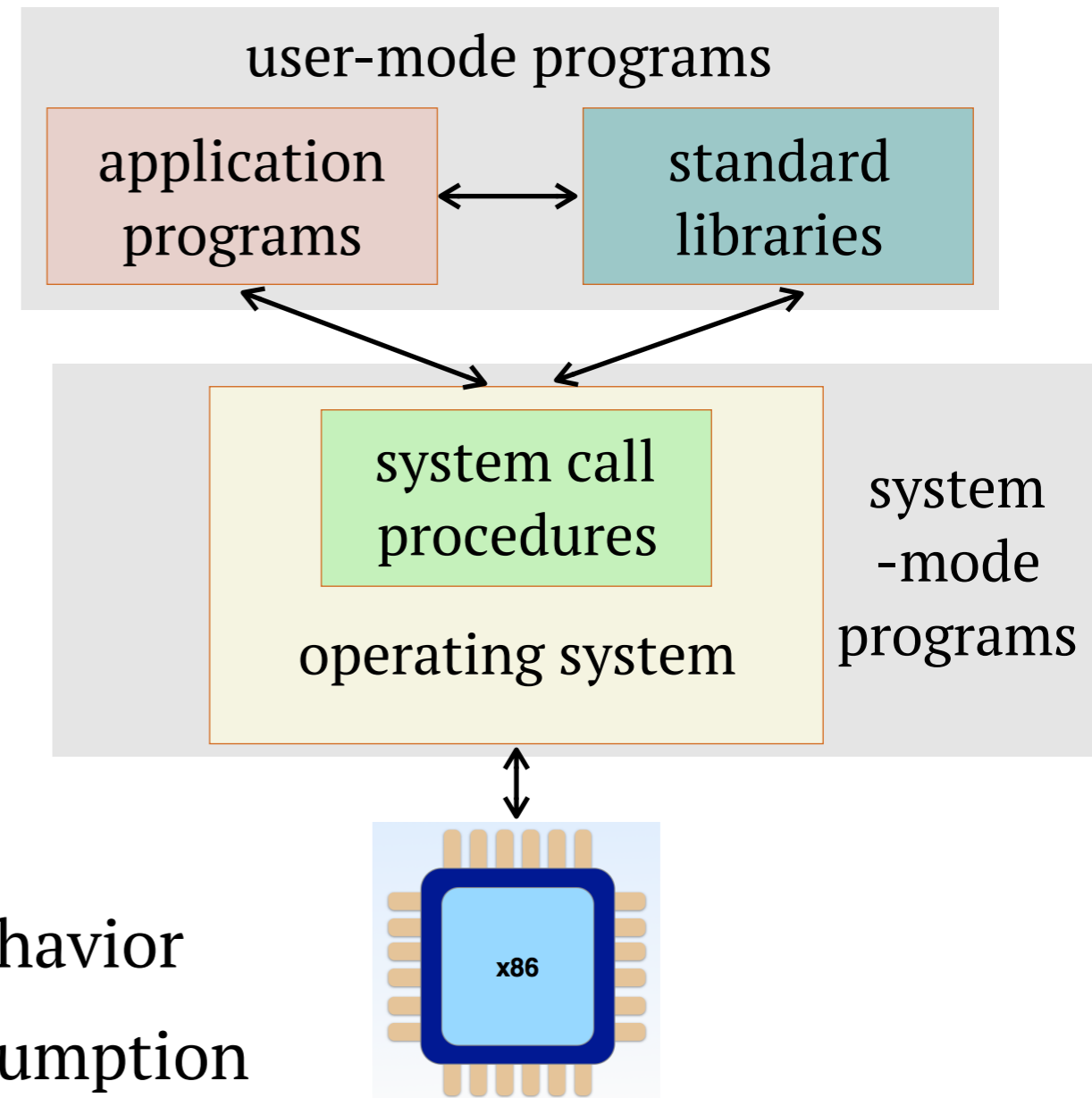
- *Both user- and system- mode programs*
 - ▶ System calls
 - ▶ Memory management
 - Paging
 - Segmentation



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
 - ▶ ...



Specification?

~3400 pages



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

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

NOTE: This document contains all seven volumes of the Intel® 64 and IA-32 Architectures Software Developer's Manual: *Basic Architecture, Instruction Set Reference, and the System Programming Guide*. It is recommended to read all seven volumes when evaluating your design needs.

~3000 pages



AMD64 Technology

AMD64 Architecture Programmer's Manual

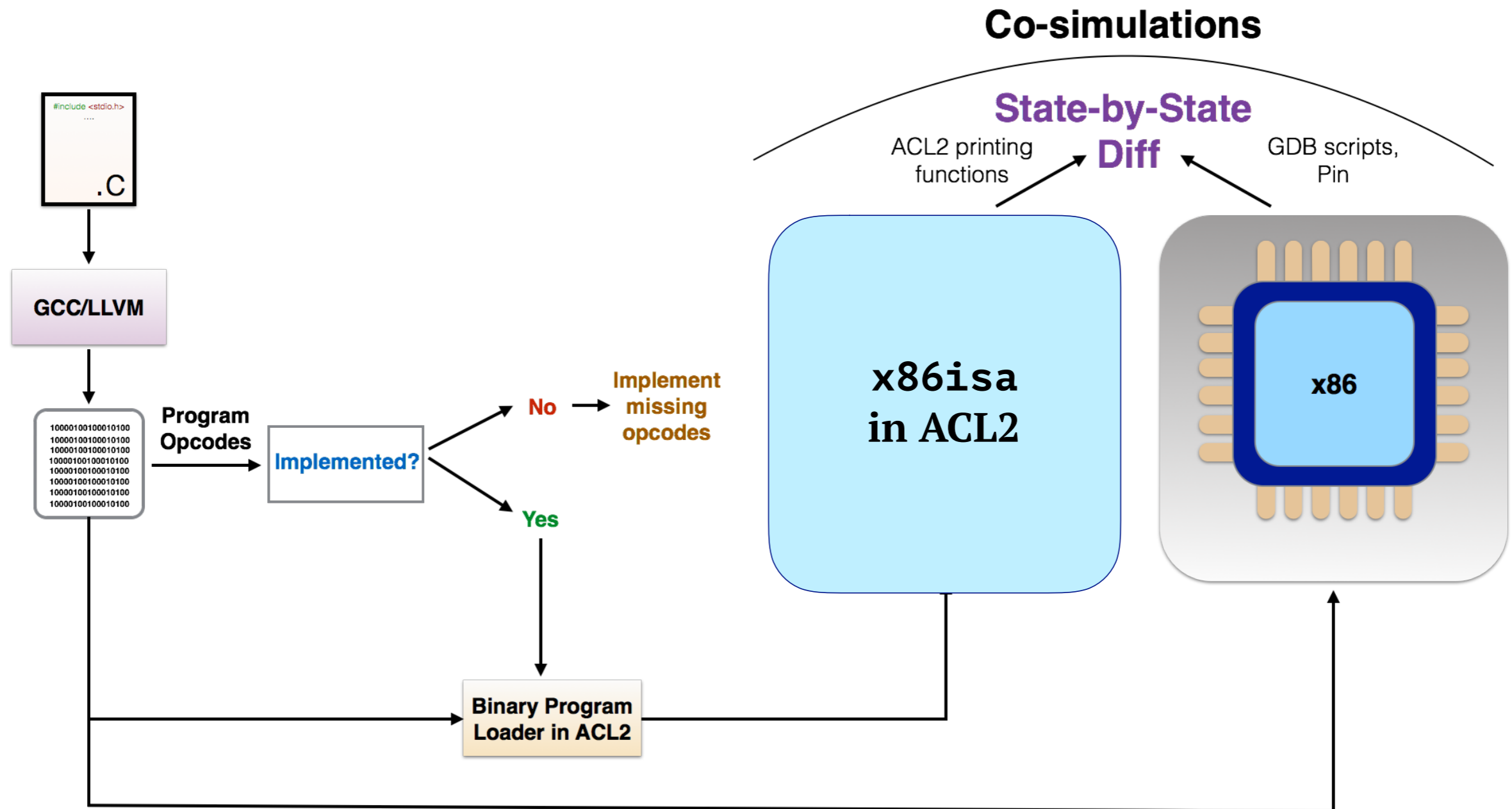
```
__asm__ volatile
("stc\n\t" // Set CF.
"mov $0, %%eax\n\t" // Set EAX = 0.
"mov $0, %%ebx\n\t" // Set EBX = 0.
"mov $0, %%ecx\n\t" // Set ECX = 0.
"mov %4, %%ecx\n\t" // Set CL = rotate_by.
"mov %3, %%edx\n\t" // Set EDX = old_cf = 1.
"mov %2, %%eax\n\t" // Set EAX = num.
"rcl %%cl, %%al\n\t" //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

"mov %%eax, %0\n\t" // Set res = EAX.
"mov %%ebx, %1\n\t" // Set cf = EBX.

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

Running tests on x86 machines

Co-Simulations for Model Validation



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

 *abstract stobjs,
guards, type
declarations,
mbe...*

Usability

Balance verification
effort and verification
utility

Reasoning Efficiency

Reduce user effort

x86isa: Design Goals

Accuracy

Reliable program
analysis



Usability

Balance verification
effort and verification
utility

Execution Efficiency

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



*abstract stobjs,
guards, type
declarations,
mbe...*

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

*modes
of operation*



*abstract stobjs,
guards, type
declarations,
mbe...*

Usability

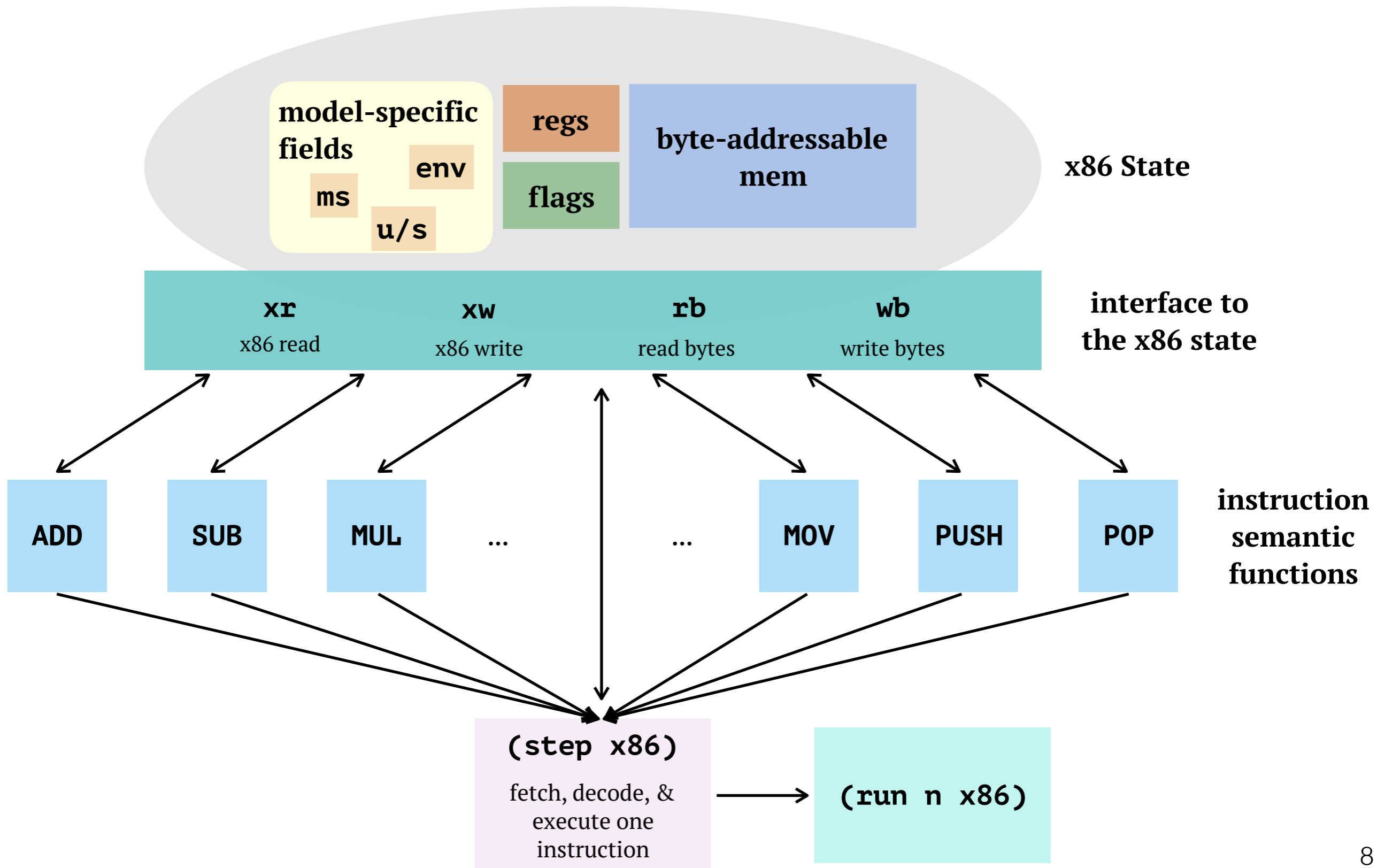
Balance verification
effort and verification
utility

Reasoning Efficiency

Reduce user effort

Development Style

Interpreter-style of Operational Semantics



x86 State: 64-bit Mode

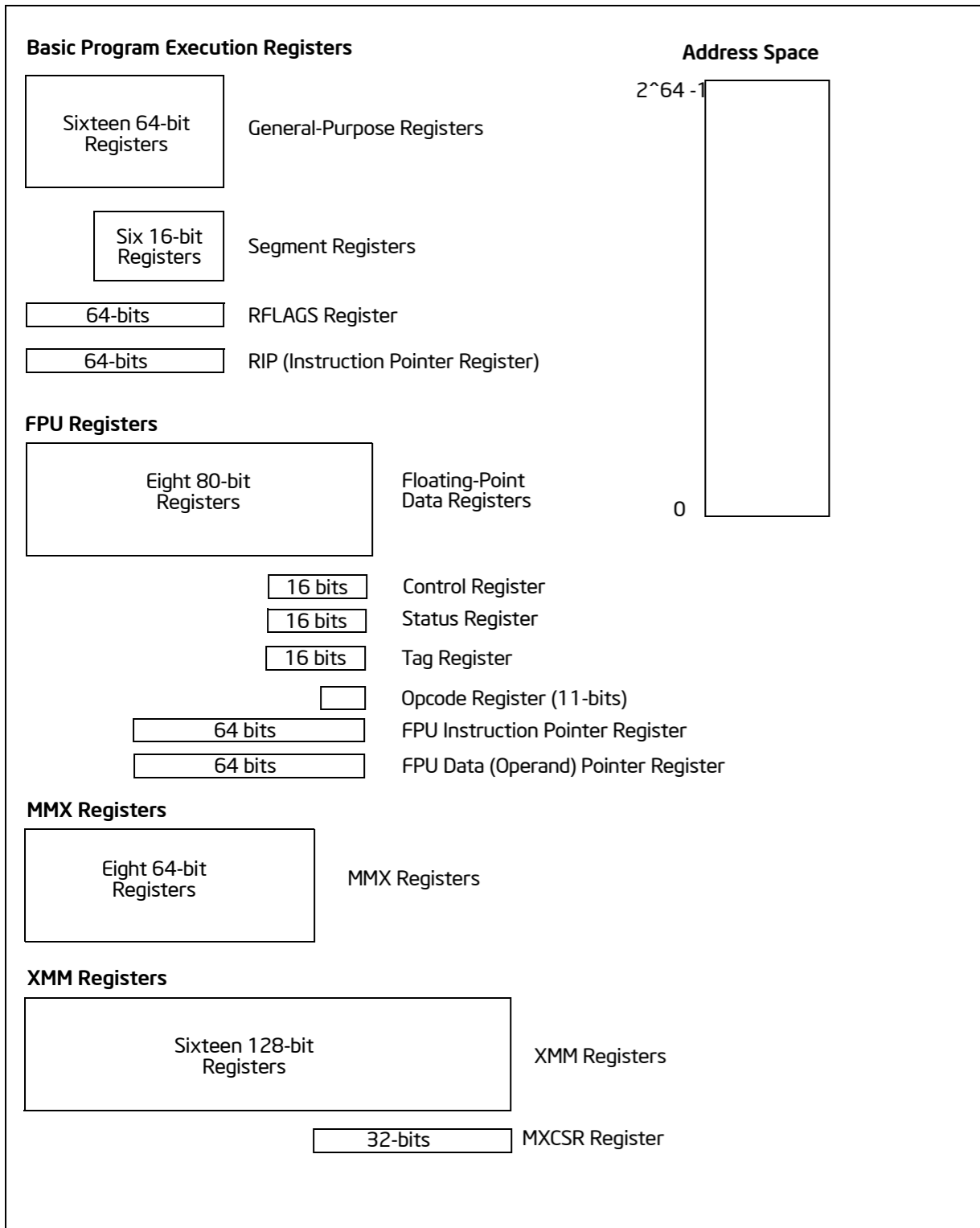


Figure 3-2. 64-Bit Mode Execution Environment

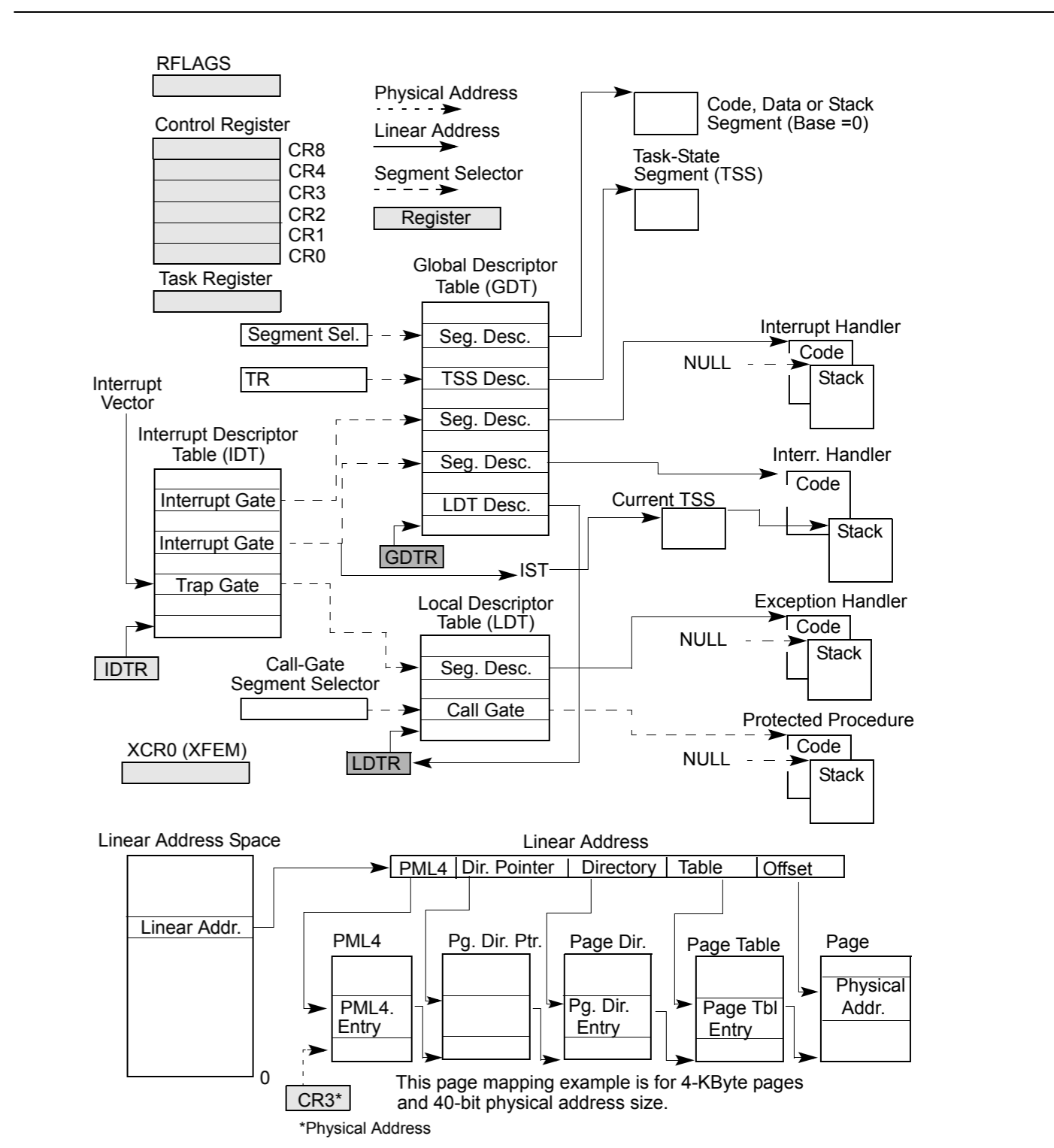


Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode

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.

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.

```
#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

User-level Mode

System-level Mode

Verification of application programs

Verification of system programs

Assumptions about correctness of
certain OS operations
(e.g., system calls)

No such assumptions

Linear memory address space*

Physical memory address space
(includes specification of paging)

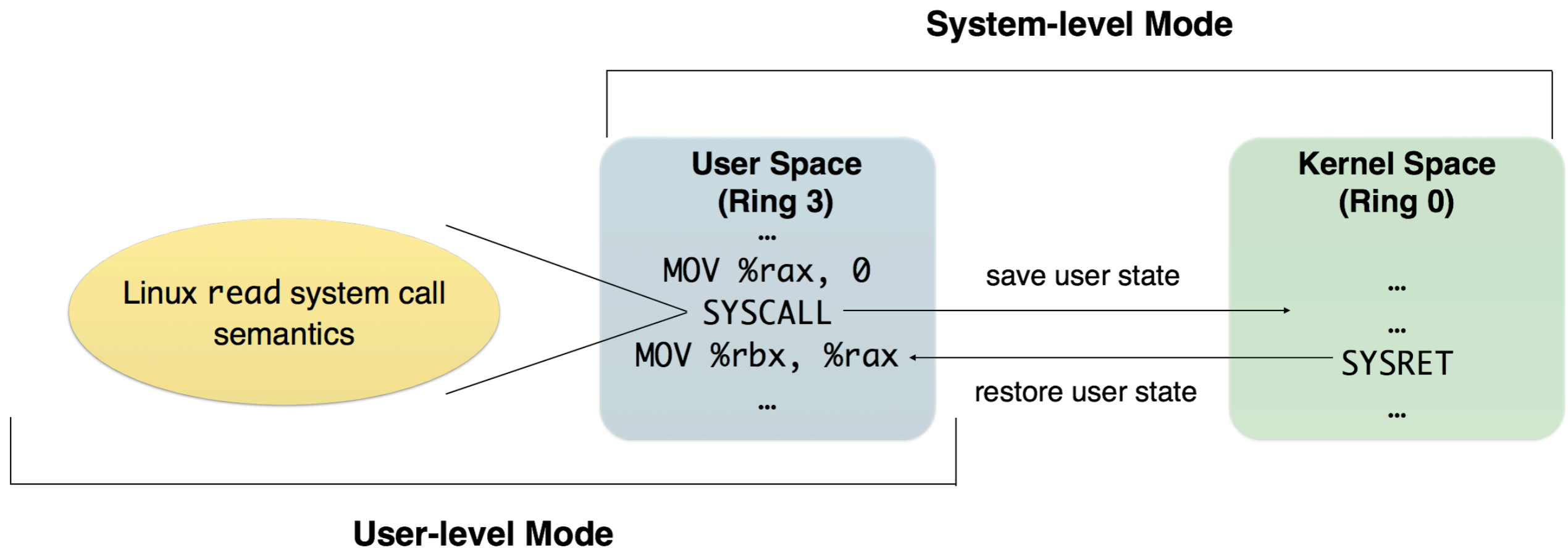
* Linear memory (2^{64} bytes) is an OS-provided abstraction of the physical memory. 64-bit programs cannot access physical memory directly.

Interface to the x86 State

- **xr and xw:**
 - Accessor and updater for all x86 state components, except memory
- **rb and 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 (*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.



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** [short-term project]
 - *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** [long-term project]
 - 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 x86isa’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

[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**