Analysis of x86 Application and System Programs via Machine-Code Verification

Shilpi Goel, Warren A. Hunt, Jr., and Matt Kaufmann

`<shigoel,hunt,kaufmann>@cs.utexas.edu`

Department of Computer Science
The University of Texas at Austin

April, 2016

Project Overview

Goal: Build robust tools to **increase software reliability**

- Verify critical properties of **application and system programs**
- Correctness with respect to **behavior**, **security**, & **resource usage**

Plan of Action:

1. Build a **formal, executable x86 ISA model** using ACL2
2. Develop a **machine-code analysis framework** based on this model
3. Employ this framework to **verify application and system programs**

Highlights of this Talk

Compile-to and Build-to Specification

A formal, executable x86 ISA model

- Specification of low-level ISA features
- Handles non-determinism

Unified Model

- *Simulator:* Executable file readers & loaders; GDB-like mode for dynamic instrumentation
- *Reasoning Framework*: ACL2 libraries to reason about x86 machine code

User Manual

- Documentation

Open Source

- Available online

Outline

Model Development

Interpreter-Style Operational Semantics

- ***x86 State:*** specifies the components of the ISA (registers, flags, memory)
- ***Instruction Semantic Functions:*** describe the effect of each instruction
- ***Step Function:*** fetches, decodes, and executes one instruction

64-bit sub-mode of Intel’s IA-32e mode

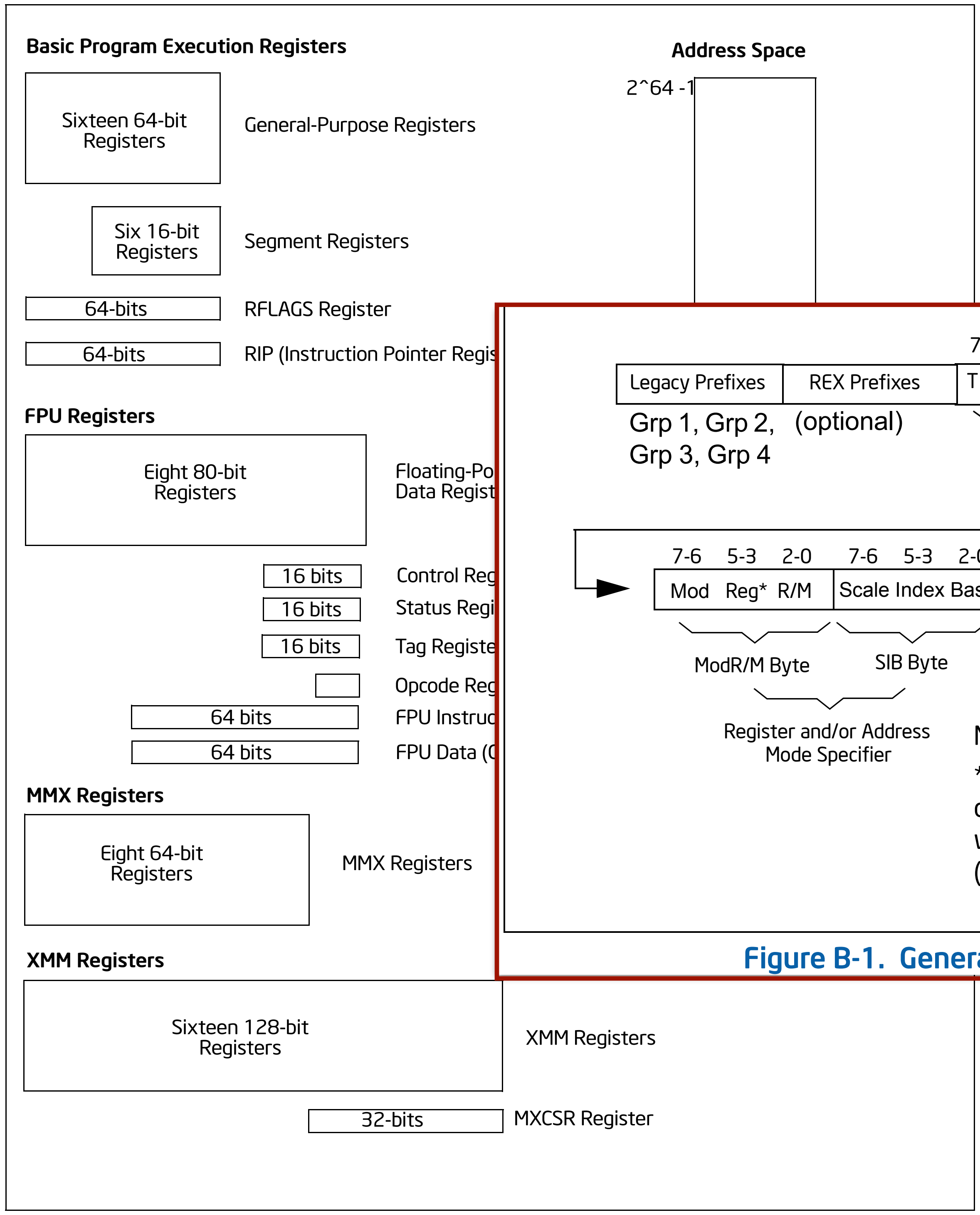


Figure 3-2. 64-Bit Mode Execution Environment

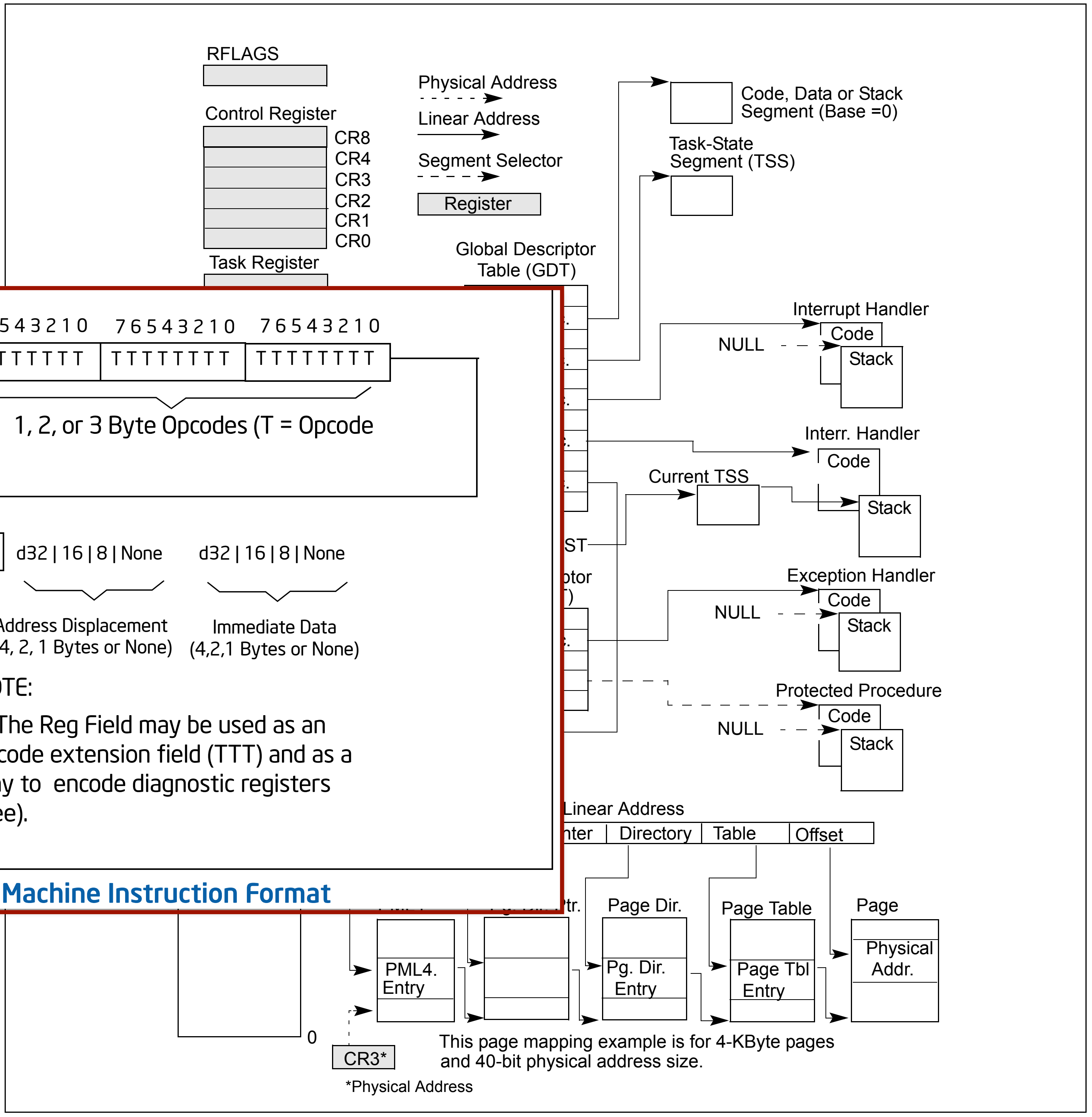
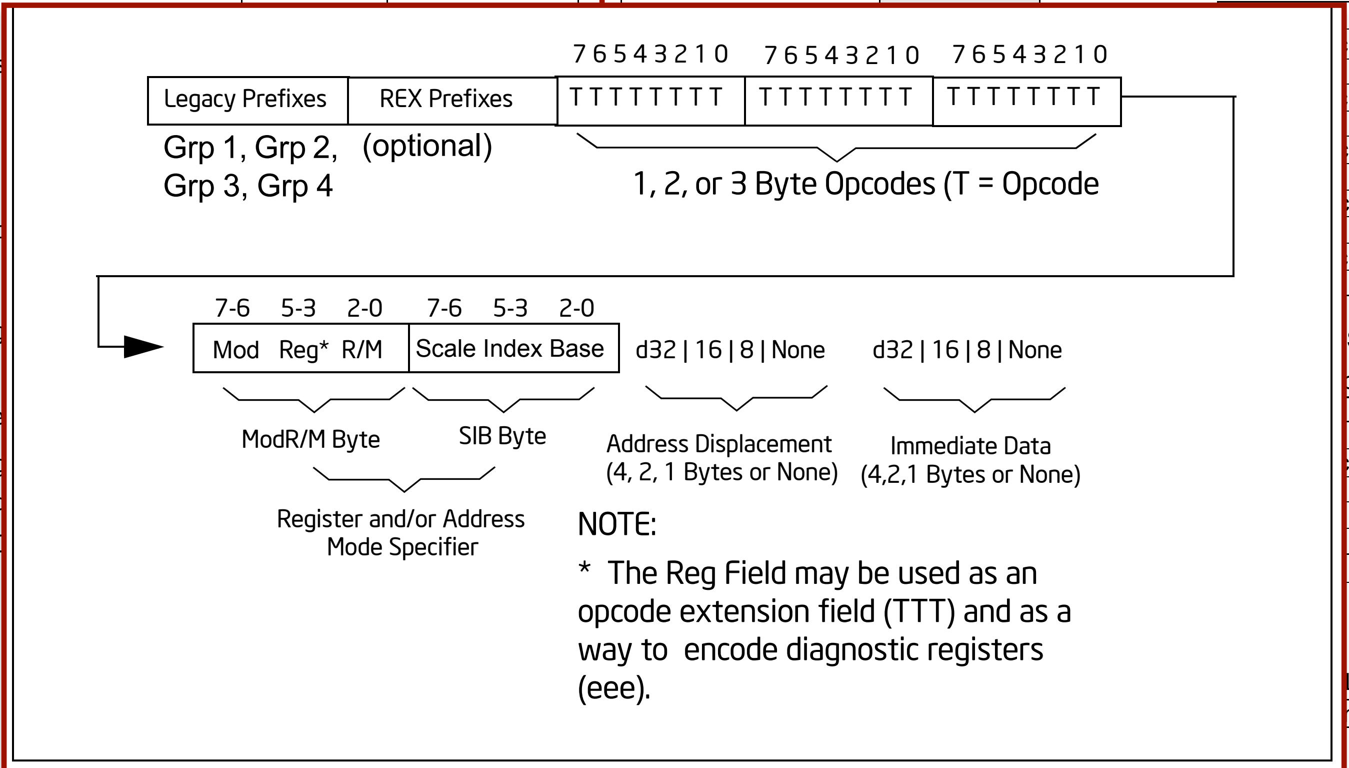


Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode



7 6 5 4 3 2 1 0
7 6 5 4 3 2 1 0
7 6 5 4 3 2 1 0
Legacy Prefixes
REX Prefixes
T T T T T T T T
T T T T T T T T
T T T T T T T T
Grp 1, Grp 2, Grp 3, Grp 4
(optional)
1, 2, or 3 Byte Opcodes (T = Opcode
7-6
5-3
2-0
7-6
5-3
2-0
Mod
Reg*
R/M
Scale Index Base
d32 | 16 | 8 | None
d32 | 16 | 8 | None
ModR/M Byte
SIB Byte
Address Displacement (4, 2, 1 Bytes or None)
Immediate Data (4,2,1 Bytes or None)
Register and/or Address Mode Specifier
NOTE:
* The Reg Field may be used as an opcode extension field (TTT) and as a way to encode diagnostic registers (eee).

Figure B-1. General Machine Instruction Format

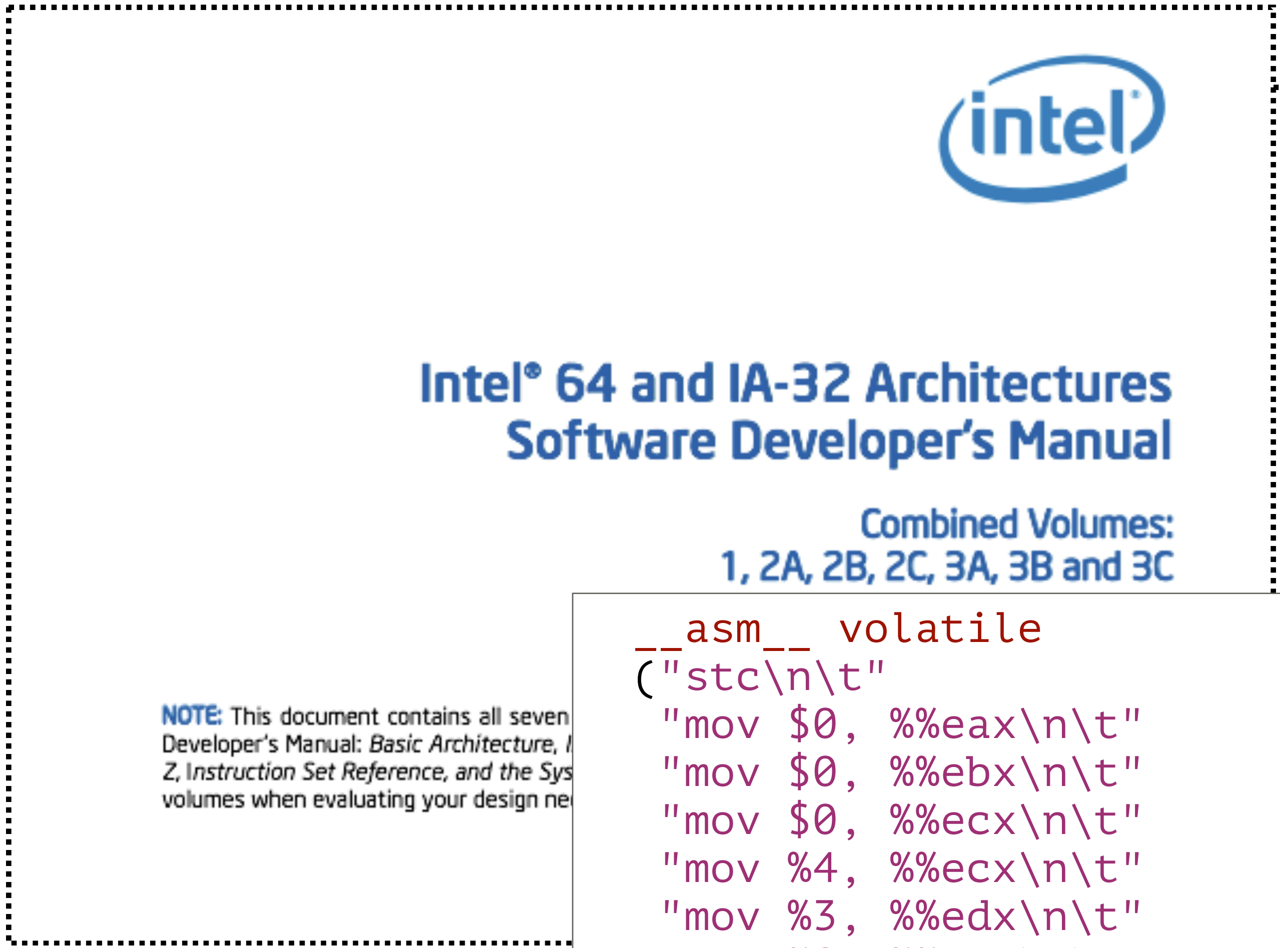
Source: Intel Manuals

Handling Non-Determinism

- Some examples of non-determinism in the ISA:
  - `RFLAGS` are undefined after the execution of some instructions.
  - Instructions like `RDRAND` are inherently non-deterministic.

- The x86 state contains an ***oracle*** field that is consulted whenever the result of a non-deterministic operation is required.
  - Every value retrieved from the oracle is ***unique*** and ***indeterminate***.
  - This allows accounting for all possible behaviors during reasoning.

Obtaining the x86 ISA Specification



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
Developer's Manual: *Basic Architecture, I*
Z, Instruction Set Reference, and the Sys
volumes when evaluating your design ne

AMD64 Technology

AMD64 Architecture
Programmer's Manual

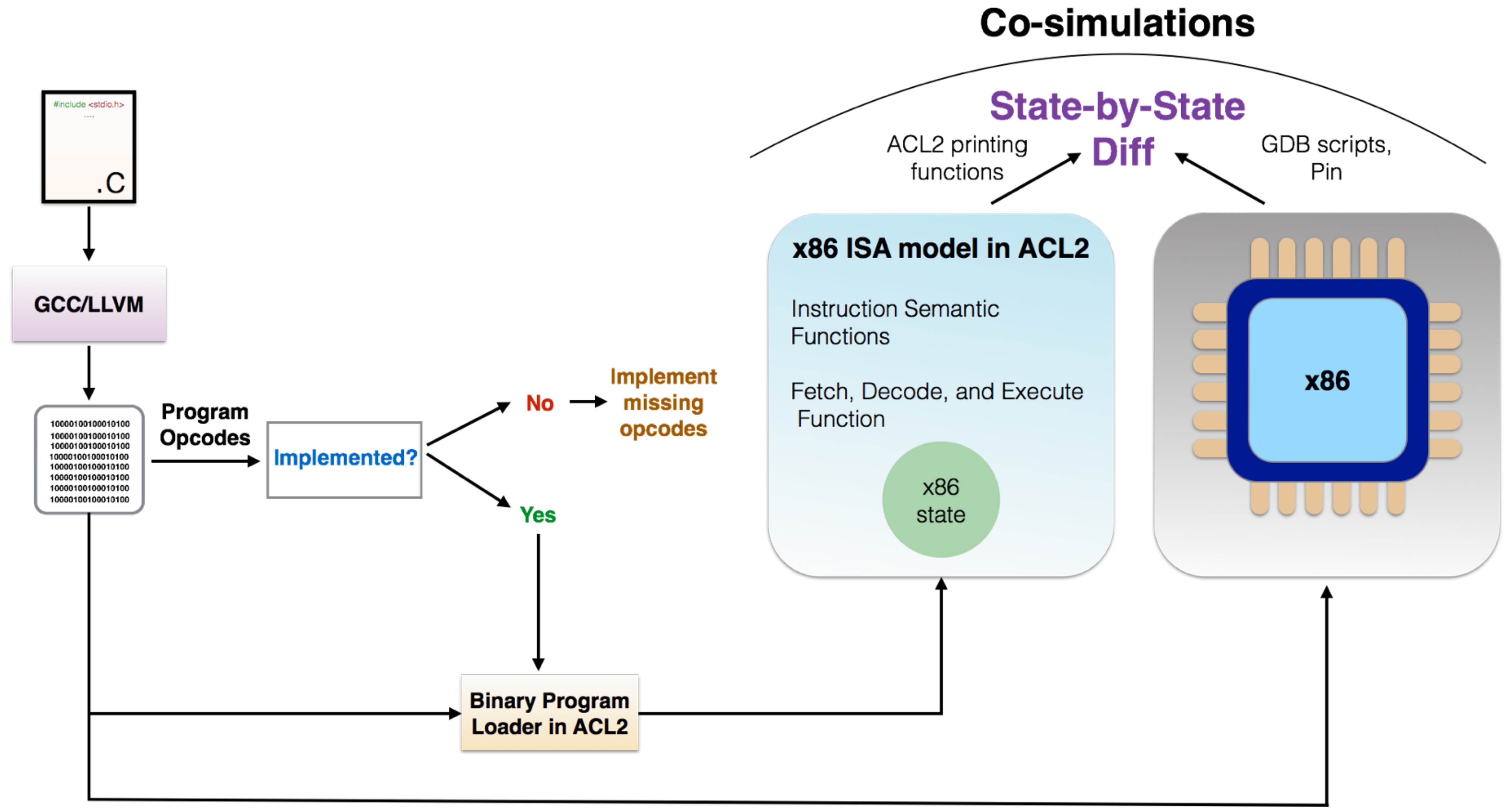
```
__asm__ volatile
("stc\n\t"                          // Set CF.
 "mov $0, %%eax\n\t"                // Set EAX = 0.
 "mov $0, %%ebx\n\t"                // Set EBX = 0.
 "mov $0, %%ecx\n\t"                // Set ECX = 0.
 "mov %4, %%ecx\n\t"                // Set CL = rotate_by.
 "mov %3, %%edx\n\t"                // 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

Model Validation

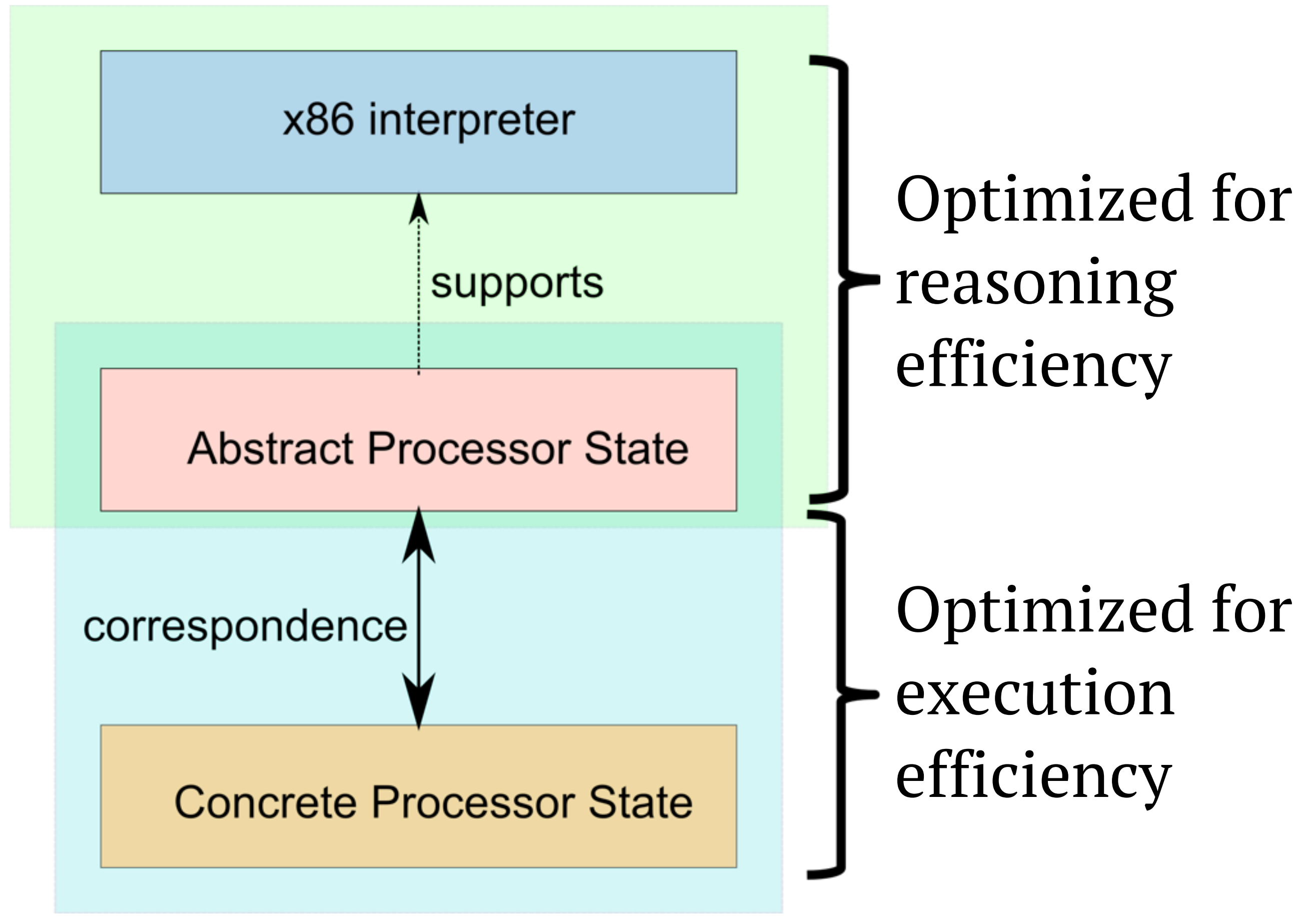
How can we know that our model faithfully represents the x86 ISA?

Validate the model to increase trust in the applicability of formal analysis.



Optimizing the Model for Efficiency

Layered modeling approach mitigates the trade-off between reasoning and execution efficiency. [ACL2'13]



~330K to 3.3 million instructions/second

This layer was introduced using an ACL2 feature called *Abstract Stobj*, which was developed in response to this need for optimizing the x86 model.

Simulation speed measured on an Intel Xeon E31280 CPU @ 3.50GHz

Focus on Usability

- Two examples that illustrate our focus on user experience:

1. **Modes of operation** to balance verification/simulation effort and utility
2. **Program debugging tools** to be used during simulation

Focus on Usability #1: Modes of Operation

Programmer-Level Mode	System-Level Mode
Verification of application programs	Verification of system programs
Virtual memory address space (2^{64} bytes)	Physical memory address space (2^{52} bytes)
Assumptions about correctness of OS operations	No assumptions about OS operations
~3.3 million instructions/second	**~330,000 instructions/second** (with 1G pages)

Simulation speed measured on an Intel Xeon E31280 CPU @ 3.50GHz

Focus on Usability #2: Tools for the Simulator

- **Executable file readers and loaders** written in ACL2 that support both Mach-O and ELF binary formats.
  - The input to the x86 model is the program binary
  - These tools use the meta-data in these binaries to automatically initialize the machine state
- A GDB-like mode is used for the **dynamic instrumentation** of machine-code.
  - Useful for debugging both the programs and the x86 specification

Current Status: x86 ISA Model

- The x86 ISA model supports **400+ instructions**
  - Can execute almost all user-level programs emitted by GCC/LLVM
  - Successfully co-simulated a contemporary SAT solver on our model
  - Successfully simulated a supervisor-mode zero-copy program
- **IA-32e paging** for all page configurations (4K, 2M, 1G)
- **Segment-based addressing**
- Lines of ACL2: ~85,000 (not including blank lines)

Outline

Current Status: Building Lemma Libraries

- General libraries include lemmas about **reads from** and **writes to the machine state**, along with the **interactions** between these operations.

- We include these libraries when we verify programs.

`add %edi, %eax`

1. **read** instruction from `mem`
2. **read** operands
3. **write** sum to `eax`
4. **write** new value to `flags`
5. **write** new value to pc

- General library construction and program verification are interdependent processes.
  - **Discover** the kinds of lemmas needed while verifying a program
  - See a general **pattern**
  - **Automate** the generation and proof of these lemmas

Outline

- Overview
- Project Description
  - [1] Developing an x86 ISA Model
  - [2] Building a Machine-Code Analysis Framework
  - **[3] Verifying Application and System Programs**
- Future Work & Conclusion

Application Program #1: *popcount*

Automatically verify snippets of straight-line machine code using **bit-blasting** [VSTTE'13]

```
int popcount_32 (unsigned int v)
{
  // From Sean Anderson's Bit-Twiddling Hacks
  v = v - ((v >> 1) & 0x55555555);
  v = (v & 0x33333333) + ((v >> 2) & 0x33333333);
  v = ((v + (v >> 4) & 0xF0F0F0F) * 0x1010101) >> 24;
  return(v);
}
```

```
8b 45 fc            mov     -0x4(%rbp),%eax
25 33 33 33 33      and     $0x33333333,%eax
8b 7d fc            mov     -0x4(%rbp),%edi
c1 ef 02            shr     $0x2,%edi
81 e7 33 33 33 33   and     $0x33333333,%edi
01 f8               add     %edi,%eax
89 45 fc            mov     %eax,-0x4(%rbp)
8b 45 fc            mov     -0x4(%rbp),%eax
8b 7d fc            mov     -0x4(%rbp),%edi
c1 ef 04            shr     $0x4,%edi
01 f8               add     %edi,%eax
25 0f 0f 0f 0f      and     $0xf0f0f0f,%eax
69 c0 01 01 01 01   imul    $0x1010101,%eax,%eax
c1 e8 18            shr     $0x18,%eax
89 45 fc            mov     %eax,-0x4(%rbp)
8b 45 fc            mov     -0x4(%rbp),%eax
5d                  pop     %rbp
c3                  retq
```

Functional Correctness:
`RAX = popcount(v)`

specification function

```
popcount(v):

if (v <= 0) then
    return 0
else
    lsb := v & 1
    v   := v >> 1
    return (lsb + popcount(v))
endif
```

Application Program #2: *word-count*

The word-count program reads input from the `stdin` using `read` system calls. System calls are ***non-deterministic*** for application programs. [FMCAD'14]

Functional Correctness: Values computed by specification functions on standard input are found in the expected memory locations of the final x86 state.

Resource Usage: Irrespective of the input, program uses a fixed amount of memory.

Security: Program does not modify unintended regions of memory.

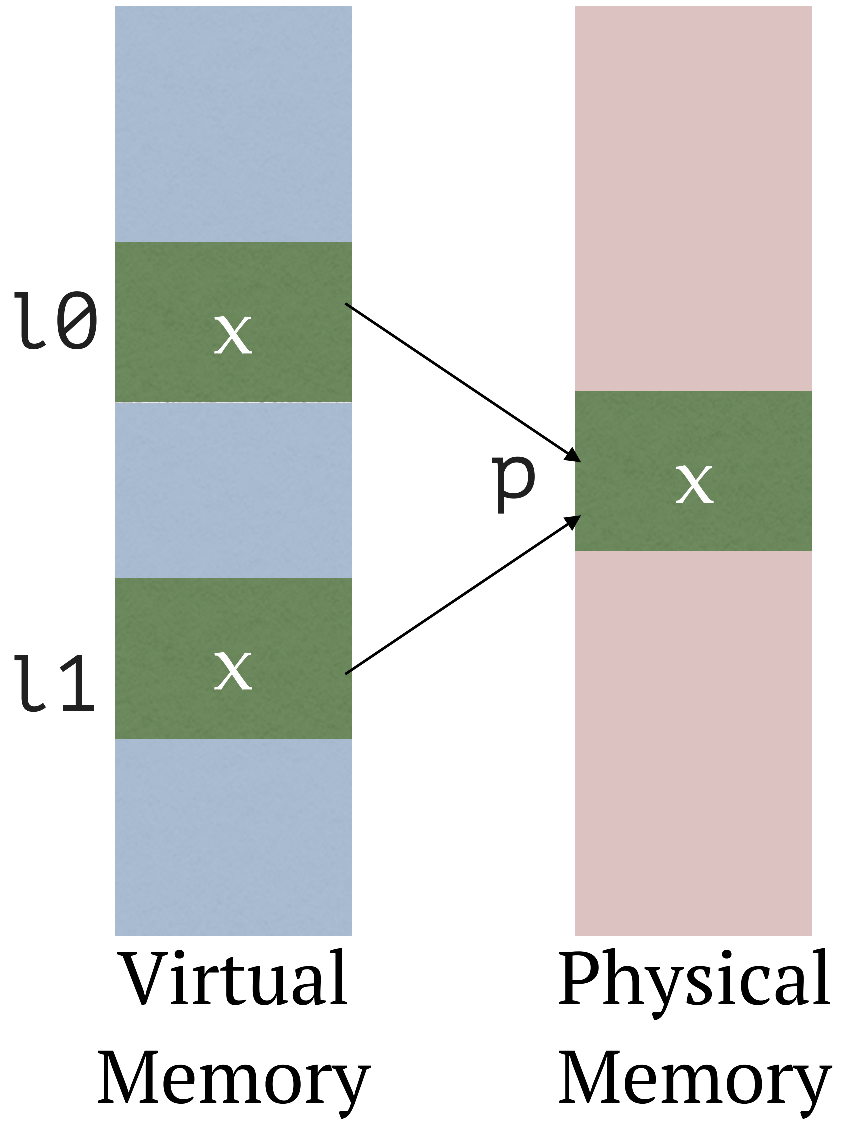
System Program: *zero-copy*

Specification:
Copy data `x` from virtual memory location `l0` to disjoint virtual memory location `l1`.

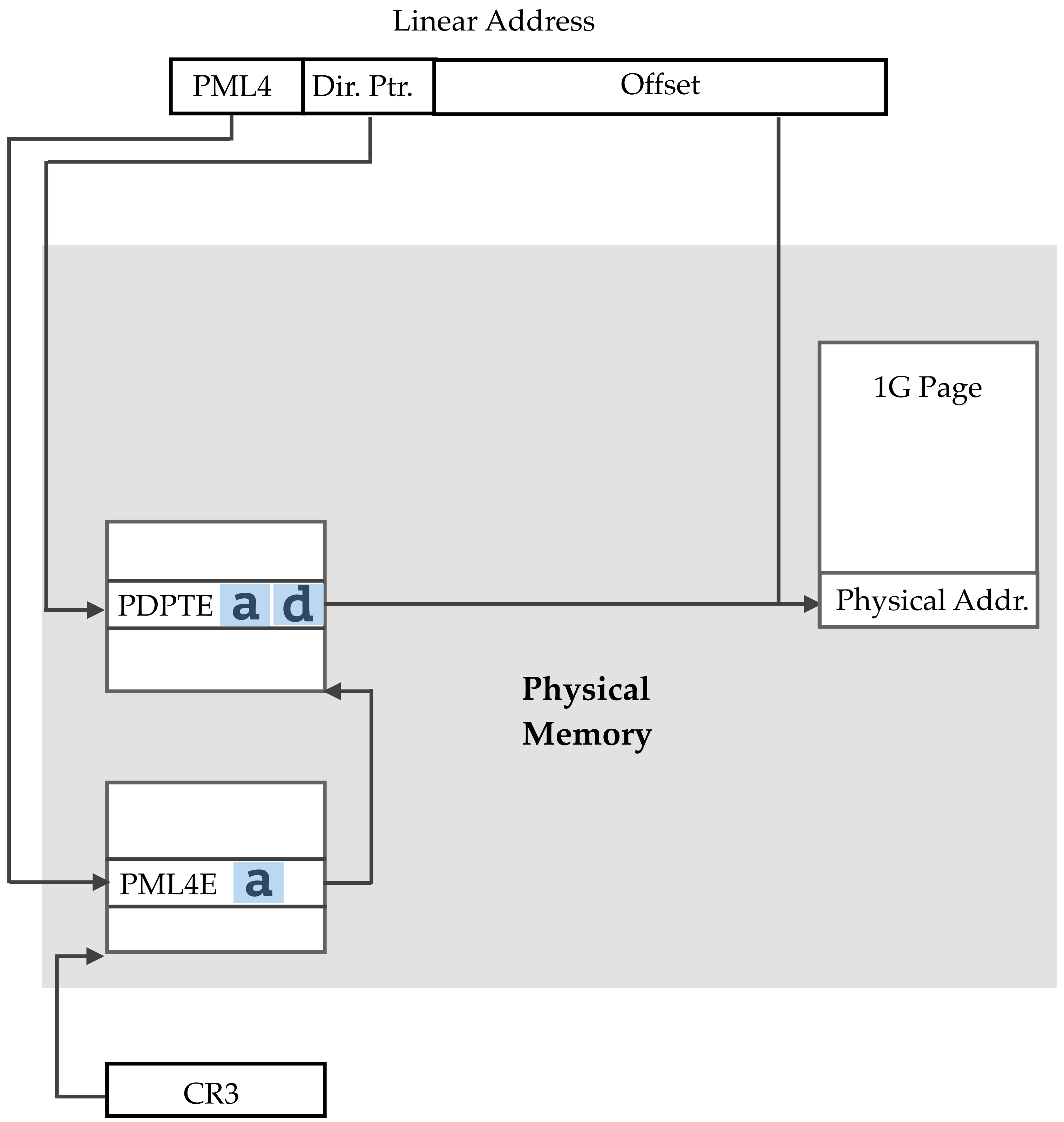
Verification Objective:
After a successful copy, `l0` and `l1` contain `x`.

Implementation:
Include the *copy-on-write* technique: `l0` and `l1` can be mapped to the same physical memory location `p`.

- Modifications to address mapping



Address Translations: IA-32e Paging (1G page)



a *accessed flag* **d** *dirty flag*

System Program: *zero-copy*

Functional Correctness: implementation of a zero-copy program meets the specification of a simple copy operation.

For simplicity, marking of x86 paging structures during traversal was turned off, i.e., no accessed and dirty bit updates were allowed.

We are currently re-doing this proof to account for updates to accessed and dirty bits.

Outline

Contributions

A new tool

- General-purpose analysis framework for x86 machine-code
- Accurate x86 ISA reference

Perform program verification cognizant of low-level ISA features

- E.g., properties of x86 memory-management data structures

Reasoning strategies

- Insight into low-level code verification in general
- Build effective lemma libraries

Foundation for future research

- Resource usage guarantees, information-flow analysis, etc.

Long-Term Goals

- **Run a 64-bit** FreeBSD **kernel** on our x86 ISA model
  - This involves identifying and implementing relevant instructions, call gates, supporting task management, etc.
- **Identify and prove critical invariants** in kernel code
  - This includes proving the correctness of context switches, privilege escalations, etc.
- Add **multiprocessor support** to the x86 ISA model

Accessing Source Code + Documentation

The `x86isa` project is available under **BSD 3-Clause license** as a part of the **ACL2 Community Books** project.

Go to https://github.com/acl2/acl2/
and see `books/projects/x86isa/README` for details.

Also, documentation and user's manual is available online at
www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____X86ISA

Highlights of this Talk

Compile-to and Build-to Specification

A formal, executable x86 ISA model

- Specification of low-level ISA features
- Handles non-determinism

Unified Model

- *Simulator:* Executable file readers & loaders; GDB-like mode for dynamic instrumentation
- *Reasoning Framework*: ACL2 libraries to reason about x86 machine code

User Manual

- Documentation

Open Source

- Available online