Formal Verification of Application and System Programs Based on a Validated x86 ISA Model

Ph.D. Final Defense

Shilpi Goel
shigoel@cs.utexas.edu
Department of Computer Science
The University of Texas at Austin

Software and Reliability

Can we rely on our software systems?

Recent example of a serious bug:

CVE-2016-5195 or "Dirty COW"

- Privilege escalation vulnerability in Linux
- E.g.: allowed a user to write to files intended to be read only
- Copy-on-Write (COW) breakage of private read-only memory mappings
- Existed since around v2.6.22 (**2007**) and was **fixed on Oct 18, 2016**

Tools for Formal Software Verification

How do we increase software reliability?

Point Tools

- Low overhead
- Limited scope

Restrictive

General-Purpose Tools

- High overhead
- Unrealistic models

Misleading

Tools for Formal Software Verification

How do we increase software reliability?

Point Tools

- Low overhead
- Limited scope

Restrictive

General-Purpose Tools

Lower overhead

Accurate models

Reliable

Tools for Formal Software Verification

How do we increase software reliability?

Point Tools

- Low overhead
- Limited scope

Restrictive

General-Purpose Tools

Lower overhead

Accurate models

Reliable

- This research:

  General-purpose tool for formal software verification based on an accurate model of the x86 ISA

- Make formal verification of machine code a ***practical*** choice

Why x86 Machine-Code Verification?

- ***Why not high-level code verification?***
  - × Sometimes, high-level code is unavailable (e.g., malware)
  - × 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

- ***Why x86?***
  - ✓ x86 is in widespread use

Overview

Goal

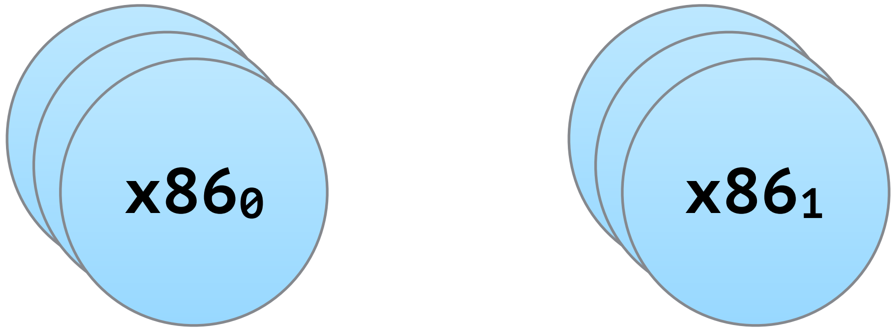
Specify and *verify* properties of x86 application and system programs

Overview

Goal

Specify and *verify* properties of x86 application and system programs

specify:
in terms of states of
computation



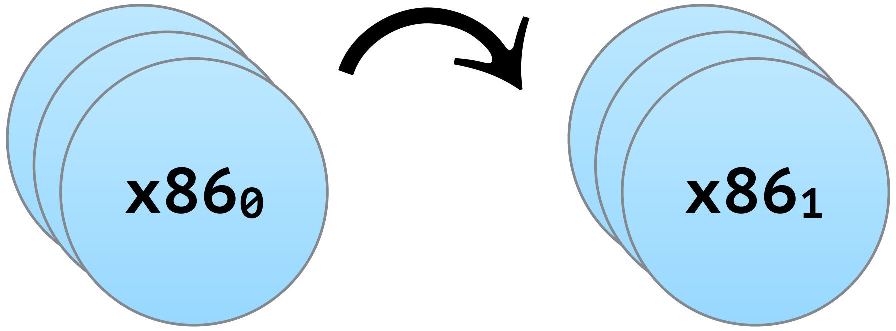
Overview

Goal

Specify and *verify* properties of x86 application and system programs

specify: in terms of states of computation

verify: reason about symbolic executions



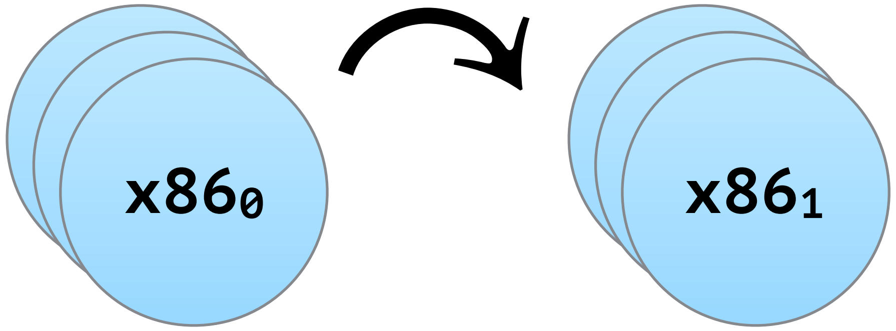
Overview

Goal

Specify and *verify* properties of x86 application and system programs

specify:
in terms of states of
computation

verify:
reason about symbolic
executions



Approach

1. Build a **formal, executable model of the x86 ISA** using ACL2
2. Develop a **machine-code analysis framework** based on this model that supports reasoning about
   (a) application programs, and (b) system programs
3. **Employ this framework** to verify
   (a) application programs, and (b) system programs

Review

My Ph.D. proposal described:

1. x86 ISA model
2. (a) Libraries to reason about application programs
3. (a) Verification of two application programs

Review

My Ph.D. proposal described:

1. x86 ISA model
2. (a) Libraries to reason about application programs
3. (a) Verification of two application programs

Focus of this talk:

1. New features of the x86 ISA model
2. (b) Libraries to reason about system programs
3. (b) Verification of a system program — Zero-Copy

Review

My Ph.D. proposal described:

1. x86 ISA model
2. (a) Libraries to reason about application programs
3. (a) Verification of two application programs

Focus of this talk: **[Diss. Ch. 7]**

1. New features of the x86 ISA model
2. (b) Libraries to reason about system programs **[Diss. Ch. 10]**
3. (b) Verification of a system program — Zero-Copy **[Diss. Ch. 12]**

Review

My Ph.D. proposal described:

1. x86 ISA model
2. (a) Libraries to reason about application programs
3. (a) Verification of two application programs

Focus of this talk: **[Diss. Ch. 7]**

1. New features of the x86 ISA model
2. (b) Libraries to reason about system programs **[Diss. Ch. 10]**
3. (b) Verification of a system program — Zero-Copy **[Diss. Ch. 12]**

	STATUS: THEN	STATUS: NOW
x86 ISA Model	220 Opcodes	413 Opcodes
Lemma Libraries	Support only for application programs	Support added for system programs
Case Studies	Application programs	Added system program (Zero-Copy)
Documentation	Largely developer-focused topics	Added user-focused topics, including a guide to debug failed proofs

Our Framework: Design Goals

Accuracy

Reliable program analysis

Our Framework: Design Goals

Accuracy
Reliable program analysis

Execution Efficiency
Aid in co-simulations and testing

Our Framework: Design Goals

Accuracy
Reliable program analysis

Execution Efficiency
Aid in co-simulations and testing

Reasoning Efficiency
Reduce user effort, e.g., support failed proofs’ debugging

Our Framework: Design Goals

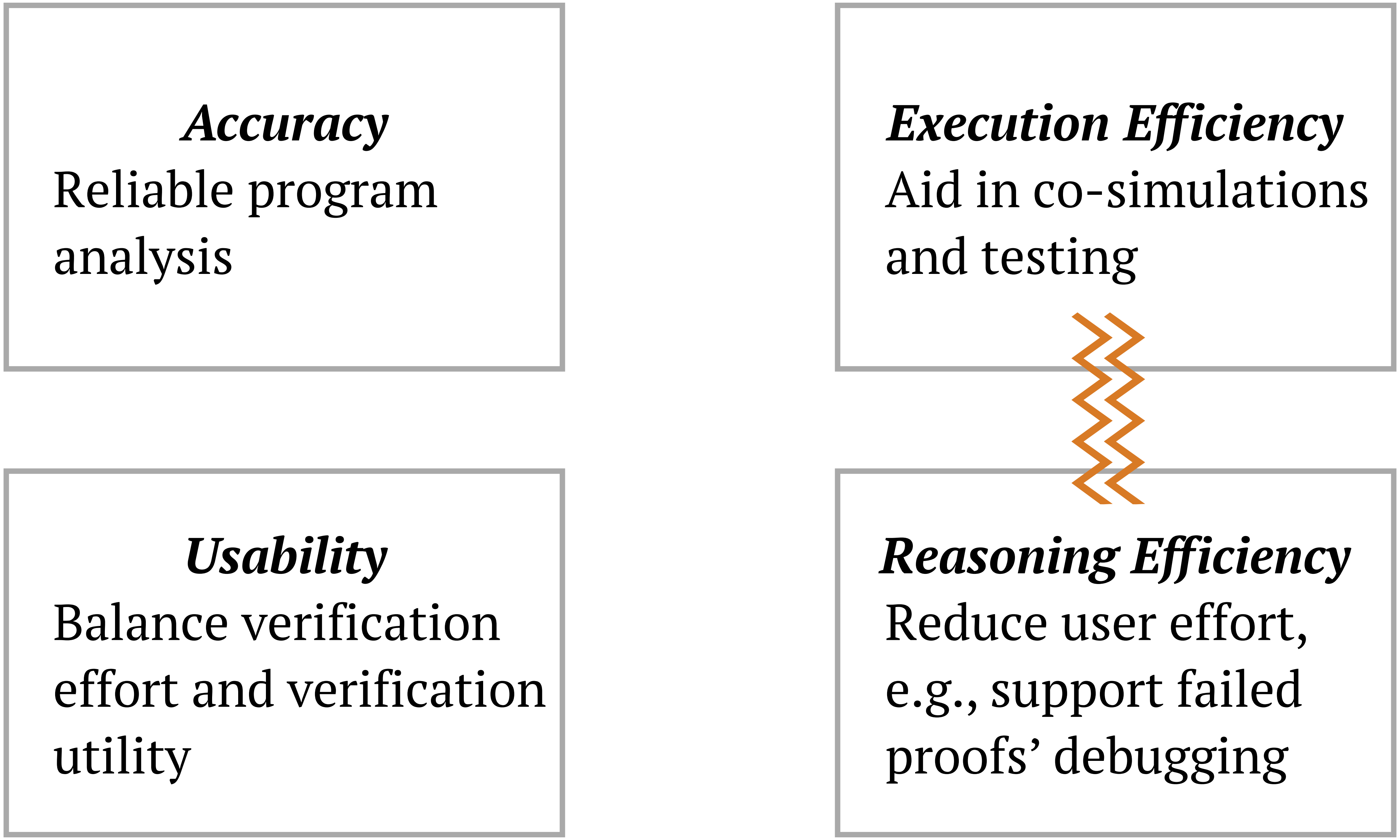
Accuracy
Reliable program analysis

Execution Efficiency
Aid in co-simulations and testing

Usability
Balance verification effort and verification utility

Reasoning Efficiency
Reduce user effort, e.g., support failed proofs' debugging

Our Framework: Design Goals



Accuracy
Reliable program analysis

Execution Efficiency
Aid in co-simulations and testing

Usability
Balance verification effort and verification utility

Reasoning Efficiency
Reduce user effort, e.g., support failed proofs' debugging

Our Framework: Design Goals

Accuracy
Reliable program analysis

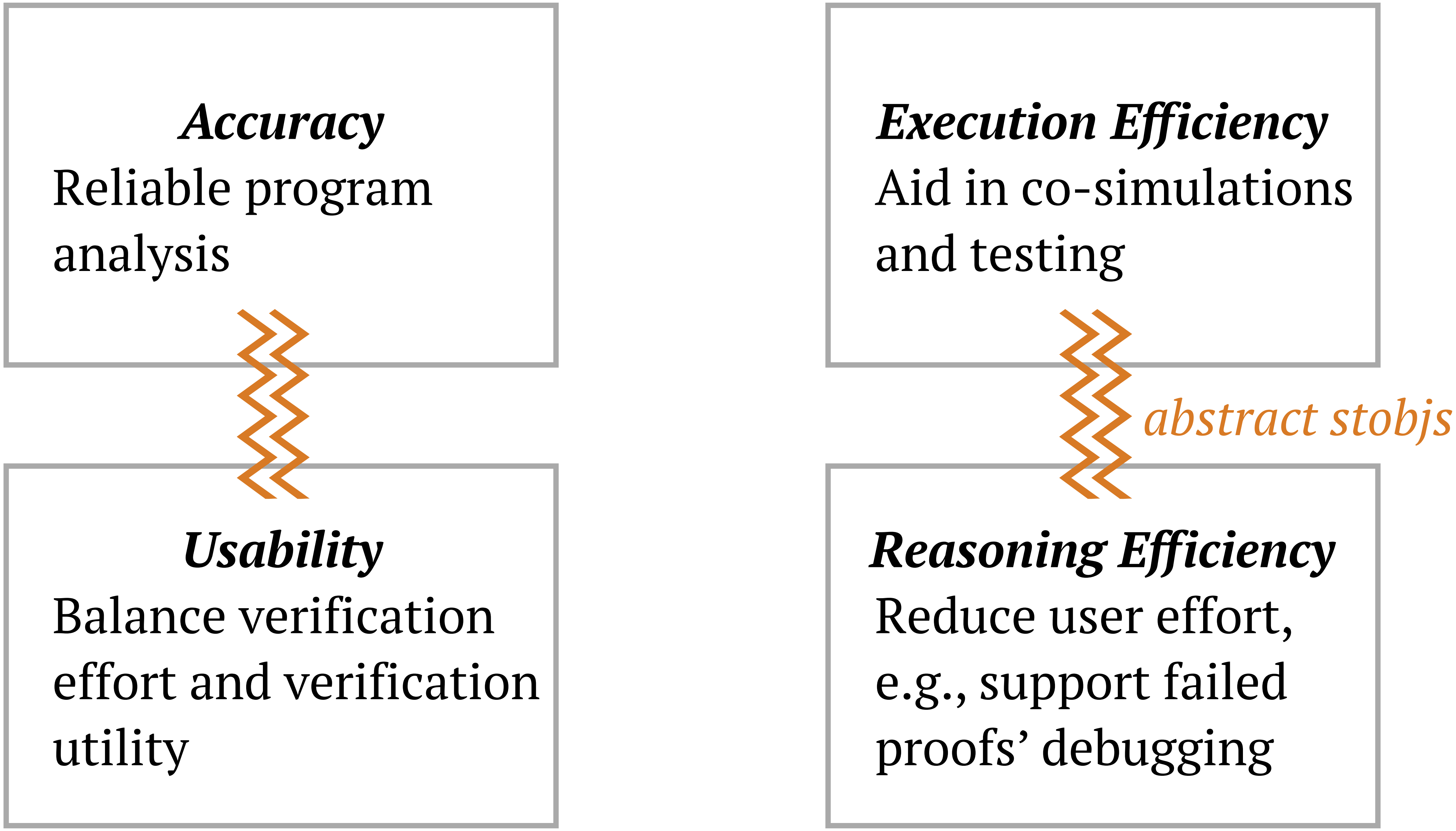
Execution Efficiency
Aid in co-simulations and testing

abstract stobjs

Usability
Balance verification effort and verification utility

Reasoning Efficiency
Reduce user effort, e.g., support failed proofs' debugging

Our Framework: Design Goals



Accuracy

Reliable program analysis

Execution Efficiency

Aid in co-simulations and testing

abstract stobjs

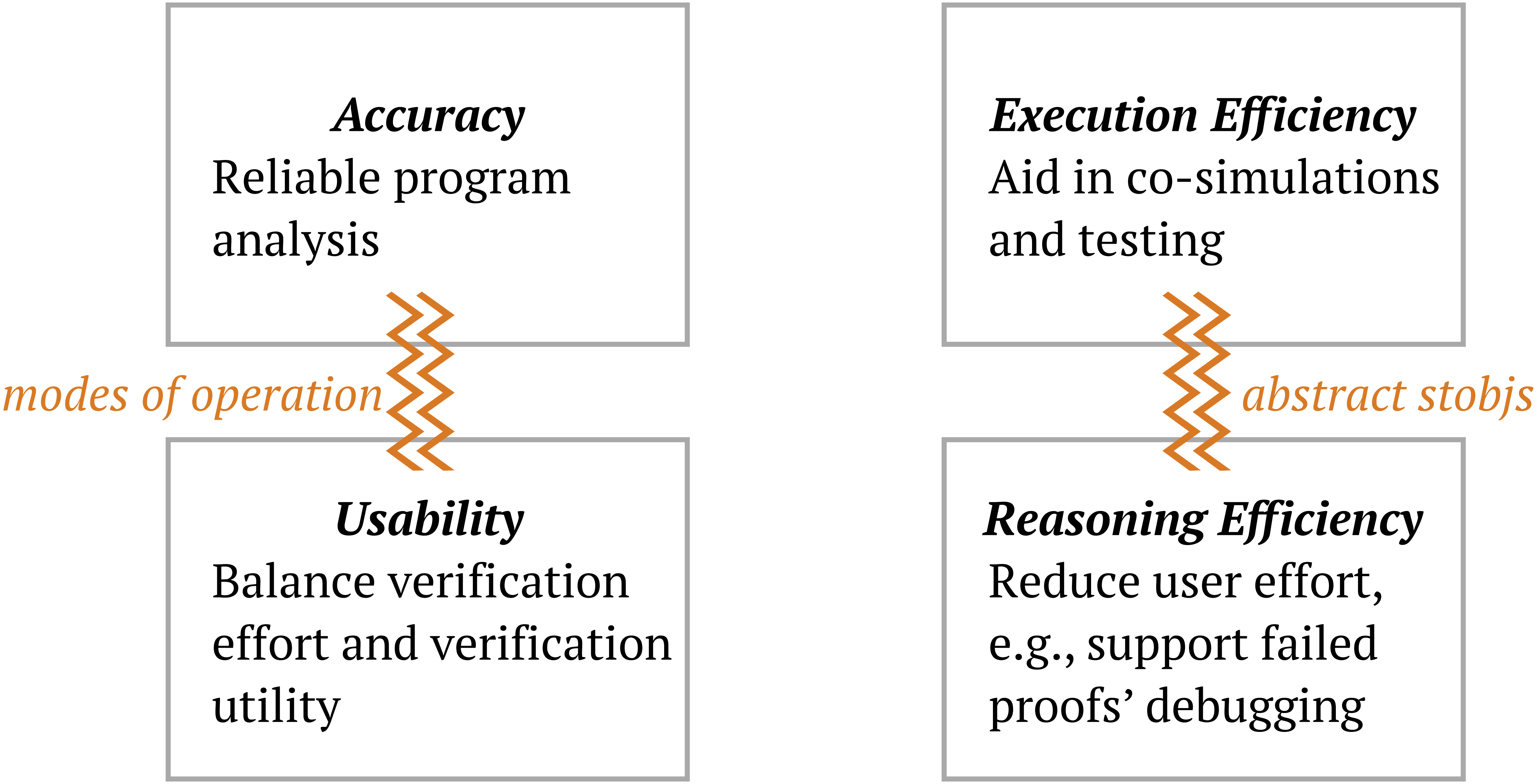
Usability

Balance verification effort and verification utility

Reasoning Efficiency

Reduce user effort, e.g., support failed proofs' debugging

Our Framework: Design Goals



Accuracy
Reliable program analysis

Usability
Balance verification effort and verification utility

modes of operation

Execution Efficiency
Aid in co-simulations and testing

Reasoning Efficiency
Reduce user effort, e.g., support failed proofs' debugging

abstract stobjs

Outline

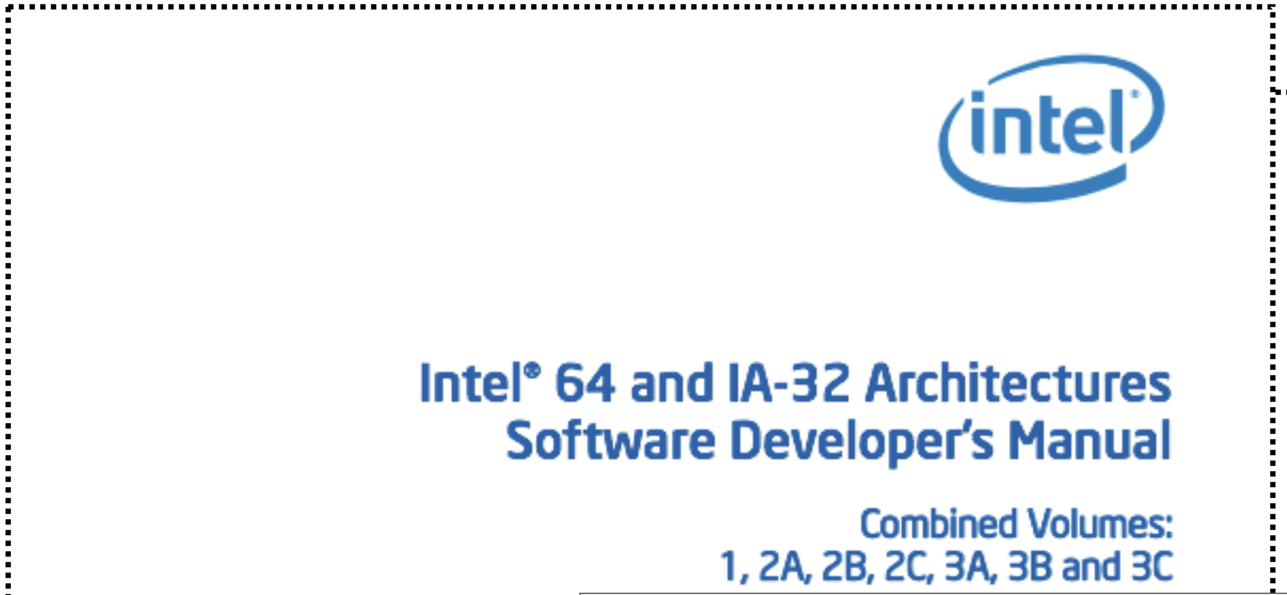
Overview

1. **Formal Model of the x86 ISA**
2. Lemma Libraries for Machine-Code Verification
3. Case Studies

Concluding Remarks and Future Work

Obtaining the x86 ISA Specification

~3400 pages



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

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

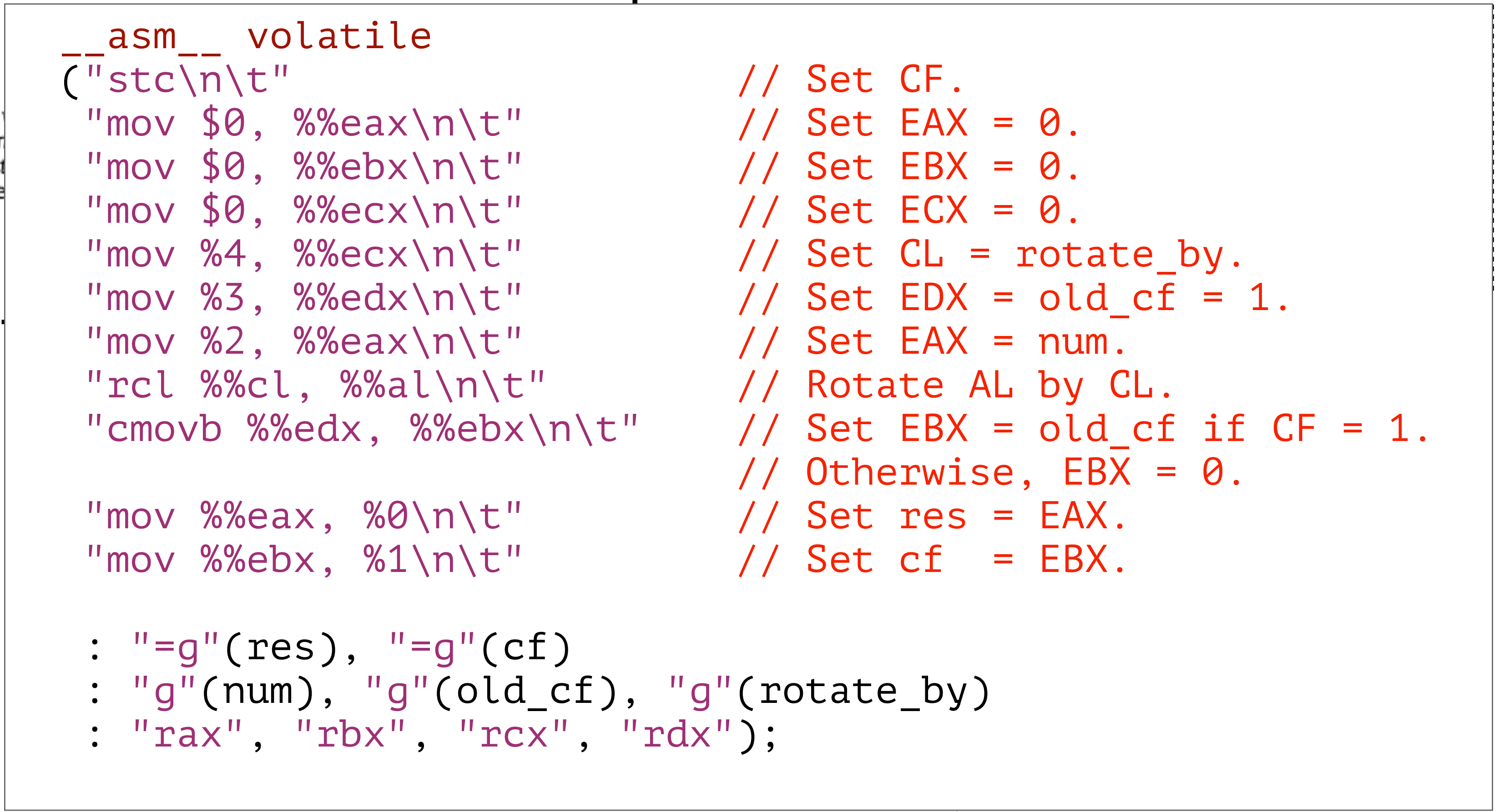
NOTE: This document contains all seven
Developer's Manual: *Basic Architecture, In*
Z, Instruction Set Reference, and the Syst
volumes when evaluating your design nee

~3000 pages

AMD

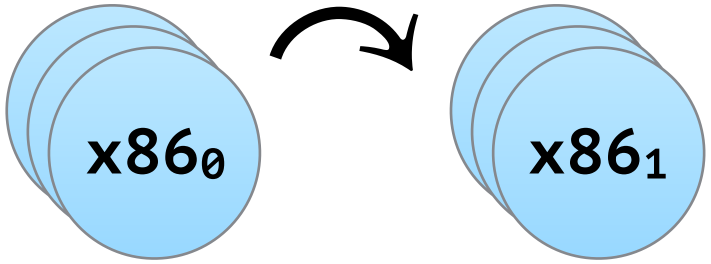
AMD64 Technology

AMD64 Architecture
Programmer's Manual



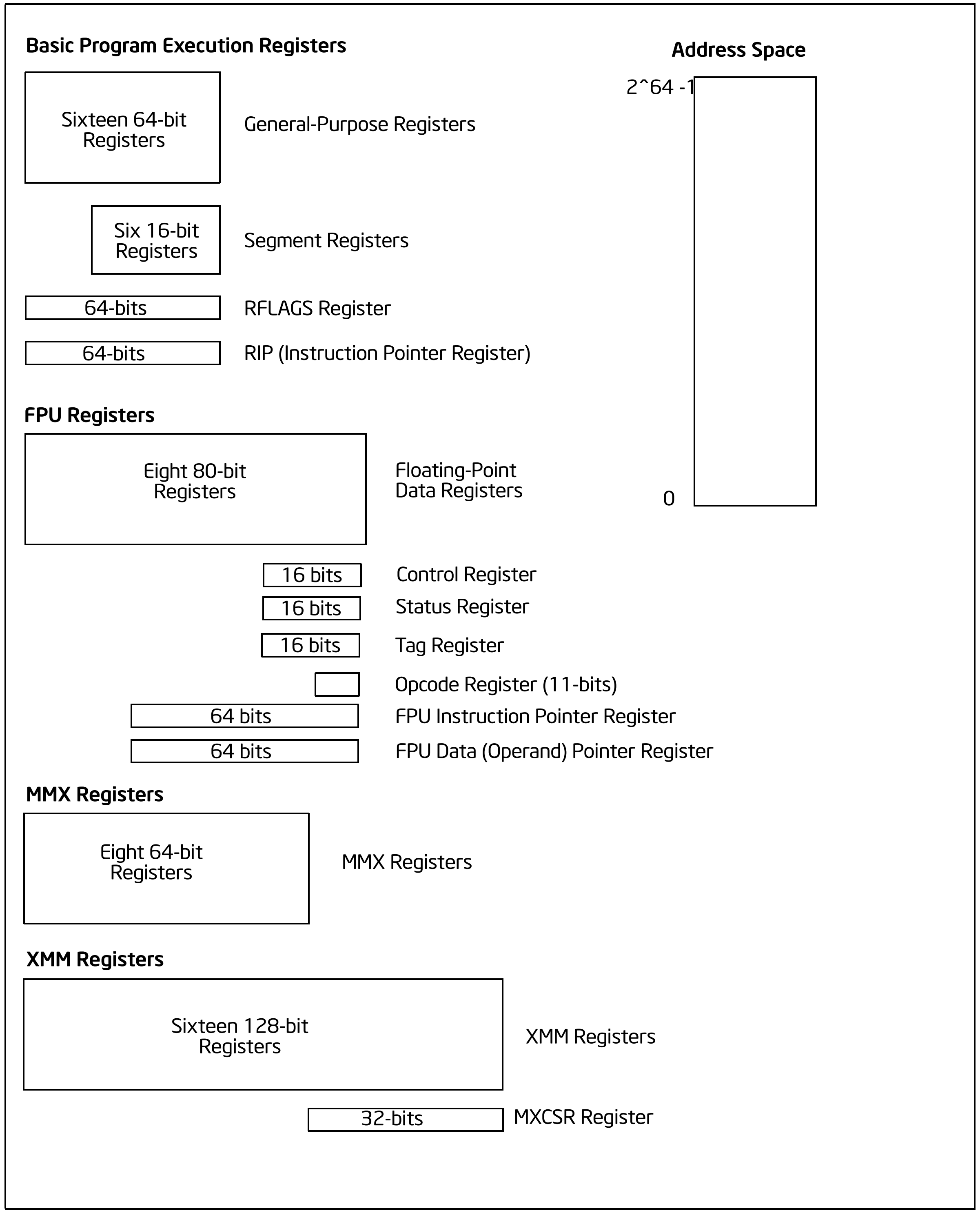
```
__asm__ volatile
("stc\n\t"                    // Set CF.
 "mov $0, %%eax\n\t"          // Set EAX = 0.
 "mov $0, %%ebx\n\t"          // Set EBX = 0.
 "mov $0, %%ecx\n\t"          // Set ECX = 0.
 "mov %4, %%ecx\n\t"          // Set CL = rotate_by.
 "mov %3, %%edx\n\t"          // Set EDX = old_cf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



x86 State

Focus: Intel's 64-bit mode



Basic Program Execution Registers

Sixteen 64-bit Registers — General-Purpose Registers

Six 16-bit Registers — Segment Registers

64-bits — RFLAGS Register

64-bits — RIP (Instruction Pointer Register)

Address Space

2^64 -1

0

FPU Registers

Eight 80-bit Registers — Floating-Point Data Registers

16 bits — Control Register

16 bits — Status Register

16 bits — Tag Register

Opcode Register (11-bits)

64 bits — FPU Instruction Pointer Register

64 bits — FPU Data (Operand) Pointer Register

MMX Registers

Eight 64-bit Registers — MMX Registers

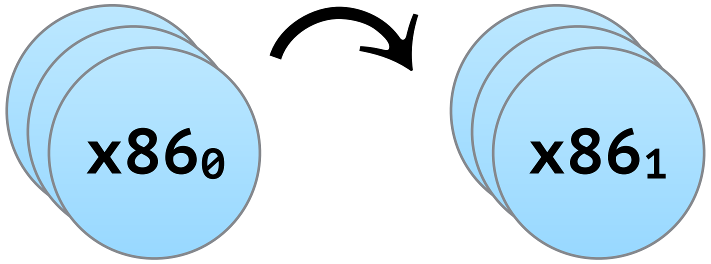
XMM Registers

Sixteen 128-bit Registers — XMM Registers

32-bits — MXCSR Register

Figure 3-2. 64-Bit Mode Execution Environment

Source: Intel Manuals



x86 State

Focus: Intel’s 64-bit mode

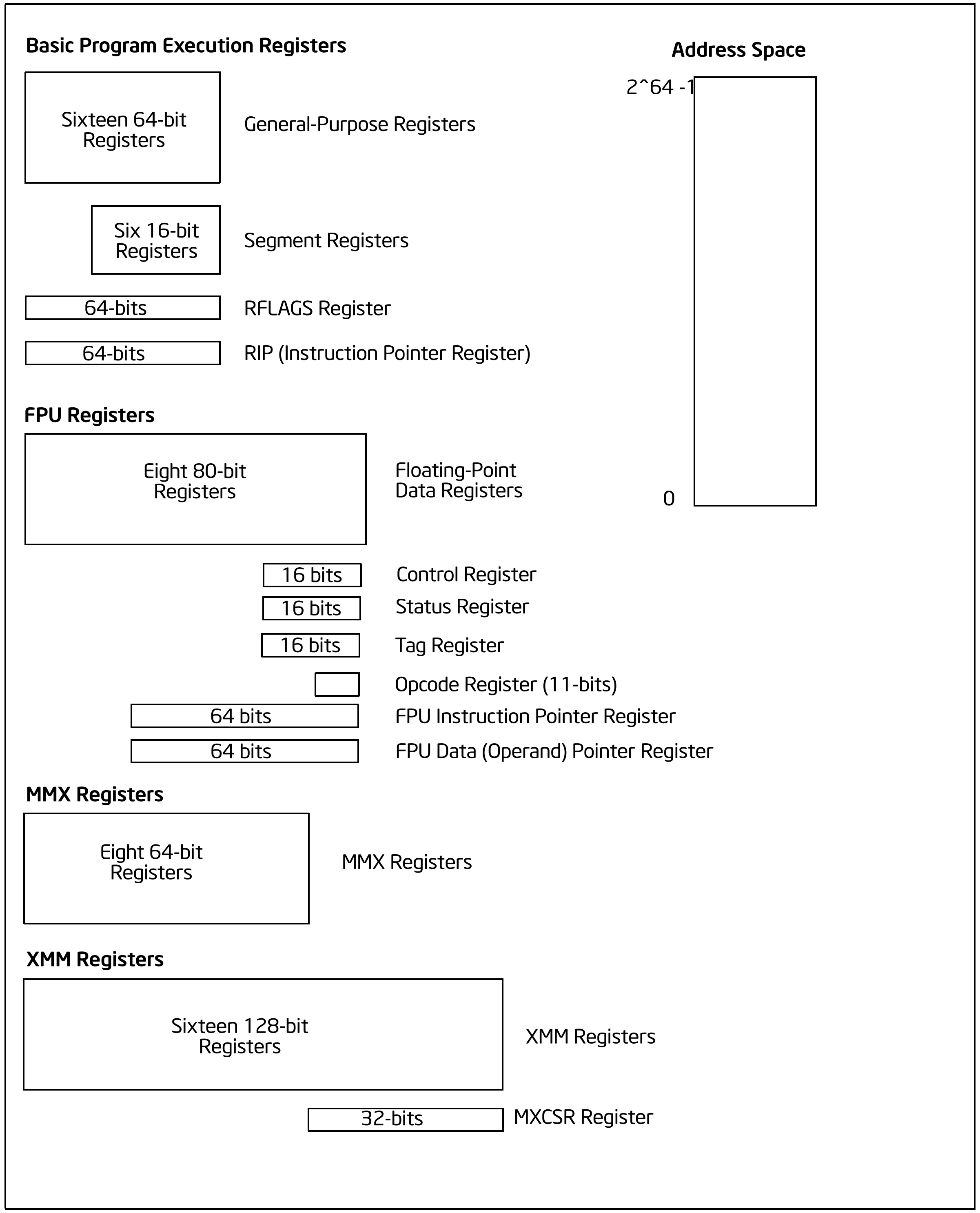


Figure 3-2. 64-Bit Mode Execution Environment

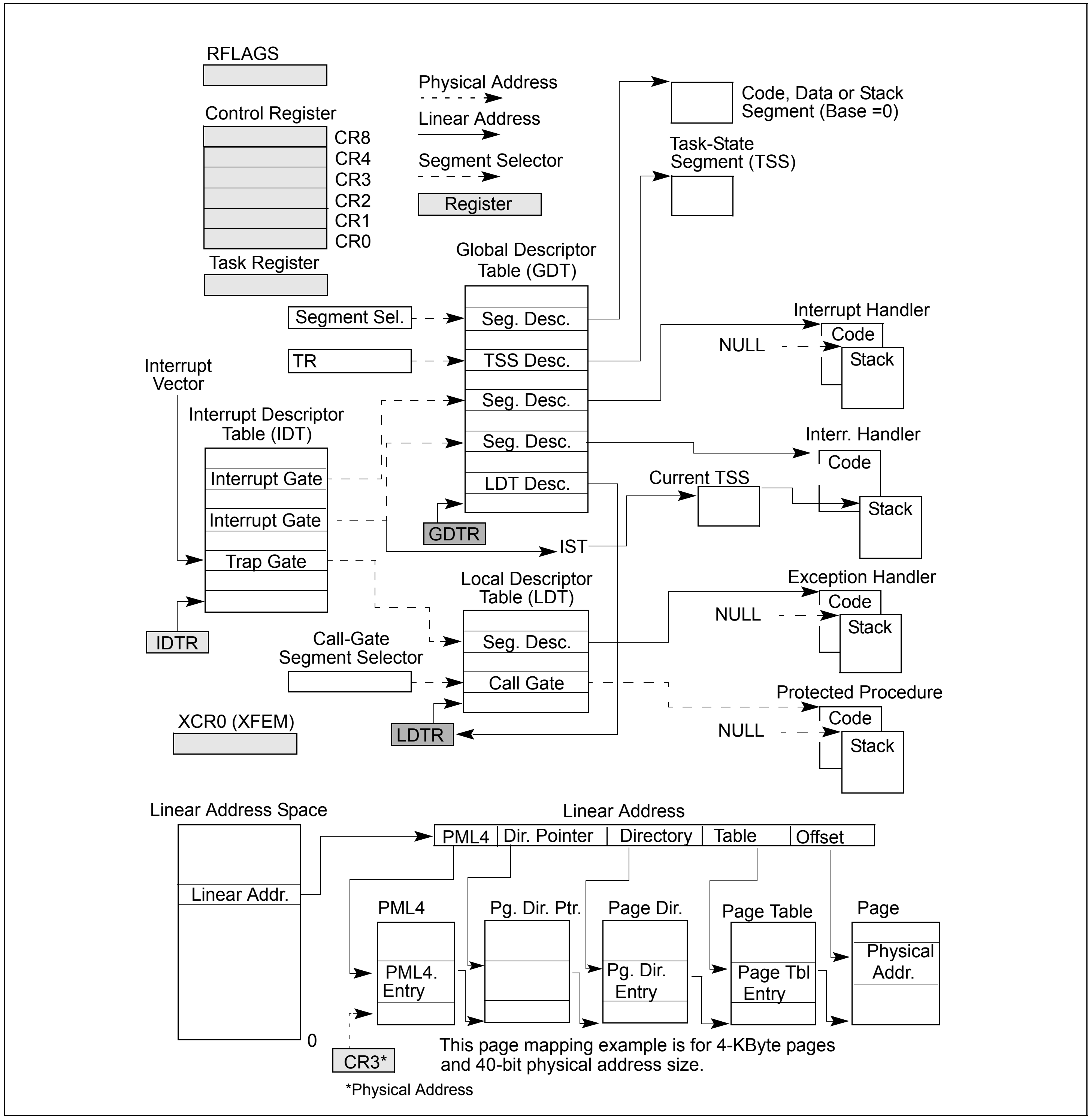


Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode

Source: Intel Manuals

Modes of Operation of the x86 ISA Model

User-level Mode

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

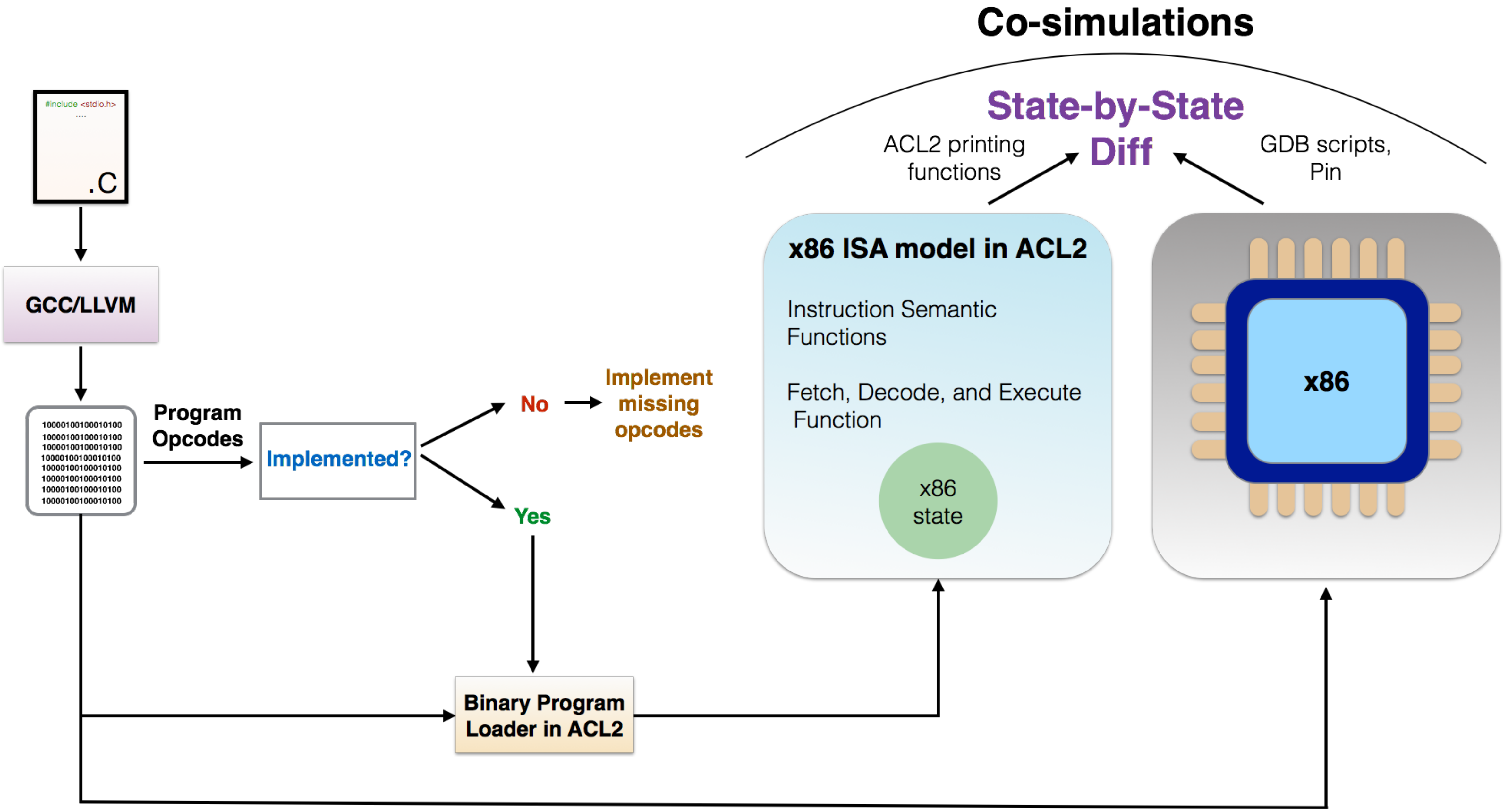
System-level Mode

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

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



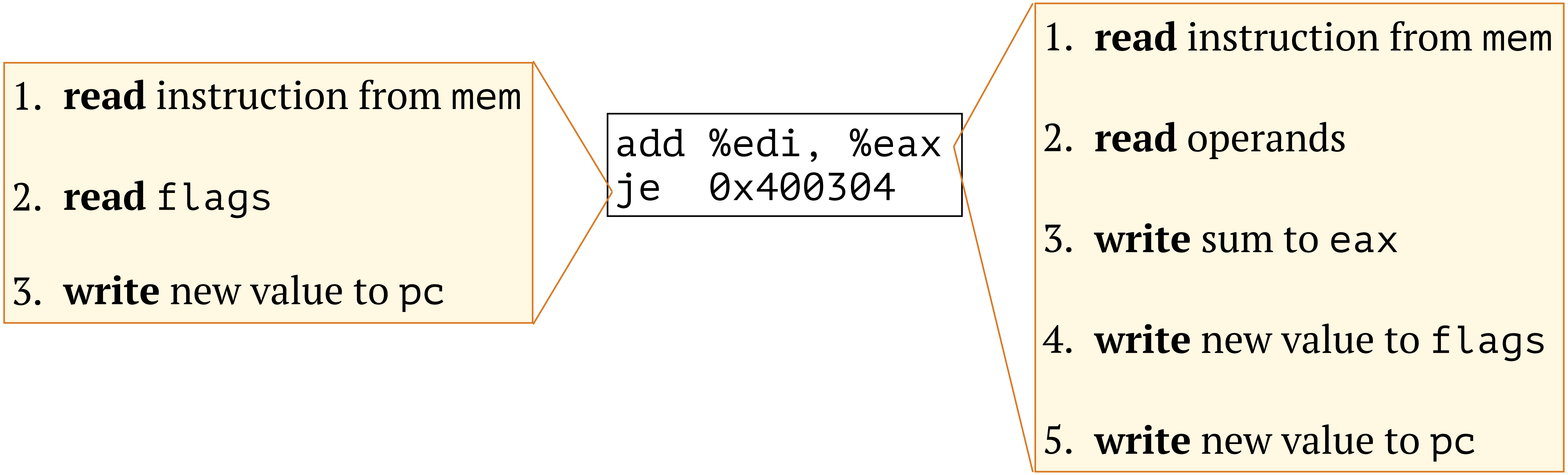
Outline

Overview

1. Formal Model of the x86 ISA
2. **Lemma Libraries for Machine-Code Verification**
3. Case Studies

Concluding Remarks and Future Work

Supporting Symbolic Execution



1. **read** instruction from `mem`
2. **read** `flags`
3. **write** new value to pc

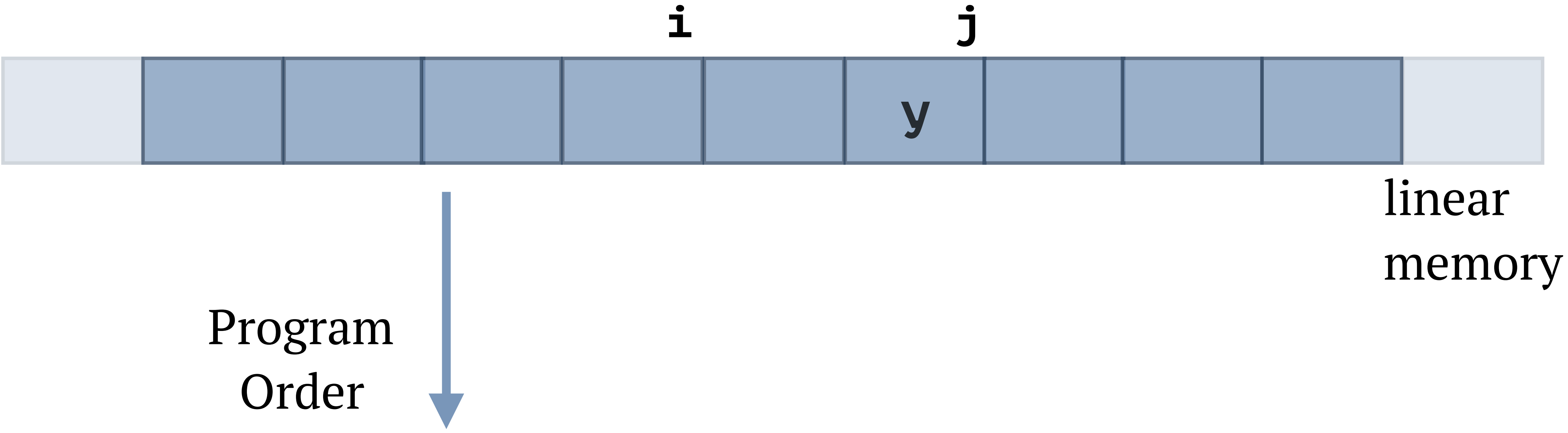
```
add %edi, %eax
je  0x400304
```

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

Rules (theorems) describing interactions between these reads and writes to the x86 state enable *symbolic execution* of programs.

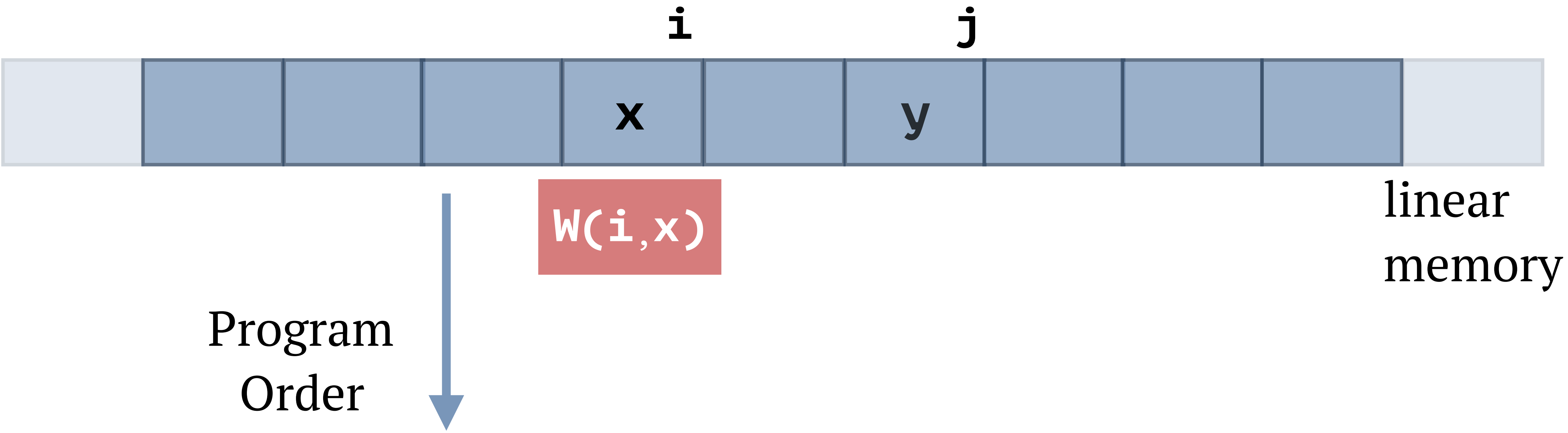
Linear Memory Non-Interference Theorem

user-level mode



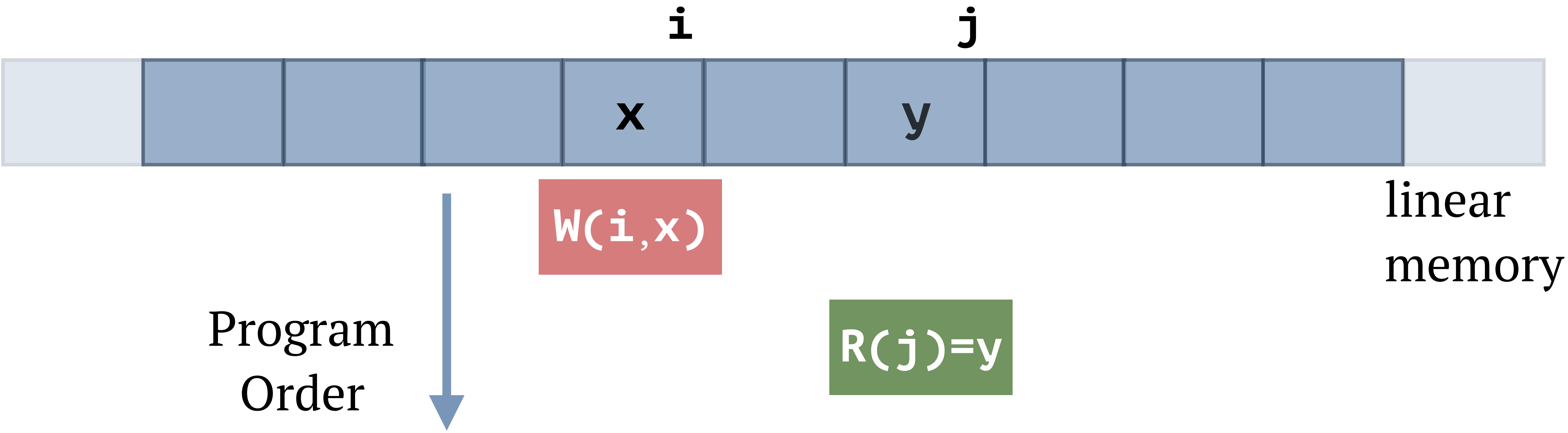
Linear Memory Non-Interference Theorem

user-level mode



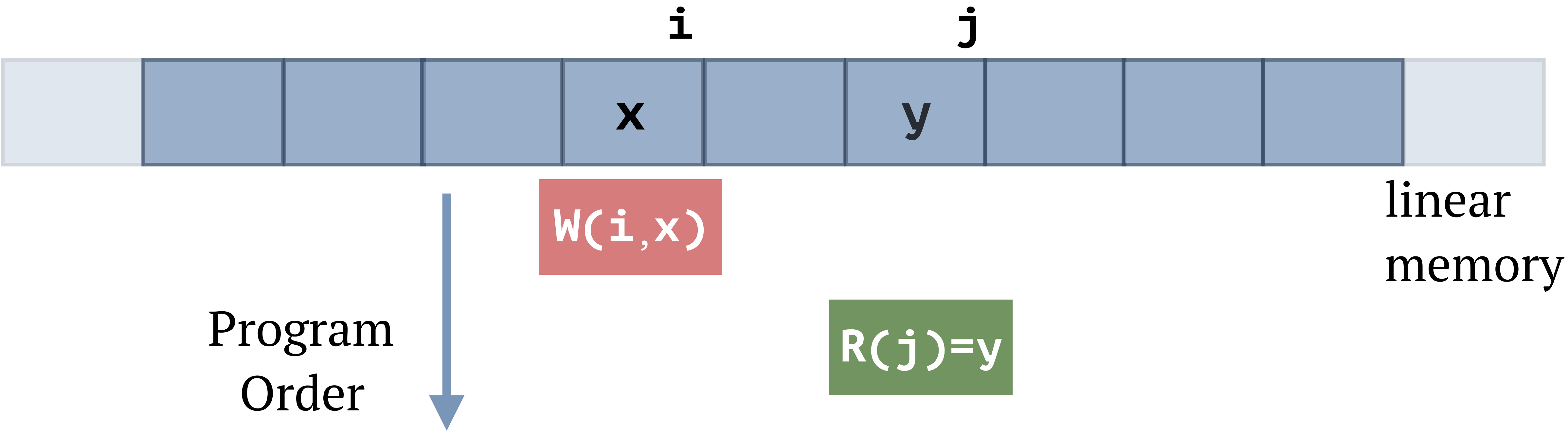
Linear Memory Non-Interference Theorem

user-level mode



Linear Memory Non-Interference Theorem

user-level mode

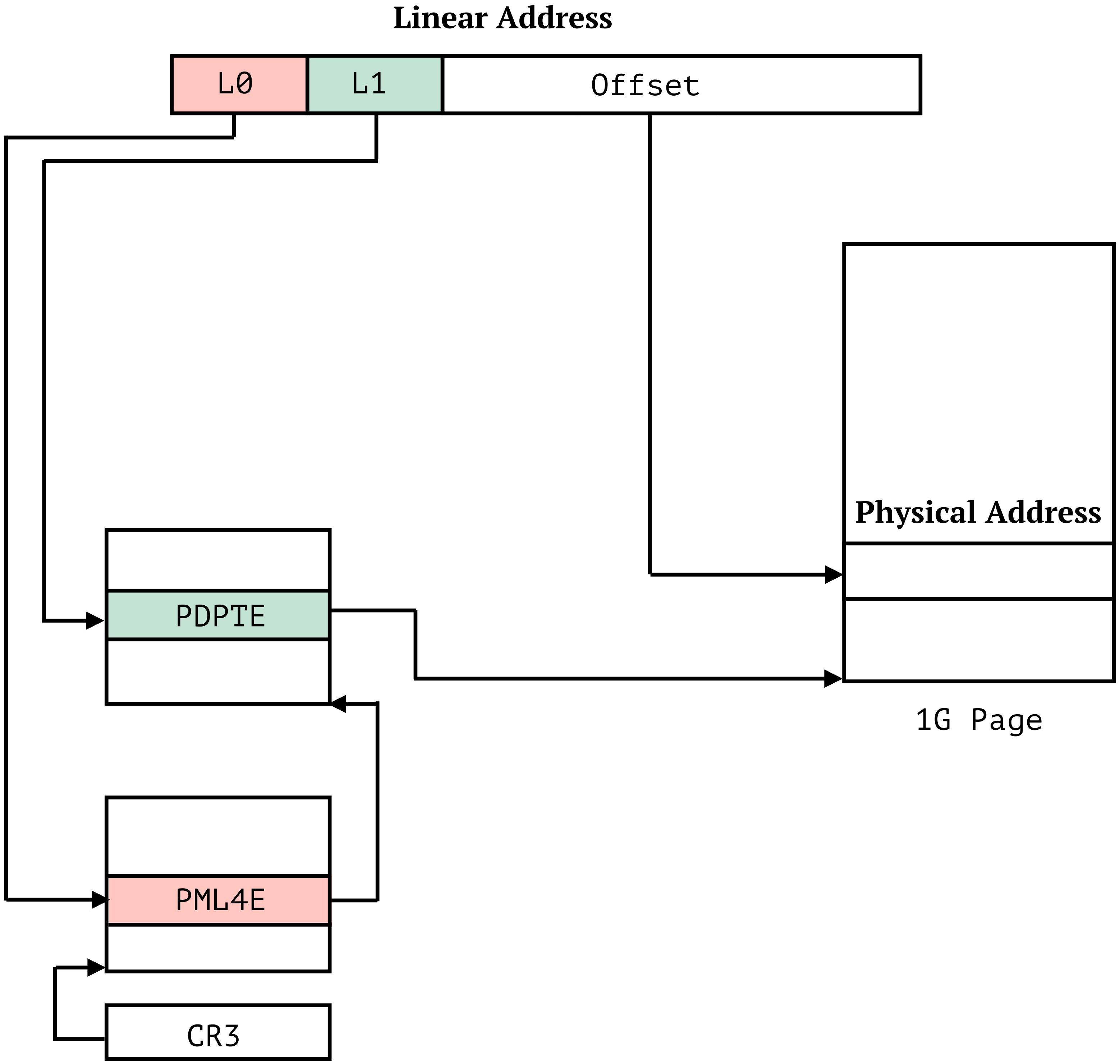


```
(defthm linear-mem-non-interference-user-level-mode
  (implies
   (and (disjoint-p las-1 las-2)
        (user-level-mode x86))
   (equal
    (read-mem las-1 r-x (write-mem las-2 bytes x86))
    (read-mem las-1 r-x x86))))
```

`las-1` `las-2` – lists of linear addresses

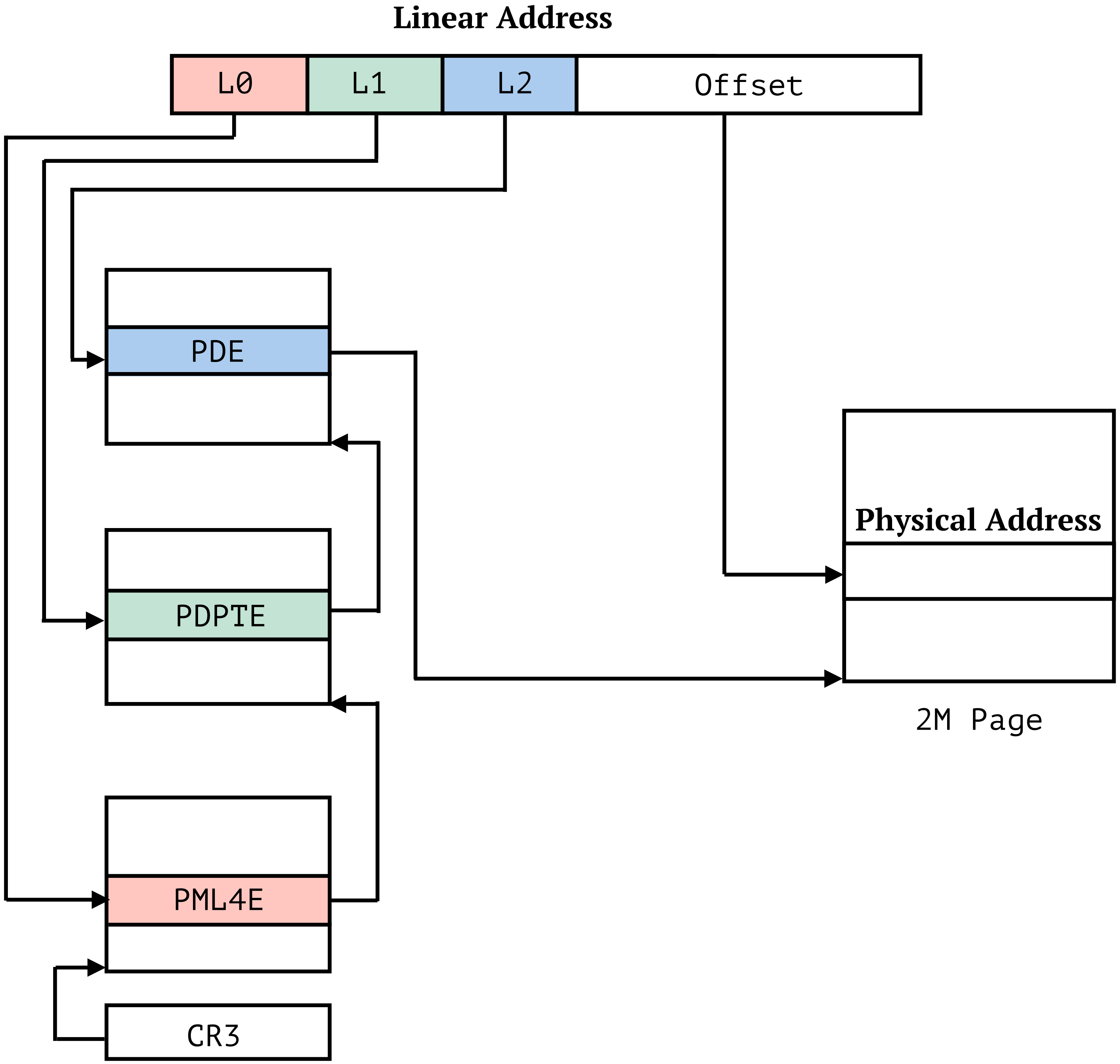
Reasoning about Paging is Complicated *#1*

1. Complicated data structures — hierarchical, with two to four levels of indirection, depending on the page configuration



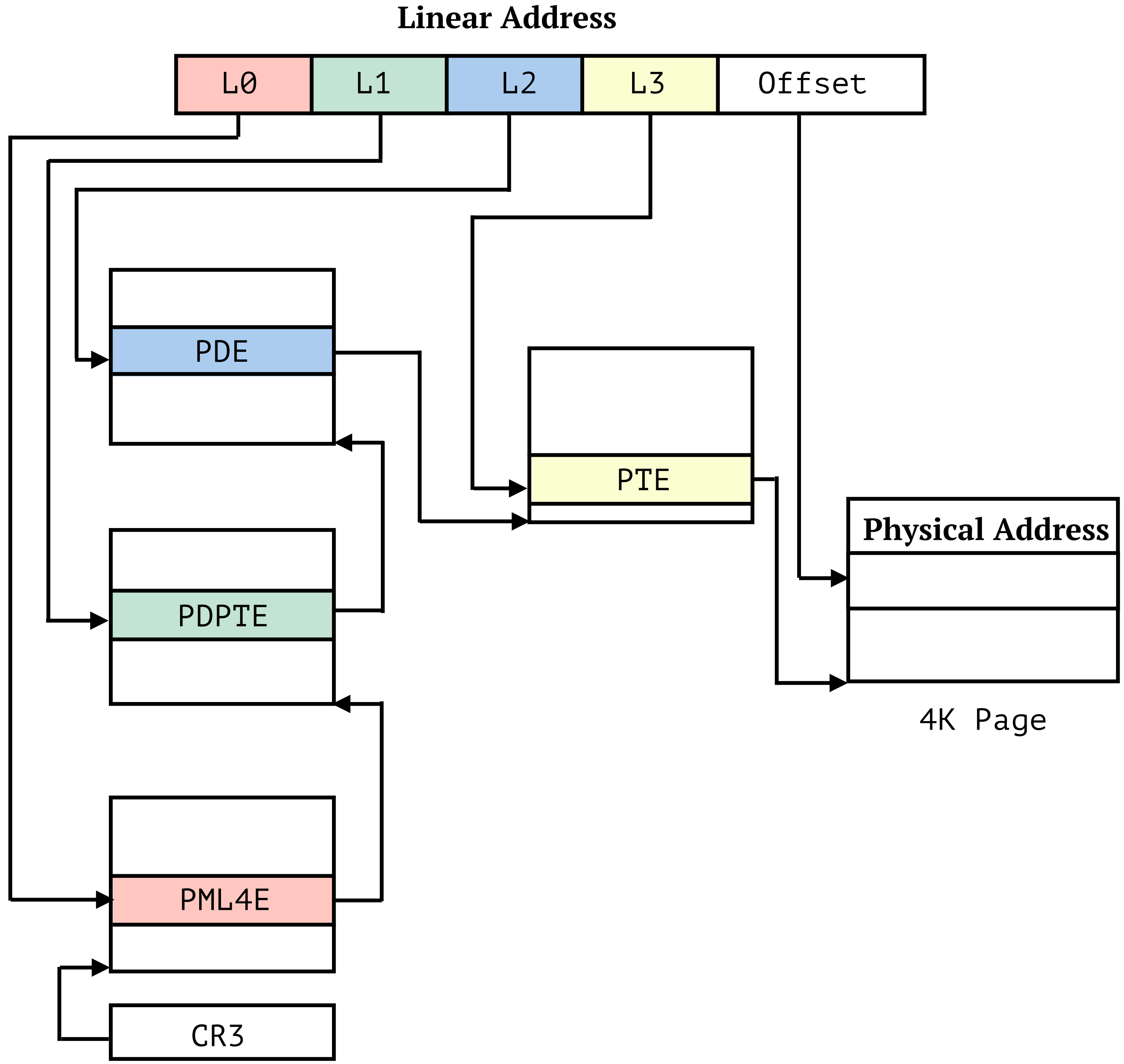
Reasoning about Paging is Complicated *#1*

1. Complicated data structures — hierarchical, with two to four levels of indirection, depending on the page configuration



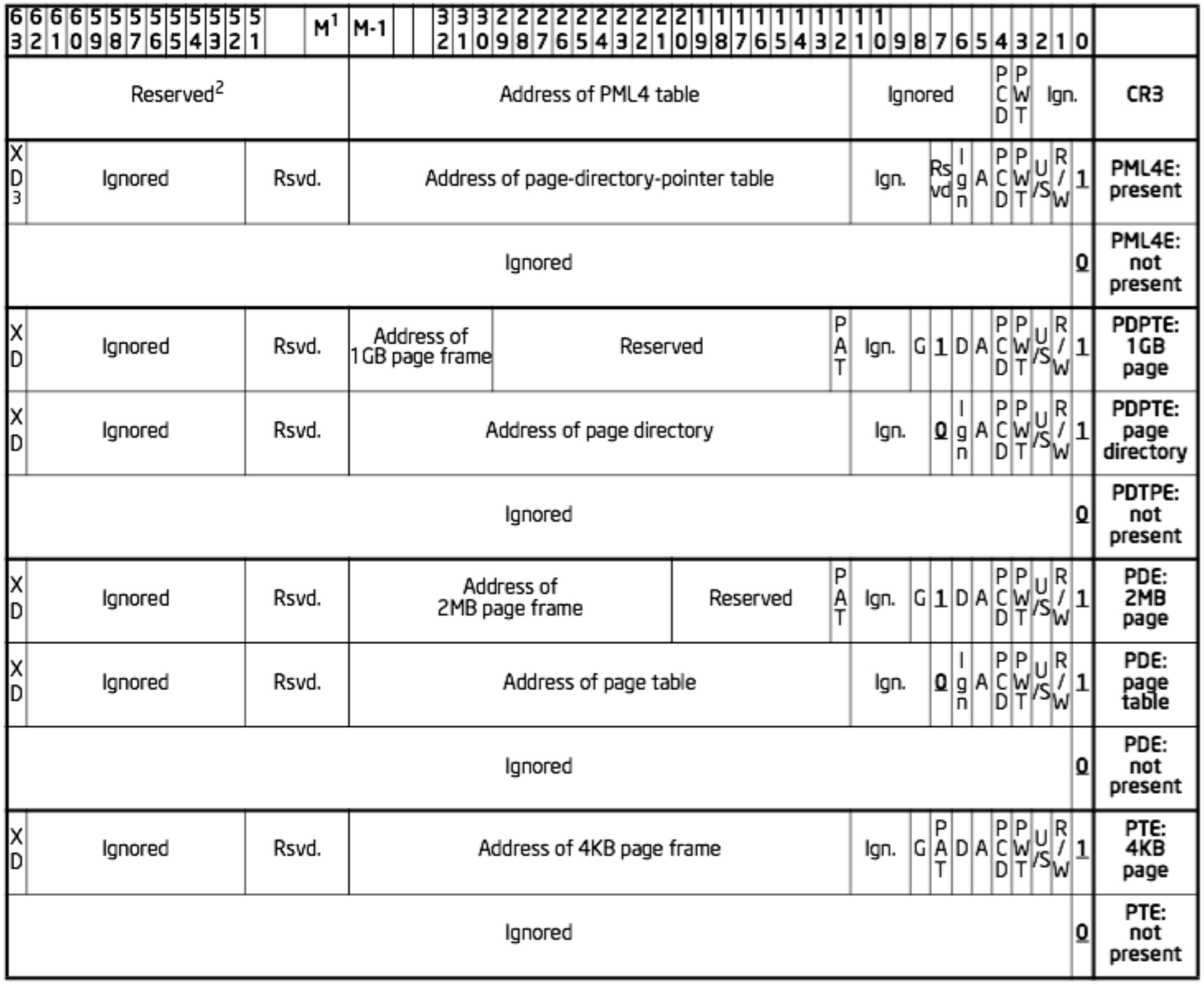
Reasoning about Paging is Complicated *#1*

1. Complicated data structures — hierarchical, with two to four levels of indirection, depending on the page configuration



Reasoning about Paging is Complicated *#1*

1. Complicated data structures — hierarchical, with two to four levels of indirection, depending on the page configuration



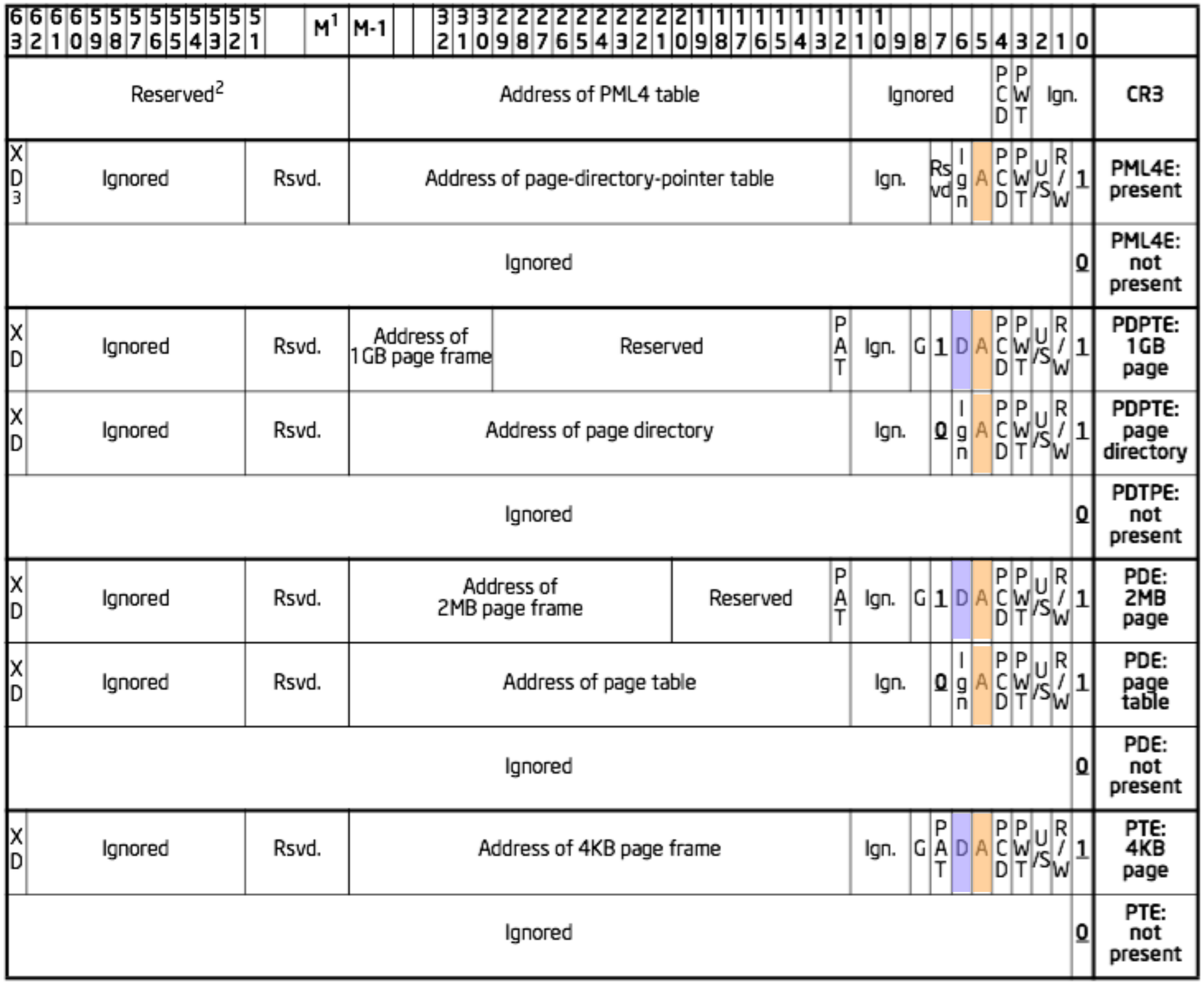
63	62 … 52	51 … M[1]	M-1 … 13	12	11 … 9	8	7	6	5	4	3	2	1	0	
Reserved[2]			Address of PML4 table		Ignored					PCD	PWT	Ign.			CR3
XD[3]	Ignored	Rsvd.	Address of page-directory-pointer table		Ign.		Rsvd	Ign	A	PCD	PWT	U/S	R/W	1	PML4E: present
Ignored														0	PML4E: not present
XD	Ignored	Rsvd.	Address of 1GB page frame Reserved	PAT	Ign.	G	1	D	A	PCD	PWT	U/S	R/W	1	PDPTE: 1GB page
XD	Ignored	Rsvd.	Address of page directory		Ign.		0	Ign	A	PCD	PWT	U/S	R/W	1	PDPTE: page directory
Ignored														0	PDTPE: not present
XD	Ignored	Rsvd.	Address of 2MB page frame Reserved	PAT	Ign.	G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 2MB page
XD	Ignored	Rsvd.	Address of page table		Ign.		0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored														0	PDE: not present
XD	Ignored	Rsvd.	Address of 4KB page frame		Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored														0	PTE: not present

Figure 4-11. Formats of CR3 and Paging-Structure Entries with IA-32e Paging

Source: Intel Manuals

Reasoning about Paging is Complicated *#1*

1. Complicated data structures — hierarchical, with two to four levels of indirection, depending on the page configuration



63	62 … 52	51 … M[1]	M-1 … 30	29 … 21	20 … 13	12	11 … 9	8	7	6	5	4	3	2	1	0	
Reserved[2]			Address of PML4 table				Ignored					PCD	PWT	Ign.			CR3
XD[3]	Ignored	Rsvd.	Address of page-directory-pointer table				Ign.		Rsvd	Ign	A	PCD	PWT	U/S	R/W	1	PML4E: present
Ignored																0	PML4E: not present
XD	Ignored	Rsvd.	Address of 1GB page frame	Reserved		PAT	Ign.	G	1	D	A	PCD	PWT	U/S	R/W	1	PDPTE: 1GB page
XD	Ignored	Rsvd.	Address of page directory				Ign.		0	Ign	A	PCD	PWT	U/S	R/W	1	PDPTE: page directory
Ignored																0	PDTPE: not present
XD	Ignored	Rsvd.	Address of 2MB page frame		Reserved	PAT	Ign.	G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 2MB page
XD	Ignored	Rsvd.	Address of page table				Ign.		0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																0	PDE: not present
XD	Ignored	Rsvd.	Address of 4KB page frame				Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																0	PTE: not present

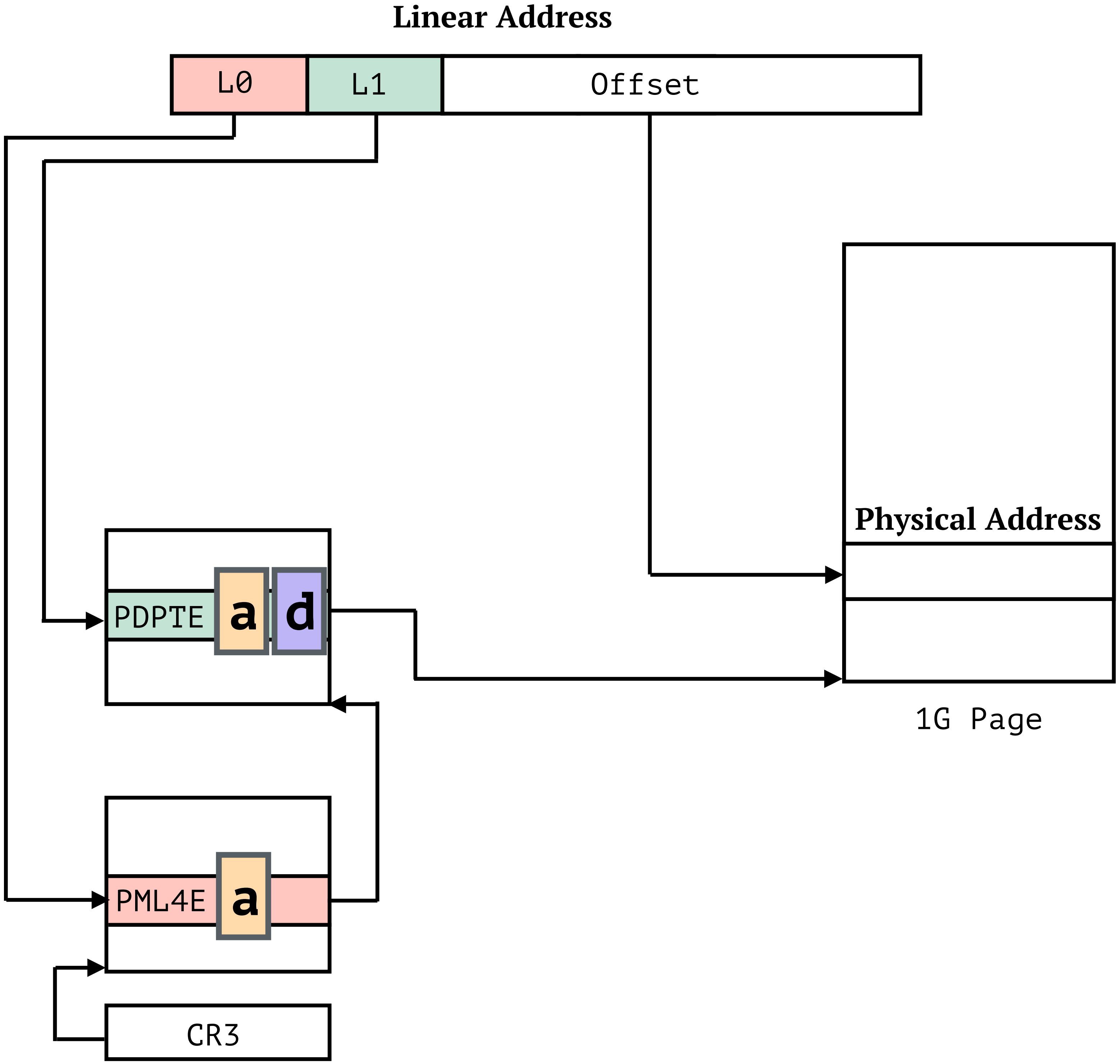
Figure 4-11. Formats of CR3 and Paging-Structure Entries with IA-32e Paging

Source: Intel Manuals

Reasoning about Paging is Complicated #2

2. *Accessed and dirty flag* updates during paging structure traversals cause side-effect writes

Paging entries governing the translation of a linear address are ***marked***.



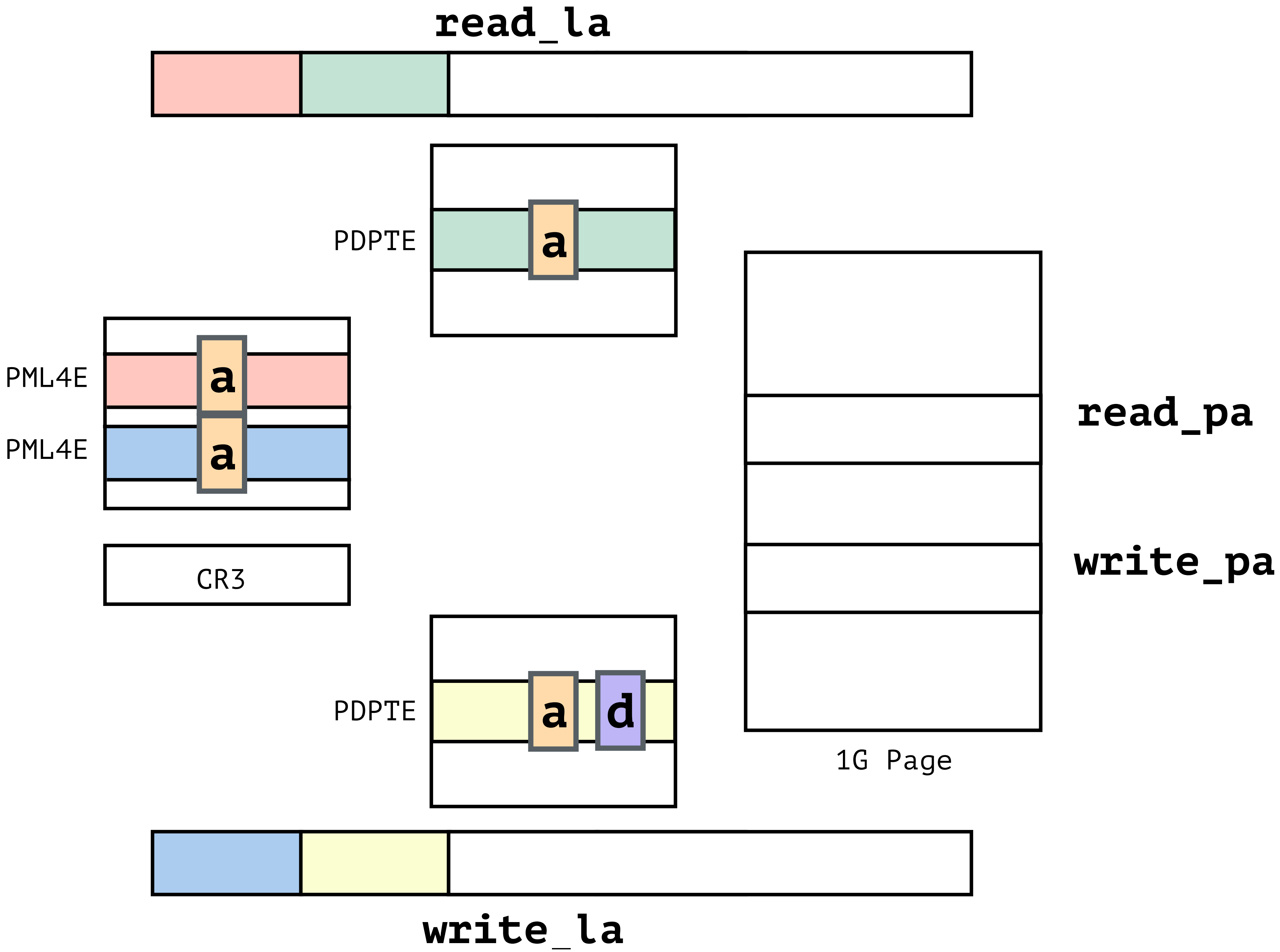
Reasoning about Paging is Complicated *#3*

3. Paging data structures are located in the physical memory
   - Physical memory cannot be accessed directly in the 64-bit mode — not even by supervisor-mode programs.
   - In order to access a paging entry, the entry's own linear address needs to be translated to a physical address first.
   - **Paging structures are mapped, too!**

Linear Memory Non-Interference Theorem

When is a linear memory read operation unaffected by a linear memory write operation?

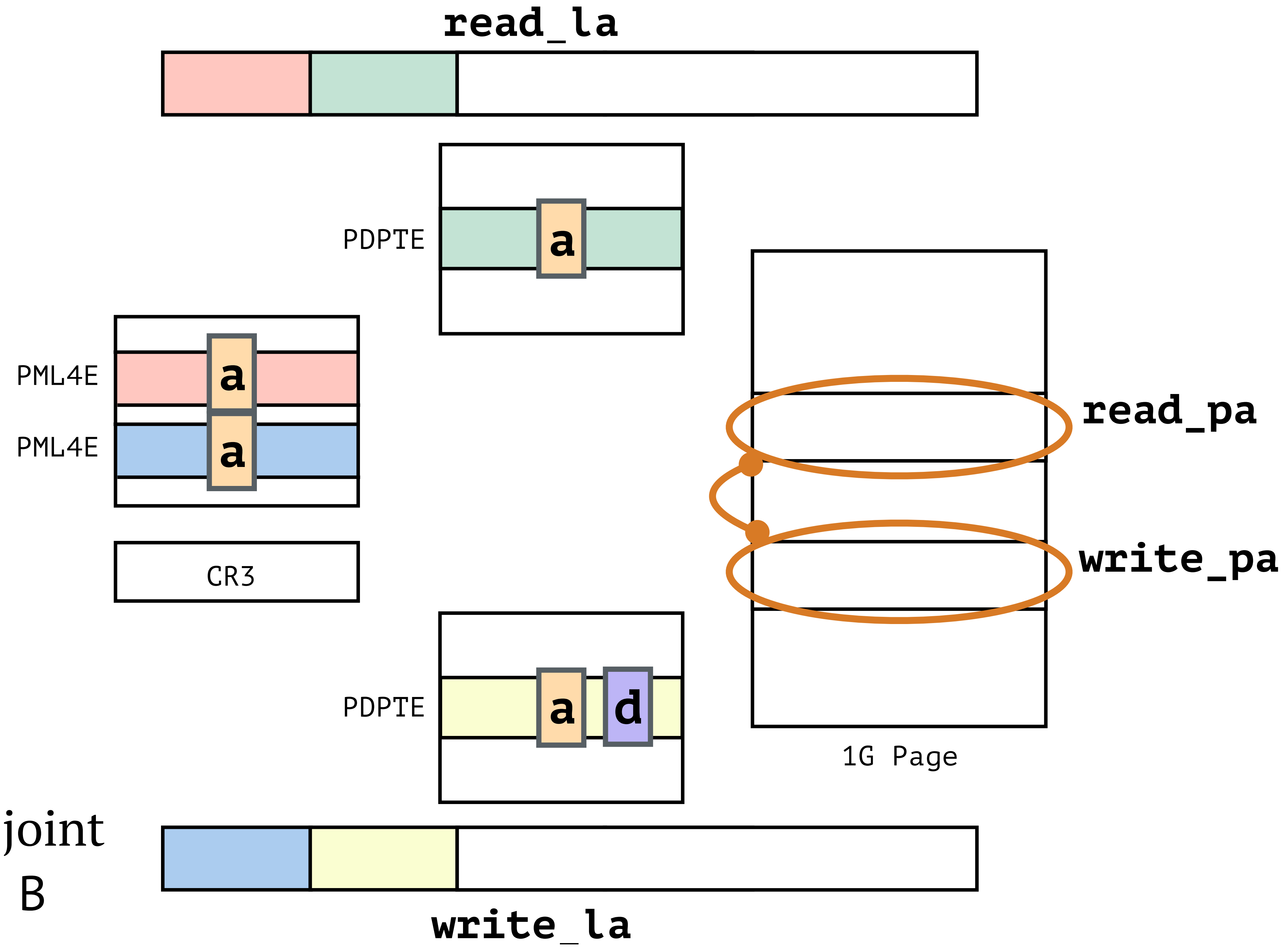
system-level mode



Linear Memory Non-Interference Theorem

When is a linear memory read operation unaffected by a linear memory write operation?

system-level mode

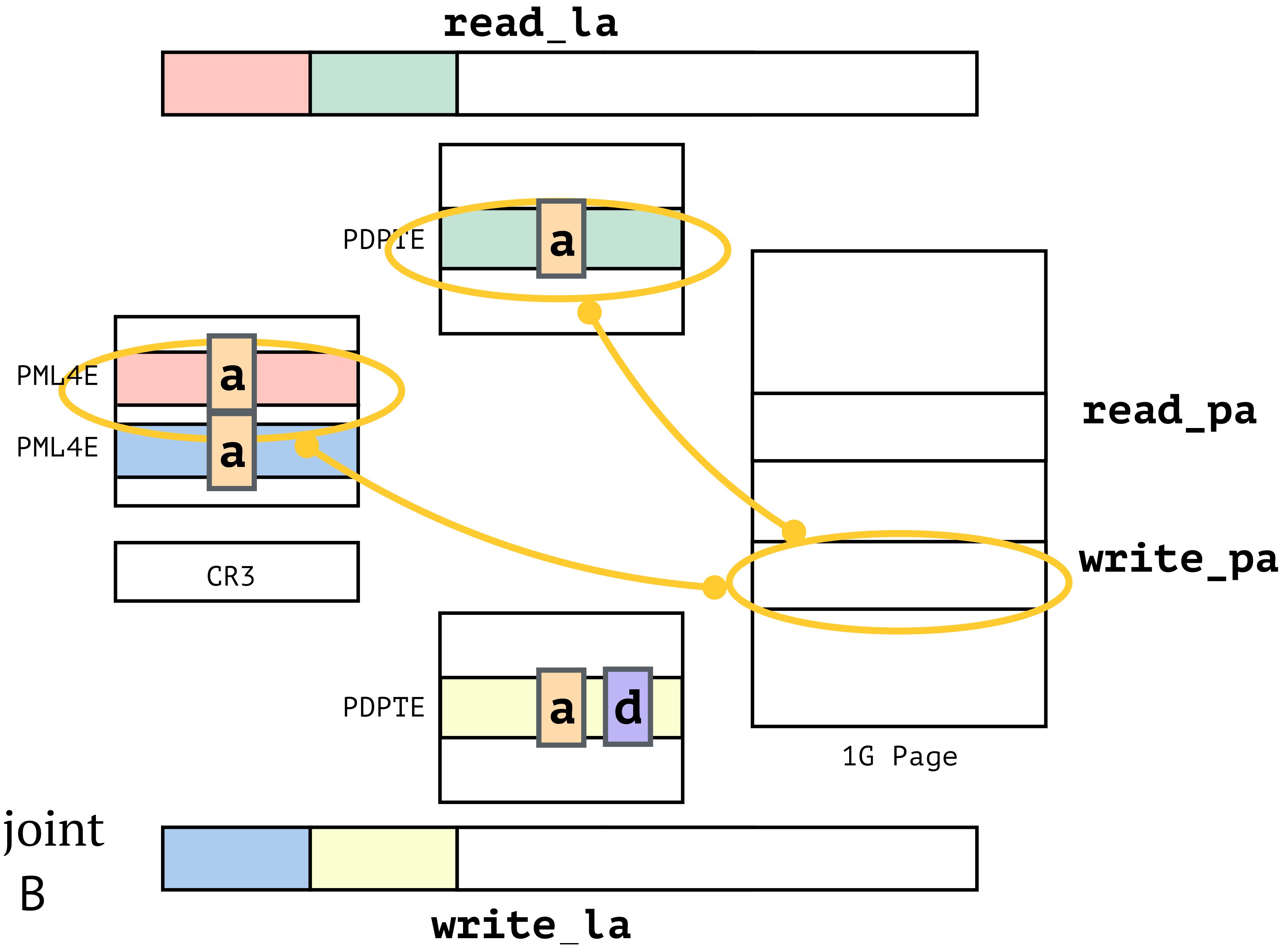


A and B are disjoint

Linear Memory Non-Interference Theorem

When is a linear memory read operation unaffected by a linear memory write operation?

system-level mode

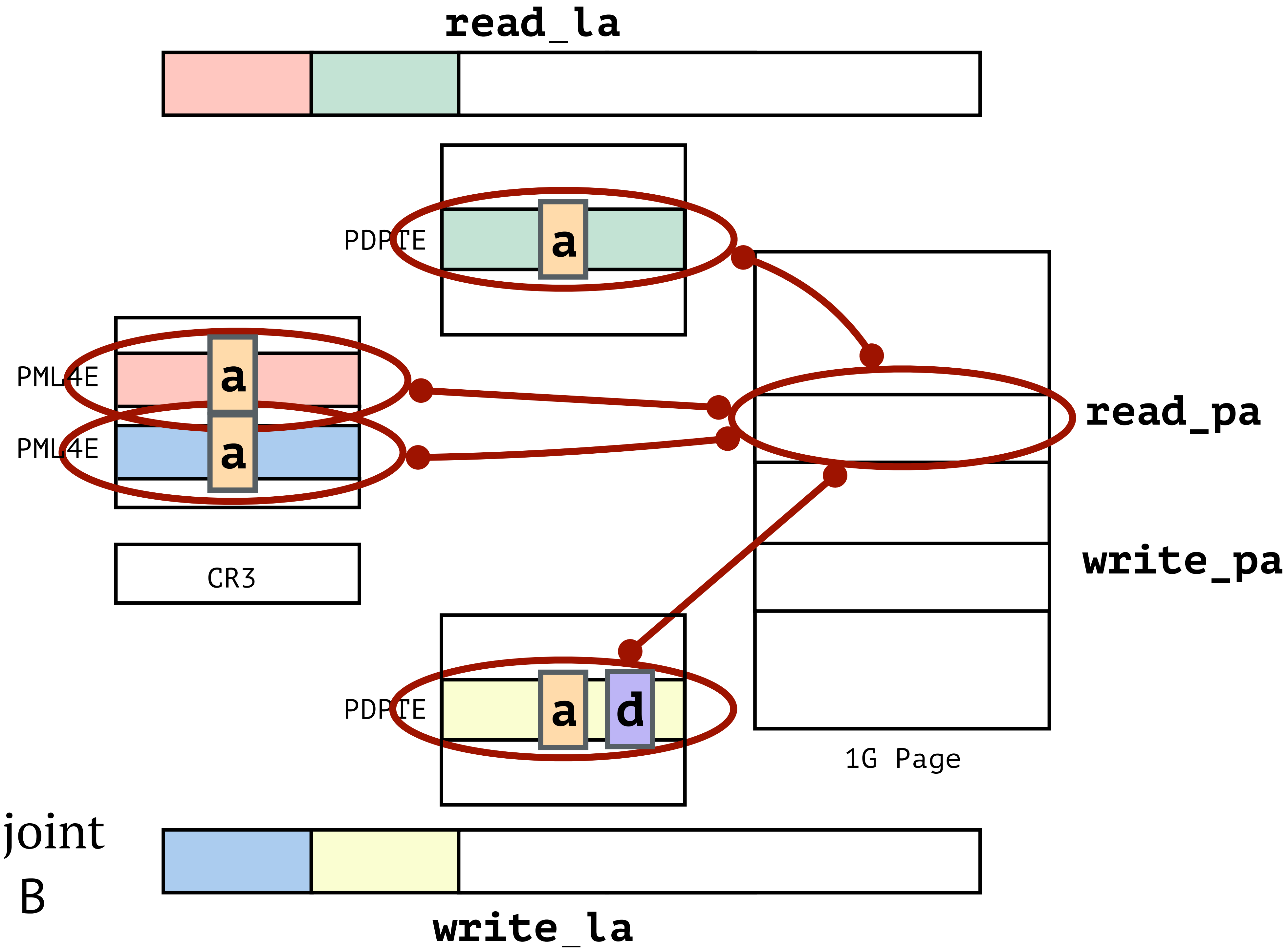


A and B are disjoint

Linear Memory Non-Interference Theorem

When is a linear memory read operation unaffected by a linear memory write operation?

system-level mode



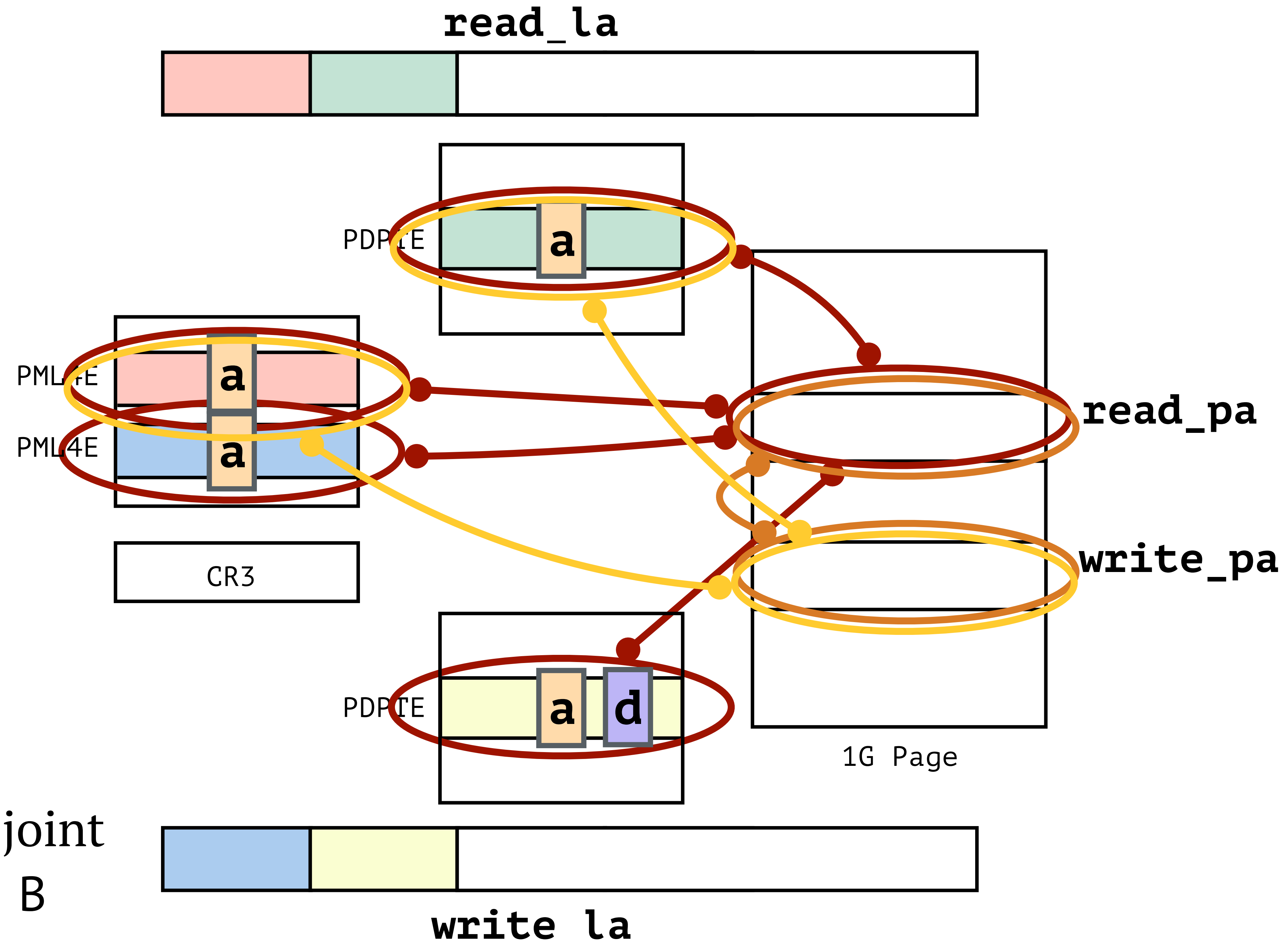
A and B are disjoint

A ●——● B

Linear Memory Non-Interference Theorem

When is a linear memory read operation unaffected by a linear memory write operation?

system-level mode



A and B are disjoint

A ●——● B

Linear Memory Non-Interference Theorem

```
(defthm linear-mem-non-interference-system-level-mode
  (let* ((pas-1 (las-to-pas las-1 r-x (cpl x86) x86))
         (pas-2 (las-to-pas las-2  :w (cpl x86) x86)))
    (implies
     (and
      (disjoint-p pas-1 pas-2)
      (disjoint-p pas-2
                  (paging-entries-paddrs las-1 x86))
      (disjoint-p pas-1
                  (paging-entries-paddrs las-2 x86))
      (disjoint-p pas-1
                  (paging-entries-paddrs las-1 x86))
      (system-level-mode x86)
      ;; <other simple hypotheses elided here...>
      )
     (equal
      (read-mem las-1 r-x (write-mem las-2 bytes x86))
      (read-mem las-1 r-x x86)))))
```

`las-1` `las-2` – lists of linear addresses

Linear Memory Non-Interference Theorem

```
(defthm linear-mem-non-interference-system-level-mode
  (let* ((pas-1 (las-to-pas las-1 r-x (cpl x86) x86))
         (pas-2 (las-to-pas las-2  :w (cpl x86) x86)))
    (implies
     (and
      (disjoint-p pas-1 pas-2)
      (disjoint-p pas-2
                  (paging-entries-paddrs las-1 x86))
      (disjoint-p pas-1
                  (paging-entries-paddrs las-2 x86))
      (disjoint-p pas-1
                  (paging-entries-paddrs las-1 x86))
```

Complicates precondition discovery

A large number of hypotheses makes it challenging to discover interesting and/or non-obvious preconditions.

```
                                                       ))
```

`las-1` `las-2` – lists of linear addresses

System-level Mode: Sub-modes of Operation

- *Common case*: reads to fetch the next instruction or obtain program's data
  - A program and its data are usually disjoint from system data structures
  - Why pay the penalty of side-effect A/D flag updates for these reads?

System-level Mode: Sub-modes of Operation

- *Common case*: reads to fetch the next instruction or obtain program's data
  - A program and its data are usually disjoint from system data structures
  - Why pay the penalty of side-effect A/D flag updates for these reads?
- *Optimization:* separate side-effect A/D flag updates from other updates

System-level Mode: Sub-modes of Operation

- *Common case*: reads to fetch the next instruction or obtain program's data
  - A program and its data are usually disjoint from system data structures
  - Why pay the penalty of side-effect A/D flag updates for these reads?
- *Optimization:* separate side-effect A/D flag updates from other updates
- Two sub-modes of operation: ***marking*** and ***non-marking*** mode
  - **Marking Mode:** true specification of the x86 ISA
  - **Non-marking Mode:** side-effect updates to A/D flags suppressed
    - Simpler theorems, easier precondition discovery

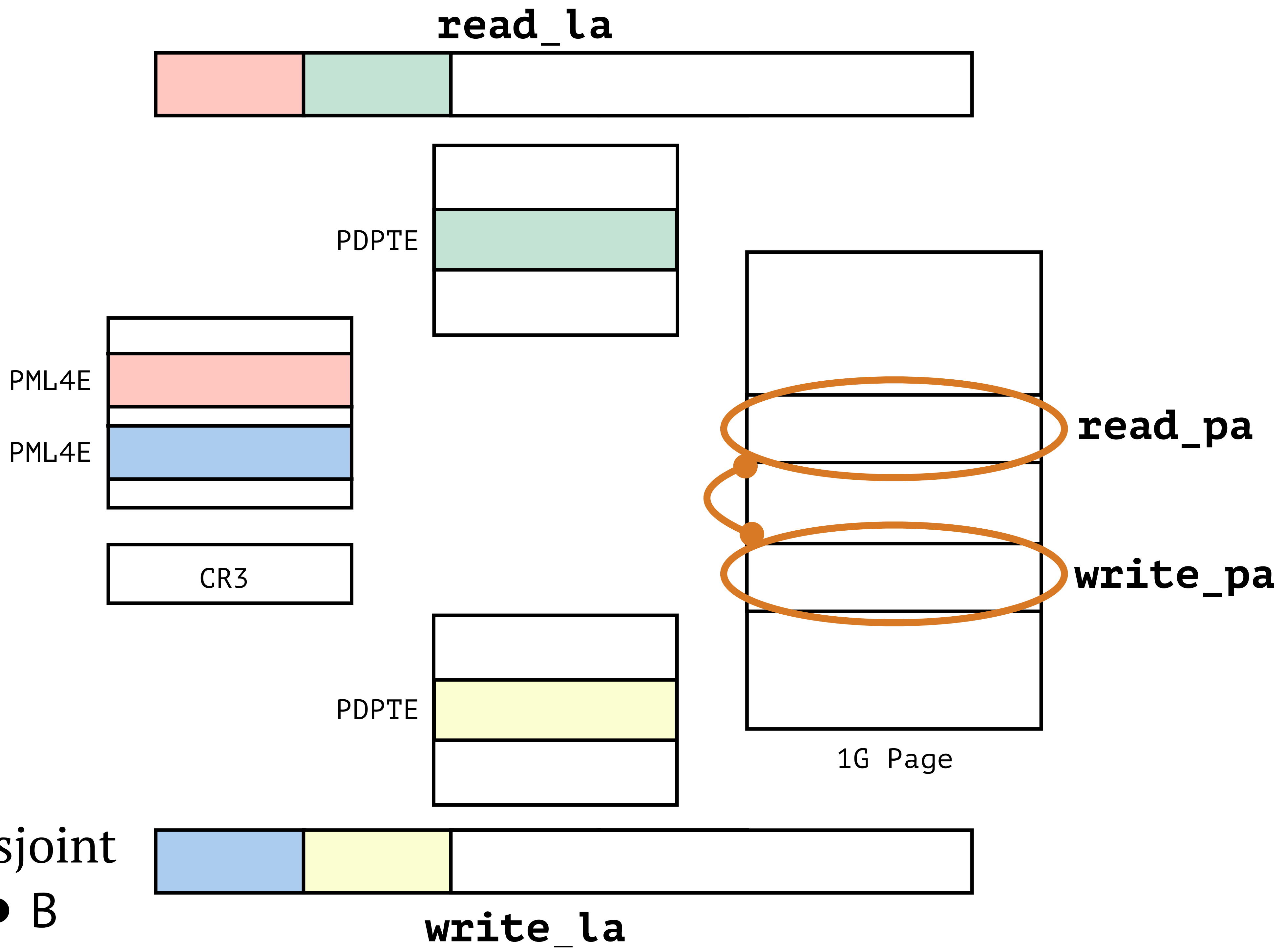
System-level Mode: Sub-modes of Operation

- Non-marking mode: simpler theorems, easier precondition discovery
- *Modus Operandi*:
  - **First verify a program in the non-marking mode and then port it over to the marking mode**
- *Caveat:*
  - Works for programs that do not rely on side-effect A/D flag updates
  - Can always reason directly in the system-level marking mode

Linear Memory Non-Interference Theorem

When is a linear memory read operation unaffected by a linear memory write operation?

system-level non-marking mode



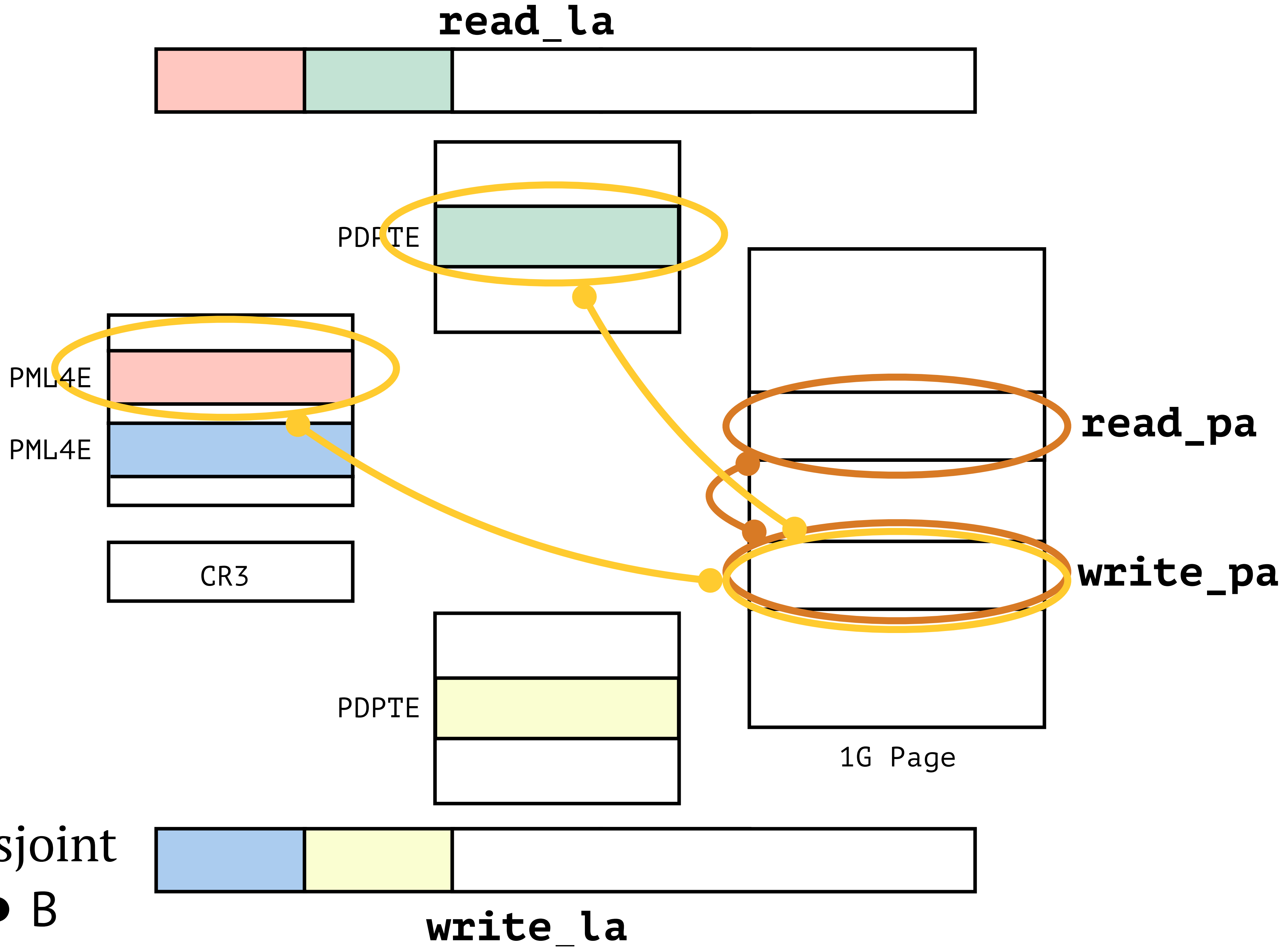
A and B are disjoint

A ●———● B

Linear Memory Non-Interference Theorem

When is a linear memory read operation unaffected by a linear memory write operation?

system-level non-marking mode



A and B are disjoint

Linear Memory Non-Interference Theorem

```
(defthm linear-mem-non-interference-system-level-non-marking—mode
  (let* ((pas-1 (las-to-pas las-1 r-x (cpl x86) x86))
         (pas-2 (las-to-pas las-2  :w (cpl x86) x86)))
    (implies
     (and
      (disjoint-p pas-1 pas-2)
      (disjoint-p pas-2 (paging-entries-paddrs las-1 x86))
      (system-level-non-marking-mode x86)
      ;; <other simple hypotheses elided here...>
      )
     (equal
      (read-mem las-1 r-x (write-mem las-2 bytes x86))
      (read-mem las-1 r-x x86)))))
```

`las-1` `las-2` – lists of linear addresses

Reducing Reasoning Overhead in Marking Mode

In the **system-level marking mode** of operation:

- Memory reads disjoint from the paging data structures *automatically ignore* side-effect updates to A/D flags
  - Provided all the additional disjointness conditions are specified

Reducing Reasoning Overhead in Marking Mode

In the **system-level marking mode** of operation:

- Memory reads disjoint from the paging data structures *automatically ignore* side-effect updates to A/D flags
  - Provided all the additional disjointness conditions are specified
- ***Conditional Congruence-based Rewriting:***
  - Rewrite `read-mem` to `read-mem-alt` if applicable; use congruence rules to allow `read-mem-alt` to ignore side-effect updates to A/D flags

Reducing Reasoning Overhead in Marking Mode

In the **system-level marking mode** of operation:

- Memory reads disjoint from the paging data structures *automatically ignore* side-effect updates to A/D flags
  - Provided all the additional disjointness conditions are specified
- ***Conditional Congruence-based Rewriting*:**
  - Rewrite `read-mem` to `read-mem-alt` if applicable; use congruence rules to allow `read-mem-alt` to ignore side-effect updates to A/D flags
- ***Program Comprehension:***
  - Memory read operation in terms of `read-mem`: target is a paging entry
  - Memory read operation in terms of `read-mem—alt`: target is disjoint from paging structures

Outline

Overview

1. Formal Model of the x86 ISA

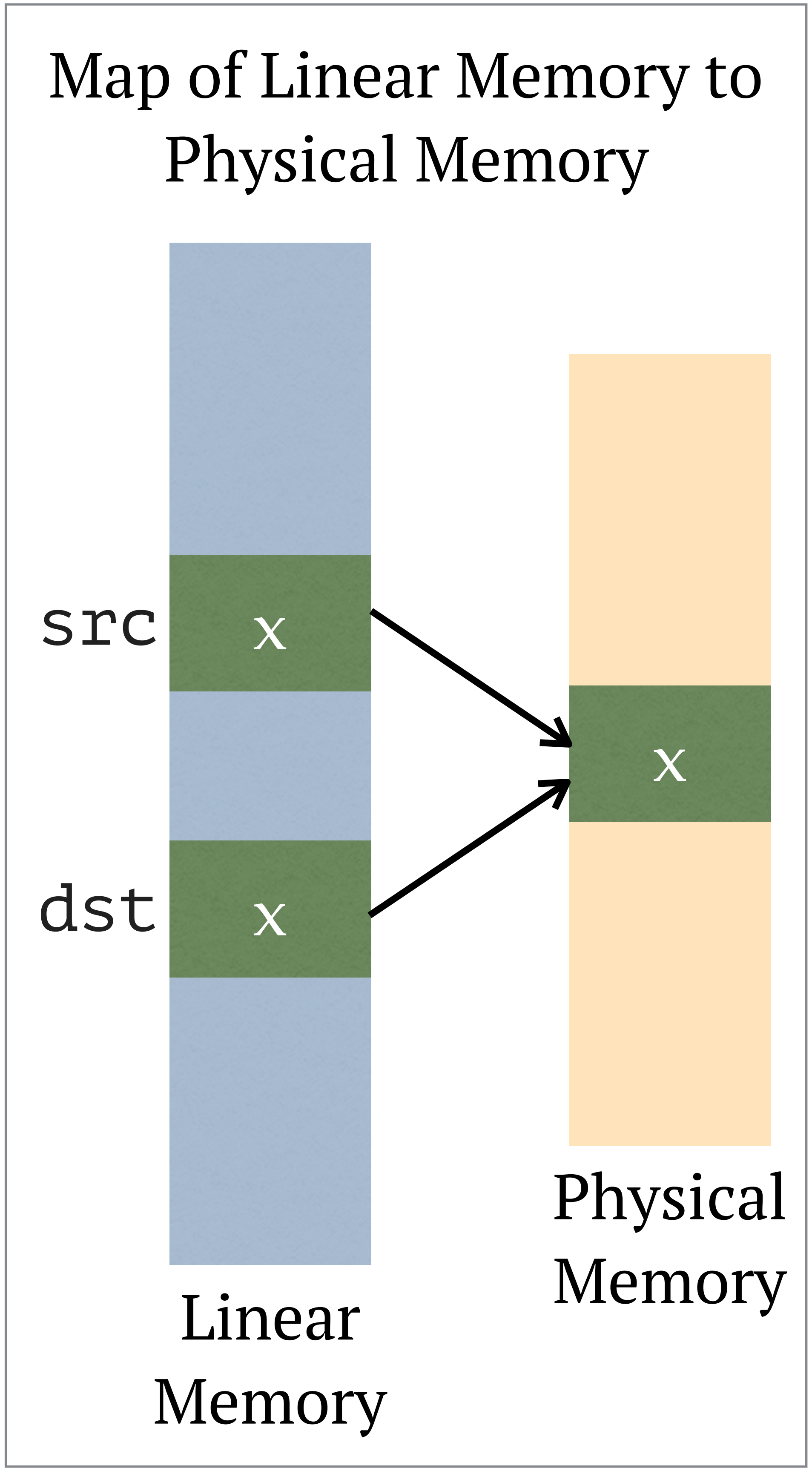
2. Lemma Libraries for Machine-Code Verification

3. **Case Studies**

Concluding Remarks and Future Work

Case Study: Zero-Copy Program

- **Copies data by modifying the paging structures** so that both the `src` and `dst` are mapped to the same physical memory location
  - Zero copies exist in reality
  - Can be used for implementing the Copy-on-Write (COW) technique

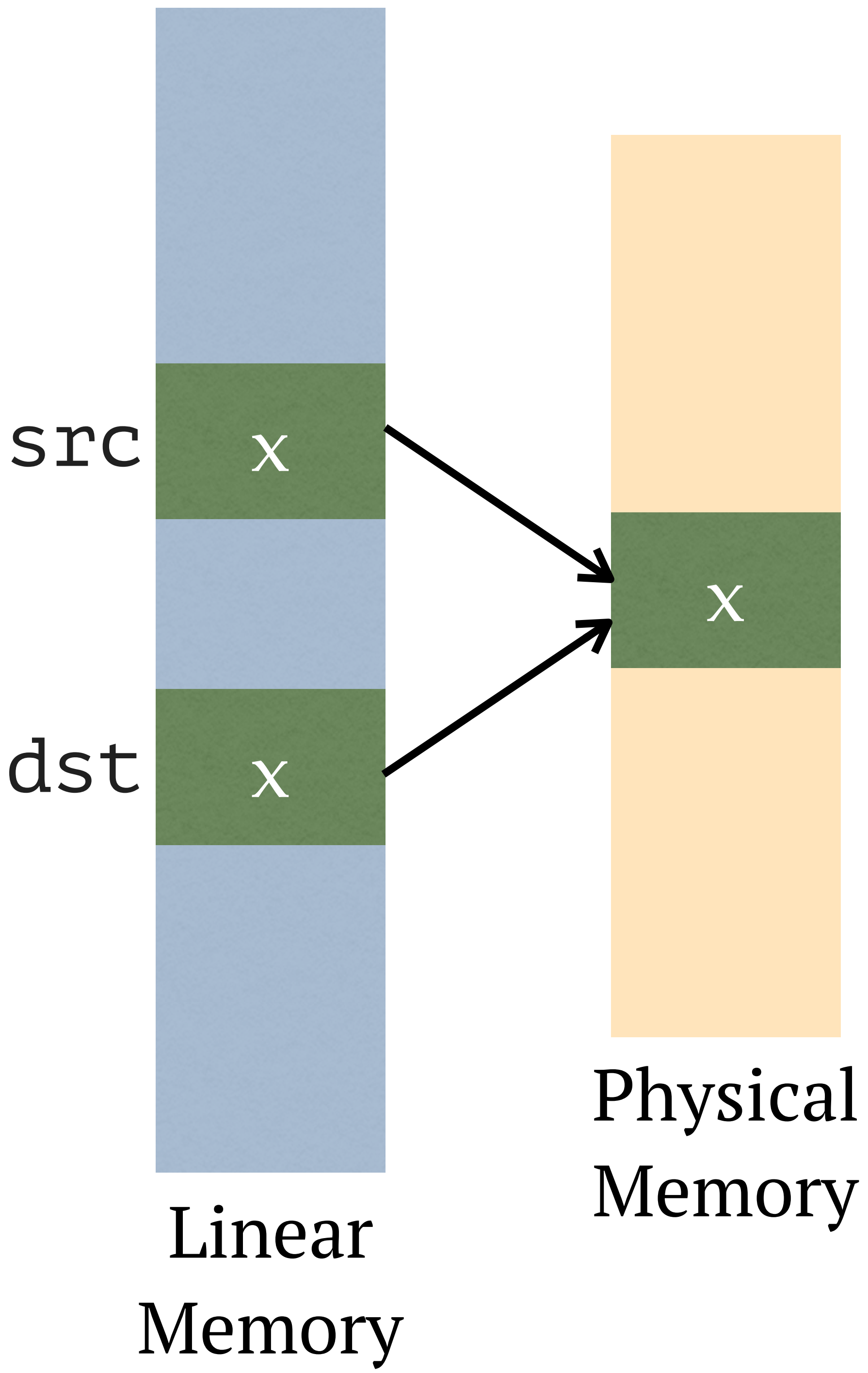


Case Study: Zero-Copy Program

- **Copies data by modifying the paging structures** so that both the `src` and `dst` are mapped to the same physical memory location
  - Zero copies exist in reality
  - Can be used for implementing the Copy-on-Write (COW) technique

- **Establishing this program's correctness is critical:**
  - Linear memory is the only view of memory available to 64-bit x86 programs.
  - An incorrect setup of paging structures can cause security leaks and crashes in otherwise correct programs.

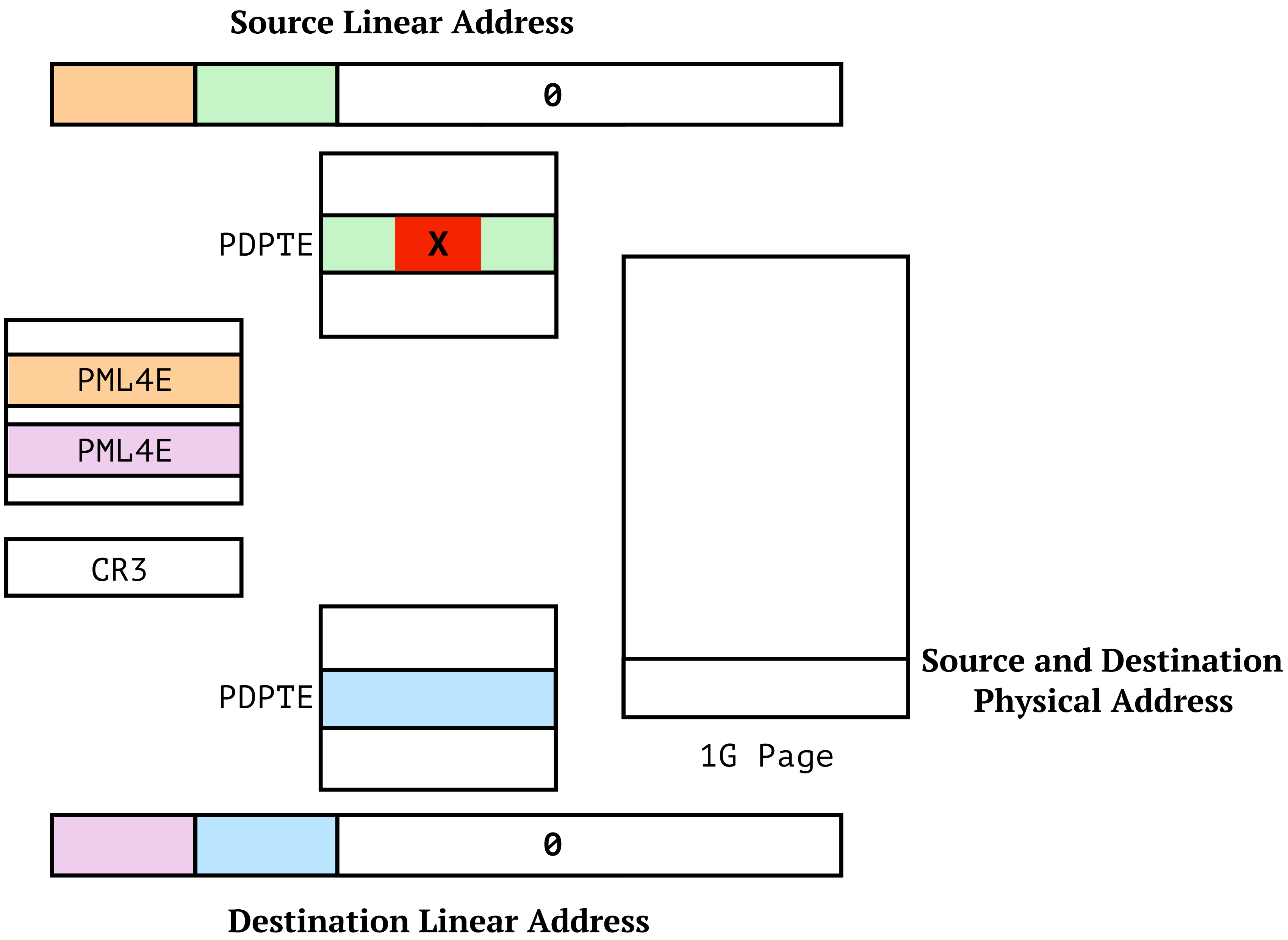
Map of Linear Memory to Physical Memory



Case Study: Zero-Copy Program

Constraints:

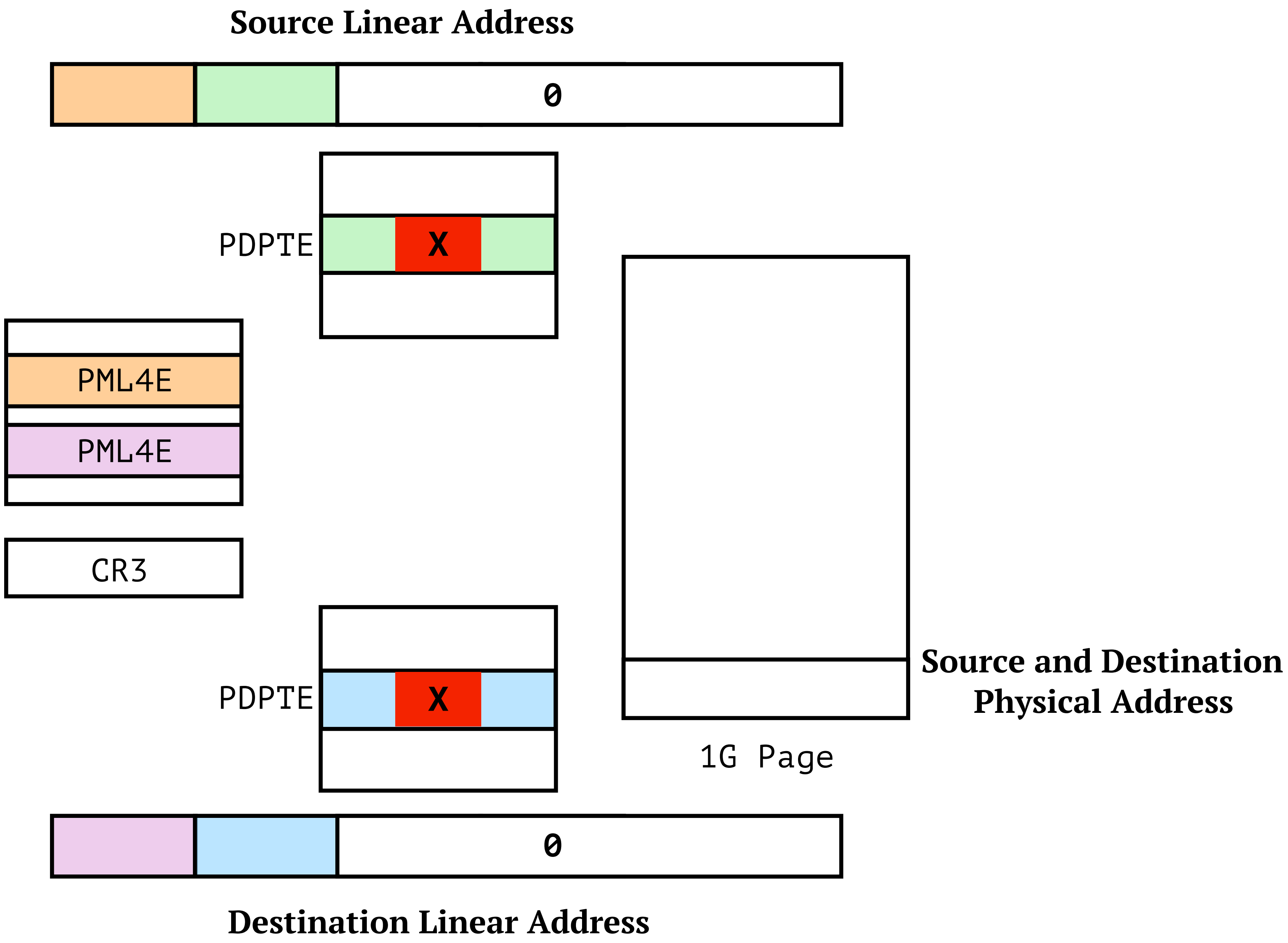
- Data to be copied: 1GB
- Source and destination are 1GB-aligned


Source Linear Address
0
PDPTE
X
PML4E
PML4E
CR3
PDPTE
1G Page
Source and Destination Physical Address
0
Destination Linear Address

Case Study: Zero-Copy Program

Constraints:

- Data to be copied: 1GB
- Source and destination are 1GB-aligned


Source Linear Address
0
PDPTE
X
PML4E
PML4E
CR3
PDPTE
X
Source and Destination Physical Address
1G Page
0
Destination Linear Address

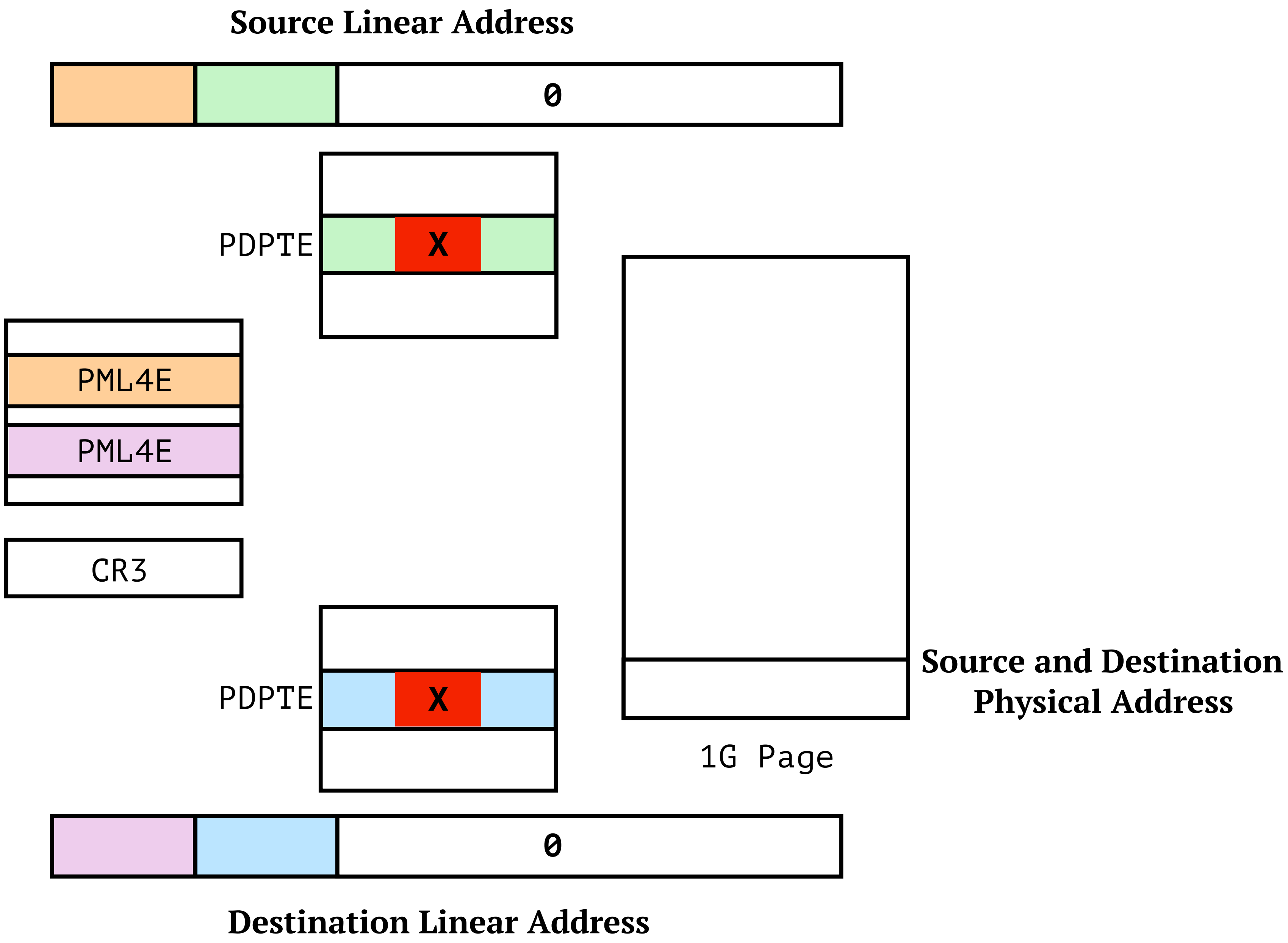
Case Study: Zero-Copy Program

Constraints:

- Data to be copied: 1GB
- Source and destination are 1GB-aligned

Key Challenge:

Discovering and specifying the conditions under which this program operates correctly


Source Linear Address
0
PDPTE
X
PML4E
PML4E
CR3
PDPTE
X
1G Page
Source and Destination Physical Address
0
Destination Linear Address

Case Study: Zero-Copy Program

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

1. **[Copy Occurs]** The 1GB of data at the destination's linear addresses in the *final x86 state* is the same as the 1GB of data at the source's linear addresses in the *initial x86 state*.
2. **[Source is Unmodified]** The 1GB of data at the source's linear addresses in the *final x86 state* is the same as the 1GB of data at the source's linear addresses in the *initial x86 state*.
3. **[Program is Unmodified]** The program in the *final x86 state* is the same as that in the *initial x86 state*.

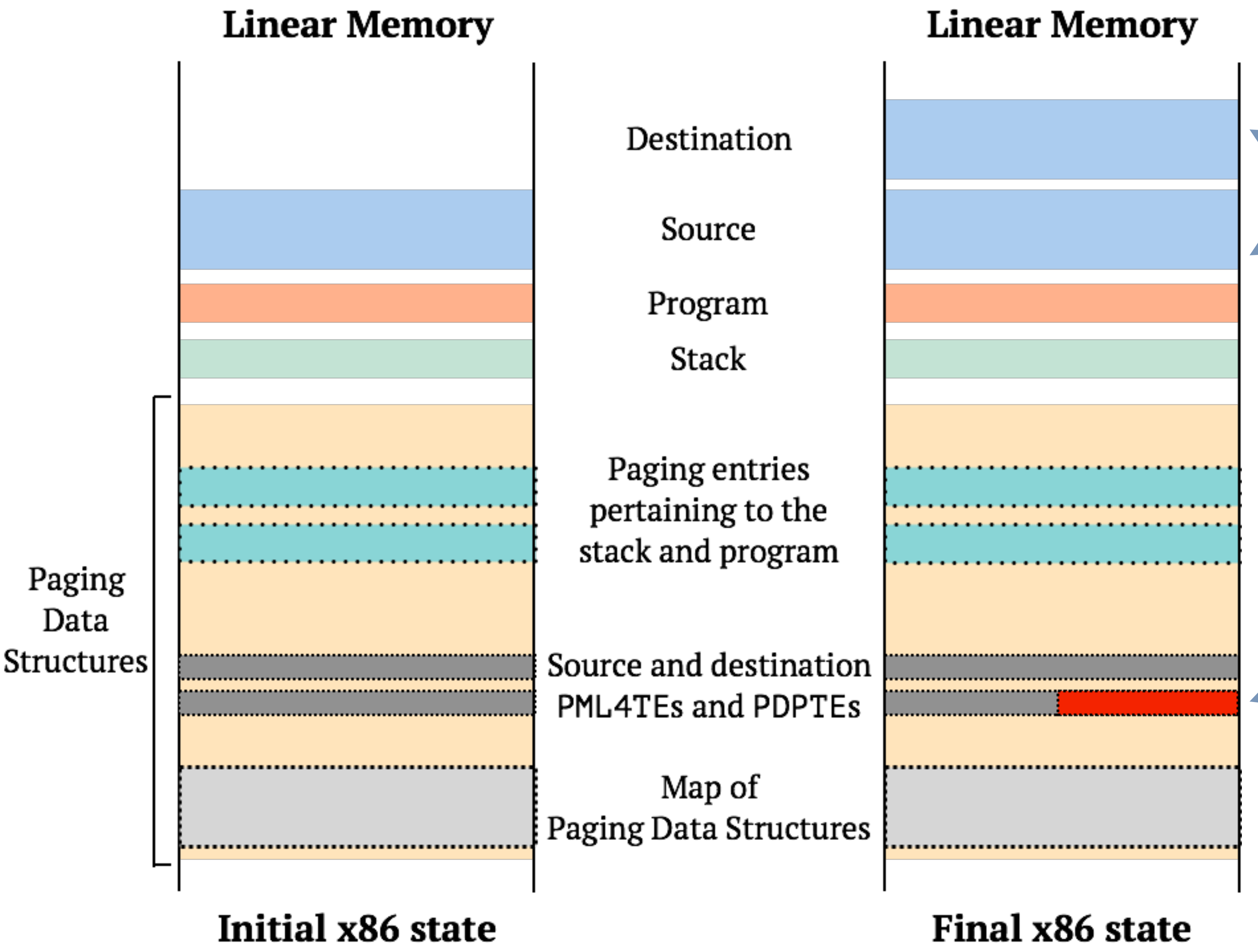
Case Study: Zero-Copy Program

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

1. **[Copy Occurs]** The 1GB of data at the destination’s linear addresses in the *final x86 state* is the same as the 1GB of data at the source’s linear addresses in the *initial x86 state*.
2. **[Source is Unmodified]** The 1GB of data at the source’s linear addresses in the *final x86 state* is the same as the 1GB of data at the source’s linear addresses in the *initial x86 state*.
3. **[Program is Unmodified]** The program in the *final x86 state* is the same as that in the *initial x86 state*.

Around **120 preconditions,** mostly about the **disjointness** of different regions of the memory (e.g., program, data, stack, paging entries)

Case Study: Zero-Copy Program



Linear Memory
Linear Memory
Destination
Source
Program
Stack
Paging entries pertaining to the stack and program
Paging Data Structures
Source and destination PML4TEs and PDPTEs
Map of Paging Data Structures
Initial x86 state
Final x86 state

two copies of data

modification to destination’s PDPTE for linear address re-mapping

View of Linear Memory

not drawn to scale

Outline

Overview

1. Formal Model of the x86 ISA

2. Lemma Libraries for Machine-Code Verification

3. Case Studies

Concluding Remarks and Future Work

Review

My Ph.D. proposal described:

1. x86 ISA model
2. (a) Libraries to reason about application programs
3. (a) Verification of two application programs

Focus of this talk: **[Diss. Ch. 7]**

1. New features of the x86 ISA model
2. (b) Libraries to reason about system programs **[Diss. Ch. 10]**
3. (b) Verification of a system program — Zero-Copy **[Diss. Ch. 12]**

	STATUS: THEN	STATUS: NOW
x86 ISA Model	220 Opcodes	413 Opcodes
Lemma Libraries	Support only for application programs	Support added for system programs
Case Studies	Application programs	Added system program (Zero-Copy)
Documentation	Largely developer-focused topics	Added user-focused topics, including a guide to debug failed proofs

Contributions

Formal, executable specification of the x86 ISA (IA-32e mode)

- Accurate reference of the x86 ISA
- Fastest formal simulator of its kind
- Tools that support its use as a practical instruction-set simulator

Reasoning framework for x86 machine-code analysis

- Automated symbolic simulation of x86 machine-code programs
- Supports reasoning about system data structures

Verification strategies that can be adopted to verify a variety of machine-code programs

Documentation of engineering aspects of building a large-scale formal analysis framework

Opportunities for Future Research

Operating System Verification

detect reliance on non-portable or undefined behaviors

User-friendly Program Analysis

automate the discovery of preconditions

Multi-process/threaded Program Verification

reason about concurrency-related issues

Reasoning about the Memory System

determine if caches are (mostly) transparent, as intended

Firmware Verification

formally specify software/hardware interfaces

Micro-architecture Verification

x86 ISA model serves as a build-to specification

Publications

Shilpi Goel, Warren A. Hunt, Jr., and Matt Kaufmann. *Abstract Stobjs and Their Application to ISA Modeling*. In Proceedings of the ACL2 Workshop 2013, EPTCS 114, pp. 54-69, 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'13), volume 8164 of Lecture Notes in Computer Science, pages 222– 241. Springer Berlin Heidelberg, 2014

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 Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design (FMCAD'14), pages 18:91–98, 2014

Shilpi Goel, Warren A. Hunt, Jr., and Matt Kaufmann. *Engineering a Formal, Executable x86 ISA Simulator for Software Verification*. In Provably Correct Systems (ProCoS), 2015

[Source Code]

Github

[Documentation]

x86isa in the ACL2+Community Books Manual

Thanks!

Extra Slides

Case Study: Pop-Count Program

```
popcount_64:
89 fa                 mov    %edi,%edx
89 d1                 mov    %edx,%ecx
d1 e9                 shr    %ecx
81 e1 55 55 55 55     and    $0x55555555,%ecx
29 ca                 sub    %ecx,%edx
89 d0                 mov    %edx,%eax
c1 ea 02              shr    $0x2,%edx
25 33 33 33 33        and    $0x33333333,%eax
81 e2 33 33 33 33     and    $0x33333333,%edx
01 c2                 add    %eax,%edx
89 d0                 mov    %edx,%eax
c1 e8 04              shr    $0x4,%eax
01 c2                 add    %eax,%edx
48 89 f8              mov    %rdi,%rax
48 c1 e8 20           shr    $0x20,%rax
81 e2 0f 0f 0f 0f     and    $0xf0f0f0f,%edx
89 c1                 mov    %eax,%ecx
d1 e9                 shr    %ecx
81 e1 55 55 55 55     and    $0x55555555,%ecx
29 c8                 sub    %ecx,%eax
89 c1                 mov    %eax,%ecx
c1 e8 02              shr    $0x2,%eax
81 e1 33 33 33 33     and    $0x33333333,%ecx
25 33 33 33 33        and    $0x33333333,%eax
01 c8                 add    %ecx,%eax
89 c1                 mov    %eax,%ecx
c1 e9 04              shr    $0x4,%ecx
01 c8                 add    %ecx,%eax
25 0f 0f 0f 0f        and    $0xf0f0f0f,%eax
69 d2 01 01 01 01     imul   $0x1010101,%edx,%edx
69 c0 01 01 01 01     imul   $0x1010101,%eax,%eax
c1 ea 18              shr    $0x18,%edx
c1 e8 18              shr    $0x18,%eax
01 d0                 add    %edx,%eax
c3                    retq
```

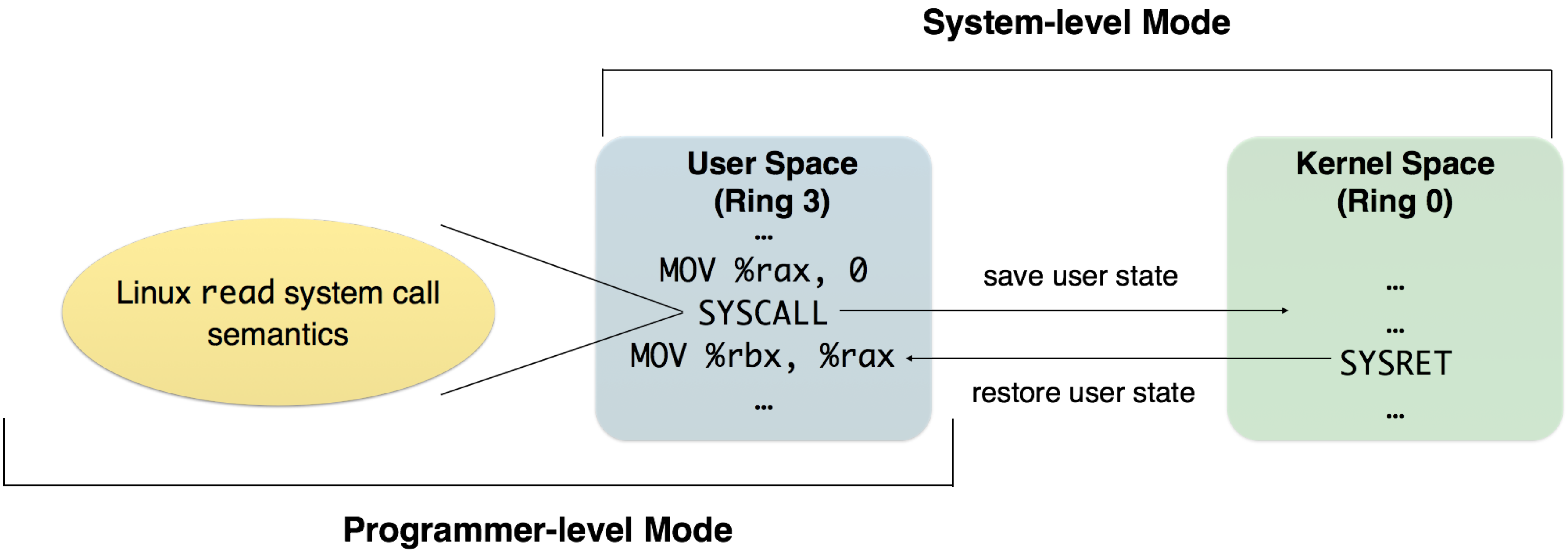
Functional Correctness:
RAX = popcount(v)

specification function

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

Case Study: Word-Count Program

- Program obtains input from the user via `read` system calls.
- System calls are ***non-deterministic*** for application programs.



Case Study: Word-Count Program

- Program obtains input from the user via `read` system calls.
- System calls are ***non-deterministic*** for application programs.

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

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
```

Case Study: Word-Count Program

- Program obtains input from the user via `read` system calls.
- System calls are ***non-deterministic*** for application programs.

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

Specification fo

```
ncSpec(offset

 if (well-fo
    c := str[
    if (c ==
       return count
    else
       count := (count + 1) mod 2^32
       ncSpec(1 + offset, str, count)
    endif
 endif
```

Resource Usage:

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

Case Study: Word-Count Program

- Program obtains input from the user via `read` system calls.
- System calls are ***non-deterministic*** for application programs.

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

Specification fo

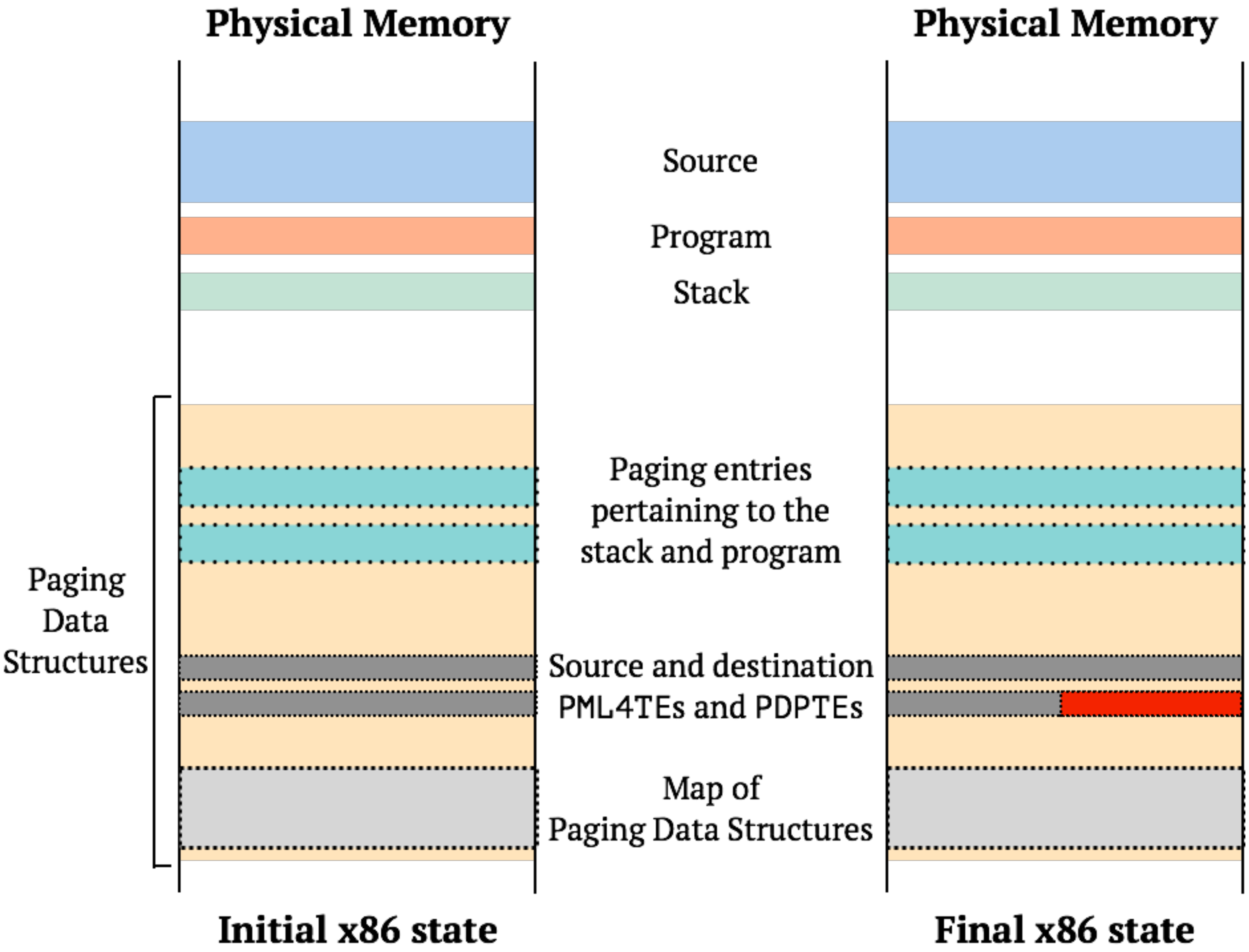
```
ncSpec(offset

 if (well-for
    c := str[
    if (c ==
       return count
    else
       count := (count + 1)
       ncSpec(1 + offset, s
    endif
 endif
```

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.

Zero-Copy: View of Physical Memory

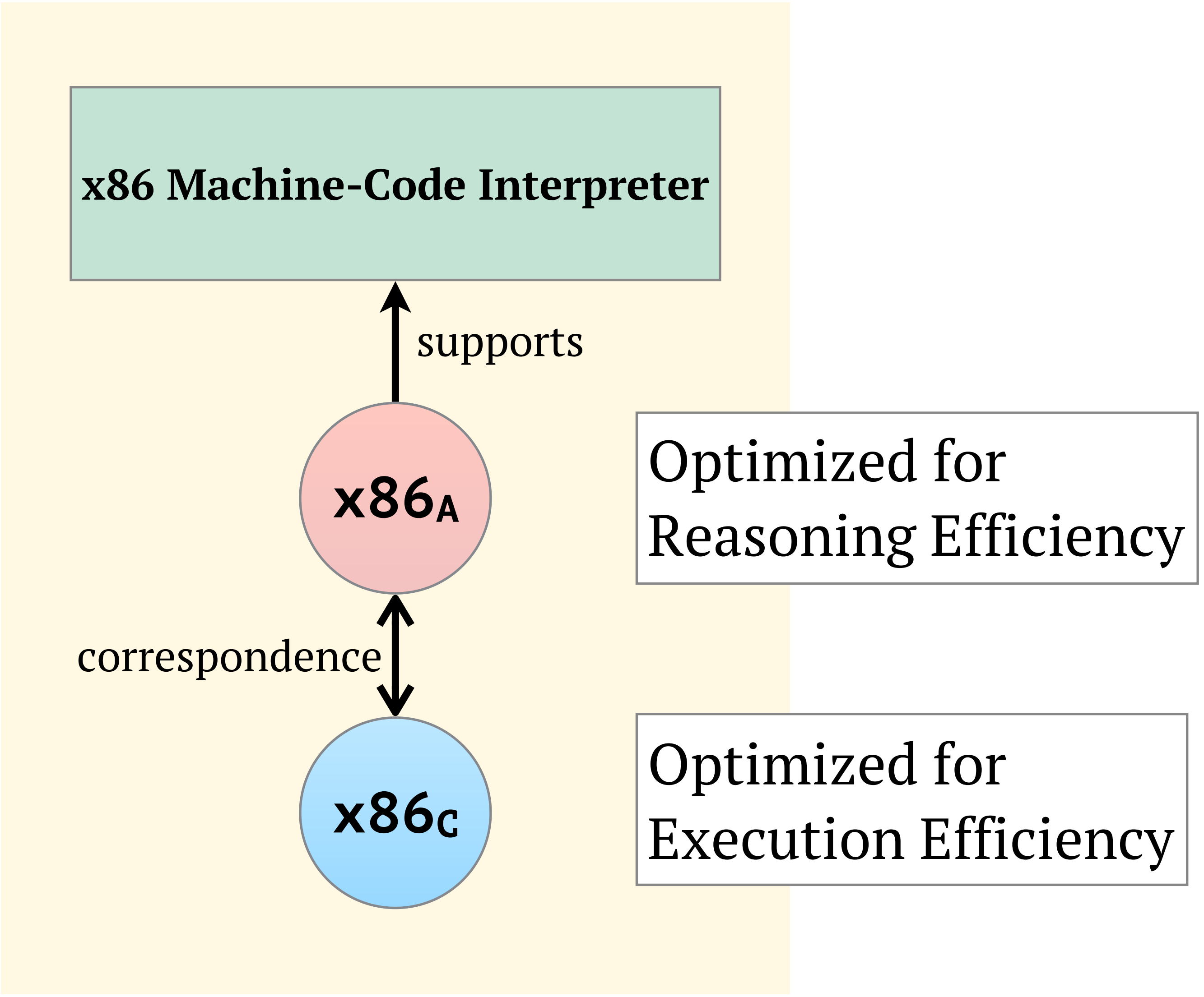

Physical Memory
Physical Memory
Source
Program
Stack
Paging entries pertaining to the stack and program
Paging Data Structures
Source and destination PML4TEs and PDPTEs
Map of Paging Data Structures
Initial x86 state
Final x86 state

one (original) copy of data

modification to destination’s PDPTE for linear address re-mapping

Reasoning & Execution Efficiency: *Abstract Stobjs*

- Layered modeling approach mitigates the trade-off between reasoning and execution efficiency.
- Abstract stobjs were added to ACL2 in response to the needs of this research project.


x86 Machine-Code Interpreter
supports
$x86_A$
Optimized for Reasoning Efficiency
correspondence
$x86_C$
Optimized for Execution Efficiency

x86 ISA Model

Review: Timeline

Adhered to the timeline envisioned in the proposal:

“*Spring 2015 – Summer 2015:* Specifying more x86 instructions; modeling the system descriptor tables to support segmentation and interrupts; formulating and proving properties about paging data structure traversals and modifications

Fall 2015: Choosing and simulating system program(s), such as an optimized data-copy program; this would identify the x86 features that need to be modeled in order to support the program’s execution and verification

Spring 2016: Verification of the target program(s)—this includes discovering and specifying properties of interest; it may also involve re-visiting modeling choices made earlier

Summer 2016 – Fall 2016: Dissertation writing and final defense”

Data point: envisioning how long a verification effort will take is becoming predictable

Deliverables

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**

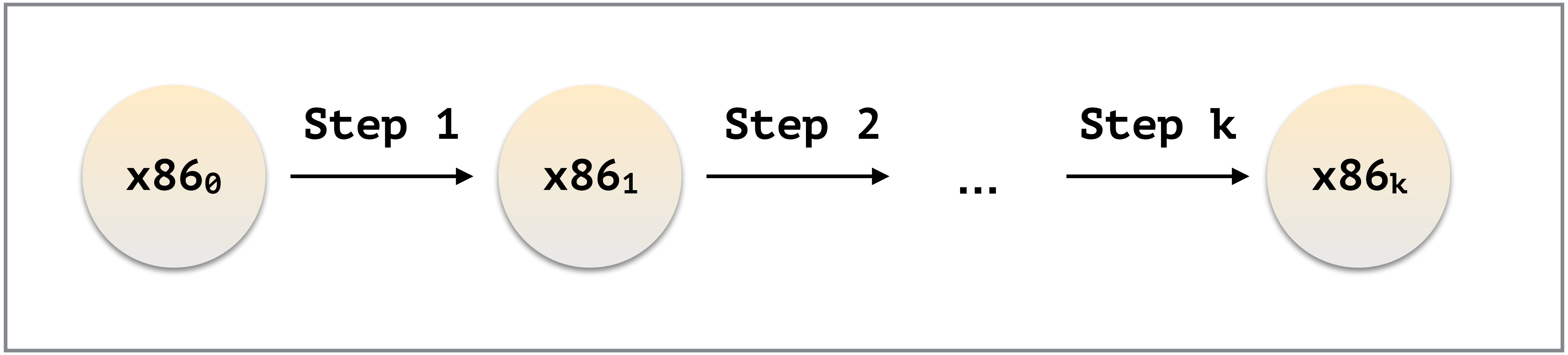
Code Proof Libraries

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

Manual

- ➡ **Documentation**

x86 ISA Model



A Run of the x86 Interpreter that executes k instructions

Interpreter-Style Operational Semantics:

- **x86 State:** specifies the components of the ISA
- **Instruction Semantic Functions:** specifies instructions' behavior
- **Step Function:** fetches, decodes, and executes one instruction
- **Run Function:** takes n steps or terminates early if an error occurs

Independence of Page Walks

- Proved using **congruence-based reasoning** in ACL2
  - Define an *equivalence relation* that states that two x86 states are equivalent if their paging structures are equal, modulo the A and D flags, and the rest of the memory is exactly equal.
  - Prove that the x86 state produced by a page walk is equivalent to the initial x86 state.
  - A page walk returns the same physical address for a linear address, given equivalent x86 states.

Successive Linear Memory Reads

System-level Marking Mode

```
(mv-nth 1 (rb las-1 r-x-1
              (mv-nth 2 (rb las-2 r-x-2 x86))))
=
(mv-nth 1 (rb las-1 r-x-1
              <writes to A flags of las-2's translation-governing entries>)
```

The above expression can be simplified to

```
(mv-nth 1 (rb las-1 r-x-1 x86))
```

only if physical addresses corresponding to `las-1` are disjoint from the physical addresses of the translation-governing entries of `las-2`.

Successive Linear Memory Reads

System-level Non-marking Mode

```
(mv-nth 1 (rb las-1 r-x-1
              (mv-nth 2 (rb las-2 r-x-2 x86))))
=
(mv-nth 1 (rb las-1 r-x-1 x86))
```

because, in the non-marking mode:

```
(mv-nth 2 (rb las-2 r-x-2 x86)))
=
x86
```

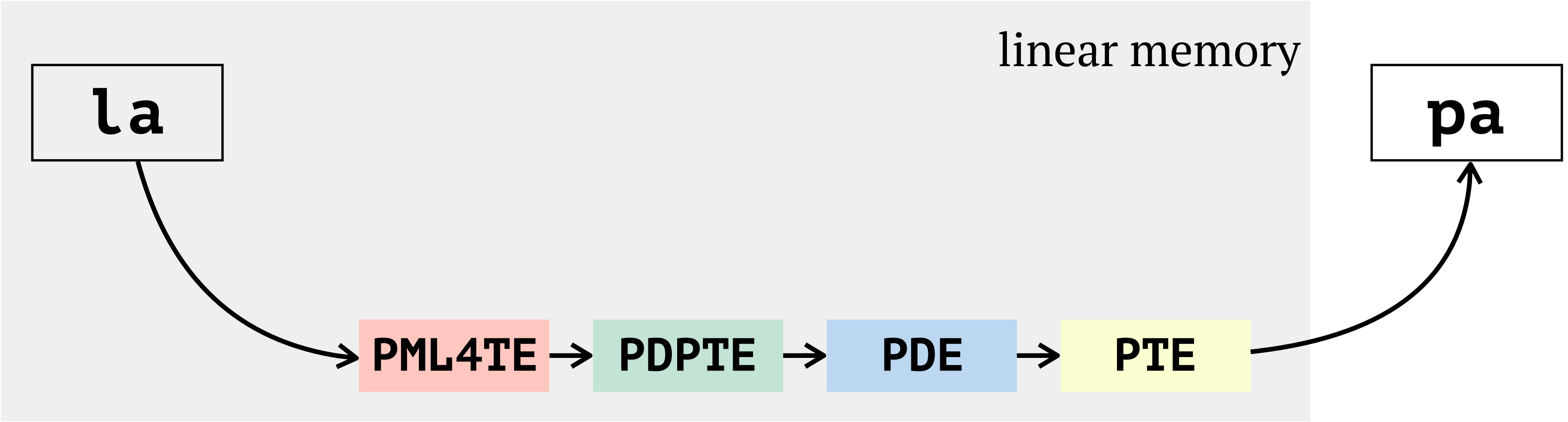
Reasoning about Paging is Complicated *#3*

3. Paging data structures are located in the physical memory
   - Physical memory cannot be accessed directly in the 64-bit mode — not even by supervisor-mode programs.
   - In order to access a paging entry, the entry’s own linear address needs to be translated to a physical address first.
   - **Paging structures are mapped, too!**

Reasoning about Paging is Complicated #3

3. Paging data structures are located in the physical memory
   - Physical memory cannot be accessed directly in the 64-bit mode — not even by supervisor-mode programs.
   - In order to access a paging entry, the entry's own linear address needs to be translated to a physical address first.
   - **Paging structures are mapped, too!**

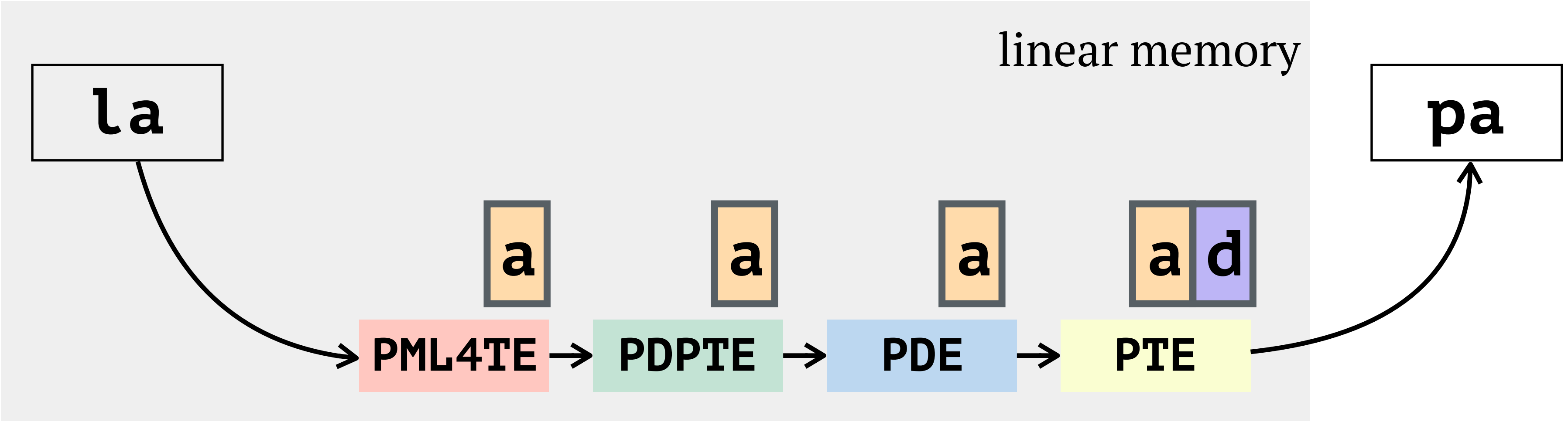
Specification: Maximum number of memory accesses to translate one linear address with a 4K configuration



Reasoning about Paging is Complicated *#3*

3. Paging data structures are located in the physical memory
   - Physical memory cannot be accessed directly in the 64-bit mode — not even by supervisor-mode programs.
   - In order to access a paging entry, the entry's own linear address needs to be translated to a physical address first.
   - **Paging structures are mapped, too!**

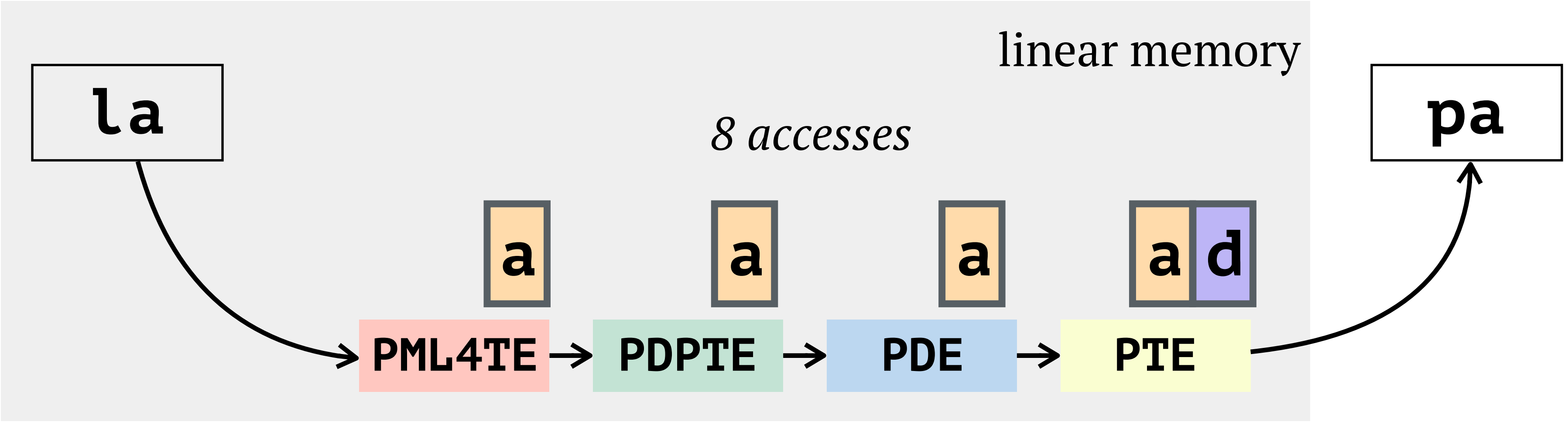
Specification: Maximum number of memory accesses to translate one linear address with a 4K configuration



Reasoning about Paging is Complicated *#3*

3. Paging data structures are located in the physical memory
   - Physical memory cannot be accessed directly in the 64-bit mode — not even by supervisor-mode programs.
   - In order to access a paging entry, the entry's own linear address needs to be translated to a physical address first.
   - **Paging structures are mapped, too!**

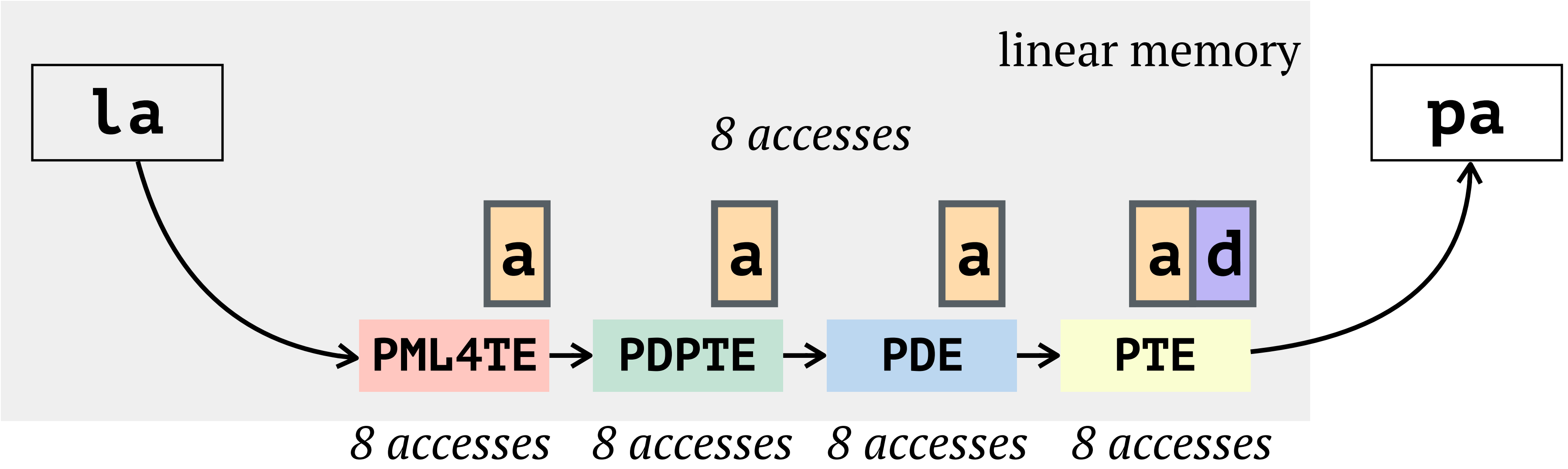
Specification: Maximum number of memory accesses to translate one linear address with a 4K configuration



Reasoning about Paging is Complicated #3

3. Paging data structures are located in the physical memory
   - Physical memory cannot be accessed directly in the 64-bit mode — not even by supervisor-mode programs.
   - In order to access a paging entry, the entry's own linear address needs to be translated to a physical address first.
   - **Paging structures are mapped, too!**

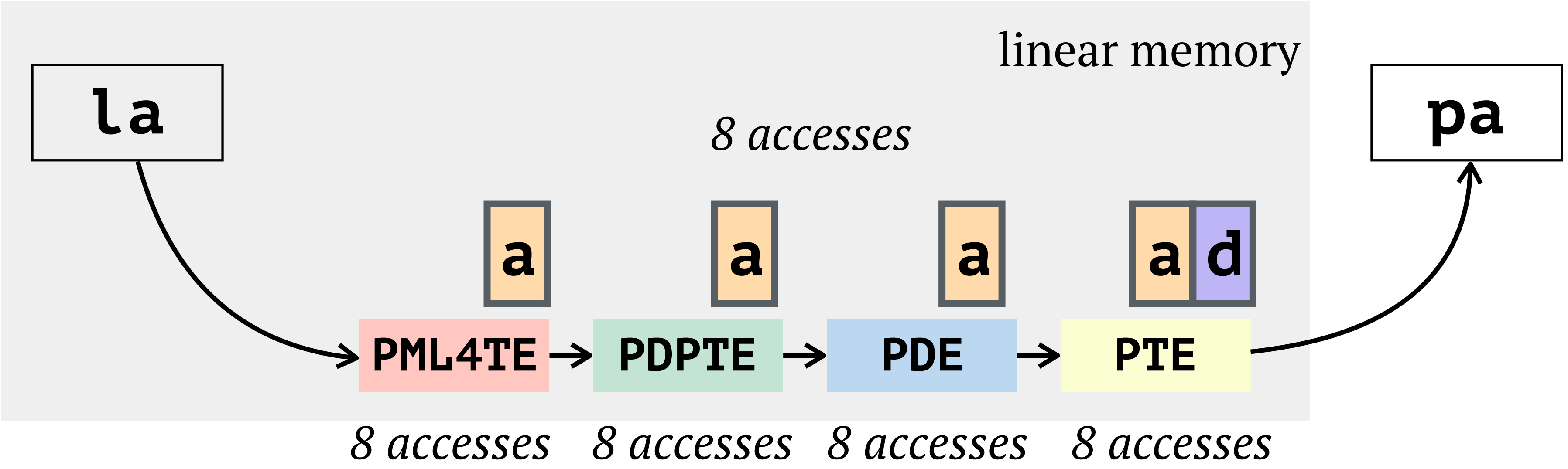
Specification: Maximum number of memory accesses to translate one linear address with a 4K configuration



Reasoning about Paging is Complicated #3

3. Paging data structures are located in the physical memory
   - Physical memory cannot be accessed directly in the 64-bit mode — not even by supervisor-mode programs.
   - In order to access a paging entry, the entry's own linear address needs to be translated to a physical address first.
   - **Paging structures are mapped, too!**

Specification: Maximum number of memory accesses to translate one linear address with a 4K configuration



Total number of memory accesses: 40