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

Shilpi Goel

The University of Texas at Austin

Ph.D. Dissertation Proposal

29th April, 2015

Outline

- Motivation
- State of the Art
- Proposed Dissertation Project
  - [Task 1] Developing an x86 ISA Model
  - [Task 2] Building a Machine-Code Analysis Framework
  - [Task 3] Verifying Application and System Programs
- Future Work
- Expected Contributions

Motivation

- Software systems are ubiquitous.
- Cost of incorrect software is extremely high.
- *Formal verification* can increase software quality.
- Approach: **Machine-code verification for x86 platforms**

Motivation

- ***Why not high-level code verification?***
  - ✗ High-level verification frameworks do not address compiler bugs
    - ✓ Verified/verifying compilers can help
    - ✗ But these compilers typically generate inefficient code
  - ✗ Need to build verification frameworks for many high-level languages
  - ✗ Sometimes, high-level code is unavailable

- ***Why x86?***
  - ✓ x86 is in widespread use — our approach will have immediate practical application

State of the Art: x86 Machine-Code Analysis

[*Morrisett et al.*] Formal x86 ISA specification	Security policy violation checks in browsers
[*Reps et al.*] Control & data flow analysis	Point tools; approximation of program behavior
[*Myreen*] Decompilation-in-Logic [*Moore*] Codewalker	Analysis of application programs
[*Klein et al.*] SeL4 [*Shao et al.*] Certified OS kernels	Clean-slate development of verified software

State of the Art: x86 Machine-Code Analysis

We need more!

- General analysis
- Precisely capture program behavior
- Analyze application *and* system programs
- Verify existing software

Security policy violation checks in browsers
Point tools; approximation of program behavior
Analysis of application programs
Clean-slate development of verified software

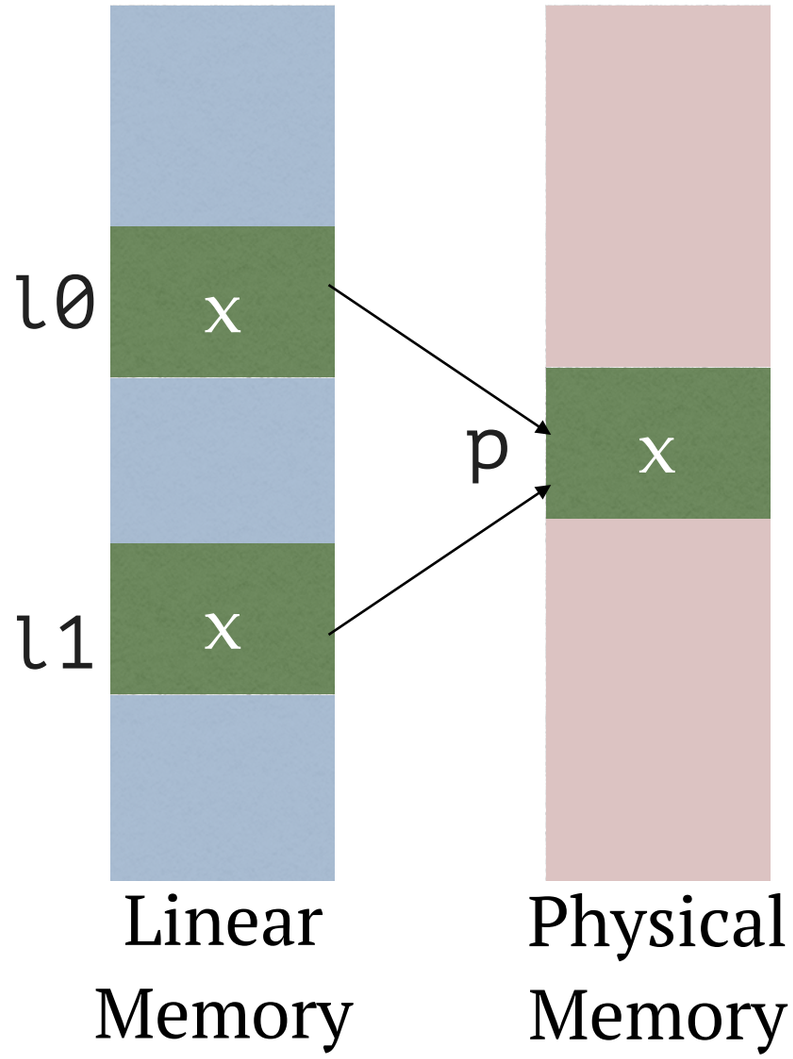
Example: Analysis of a Data-Copy Program

Specification:
Copy data `x` from linear (virtual) memory location `l0` to disjoint linear 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`.

- System calls
- Modifications to address mapping
- Access control management



Proposed Dissertation Project

- **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, and
  3. Employ this framework to verify application and system programs.

Expected Contributions

Briefly:

- ***A new tool:***
  general-purpose analysis framework for x86 machine-code

- ***Program verification taking memory management into account:***
  analysis of programs, including low-level system & ISA features

- ***Reasoning strategies:***
  insight into low-level code verification in general

- ***Foundation for future research:***
  target for verified/verifying compilers

Outline

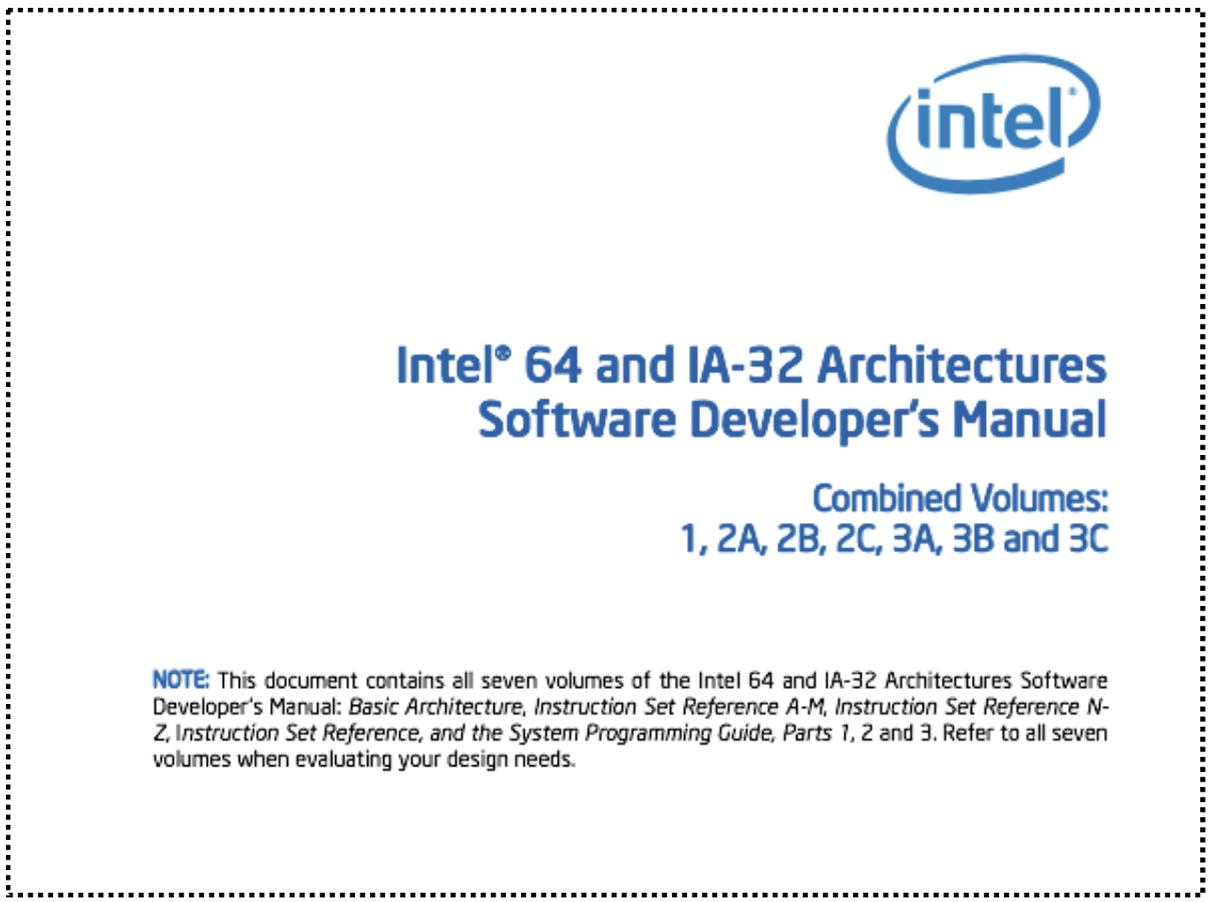
- Motivation
- State of the Art
- Proposed Dissertation Project
  - **[Task 1] Developing an x86 ISA Model**
  - [Task 2] Building a Machine-Code Analysis Framework
  - [Task 3] Verifying Application and System Programs
- Future Work
- Expected Contributions

Model Development

Obtaining the x86 ISA Specification

Model Development

Obtaining the x86 ISA Specification



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

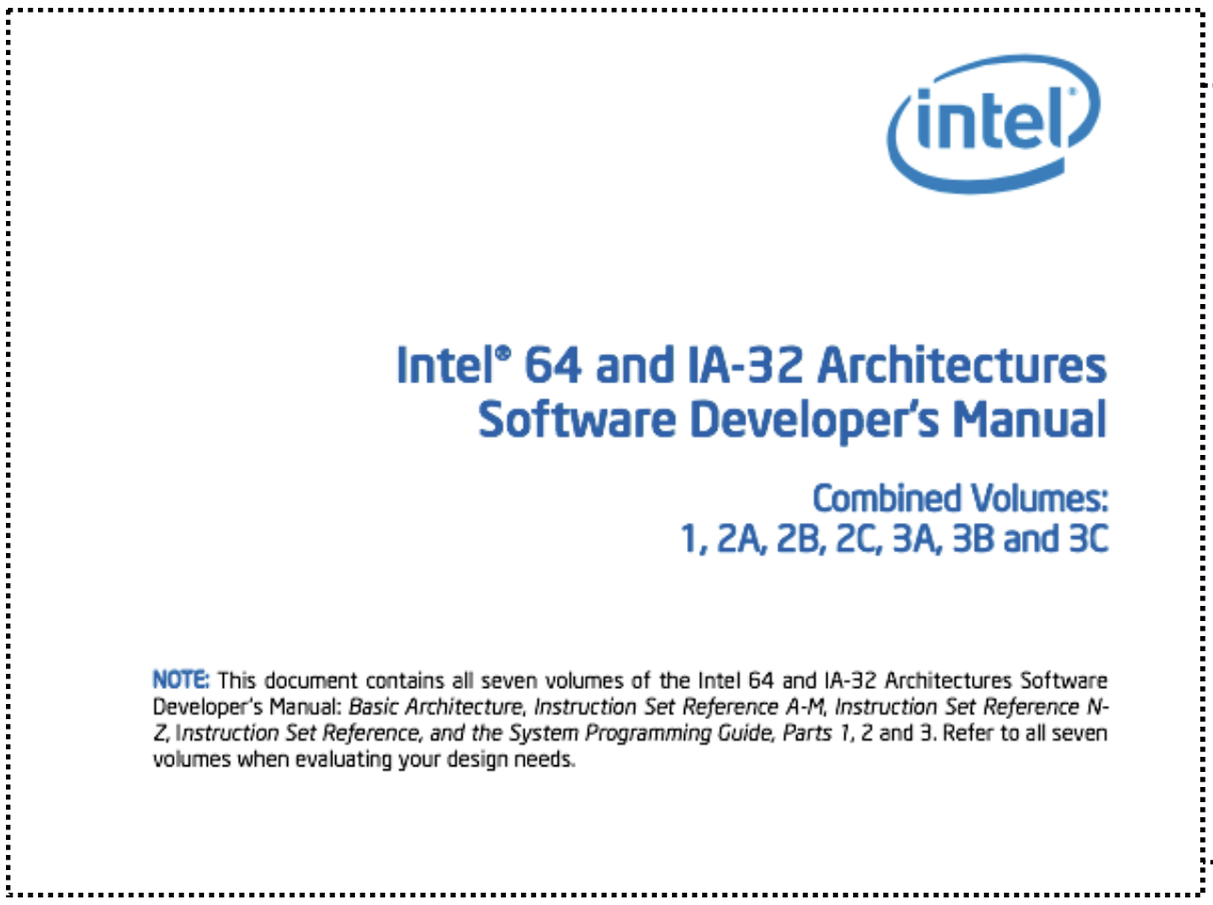
Combined Volumes:
1, 2A, 2B, 2C, 3A, 3B and 3C

NOTE: This document contains all seven volumes of the Intel 64 and IA-32 Architectures Software Developer's Manual: *Basic Architecture*, *Instruction Set Reference A-M*, *Instruction Set Reference N-Z*, *Instruction Set Reference*, *and the System Programming Guide*, *Parts* 1, 2 and 3. Refer to all seven volumes when evaluating your design needs.

~3400 pages

Model Development

Obtaining the x86 ISA Specification

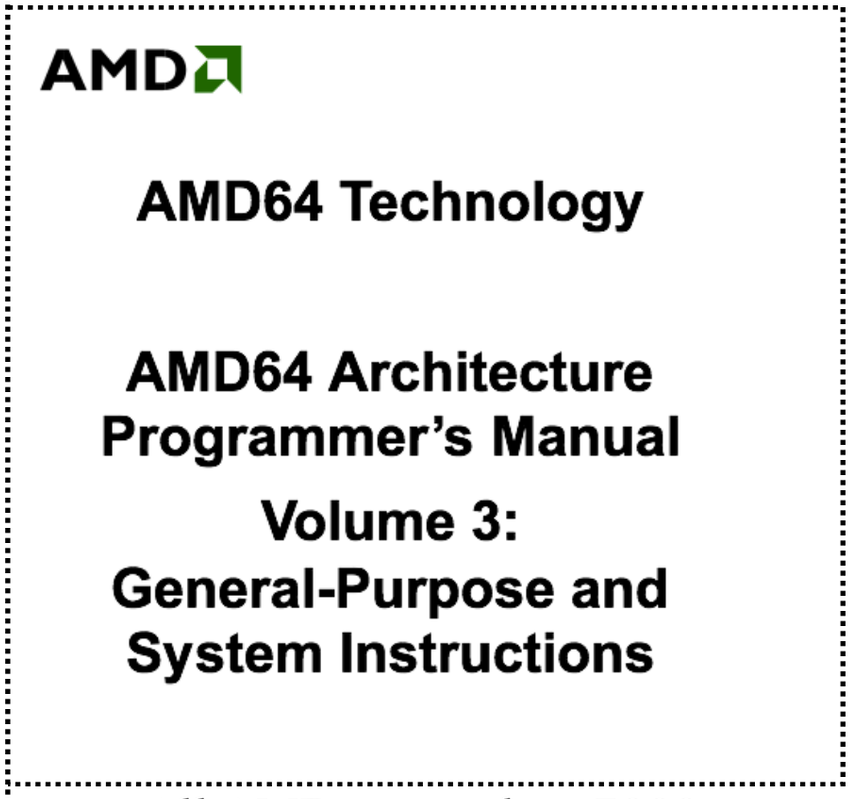


Intel® 64 and IA-32 Architectures
Software Developer's Manual

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

NOTE: This document contains all seven volumes of the Intel 64 and IA-32 Architectures Software Developer's Manual: *Basic Architecture*, *Instruction Set Reference A-M*, *Instruction Set Reference N-Z*, *Instruction Set Reference*, and the *System Programming Guide*, *Parts 1*, 2 and 3. Refer to all seven volumes when evaluating your design needs.

~3400 pages



AMD

AMD64 Technology

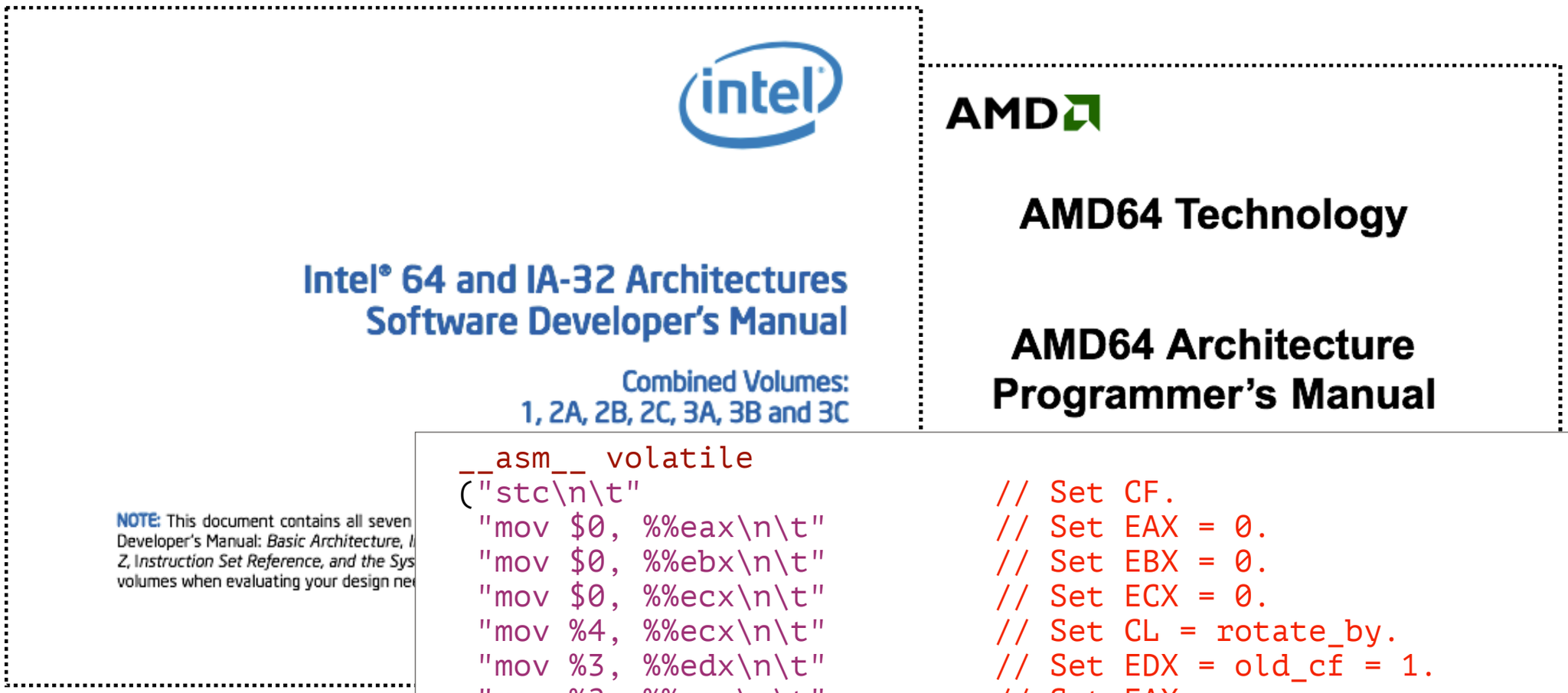
AMD64 Architecture
Programmer's Manual

Volume 3:
General-Purpose and
System Instructions

All AMD manuals: ~3000 pages

Model Development

Obtaining the x86 ISA Specification




```
__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 Development

Focus: 64-bit sub-mode of Intel's IA-32e mode

Model Development

Focus: 64-bit sub-mode of Intel's IA-32e mode

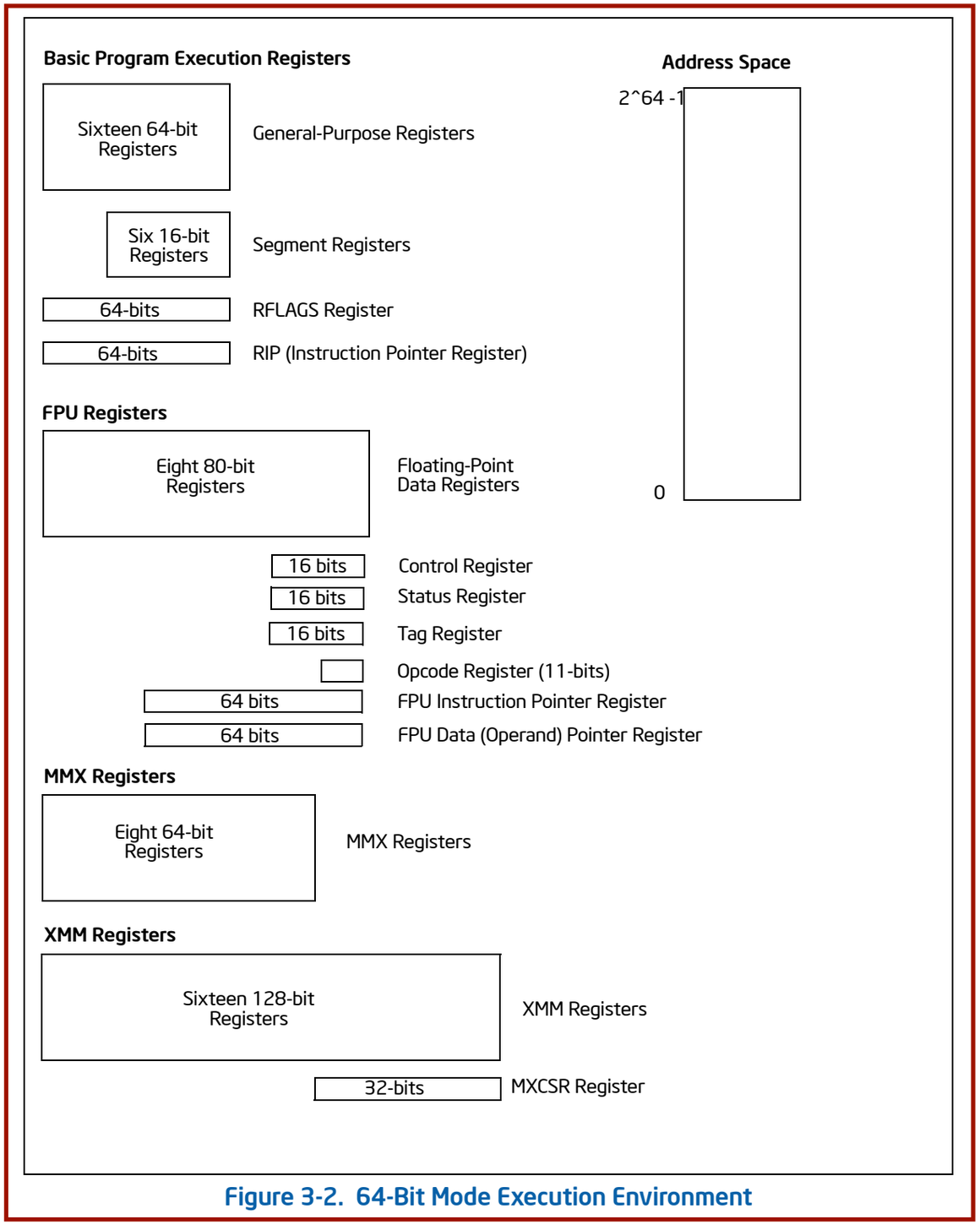


Figure 3-2. 64-Bit Mode Execution Environment

Source: Intel Manuals

Model Development

Focus: 64-bit sub-mode

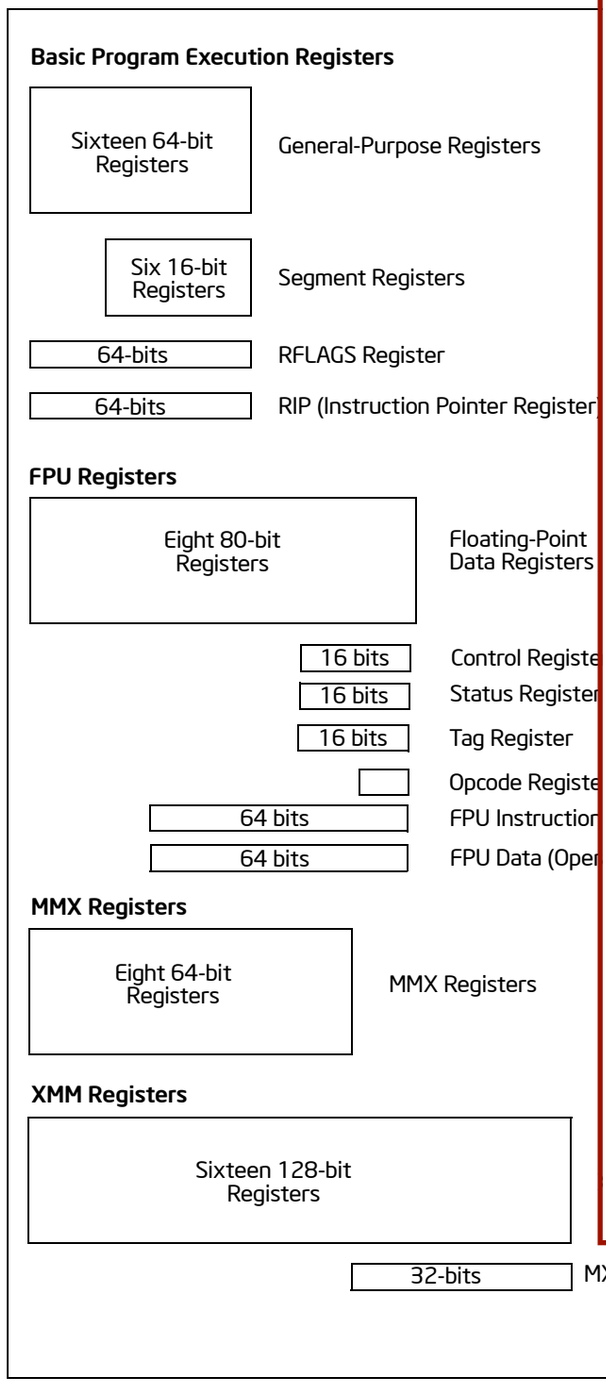


Figure 3-2. 64-Bit Mode Execution Environment

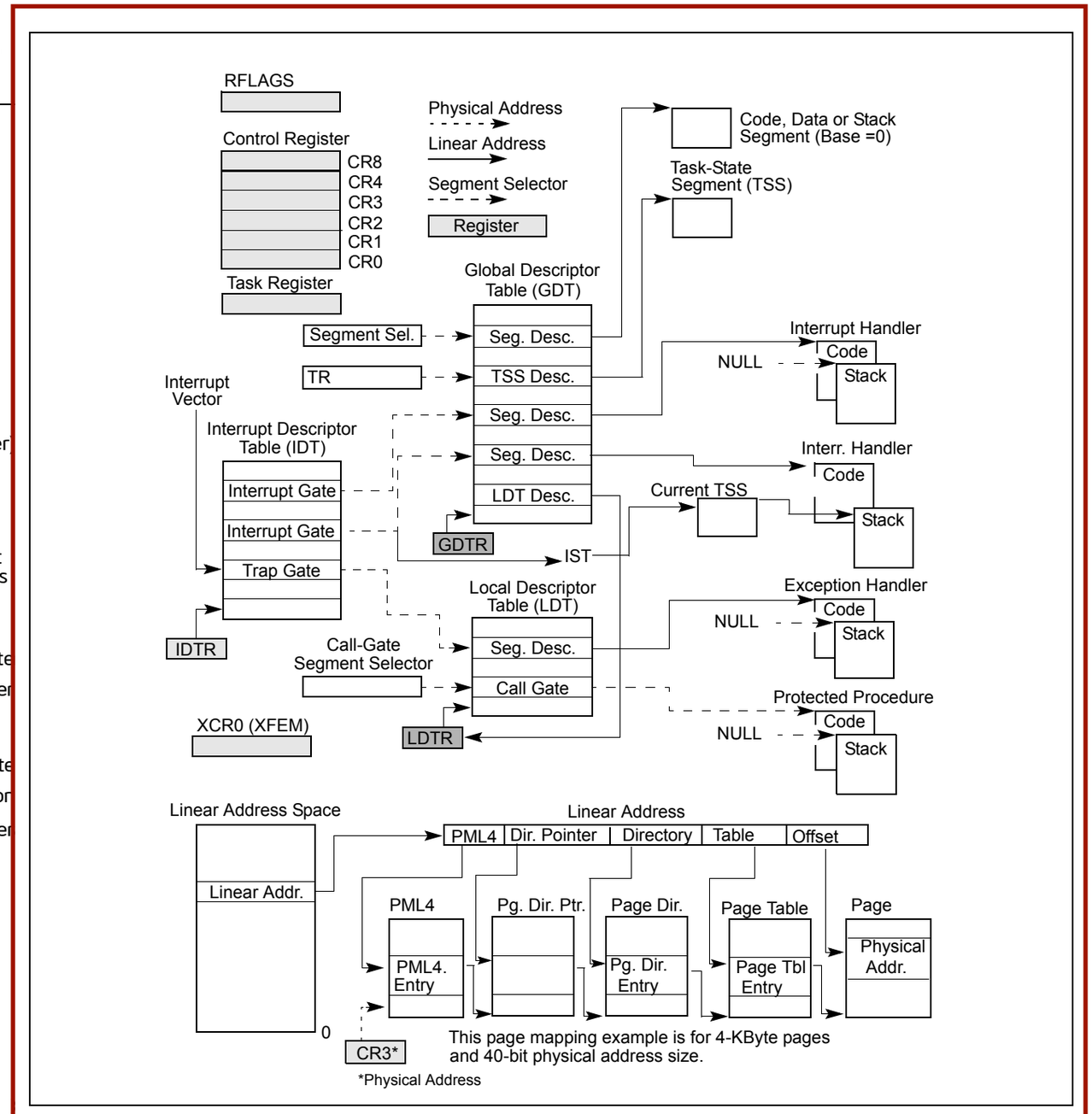


Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode

Source: Intel Manuals

Model Development

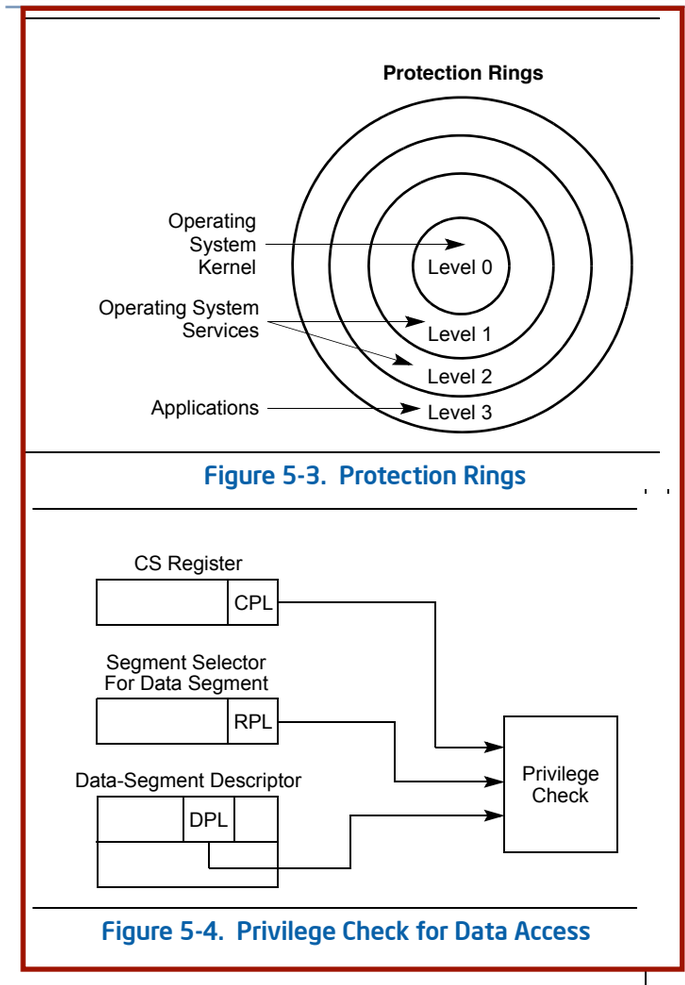
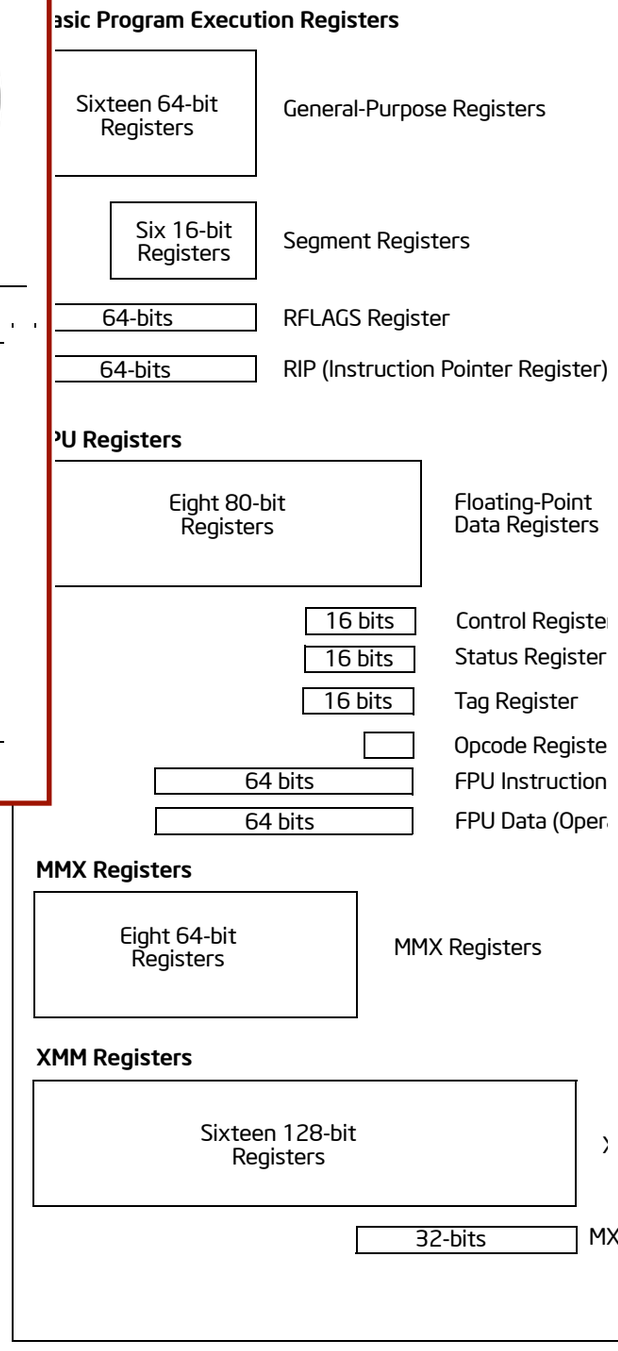


Figure 5-3. Protection Rings

Figure 5-4. Privilege Check for Data Access

64-bit sub-mode



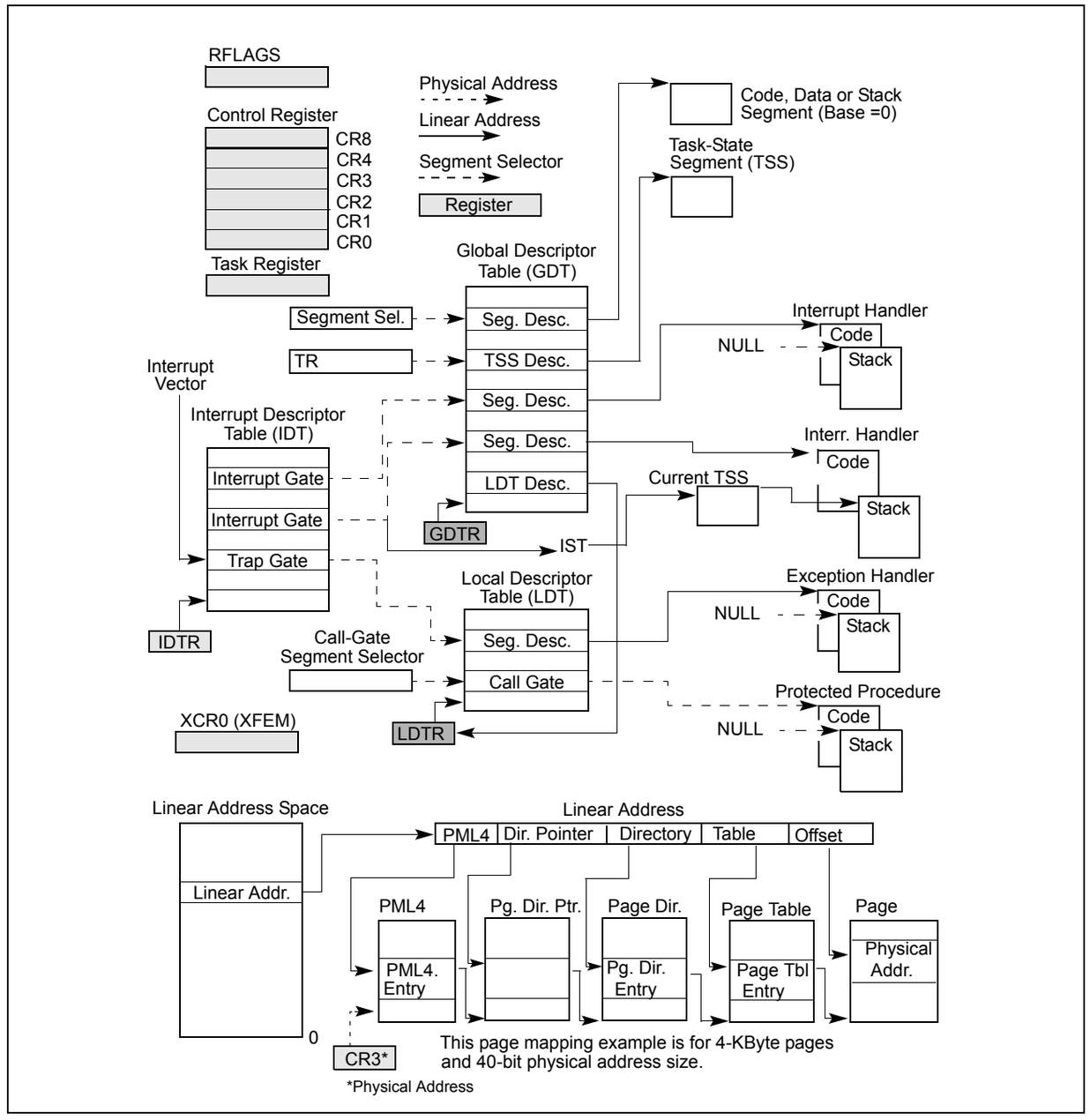


Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode

Figure 3-2. 64-Bit Mode Execution Environment

Source: Intel Manuals

Model Development

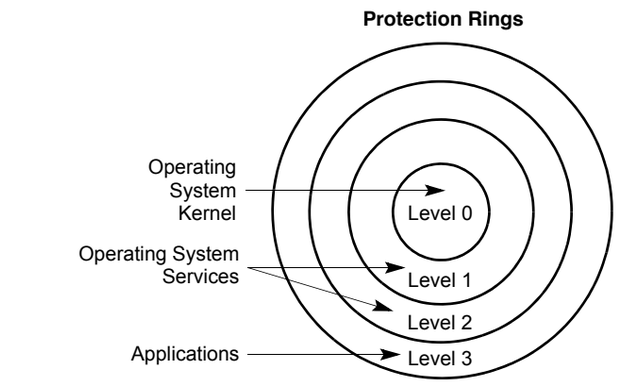


Figure 5-3. Protection Rings

Figure 6-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines

64-bit sub-mode

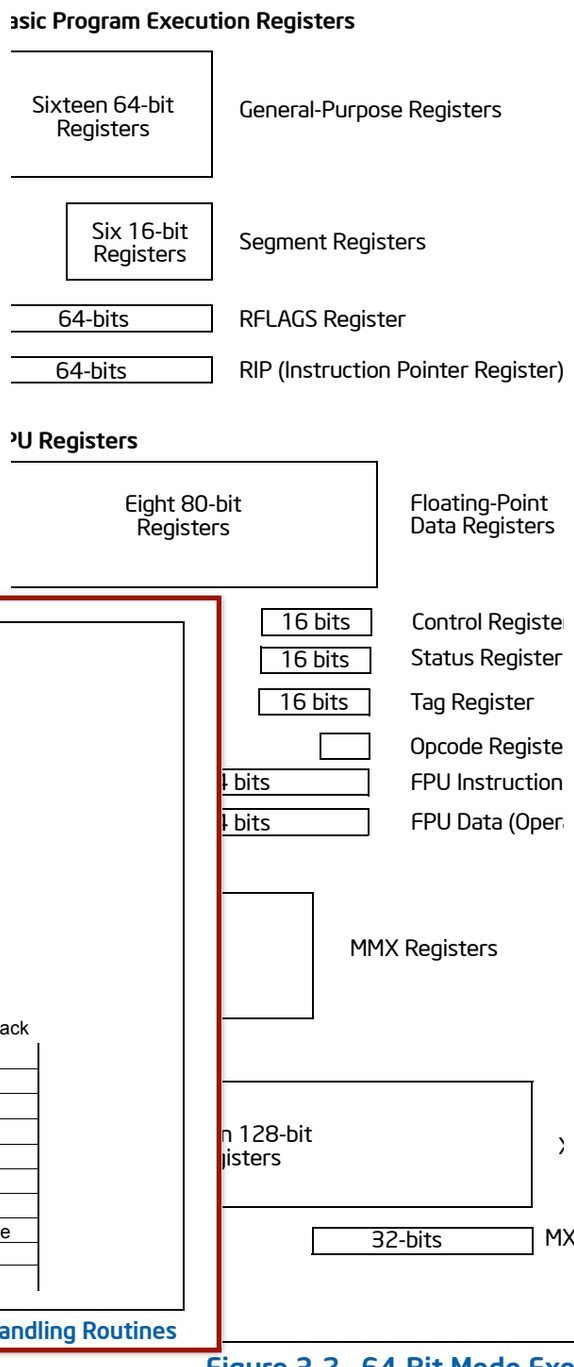


Figure 3-2. 64-Bit Mode Execution Environment

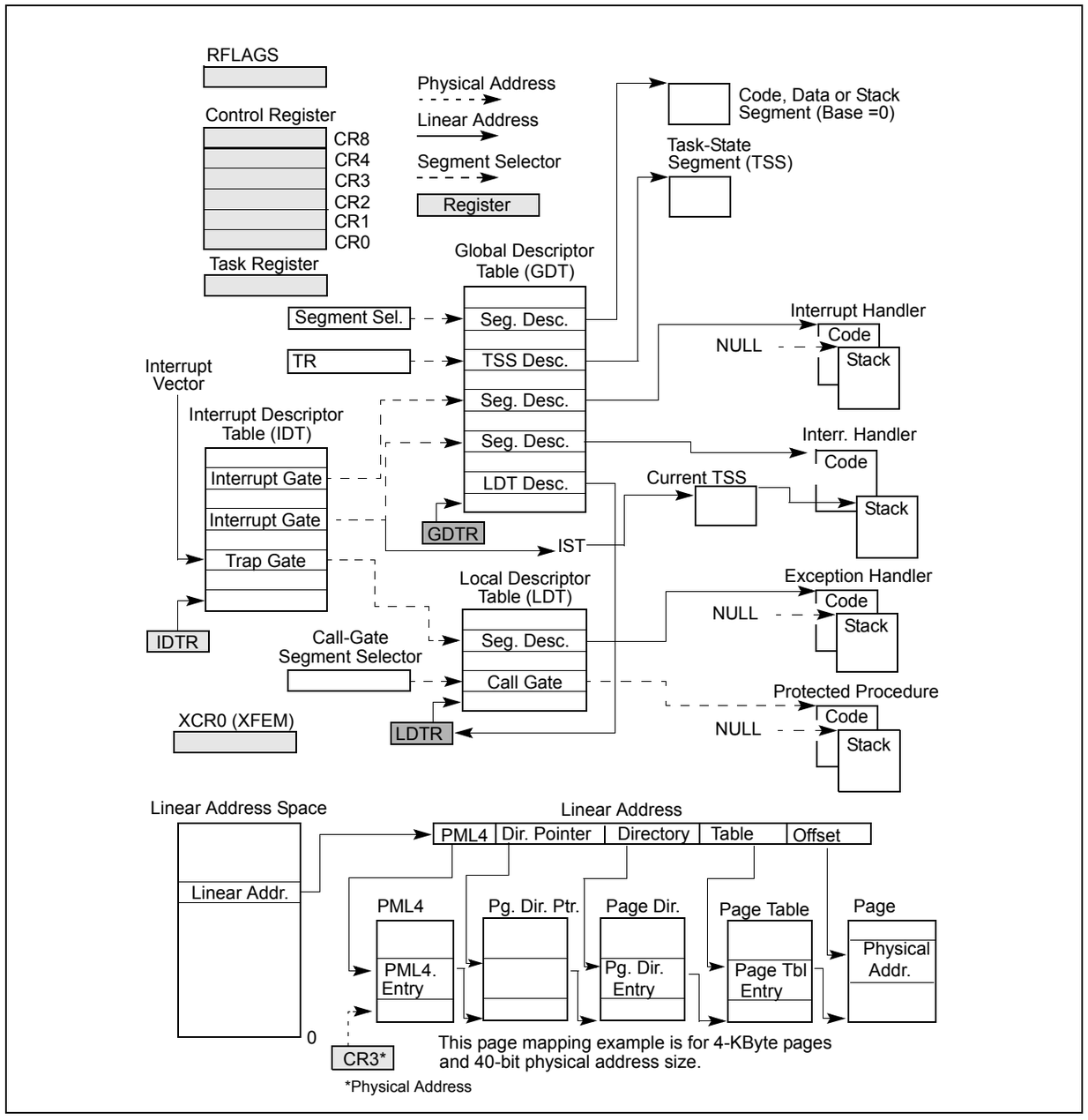


Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode

Source: Intel Manuals

Model Development

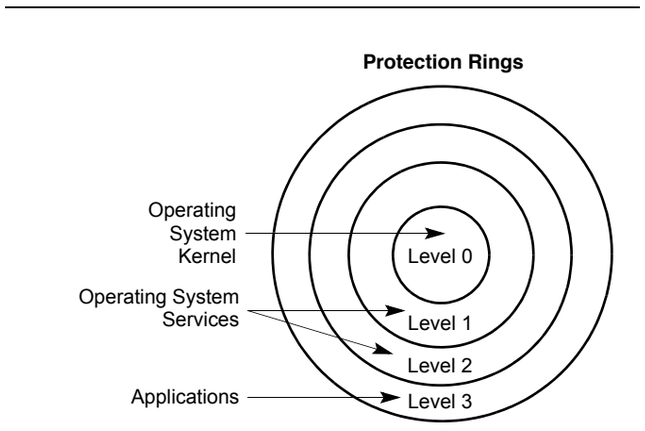


Figure 5-3. Protection Rings

64-bit sub-mode

asic Program Execution Registers

Sixteen 64-bit Registers	General-Purpose Registers
Six 16-bit Registers	Segment Registers
64-bits	RFLAGS Register

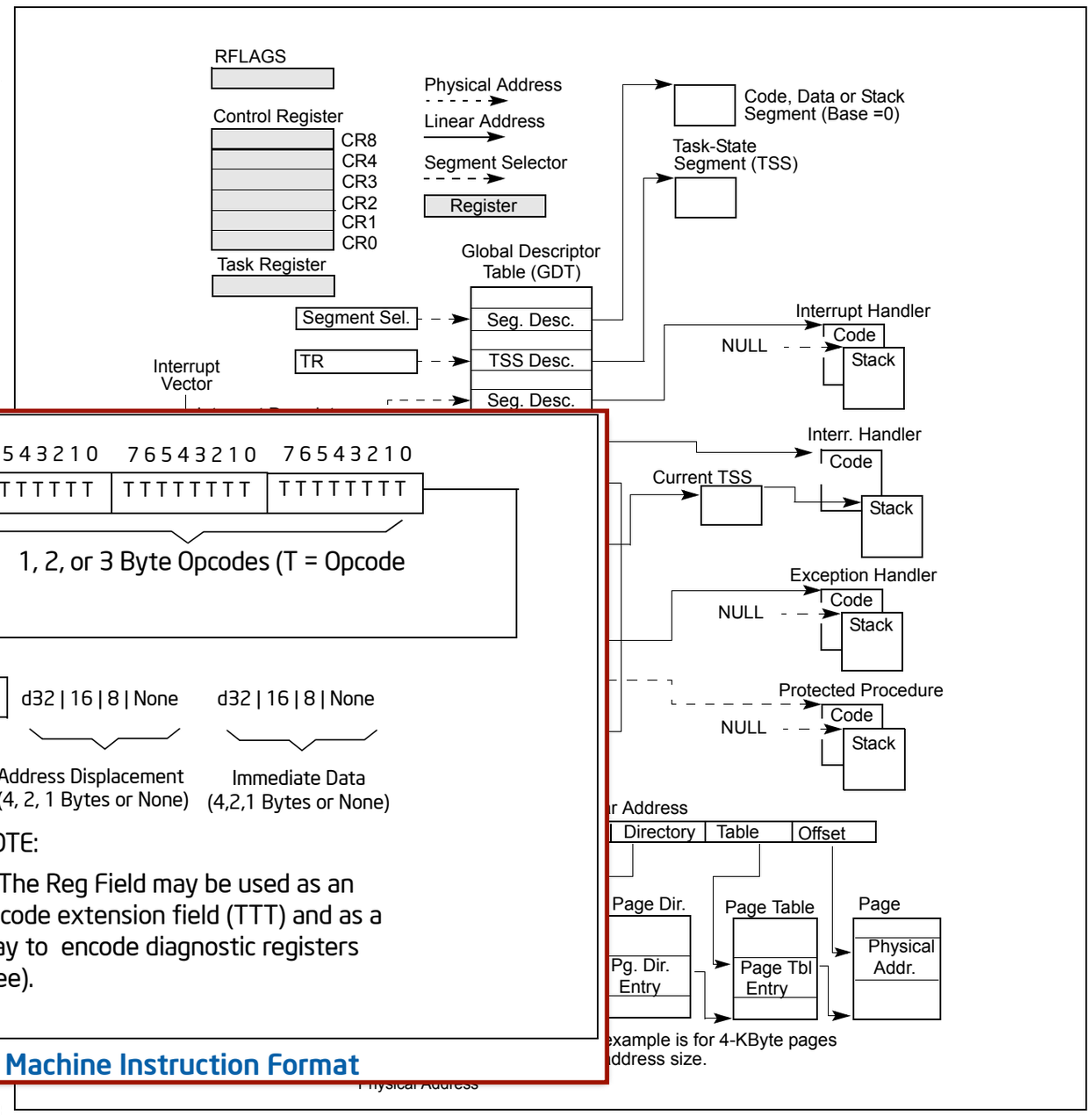


Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode

Figure B-1. General Machine Instruction Format

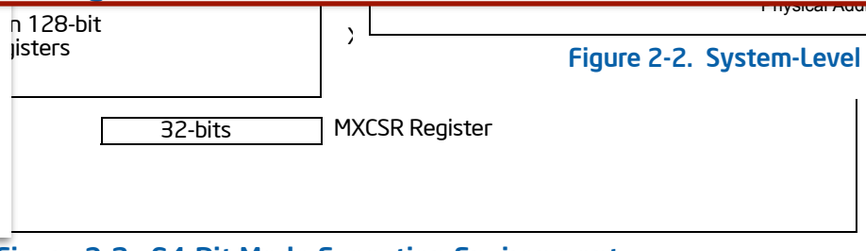


Figure 3-2. 64-Bit Mode Execution Environment

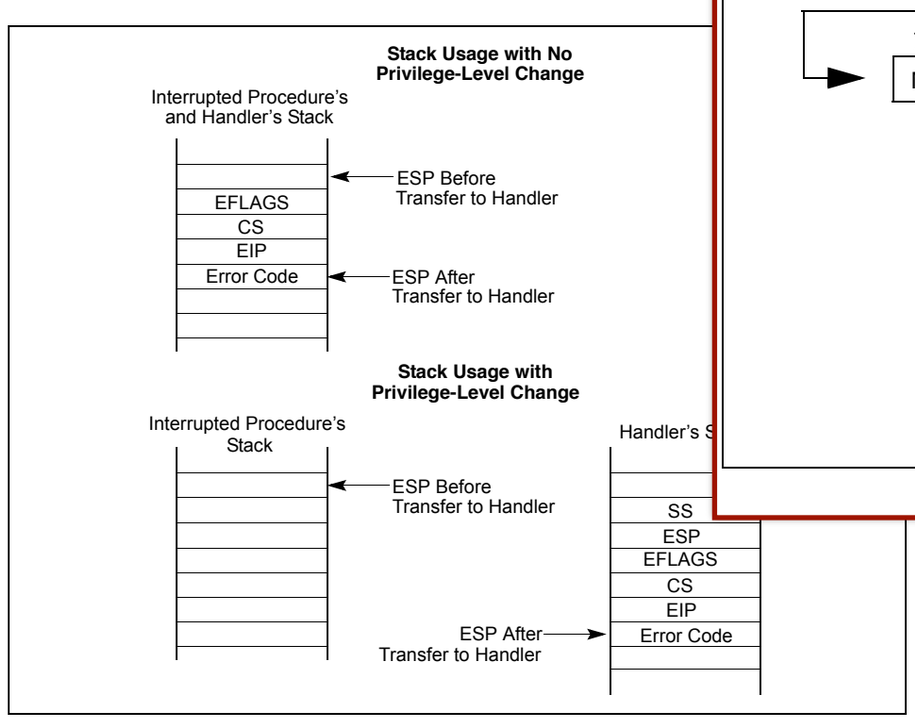


Figure 6-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines

Model Development

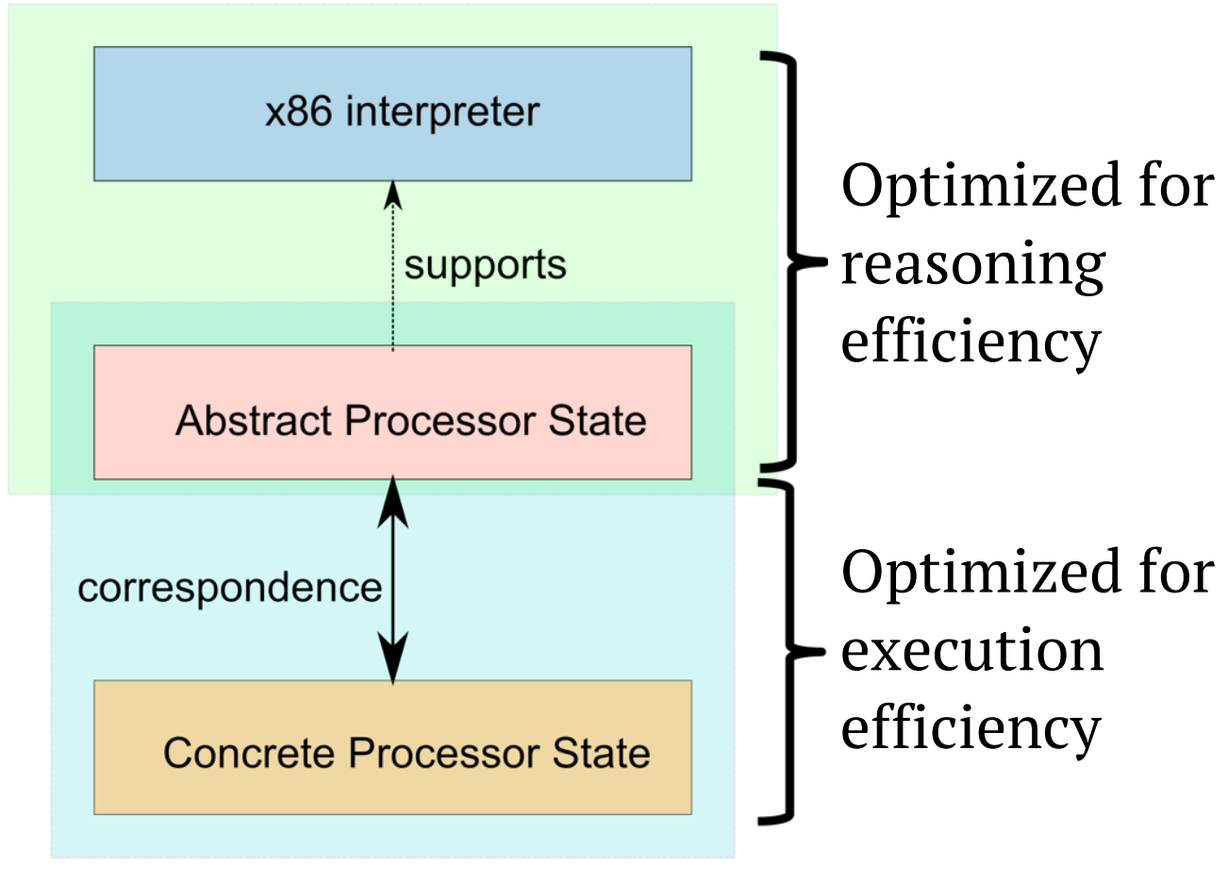
Under active development: an x86 ISA model in ACL2

- ***x86 State:*** specifies the components of the ISA (registers, flags, memory)

- ***Instruction Semantic Functions:*** specify the effect of each instruction

- ***Step Function:*** fetches, decodes, and executes one instruction

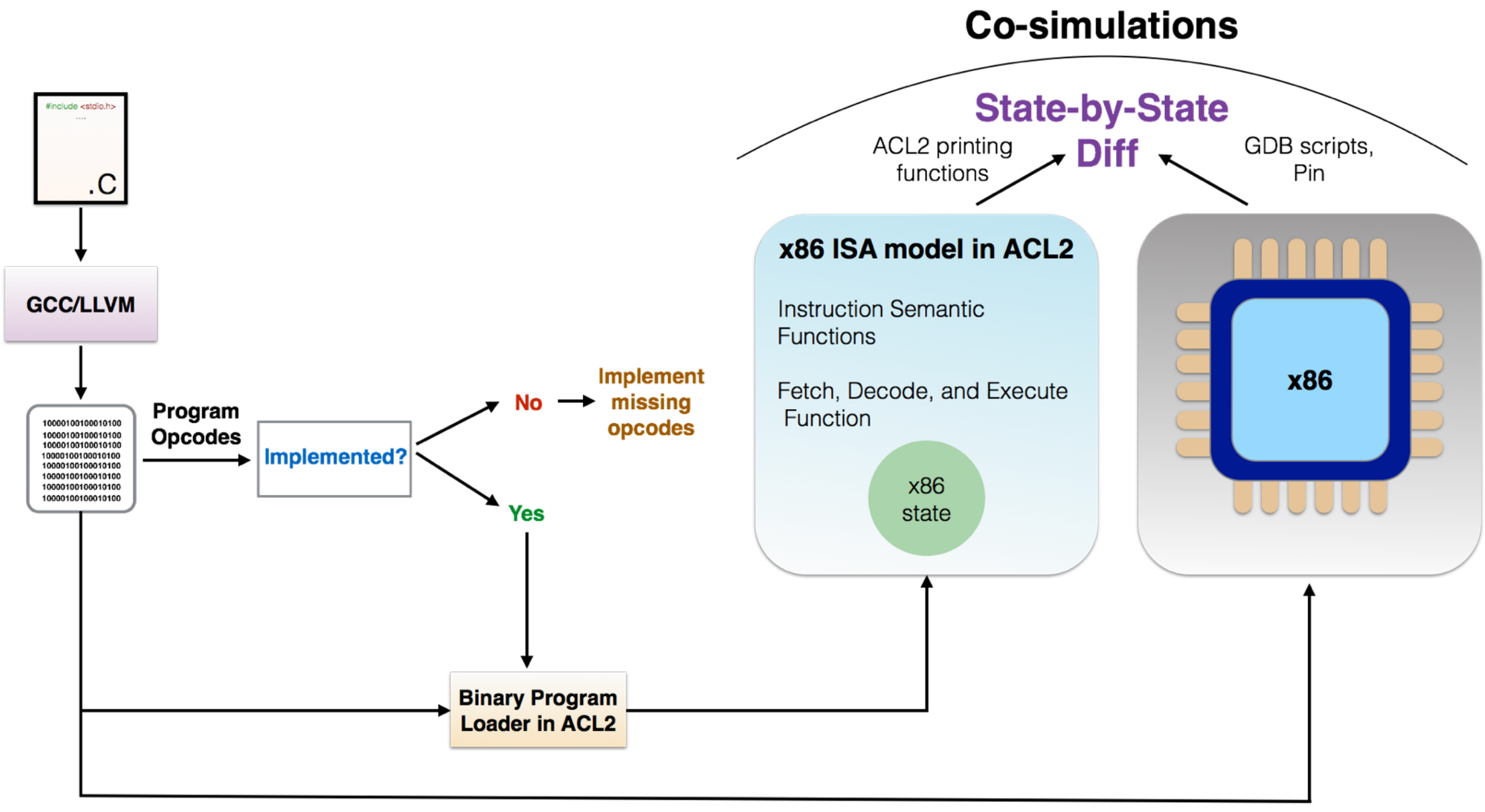
Layered modeling approach mitigates the trade-off between reasoning and execution efficiency [ACL2'13]



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.



Current Status: x86 ISA Model

- The x86 ISA model supports 100+ instructions (~220 opcodes)
  - Can execute almost all user-level programs emitted by GCC/LLVM
  - Successfully co-simulated a contemporary SAT solver on our model
- IA-32e paging for all page configurations (4K, 2M, 1G)
- Segment-based addressing
- Privileged instructions and system state
- Simulation speed*:
  - ~3.3 million instructions/second (paging disabled)
  - ~330,000 instructions/second (with 1G pages)

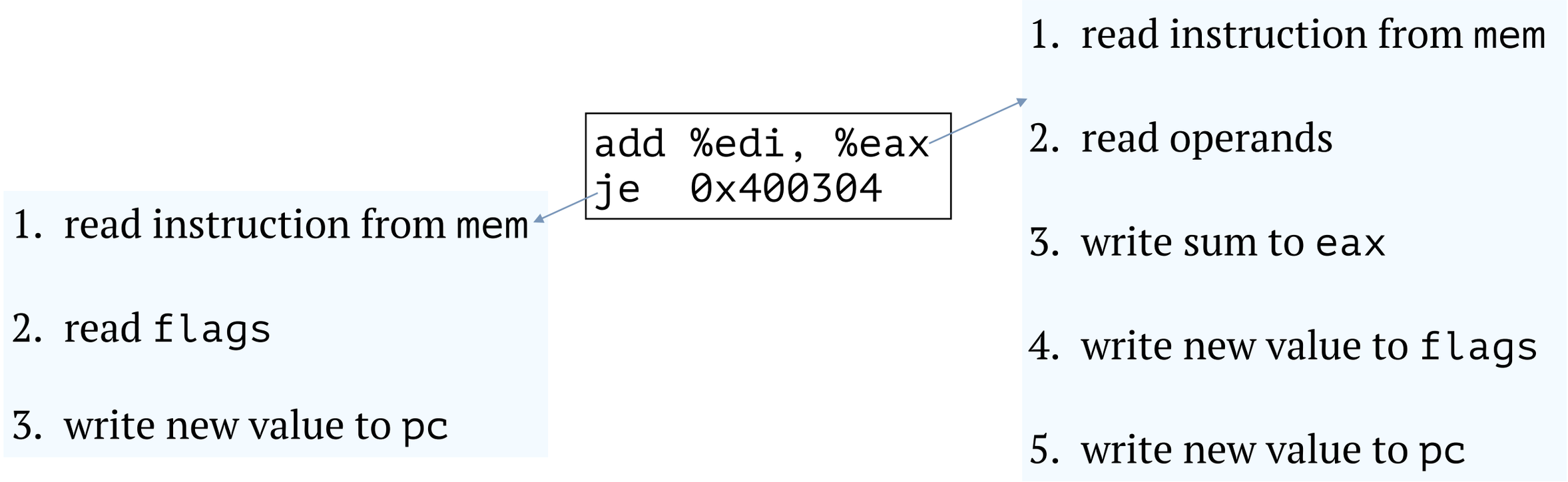
* Simulation speed measured on an Intel Xeon E31280 CPU @ 3.50GHz

Outline

- Motivation
- State of the Art
- Proposed Dissertation Project
  - [Task 1] Developing an x86 ISA Model
  - **[Task 2] Building a Machine-Code Analysis Framework**
  - [Task 3] Verifying Application and System Programs
- Future Work
- Expected Contributions

Lemma Database

- Semantics of the program is given by the effect it has on the machine state.

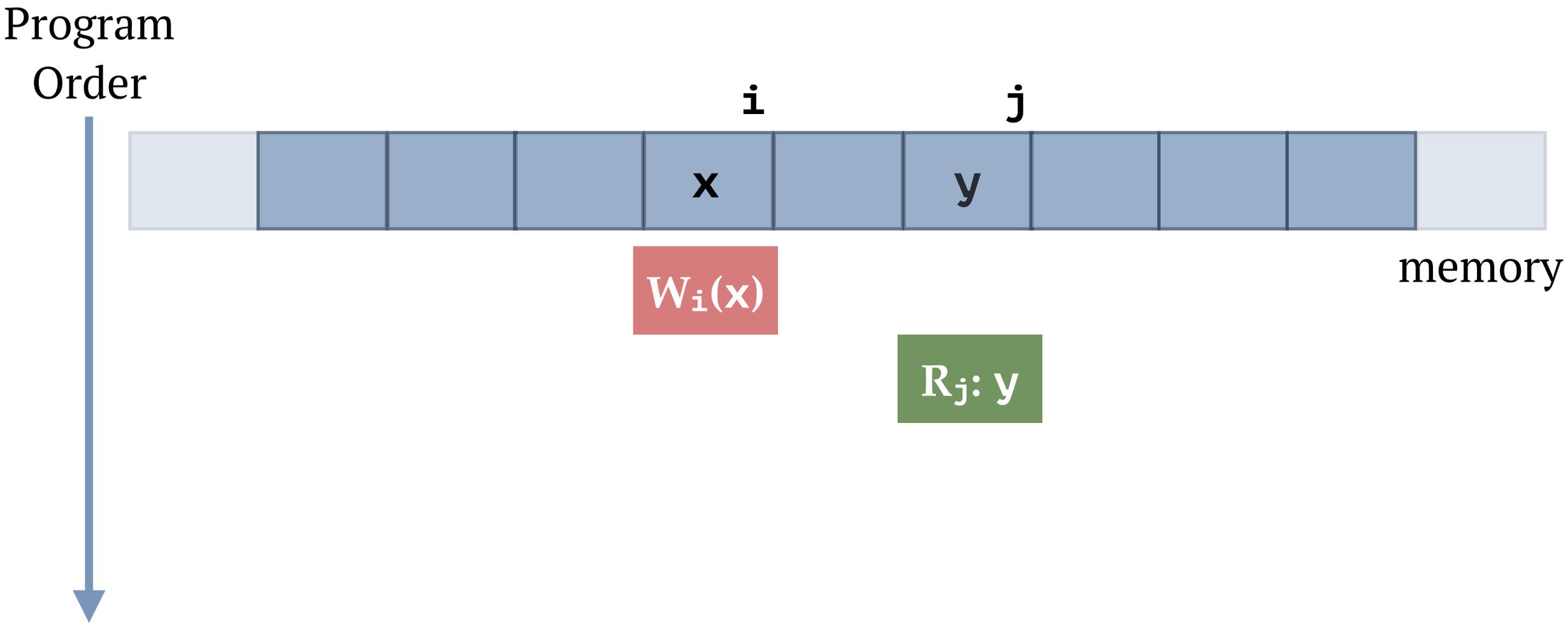


- Need to reason about:
  - Reads from machine state
  - Writes to machine state

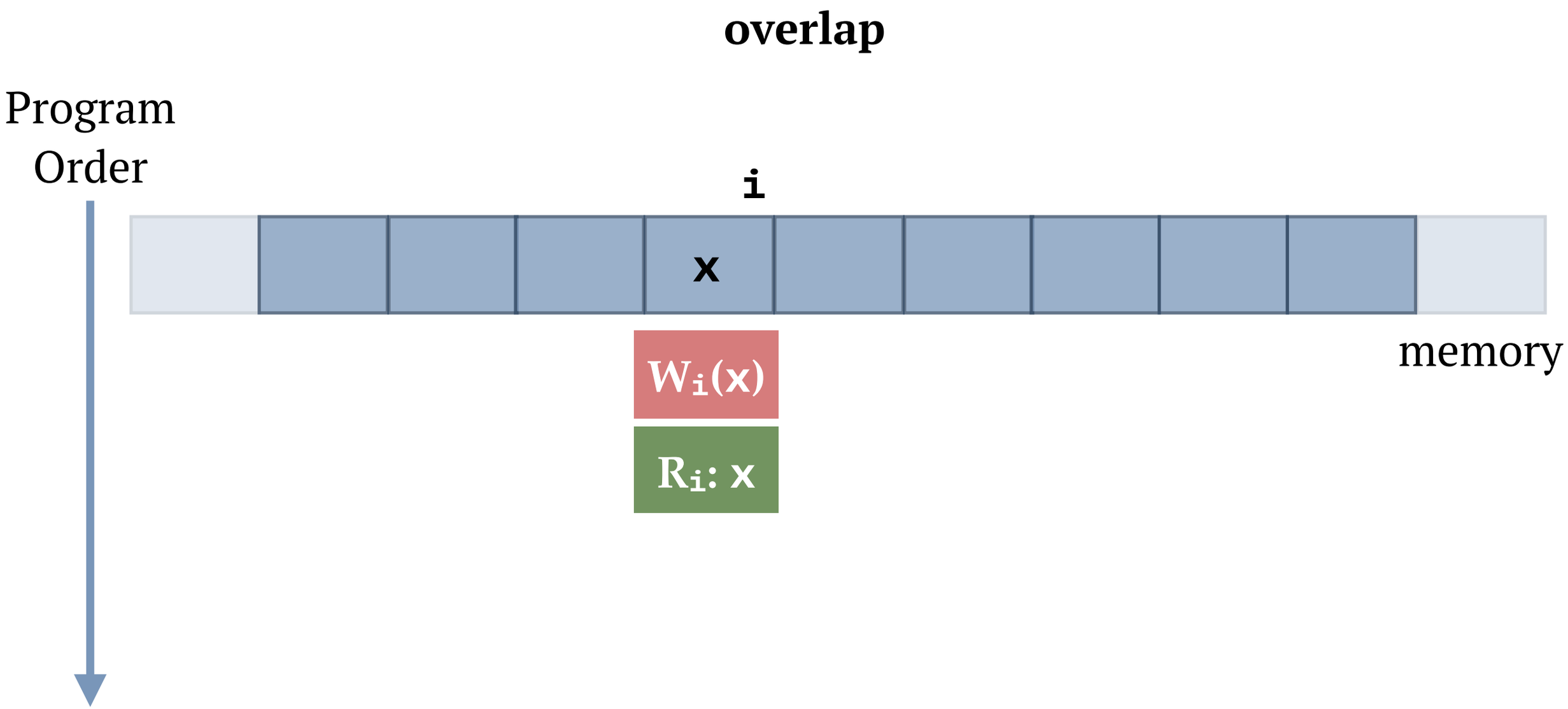
- Three kinds of theorems:
  - Read-over-Write Theorems
  - Write-over-Write Theorems
  - Preservation Theorems

Read-over-Write Theorem: #1

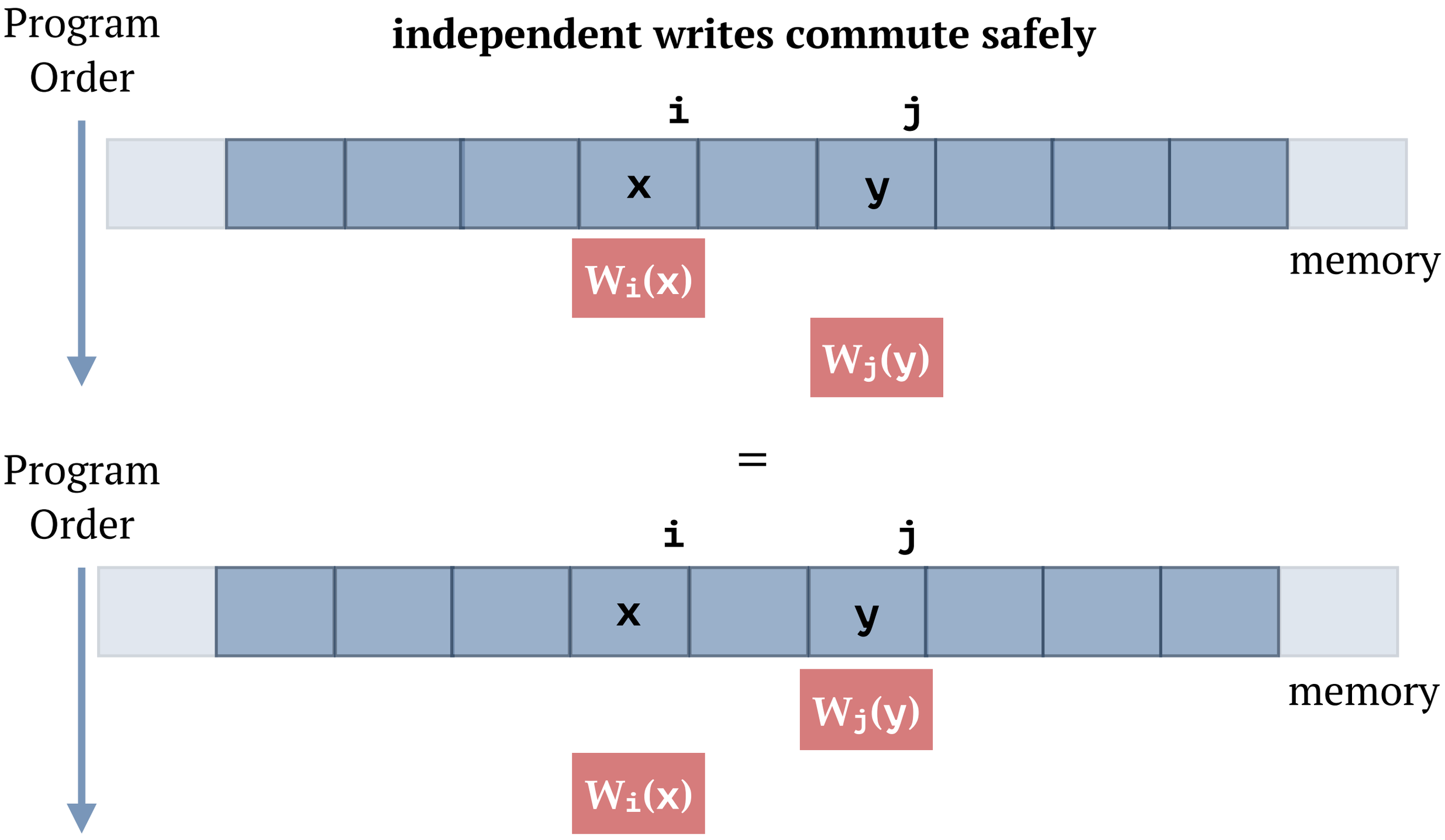
non-interference



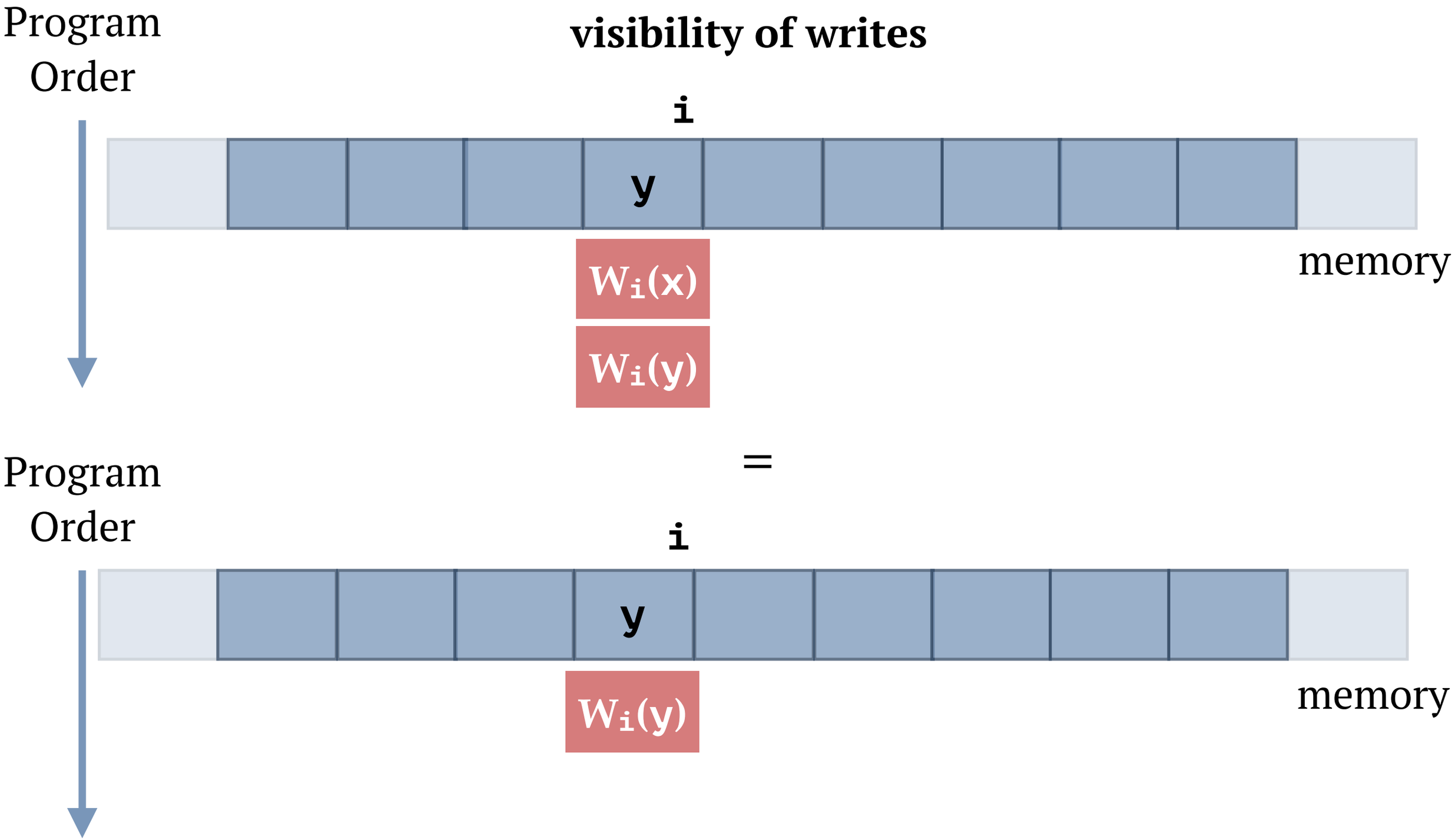
Read-over-Write Theorem: #2



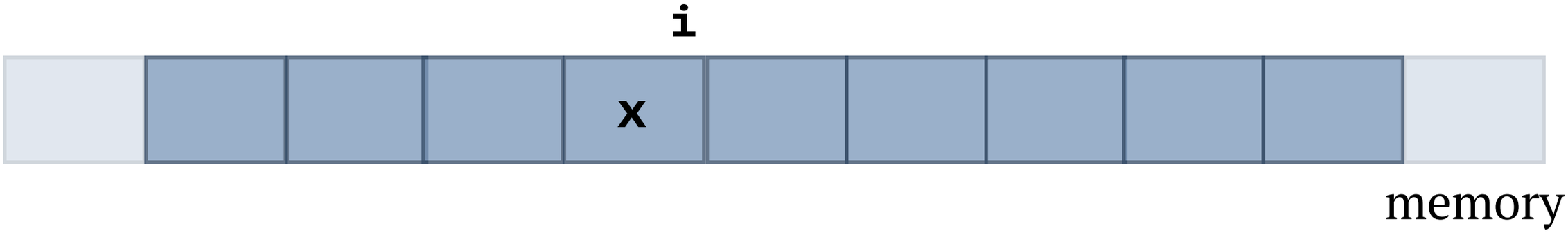
Write-over-Write Theorem: #1



Write-over-Write Theorem: #2



Preservation Theorems



reading from a valid x86 state

```
valid-address-p(i) ∧
valid-x86-p(x86)
⇒
valid-value-p( Ri: x ) ∧
valid-x86-p(x86)
```

writing to a valid x86 state

```
valid-address-p(i) ∧
valid-value-p(x) ∧
valid-x86-p(x86)
⇒
valid-x86-p( Wi(x) )
```

Address translations for **R_i: x** **W_i(x)**

Logical Address

Segment Selector

Offset

Descriptor Table(s)

Segment Descriptor

Linear Addr.

4K Page

Segment

Linear Memory

Global or Local Descriptor Table Register

Linear Address

PML4 | Dir. Ptr. | Dir. | Table | Offset

PDE **a**

PDPTE **a**

PML4E **a**

CR3

Control Register has the base address of these structures.

PTE **a** **d**

Physical Addr.

4K Page

Physical Memory

SEGMENTATION

a *accessed flag*

d *dirty flag*

IA-32e PAGING (4K page)

Current Status: Analysis Framework

- Automatically generate and prove:
  - Read-over-Write theorems
  - Write-over-Write theorems
  - Preservation theorems
- Libraries to reason about (non-)interference of memory regions
- Predicates that recognize valid paging structure entries
- Proved some properties about paging data structure traversals

Outline

- Motivation
- State of the Art
- Proposed Dissertation Project
  - [Task 1] Developing an x86 ISA Model
  - [Task 2] Building a Machine-Code Analysis Framework
  - **[Task 3] Verifying Application and System Programs**
- Future Work
- Expected Contributions

Verification Effort vs. Verification Utility

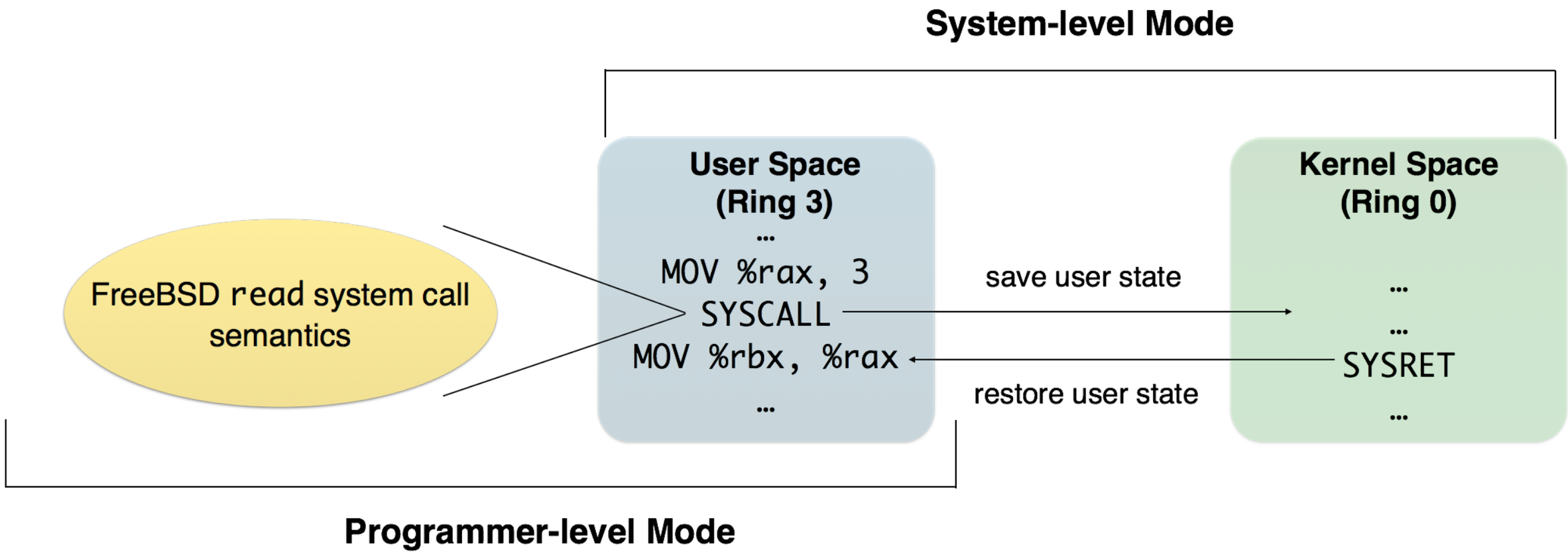
Programmer-level Mode

- Verification of *application* programs
- *Linear* memory address space (2^{64} bytes)
- *Assumptions* about correctness of OS operations

System-level Mode

- Verification of *system* programs
- *Physical* memory address space (2^{52} bytes)
- *No assumptions* about OS operations

Verification Effort vs. Verification Utility



Application Program #1: *popcount*

Automatically verify snippets of straight-line machine code using symbolic simulation [VSTTE'13]

```
55                  push   %rbp
48 89 e5            mov    %rsp,%rbp
89 7d fc            mov    %edi,-0x4(%rbp)
8b 7d fc            mov    -0x4(%rbp),%edi
8b 45 fc            mov    -0x4(%rbp),%eax
c1 e8 01            shr    $0x1,%eax
25 55 55 55 55      and    $0x55555555,%eax
29 c7               sub    %eax,%edi
89 7d fc            mov    %edi,-0x4(%rbp)
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
```

unsigned int

```
RAX = popcount(input)
```

specification function

```
popcount(x):

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

Application Program #2: *word-count*

- Proved the functional correctness of a word-count program that reads input from the user using `read` system calls [FMCAD'14]
- Interesting; system calls are ***non-deterministic*** for application programs

Specification for counting the characters in `str`:

```
ncSpec(offset, str, count):

if (well-formed(str) && offset < len(str)) then
   c := str[offset]
   if (c == EOF) then
      return count
   else
      count := (count + 1) mod 2^32
      ncSpec(1 + offset, str, count)
   endif
endif
```

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

Application Program #2: *word-count*

Other properties verified using our machine-code framework:

- **Resource Usage:**
  - Program and its stack are disjoint for all inputs.
  - Irrespective of the input, program uses a fixed amount of memory.

- **Security:**
  - Program does not modify unintended regions of memory.

Future Work

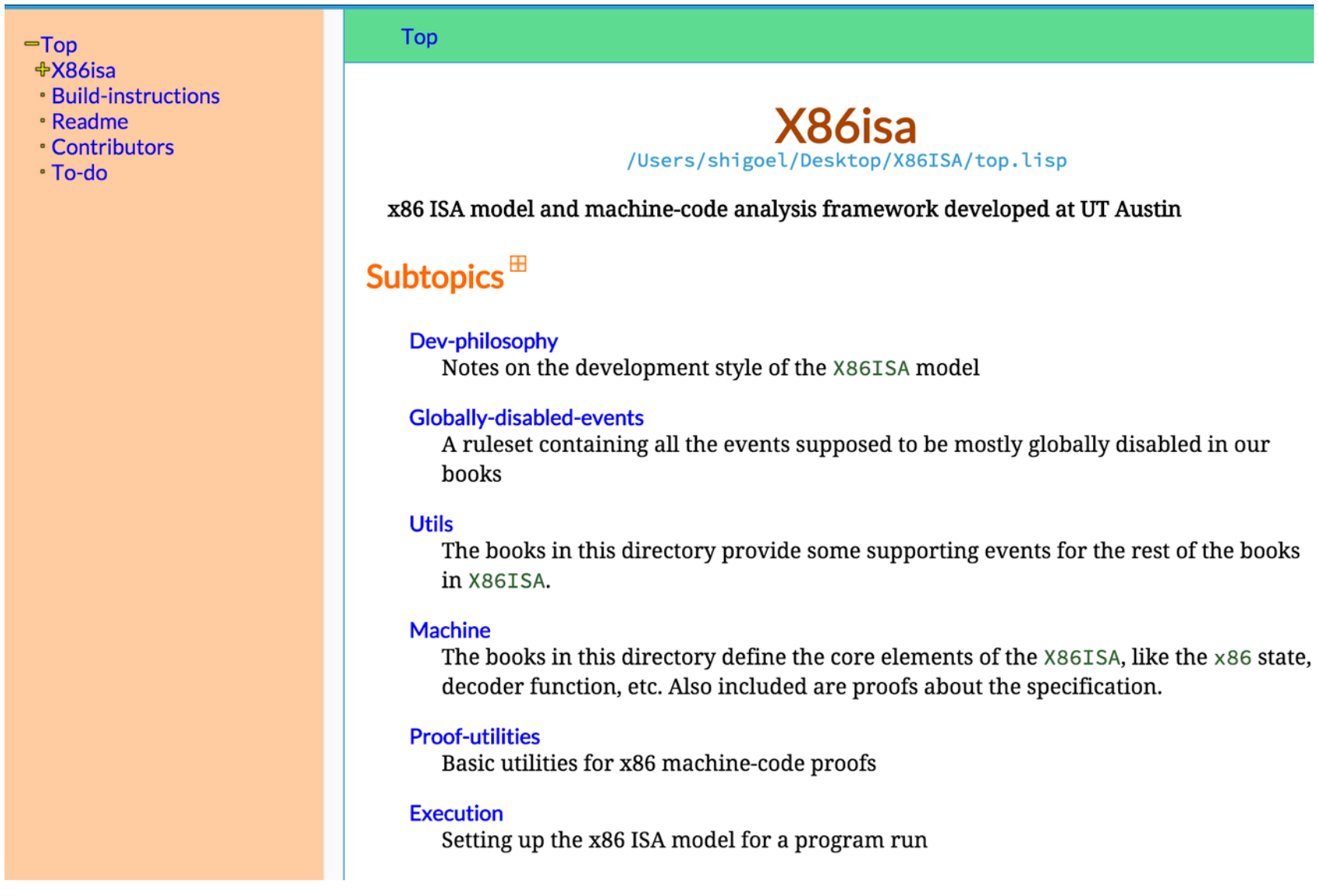
[Task 3]: System Program Verification:
The optimized data-copy program that accesses and manipulates x86 memory-management data structures

- In support of **[Task 3]**, we will continue to:
  - **[Task 1]:** *Model additional features* of x86 ISA
  - **[Task 2]:** Reason about reads from, writes to, and non-interference of x86 *memory-management data structures*

Overview: Current Status

Tasks	% Completed
[Task 1] x86 ISA Model	90%
[Task 2] Machine-Code Analysis Framework	50%
[Task 3] Program Verification	40%

Comprehensive documentation and user's manual



Expected Contributions

- ***A new tool:*** General-purpose analysis framework for x86 machine-code
  - Accurate x86 ISA reference

- ***Program verification taking memory management into account:***
  - Properties of x86 memory-management data structures
  - Analysis of programs, including low-level system & ISA features

- ***Reasoning strategies:*** Insight into low-level code verification in general
  - Build effective lemma libraries

- ***Foundation for future research:***
  - Target for verified/verifying compilers
  - Resource usage guarantees
  - Information-flow analysis
  - Ensuring process isolation

Publications

- ***Shilpi Goel**, Warren A. Hunt, Jr., and Matt Kaufmann.*
  Abstract Stobjs and Their Application to ISA Modeling
  In ACL2 Workshop, 2013

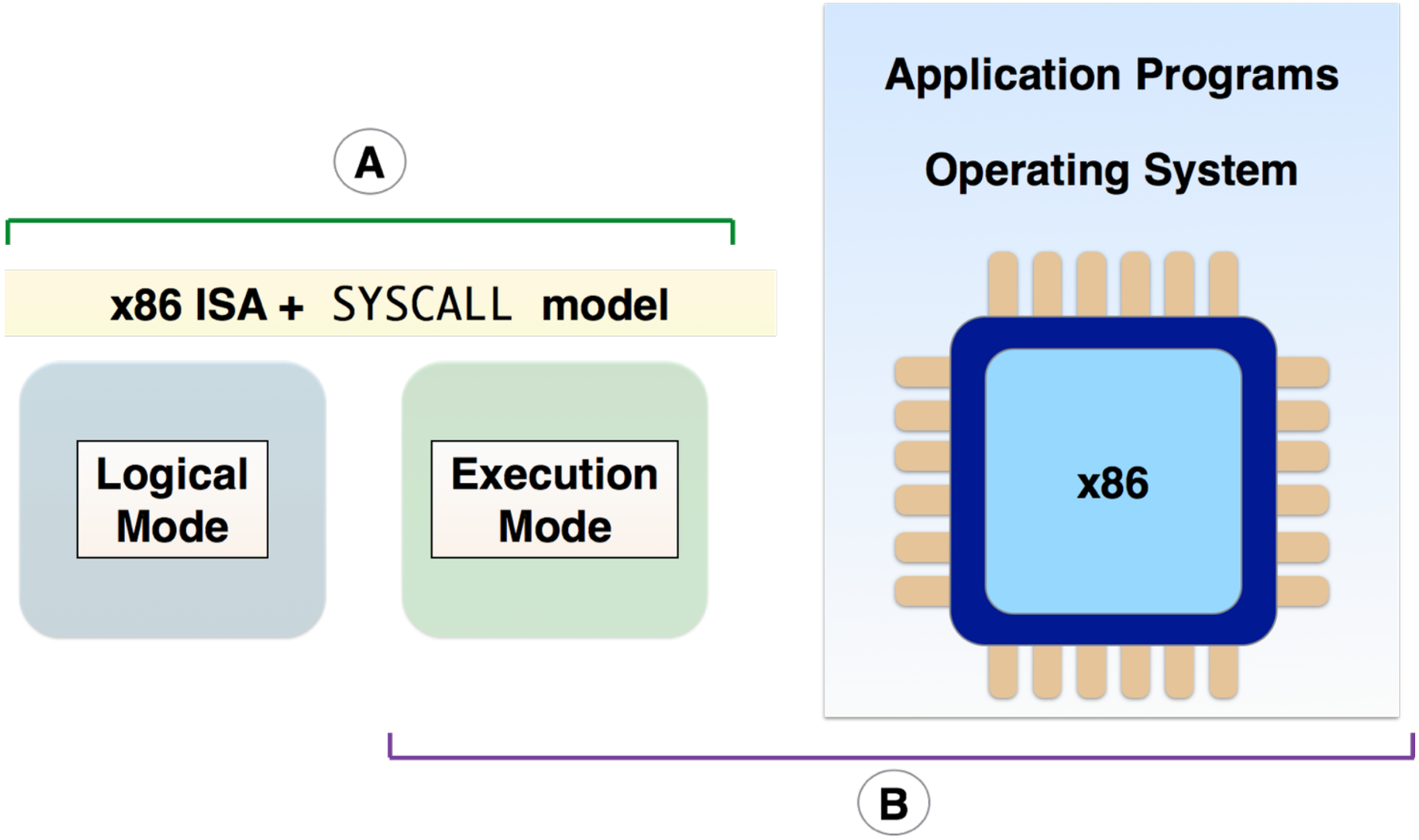
- ***Shilpi Goel** and Warren A. Hunt, Jr.*
  Automated Code Proofs on a Formal Model of the x86
  In Verified Software: Theories, Tools, Experiments (VSTTE), 2013

- ***Shilpi Goel**, Warren A. Hunt, Jr., Matt Kaufmann, and Soumava Ghosh.*
  Simulation and Formal Verification of x86 Machine-Code Programs That Make System Calls
  In Formal Methods in Computer-Aided Design (FMCAD), 2014

Thanks!

Extra Slides

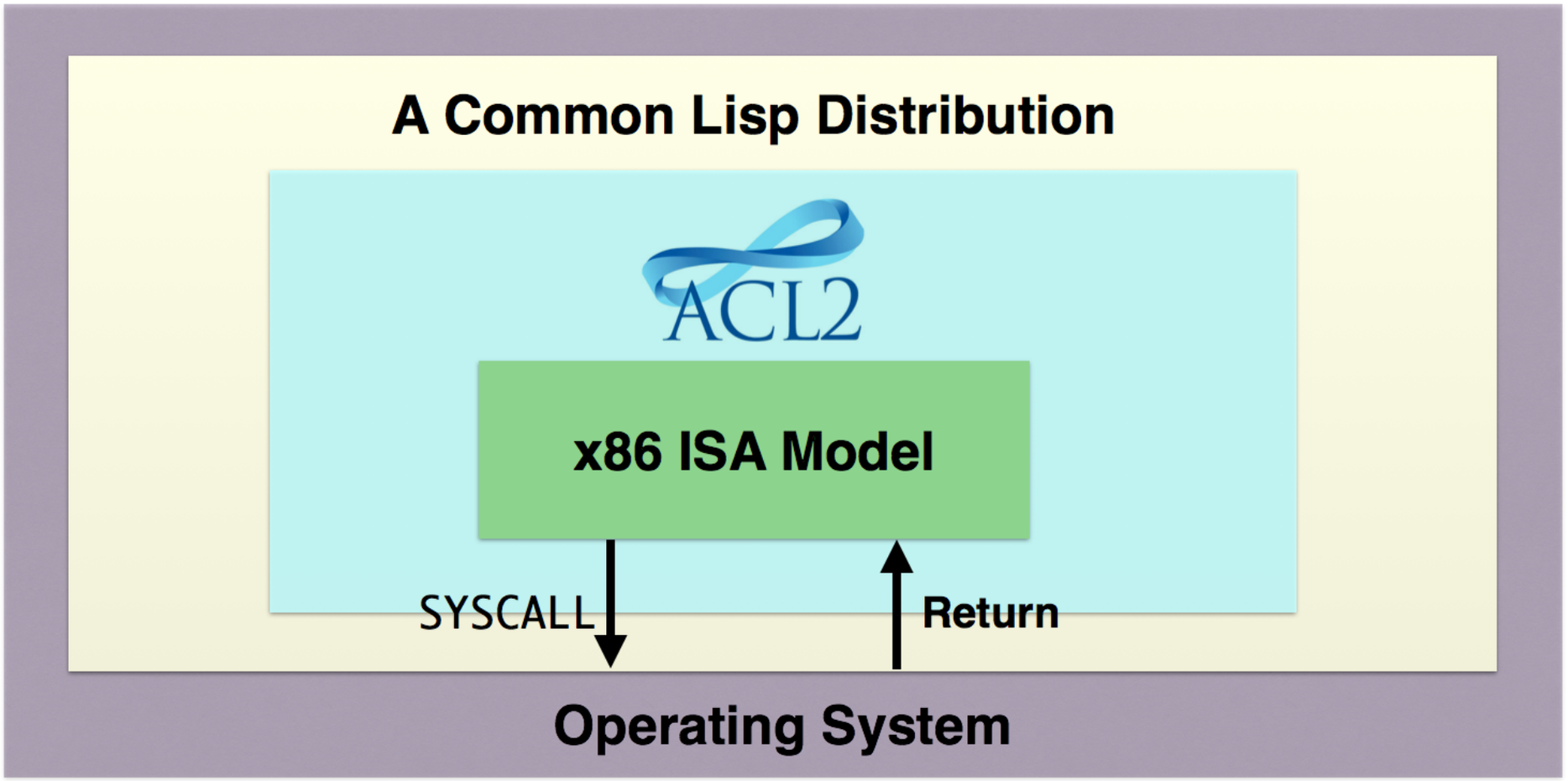
Programmer-level Mode: Model Validation



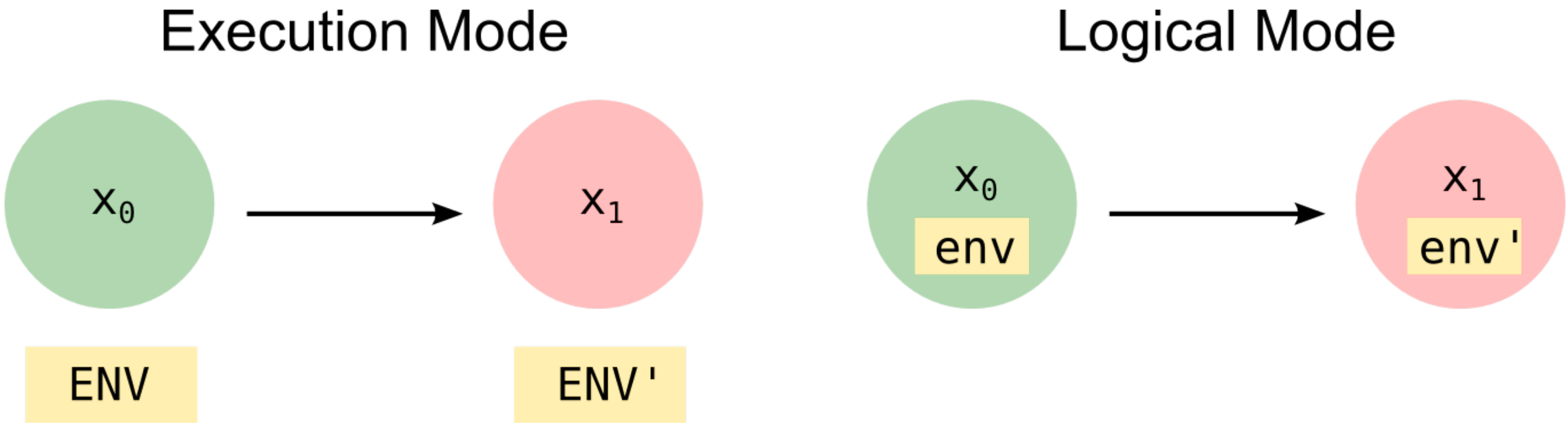
Task A: Validate the logical mode against the execution mode

Task B: Validate the execution mode against the processor + system call service provided by the OS

Programmer-level Mode: Execution Mode



Programmer-level Mode: Execution and Reasoning



Verification Landscape

Verification Tools:

limited analysis capabilities

state explosion problem

high degree of manual effort

can be applied to large systems

Static & dynamic analyzers

Model checkers

SAT & SMT solvers

Interactive theorem provers coupled with automatic tools

Interactive theorem provers

Automatic

Interactive