Adding 32-bit Mode to the ACL2 Model of the x86 ISA

Alessandro Coglio

Kestrel Technology

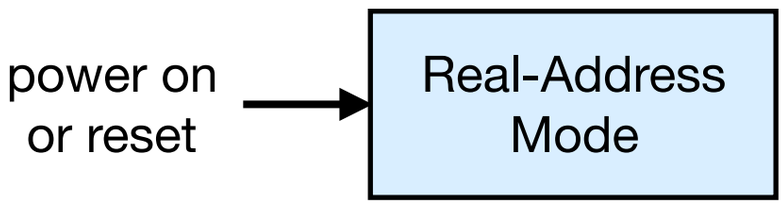
Shilpi Goel

Centaur Technology

ACL2 *Workshop 2018*

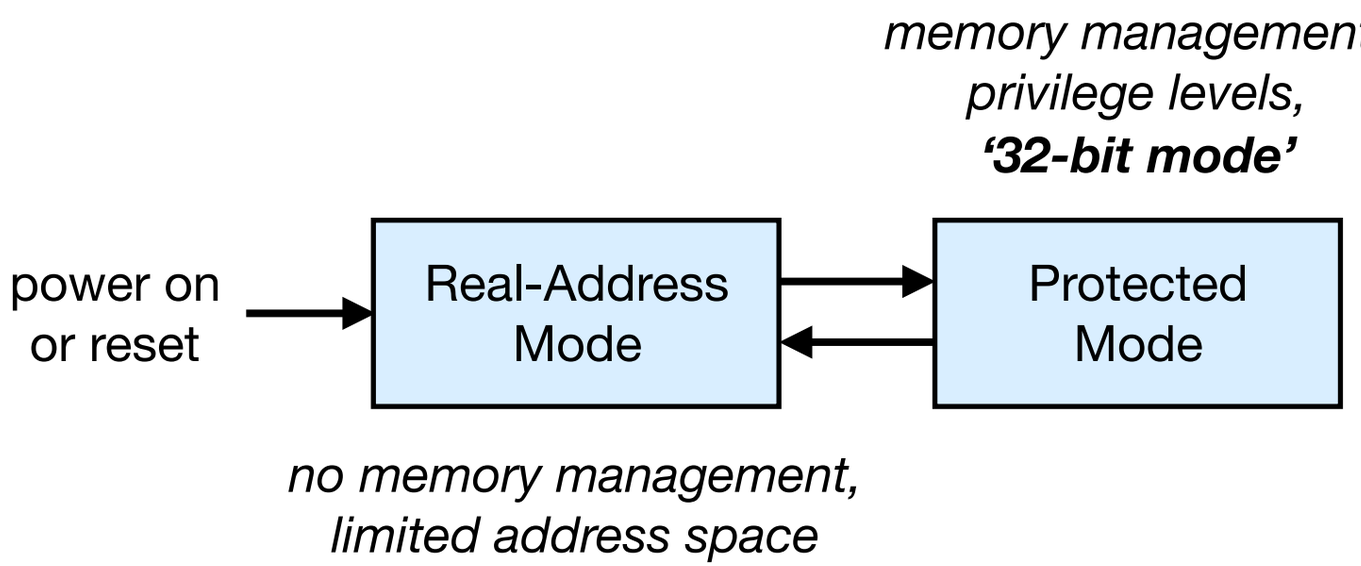
x86 Modes of Operation

x86 Modes of Operation

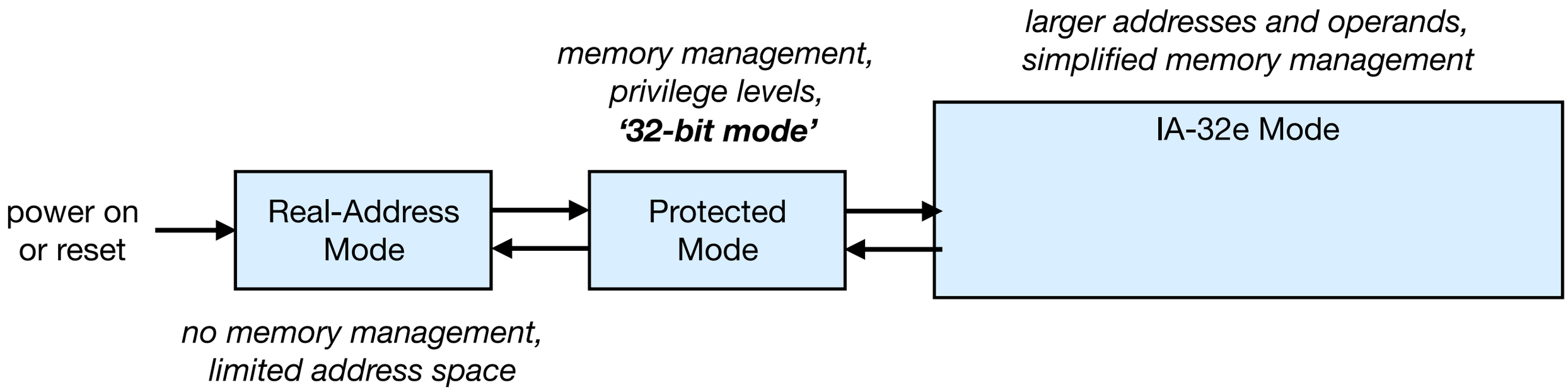


no memory management,
limited address space

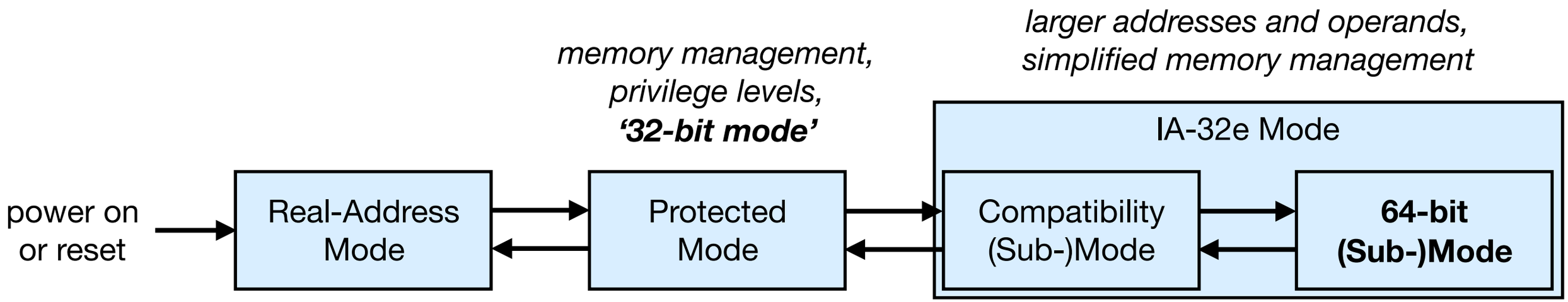
x86 Modes of Operation



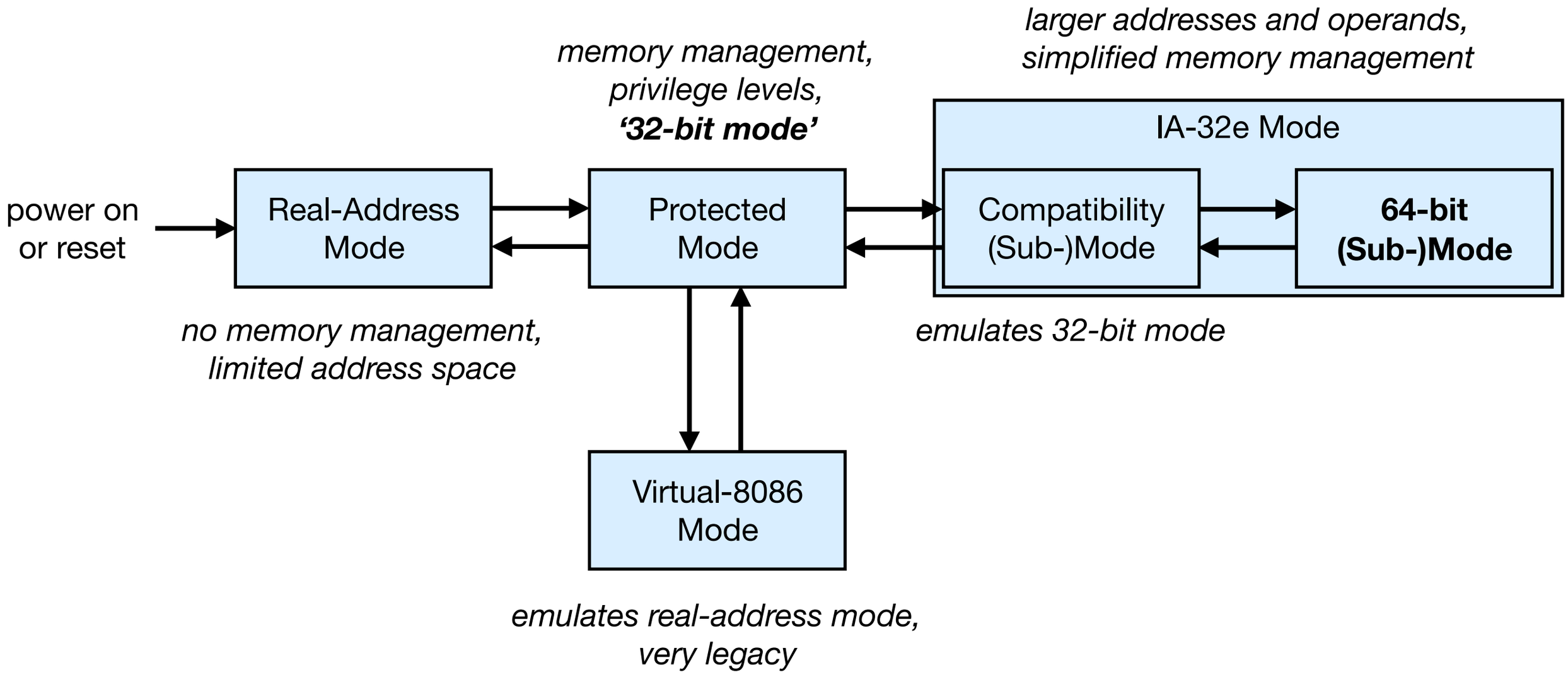
x86 Modes of Operation



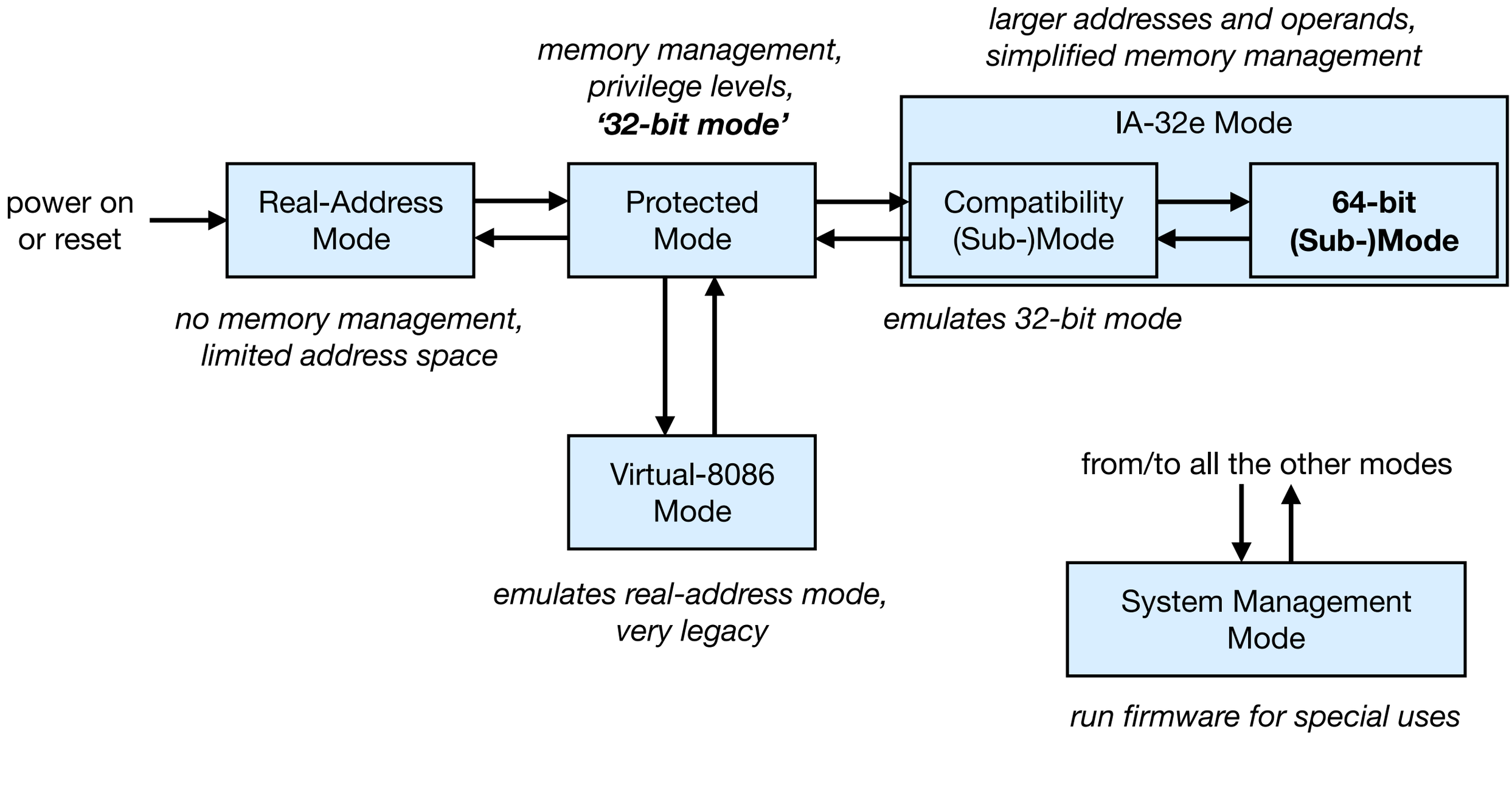
x86 Modes of Operation



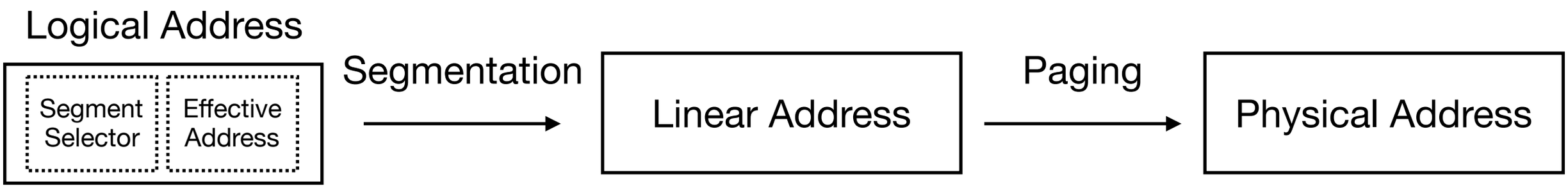
x86 Modes of Operation



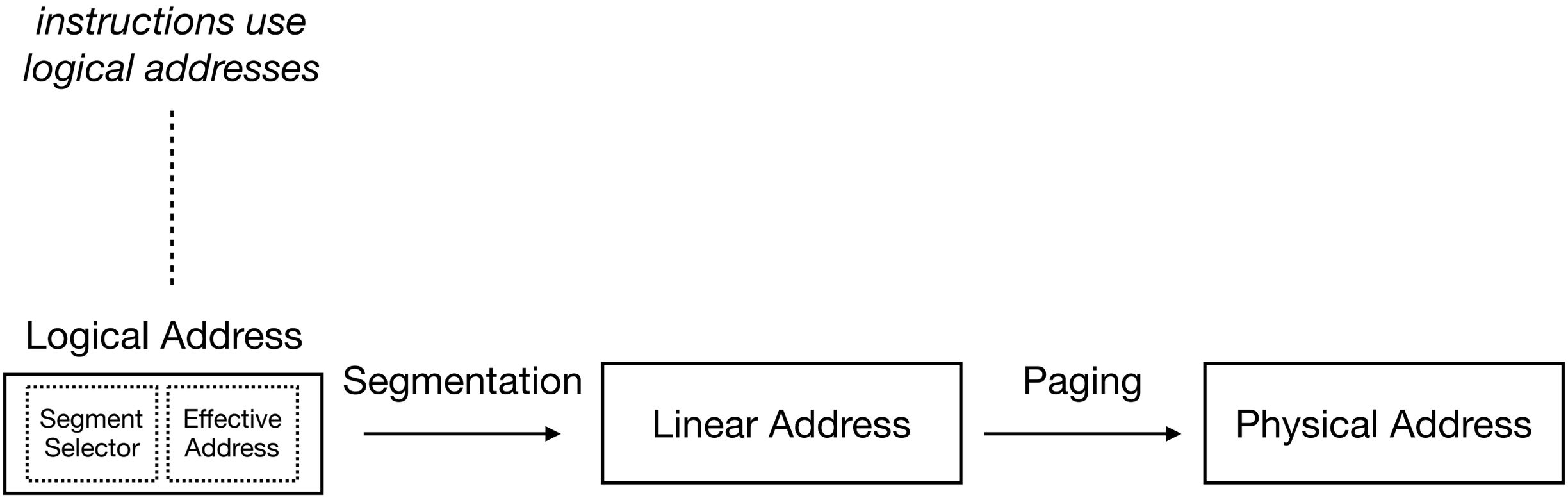
x86 Modes of Operation



Memory Management



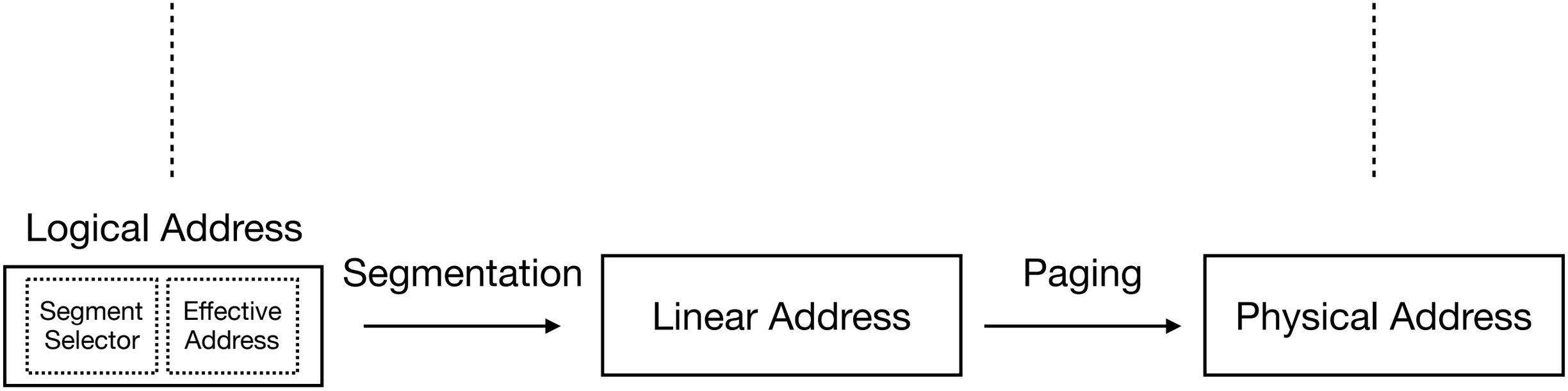
Memory Management



Memory Management

instructions use logical addresses

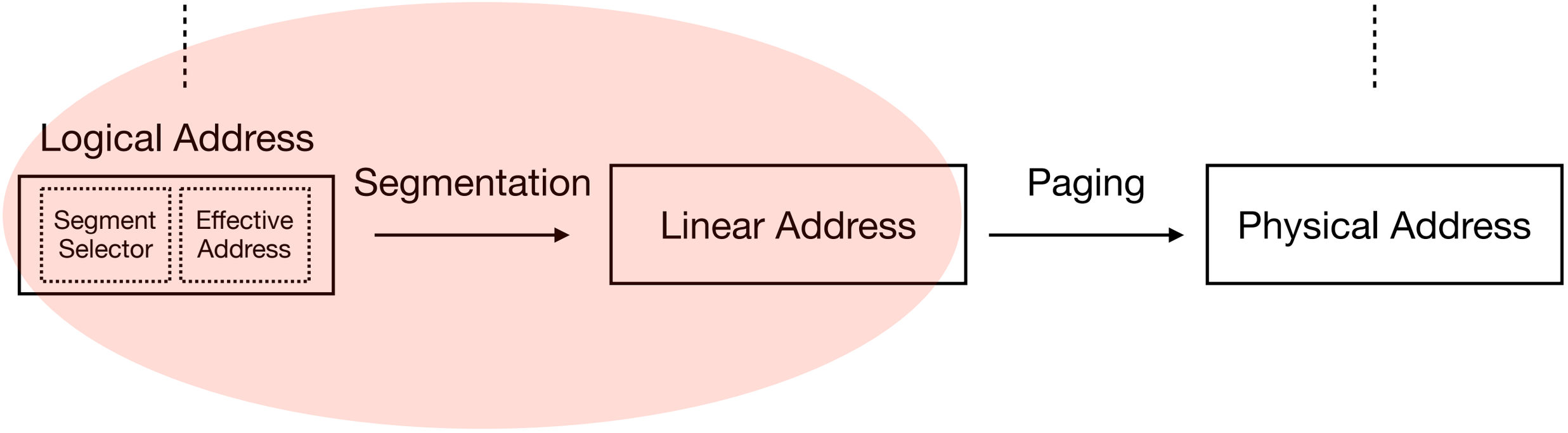
the bus uses physical addresses



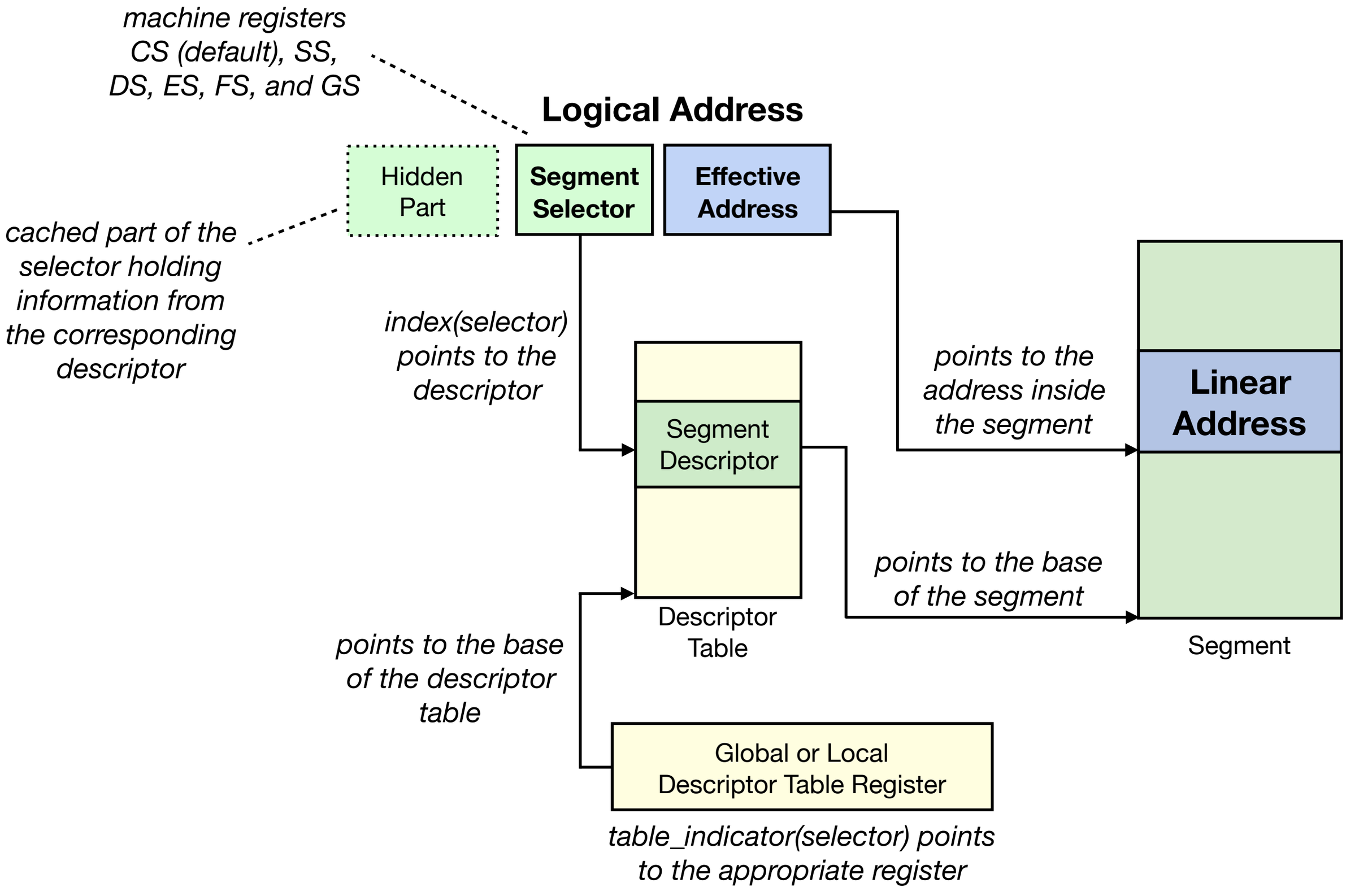
Memory Management

instructions use logical addresses

the bus uses physical addresses



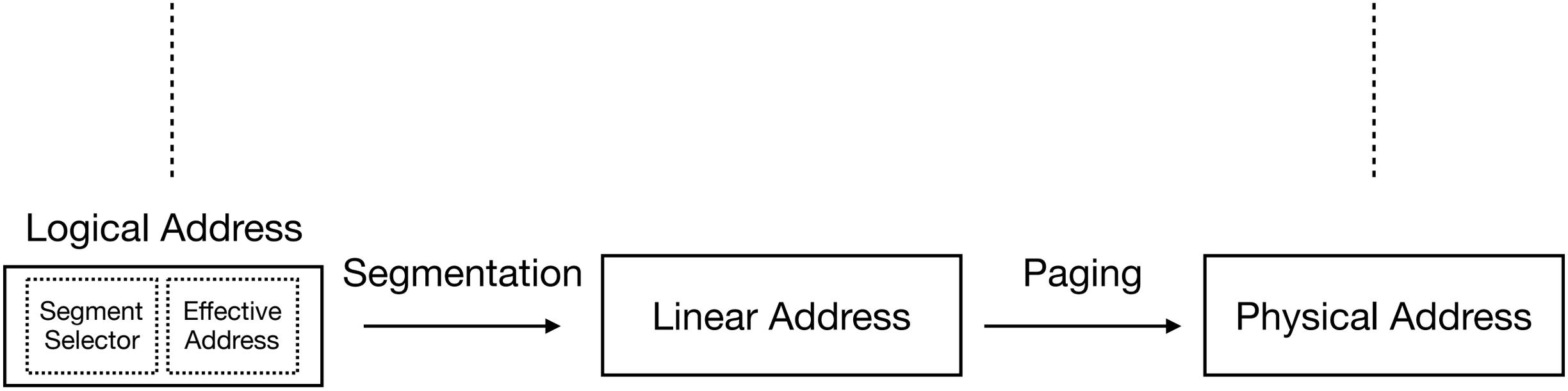
Segmentation



Memory Management

instructions use logical addresses

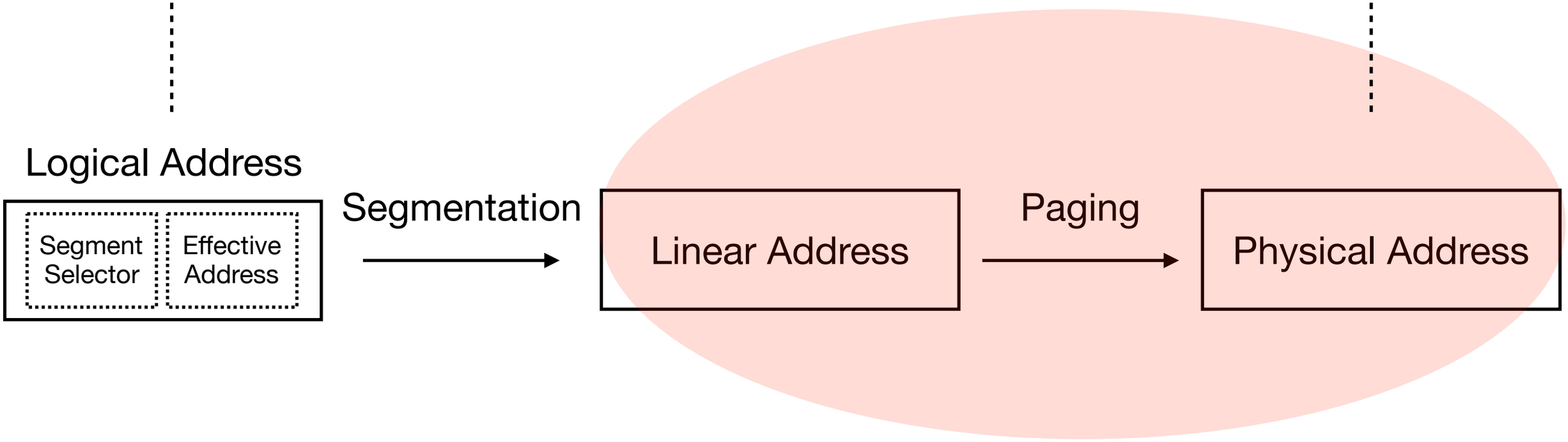
the bus uses physical addresses



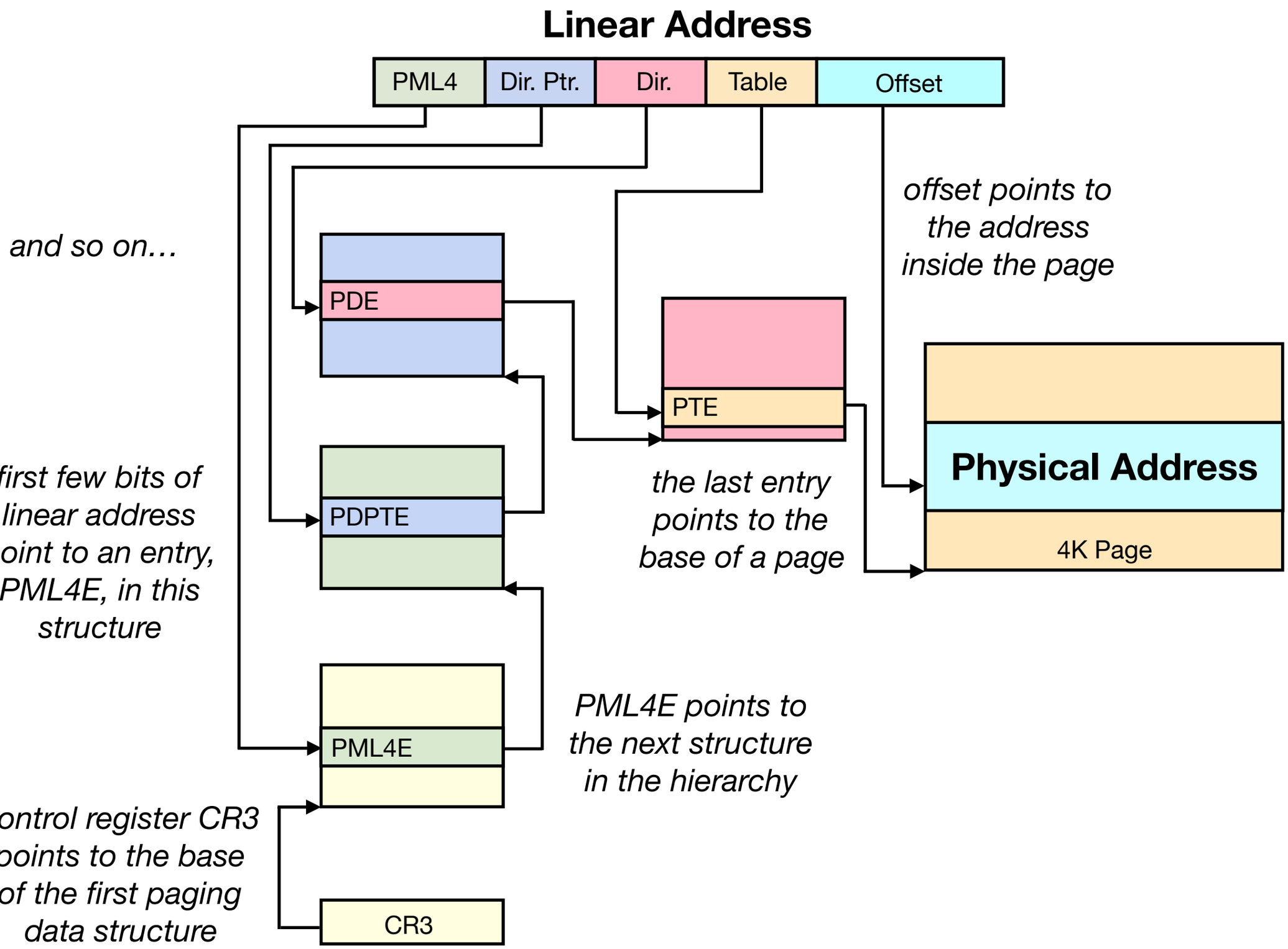
Memory Management

instructions use logical addresses

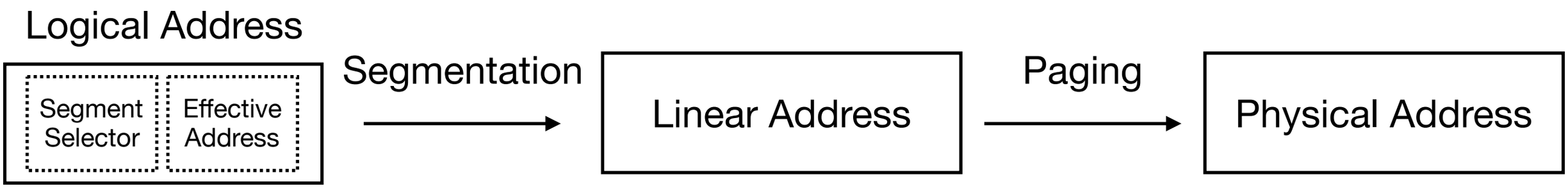
the bus uses physical addresses



IA-32e Paging (4K Pages)

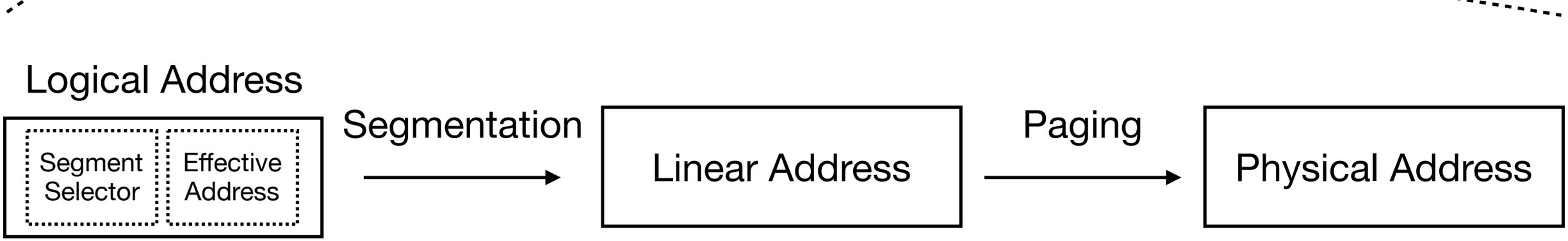


Memory Management



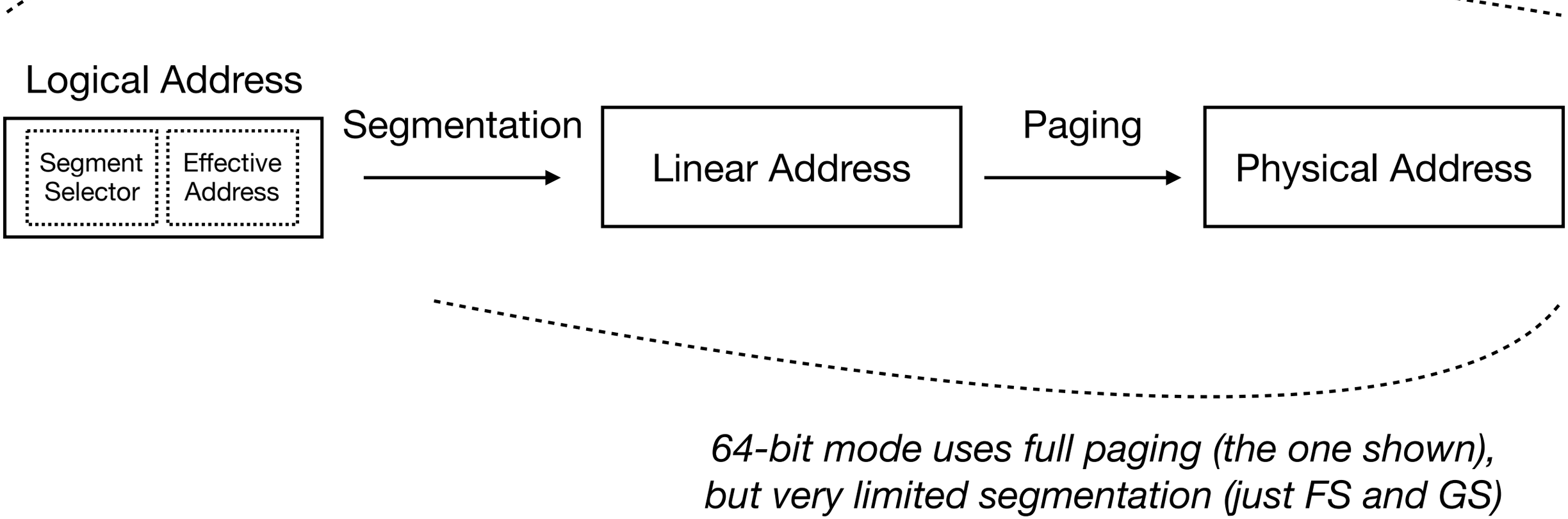
Memory Management

32-bit mode uses full segmentation and paging
(with different paging modes than the one shown)



Memory Management

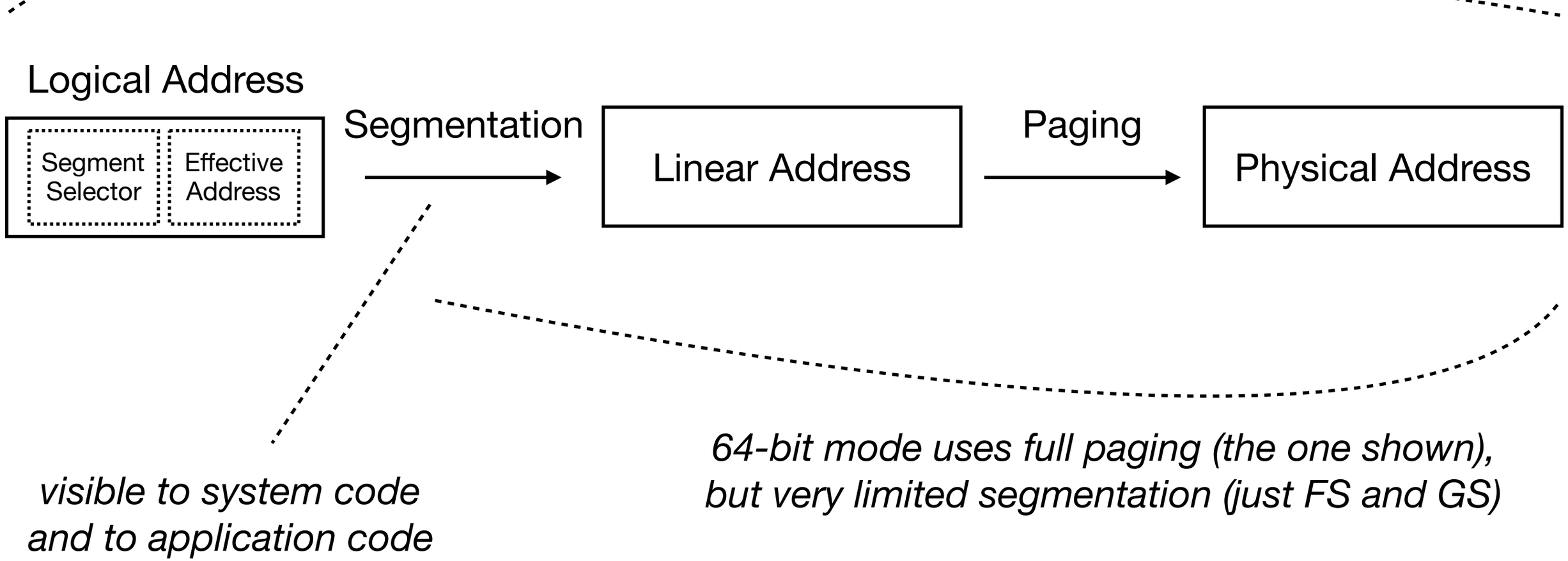
32-bit mode uses full segmentation and paging
(with different paging modes than the one shown)



64-bit mode uses full paging (the one shown),
but very limited segmentation (just FS and GS)

Memory Management

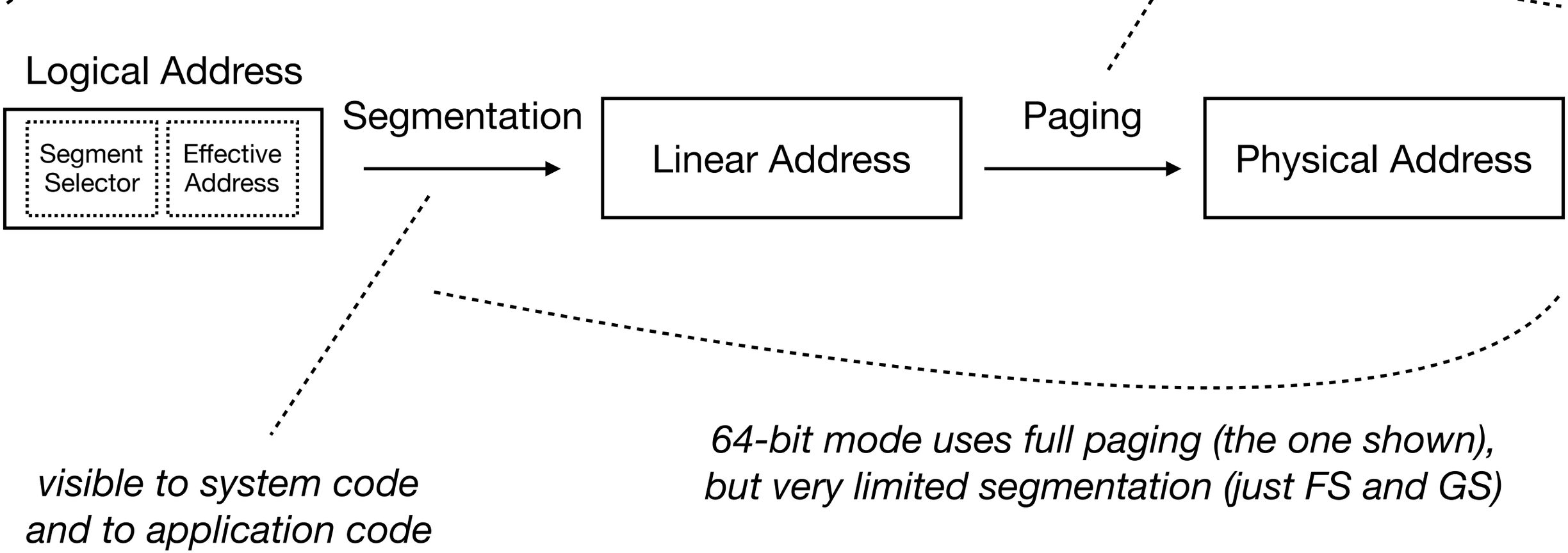
32-bit mode uses full segmentation and paging
(with different paging modes than the one shown)



Memory Management

32-bit mode uses full segmentation and paging
(with different paging modes than the one shown)

visible to system code
but not to application code



visible to system code
and to application code

64-bit mode uses full paging (the one shown),
but very limited segmentation (just FS and GS)

X86ISA: The ACL2 Formal Model of the x86 ISA

All measurements done on an Intel Xeon E31280 CPU @ 3.50GHz with 32GB RAM.

X86ISA: The ACL2 Formal Model of the x86 ISA

- Number of instructions: 413 (e.g., arithmetic, floating-point, control-flow, some system-mode opcodes).
  - See `:doc x86isa::implemented-opcodes`.

All measurements done on an Intel Xeon E31280 CPU @ 3.50GHz with 32GB RAM.

X86ISA: The ACL2 Formal Model of the x86 ISA

- Number of instructions: 413 (e.g., arithmetic, floating-point, control-flow, some system-mode opcodes).
  - See `:doc x86isa::implemented-opcodes`.

- Simulation speed in instructions/second:
  - Application programs: ~3.3 million.
  - System programs: ~320,000 (with 1G paging).

All measurements done on an Intel Xeon E31280 CPU @ 3.50GHz with 32GB RAM.

X86ISA: The ACL2 Formal Model of the x86 ISA

- Number of instructions: 413 (e.g., arithmetic, floating-point, control-flow, some system-mode opcodes).
  - See `:doc x86isa::implemented-opcodes`.

- Simulation speed in instructions/second:
  - Application programs: ~3.3 million.
  - System programs: ~320,000 (with 1G paging).

- Some 64-bit programs verified using X86ISA:
  - Application programs: bit count, word count, array copy.
  - System program: zero copy.

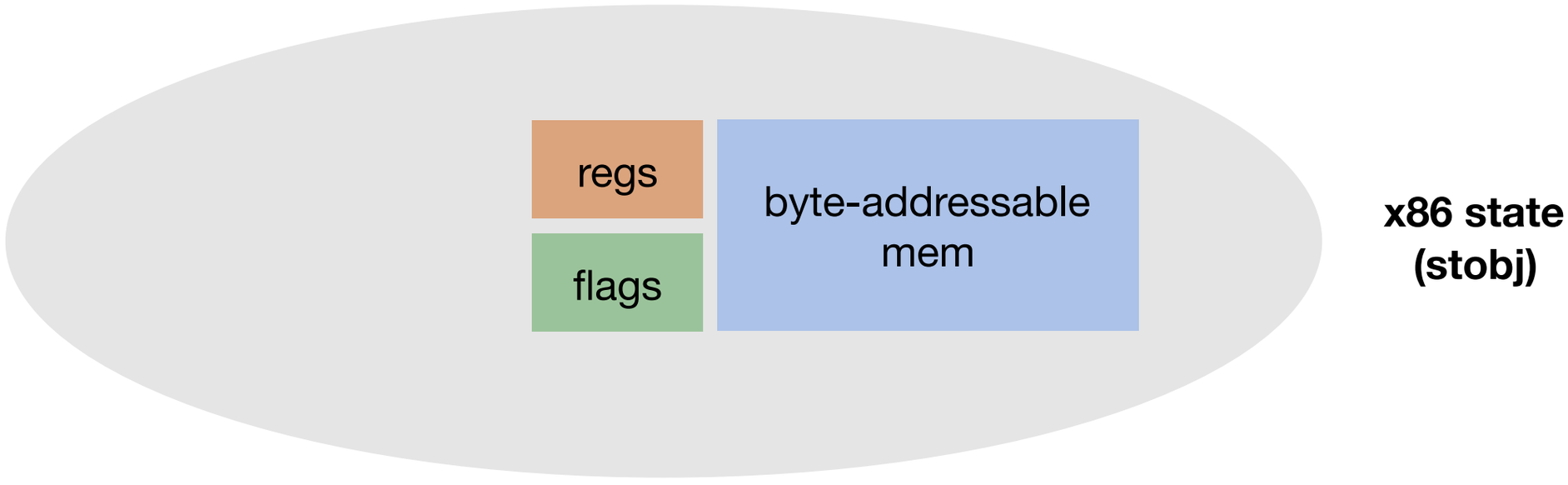
All measurements done on an Intel Xeon E31280 CPU @ 3.50GHz with 32GB RAM.

X86ISA: Overview

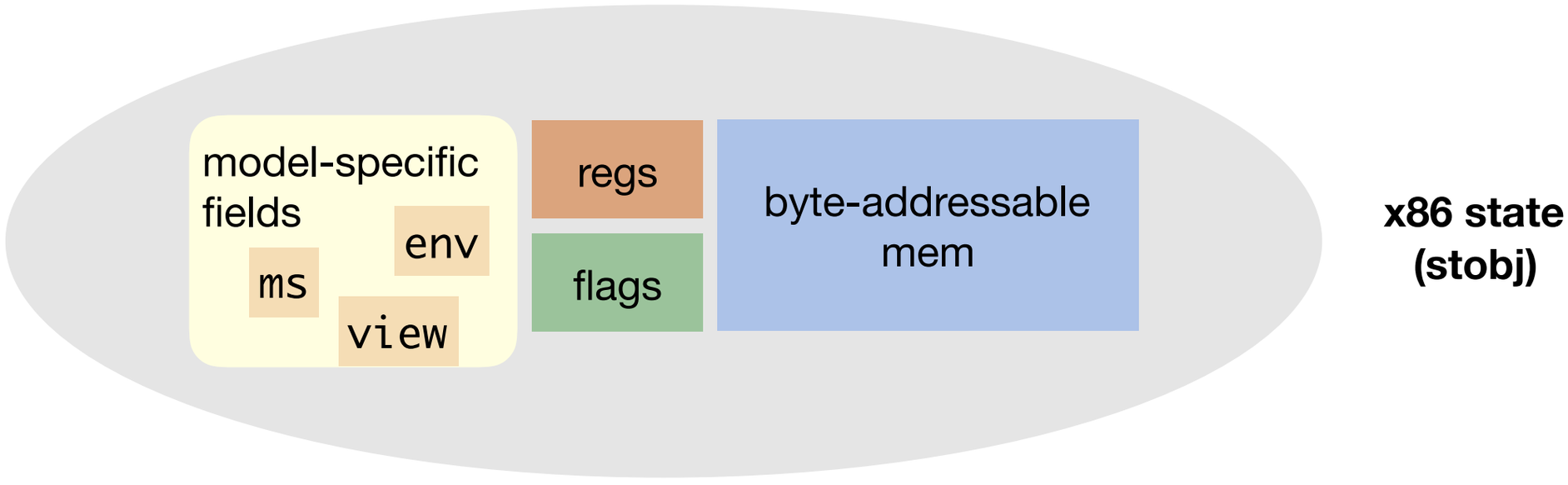
X86ISA: Overview

x86 state (stobj)

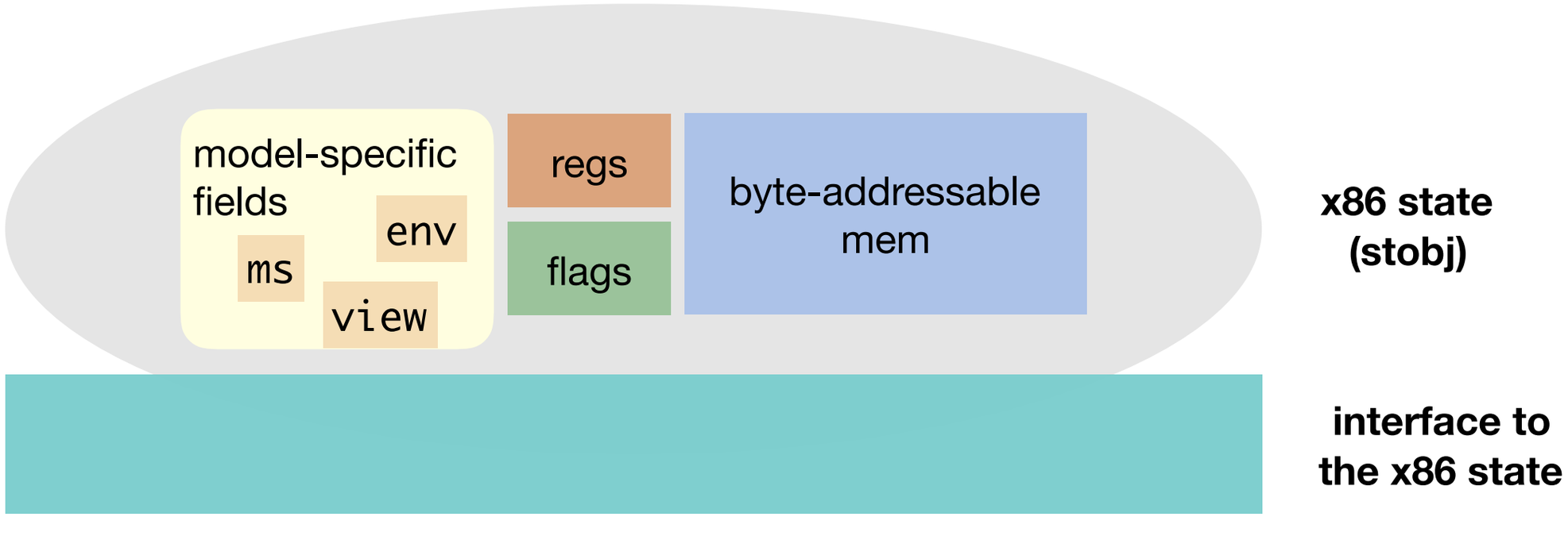
X86ISA: Overview



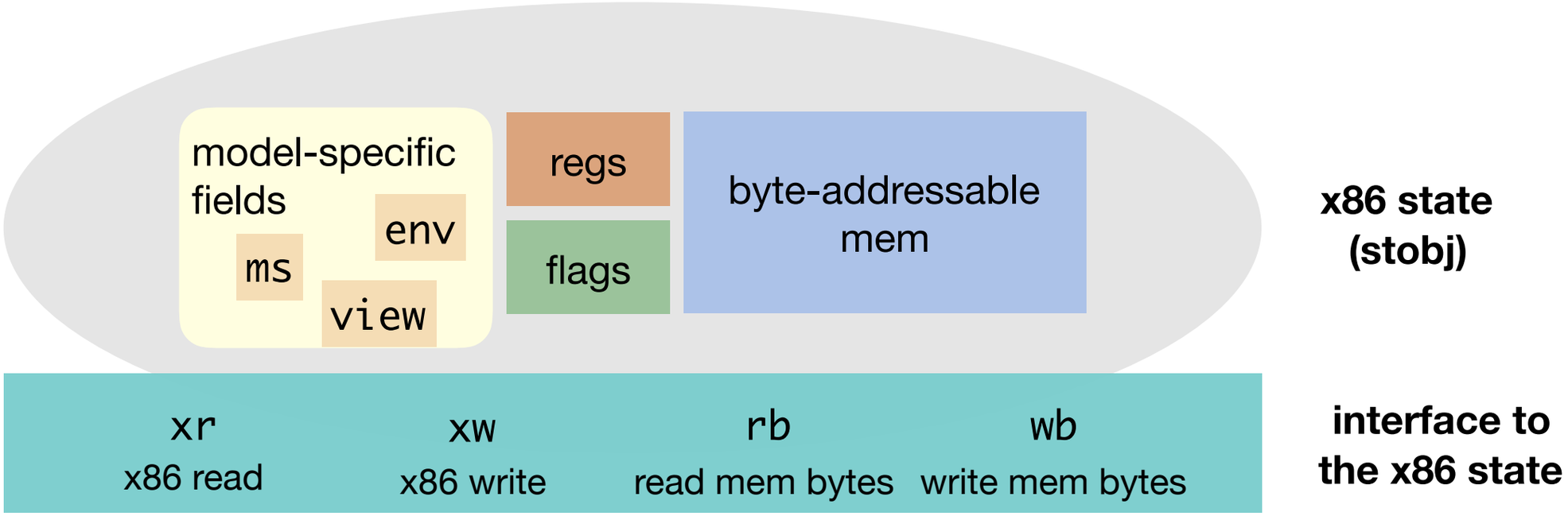
X86ISA: Overview



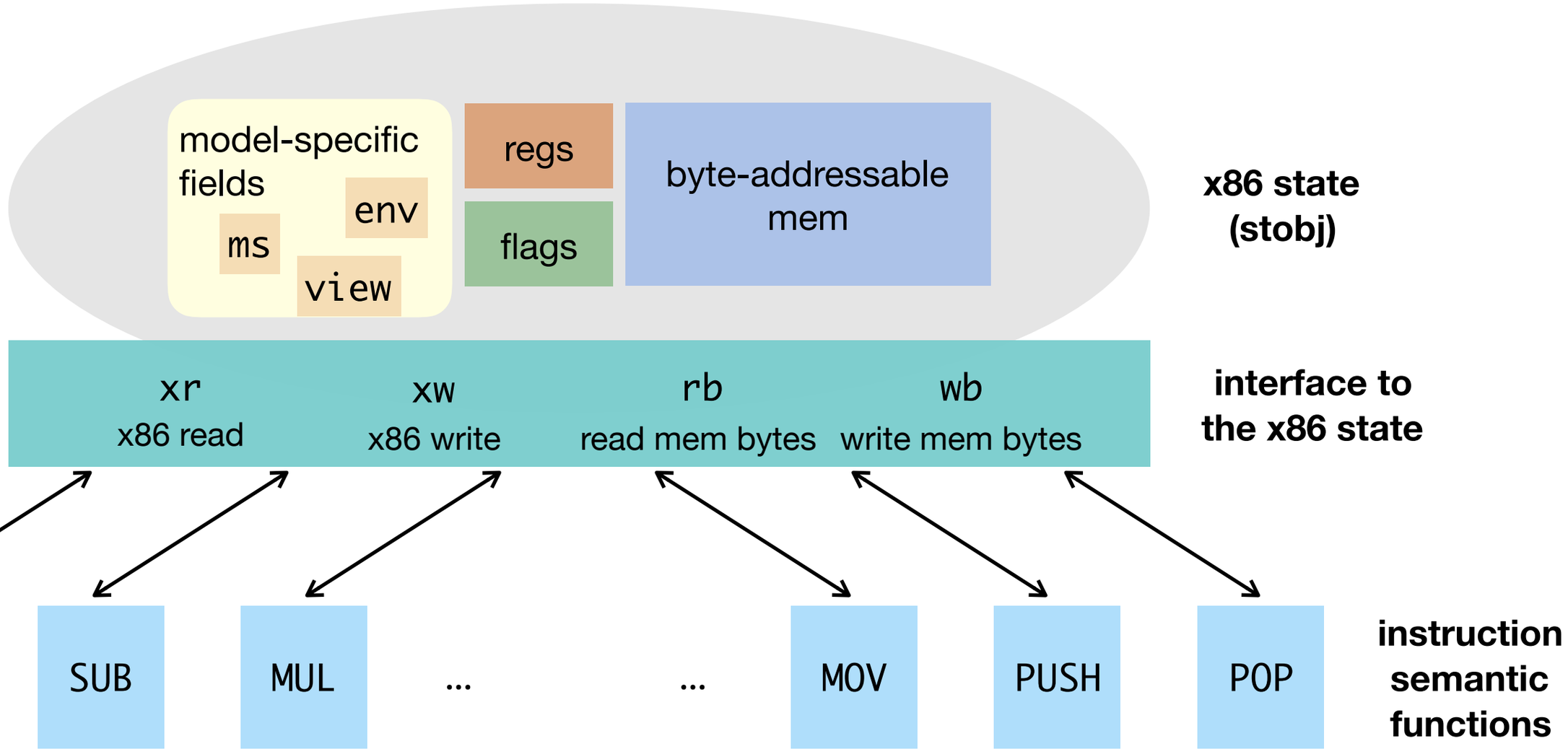
X86ISA: Overview



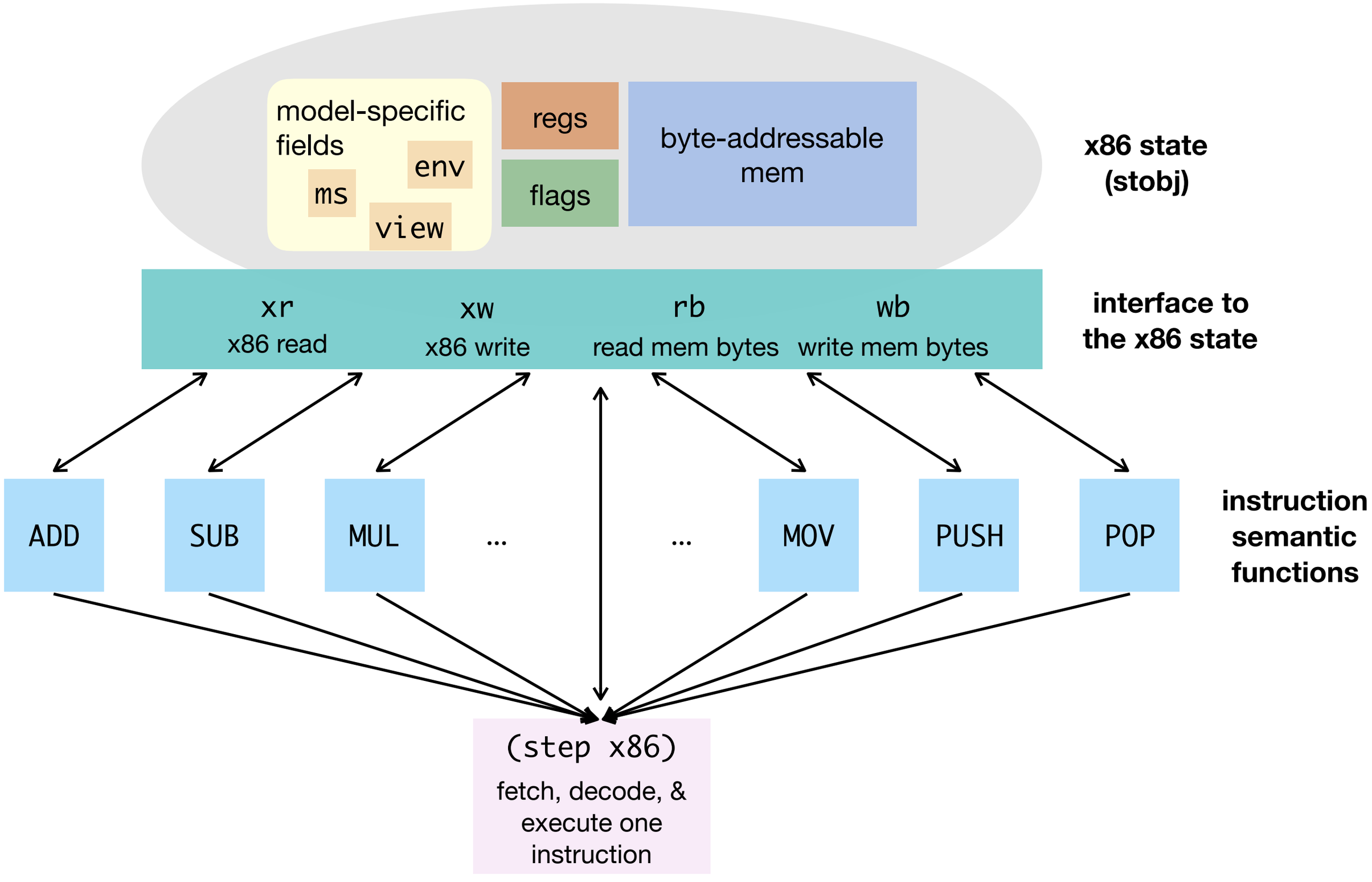
X86ISA: Overview



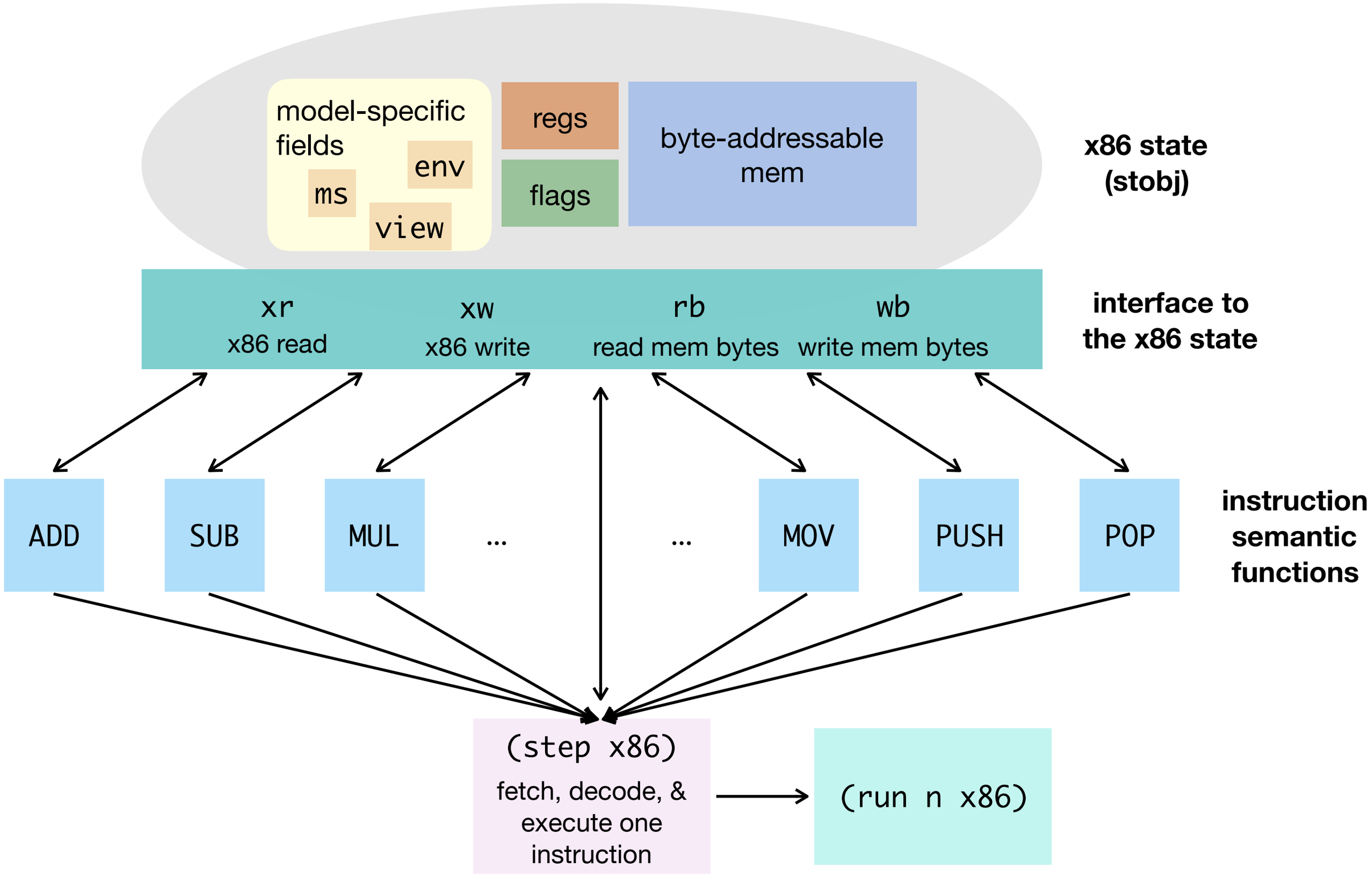
X86ISA: Overview



X86ISA: Overview



X86ISA: Overview



X86ISA: Views

Modes of Operation of the Model (NOT of the Processor)

Application View

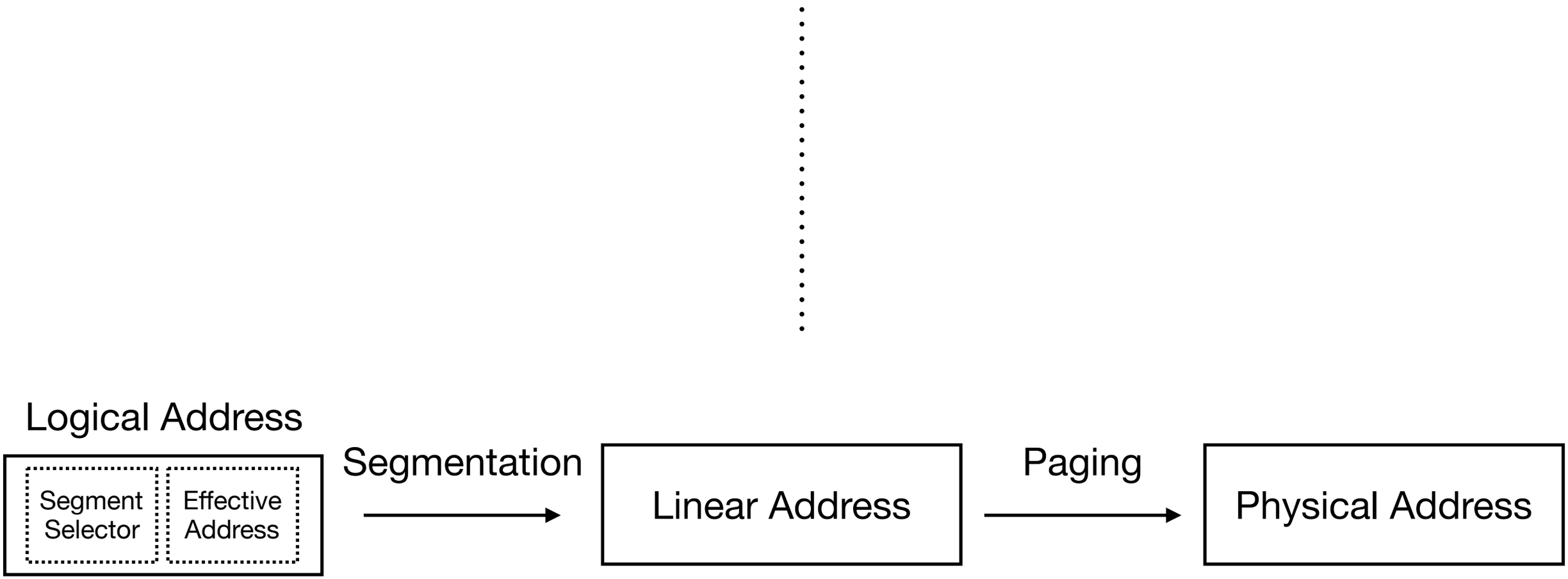
System View

X86ISA: Views

Modes of Operation of the Model (NOT of the Processor)

Application View

System View



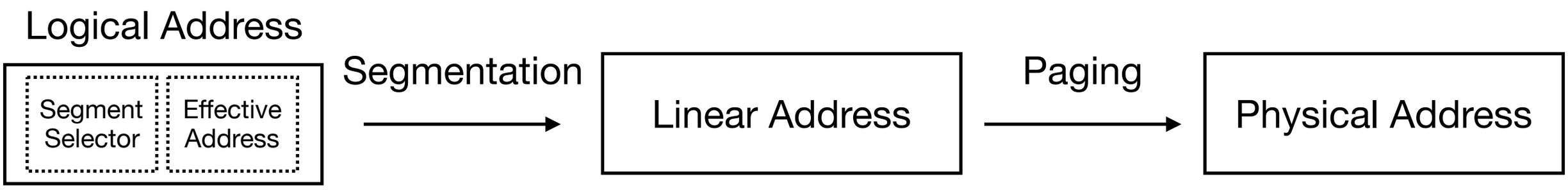
X86ISA: Views

Modes of Operation of the Model (NOT of the Processor)

Application View

- Lowest level of memory address: **linear address.**

System View



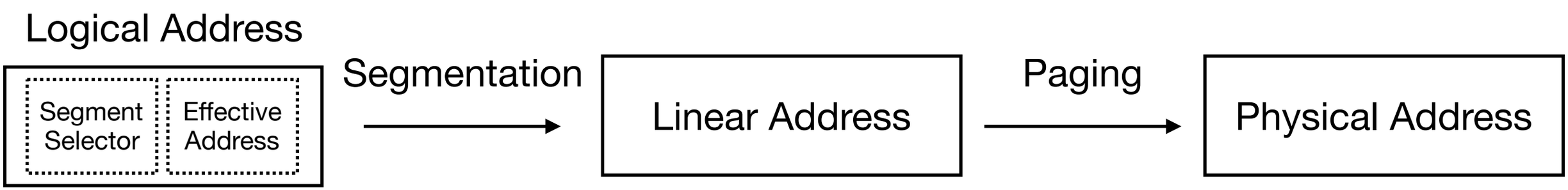
X86ISA: Views

Modes of Operation of the Model (NOT of the Processor)

Application View

- Lowest level of memory address: **linear address.**
  - User-level segmentation visible. Access only to segment selector and its hidden part; none to segmentation data structures.

System View



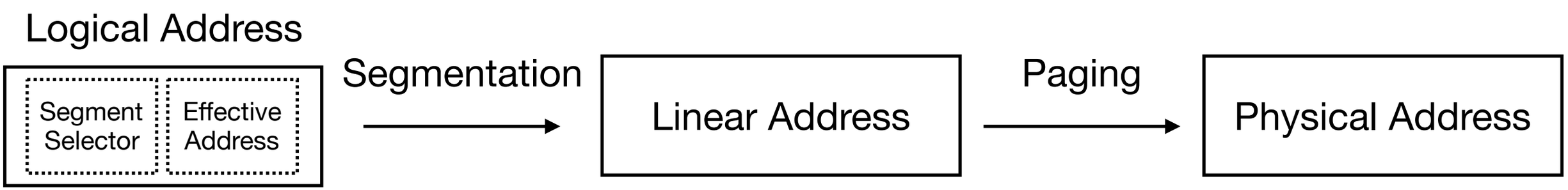
X86ISA: Views

Modes of Operation of the Model (NOT of the Processor)

Application View

- Lowest level of memory address: **linear address.**
  - User-level segmentation visible. Access only to segment selector and its hidden part; none to segmentation data structures.
  - Paging abstracted away.

System View



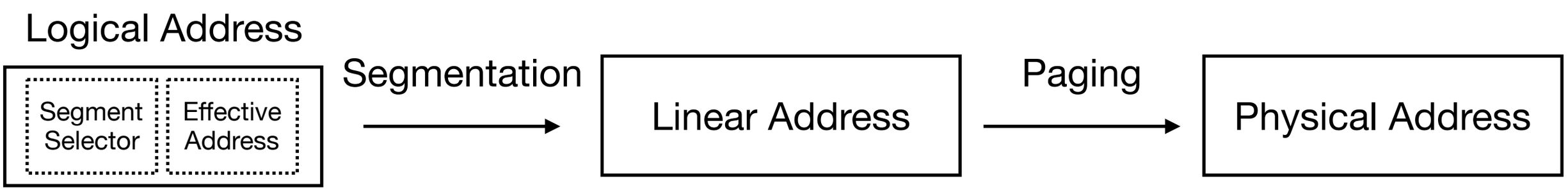
X86ISA: Views

Modes of Operation of the Model (NOT of the Processor)

Application View

- Lowest level of memory address: **linear address**.
  - User-level segmentation visible. Access only to segment selector and its hidden part; none to segmentation data structures.
  - Paging abstracted away.
- **Suitable** level of abstraction for verification of application programs.

System View



X86ISA: Views

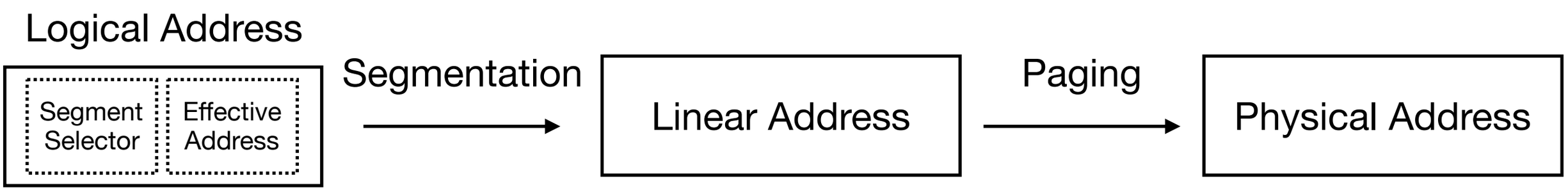
Modes of Operation of the Model (NOT of the Processor)

Application View

- Lowest level of memory address: **linear address**.
  - User-level segmentation visible. Access only to segment selector and its hidden part; none to segmentation data structures.
  - Paging abstracted away.
- **Suitable** level of abstraction for verification of application programs.

System View

- Lowest level of memory address: **physical address**.



X86ISA: Views

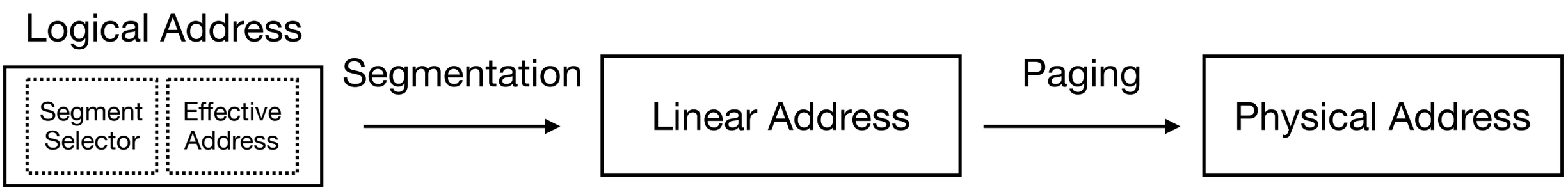
Modes of Operation of the Model (NOT of the Processor)

Application View

- Lowest level of memory address: **linear address**.
  - User-level segmentation visible. Access only to segment selector and its hidden part; none to segmentation data structures.
  - Paging abstracted away.
- **Suitable** level of abstraction for verification of application programs.

System View

- Lowest level of memory address: **physical address**.
  - Full access to segmentation and paging data structures.



X86ISA: Views

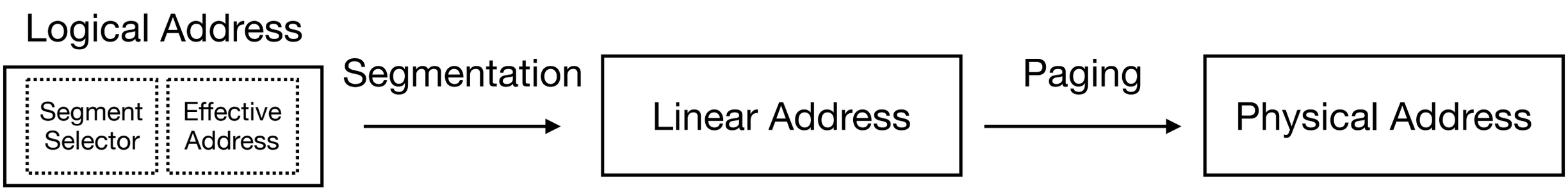
Modes of Operation of the Model (NOT of the Processor)

Application View

- Lowest level of memory address: **linear address**.
  - User-level segmentation visible. Access only to segment selector and its hidden part; none to segmentation data structures.
  - Paging abstracted away.
- **Suitable** level of abstraction for verification of application programs.

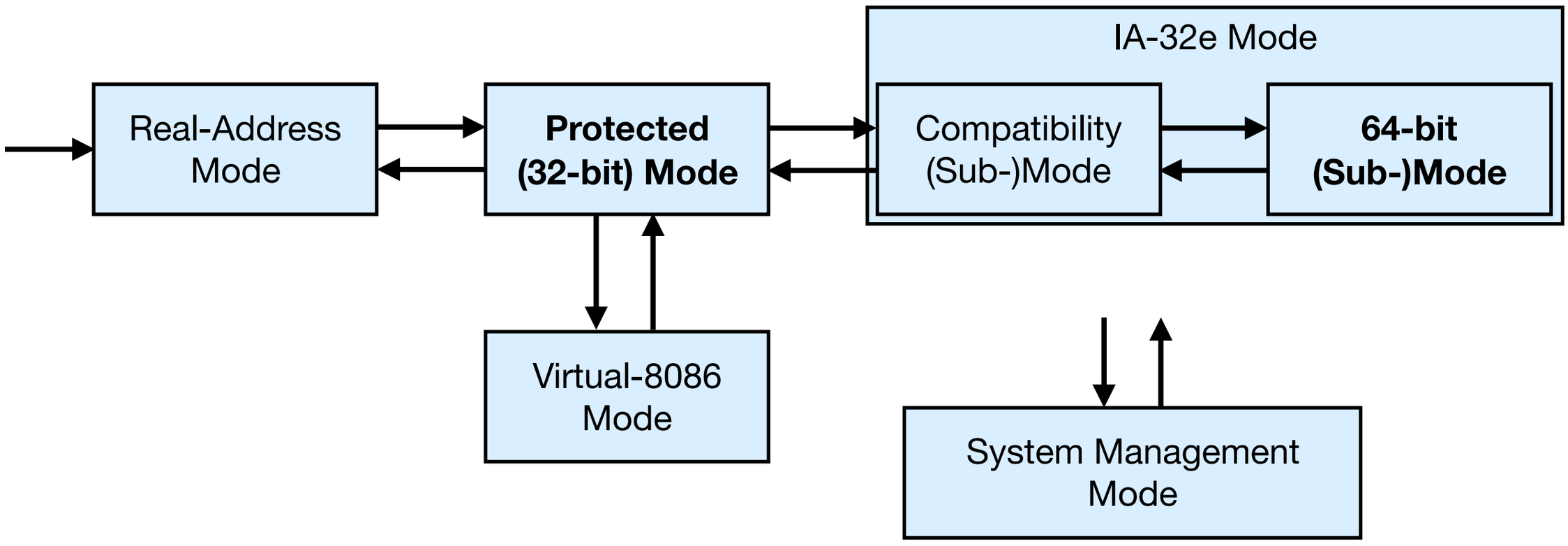
System View

- Lowest level of memory address: **physical address**.
  - Full access to segmentation and paging data structures.
- **Necessary** level of operation for verification of system programs.

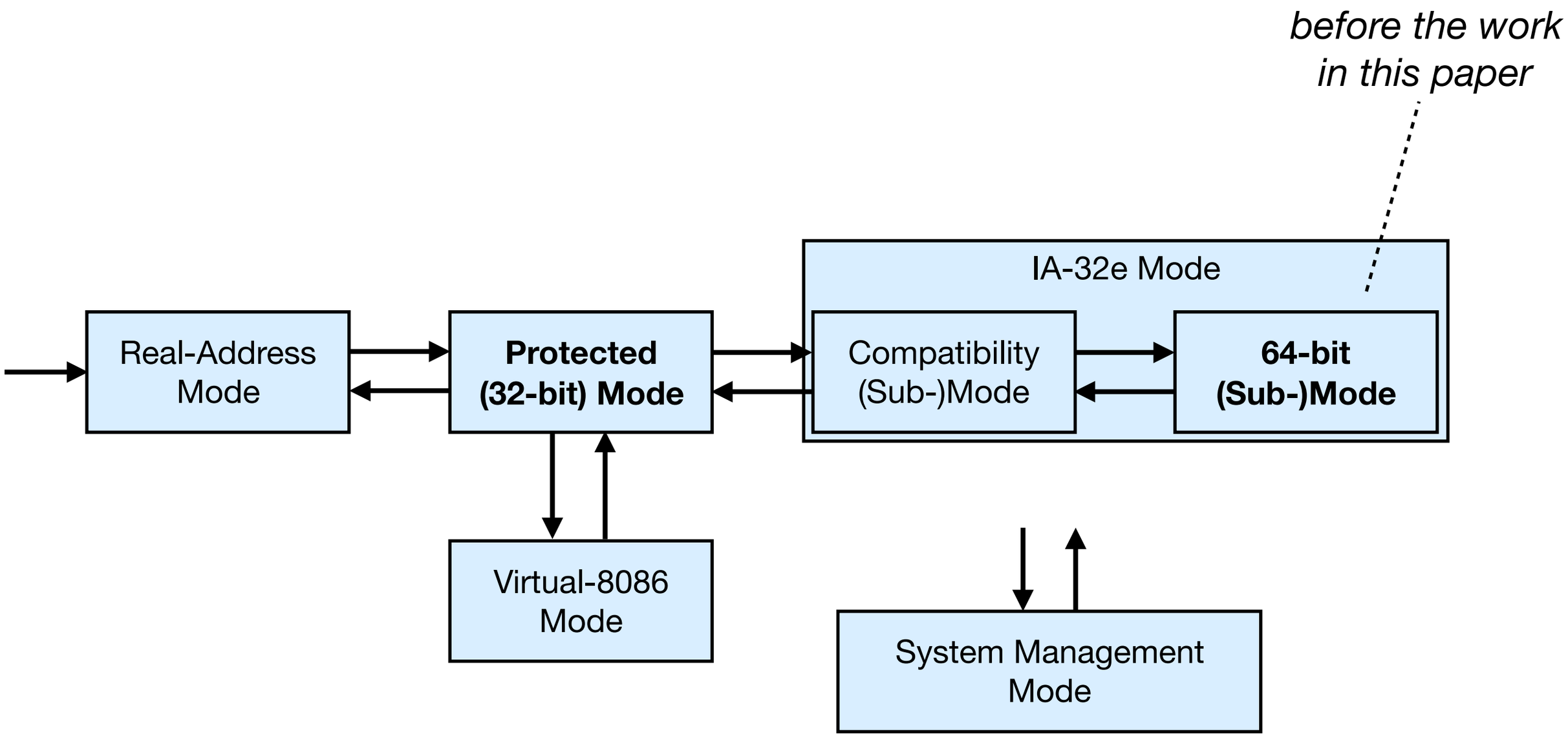


Coverage of the Model

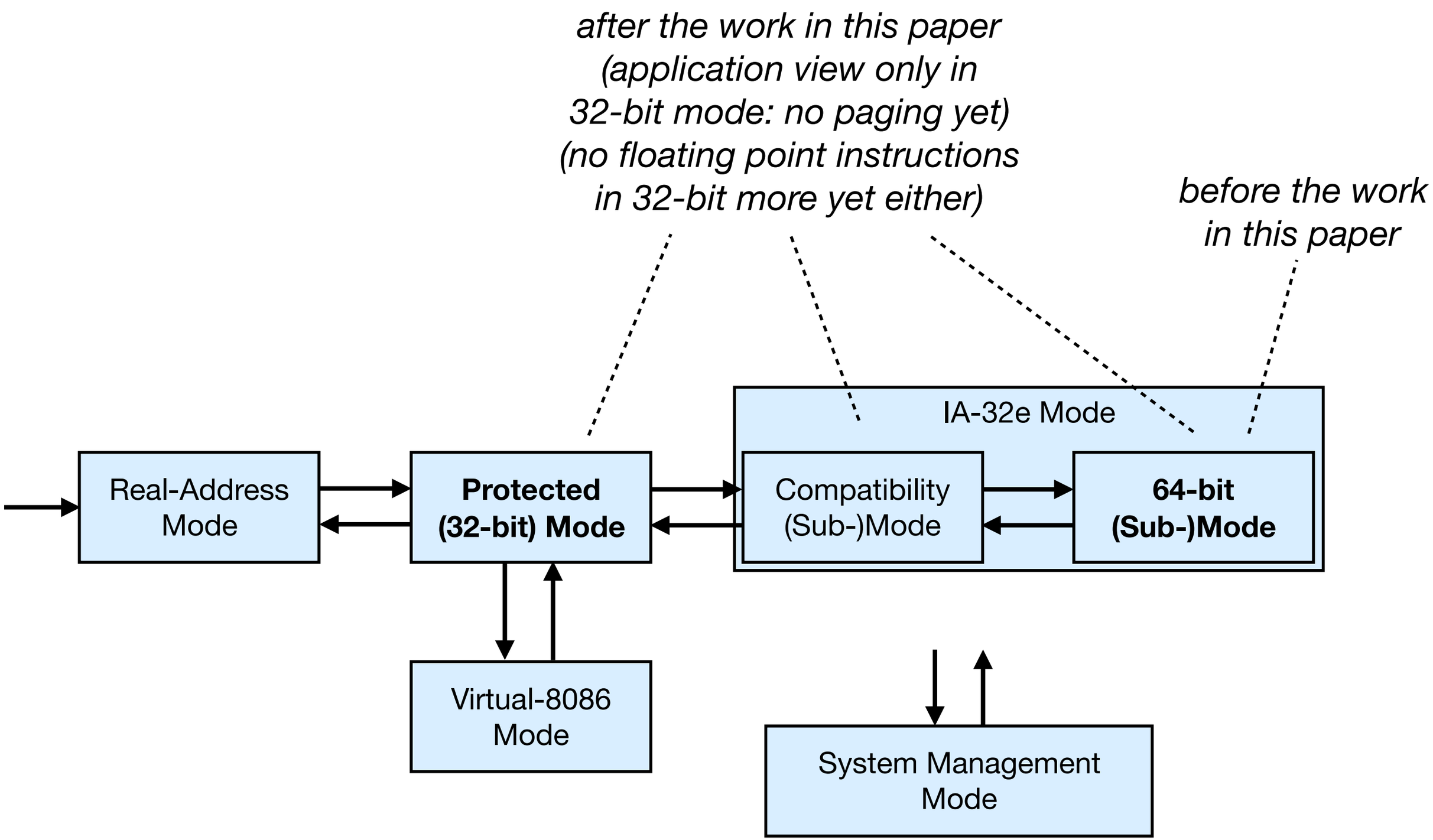
Coverage of the Model



Coverage of the Model



Coverage of the Model



Challenges of Extending the Model to 32-bit Mode

Challenges of Extending the Model to 32-bit Mode

- Much more than generalizing the sizes of operands and addresses manipulated by instructions.

Challenges of Extending the Model to 32-bit Mode

- Much more than generalizing the sizes of operands and addresses manipulated by instructions.

- Memory accesses are more complicated in 32-bit mode.

Challenges of Extending the Model to 32-bit Mode

- Much more than generalizing the sizes of operands and addresses manipulated by instructions.

- Memory accesses are more complicated in 32-bit mode.

- Add full (application-visible) segmentation.

Challenges of Extending the Model to 32-bit Mode

- Much more than generalizing the sizes of operands and addresses manipulated by instructions.

- Memory accesses are more complicated in 32-bit mode.

- Add full (application-visible) segmentation.

- Make small, incremental changes.

Challenges of Extending the Model to 32-bit Mode

- Much more than generalizing the sizes of operands and addresses manipulated by instructions.

- Memory accesses are more complicated in 32-bit mode.

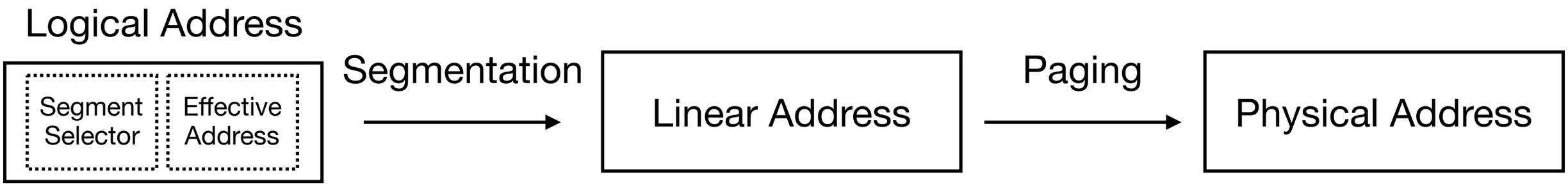
- Add full (application-visible) segmentation.

- Make small, incremental changes.

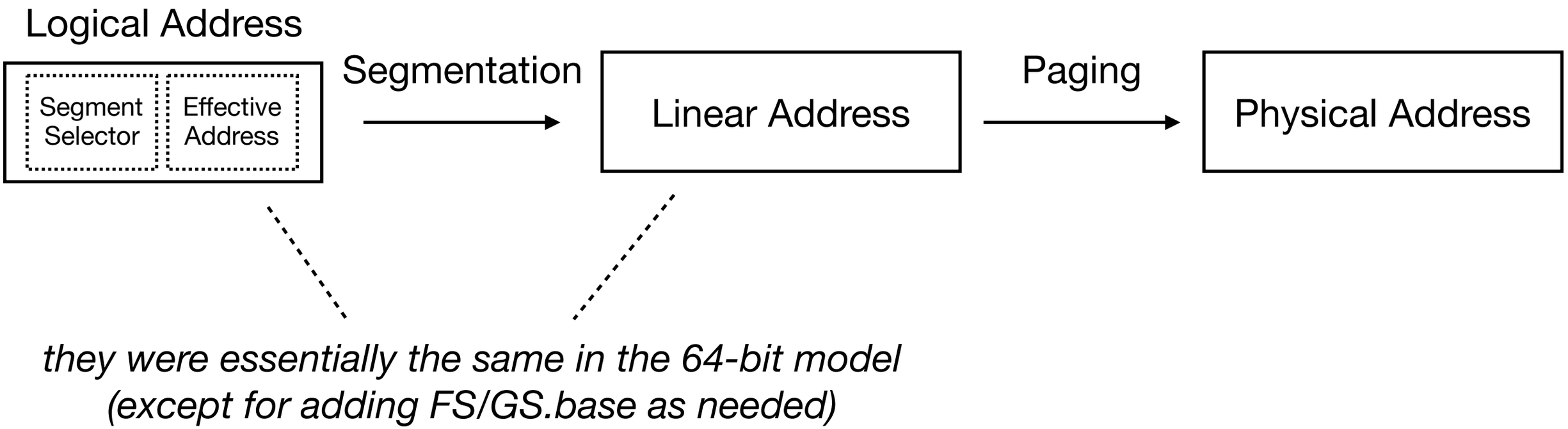
- Keep all existing proofs working — guards, return types, 64-bit programs.

Distinguish between Effective and Linear Addresses

Distinguish between Effective and Linear Addresses



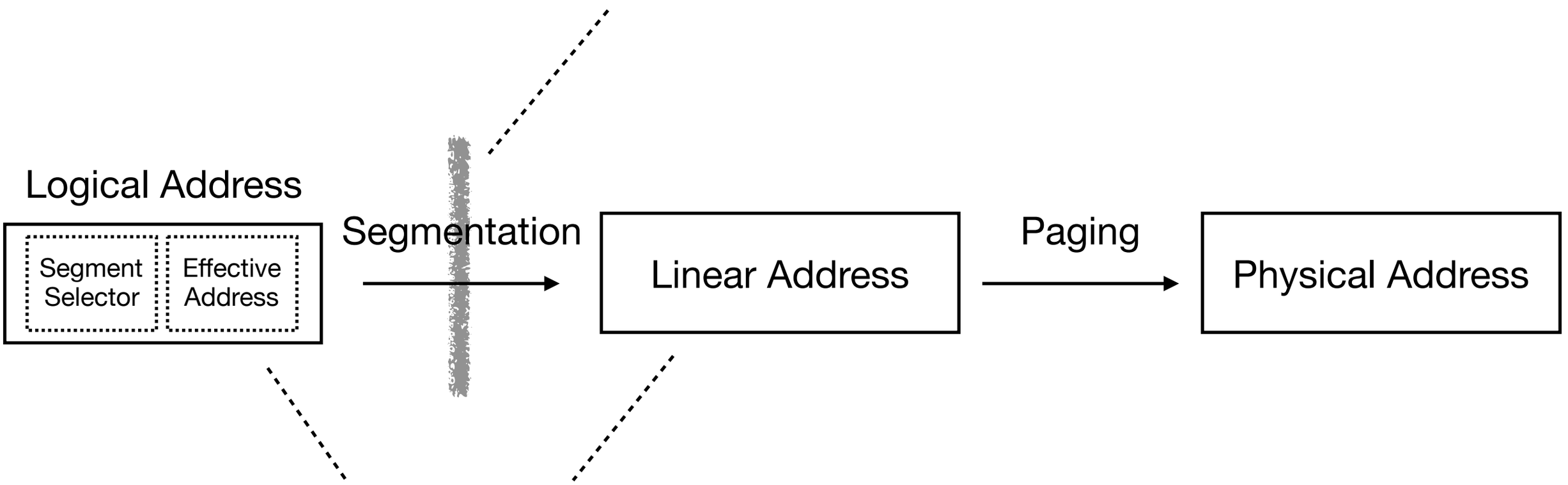
Distinguish between Effective and Linear Addresses



they were essentially the same in the 64-bit model
(except for adding FS/GS.base as needed)

Distinguish between Effective and Linear Addresses

they had to be separated
in the 64/32-bit model



they were essentially the same in the 64-bit model
(except for adding FS/GS.base as needed)

Add Mode Discrimination

64-bit model

64/32-bit model

Add Mode Discrimination

64-bit model

64/32-bit model

```
(defun 64-bit-modep (x86)
  t)
```

predicate to check whether the current mode is 64-bit (always true, rarely called)

Add Mode Discrimination

64-bit model

```
(defun 64-bit-modep (x86)
  t)
```

predicate to check whether the current mode is 64-bit (always true, rarely called)

64/32-bit model

```
(defun 64-bit-modep (x86)
  ;; return T iff
  ;; IA32_EFER.LMA = 1
  ;; and CS.D = 1
  )
```

modify definition to check for IA-32e mode (1st condition) and 64-bit sub-mode (2nd condition)

Add Mode Discrimination

64-bit model

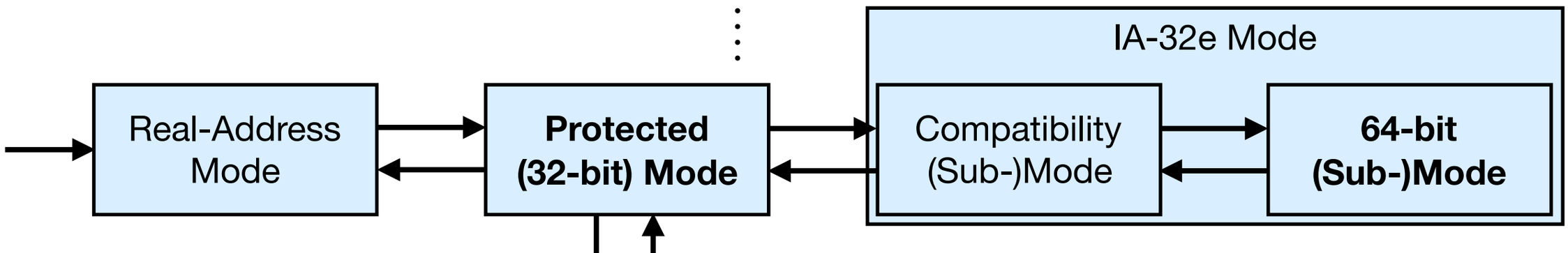
```
(defun 64-bit-modep (x86)
  t)
```

predicate to check whether the current mode is 64-bit (always true, rarely called)

64/32-bit model

```
(defun 64-bit-modep (x86)
  ;; return T iff
  ;; IA32_EFER.LMA = 1
  ;; and CS.D = 1
  )
```

modify definition to check for IA-32e mode (1st condition) and 64-bit sub-mode (2nd condition)



Add Temporary Wrappers in Top-Level Instruction Dispatch

64-bit model

64/32-bit model

Add Temporary Wrappers in Top-Level Instruction Dispatch

64-bit model

```
;; fetch and decode...
;; dispatch:
(case opcode
  (#x00 (execute-00 x86))
  (#x01 (execute-01 x86))
  ...)
```

simplified version of the actual code

64/32-bit model

Add Temporary Wrappers in Top-Level Instruction Dispatch

64-bit model

```
;; fetch and decode...
;; dispatch:
(case opcode
  (#x00 (execute-00 x86))
  (#x01 (execute-01 x86))
  ...)
```

simplified version of the actual code

64/32-bit model

```
;; fetch and decode...
;; dispatch:
(case opcode
  (#x00 (if (64-bit-modep x86)
            (execute-00 x86)
          <throw-error>))
  (#x01 (if (64-bit-modep x86)
            (execute-01 x86)
          <throw-error>))
  ...)
```

return 'unimplemented error' initially; remove wrappers as each `execute-XX` *is extended to work in 32-bit mode*

Add Translation from Logical to Linear Address

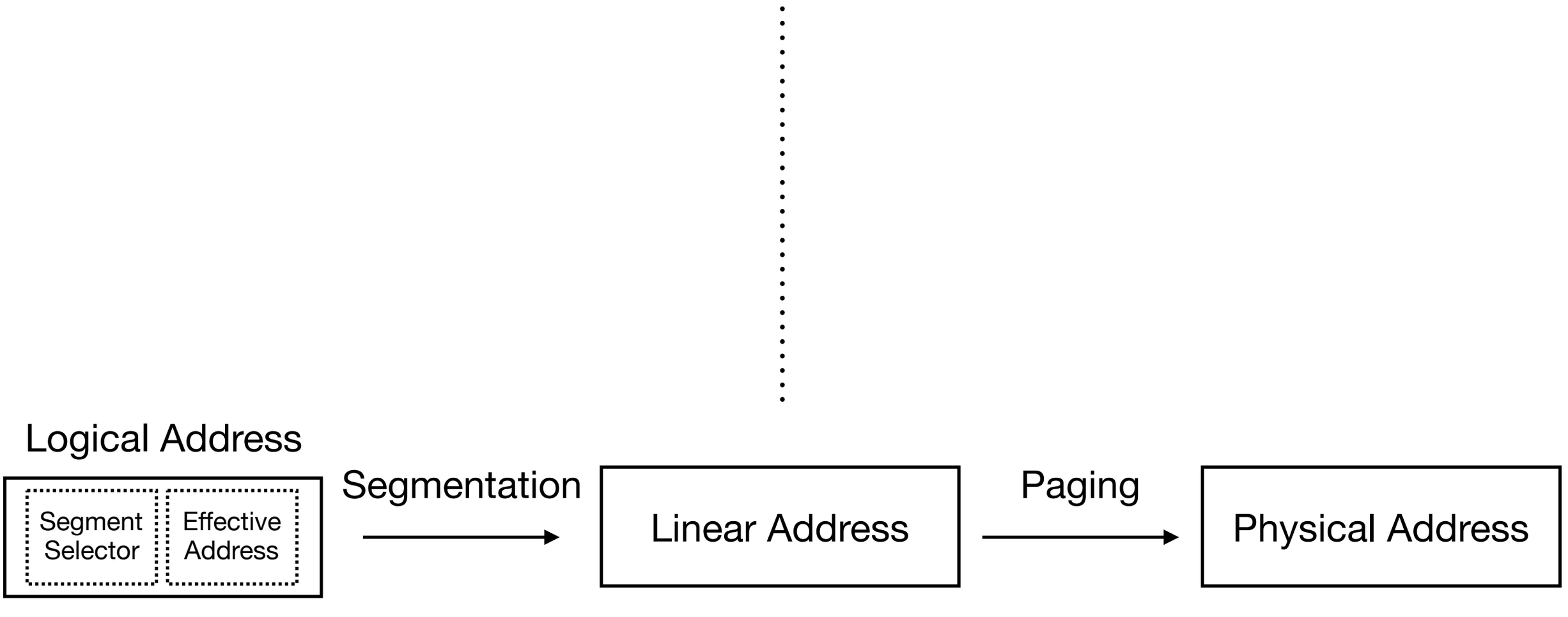
64-bit model

64/32-bit model

Add Translation from Logical to Linear Address

64-bit model

64/32-bit model



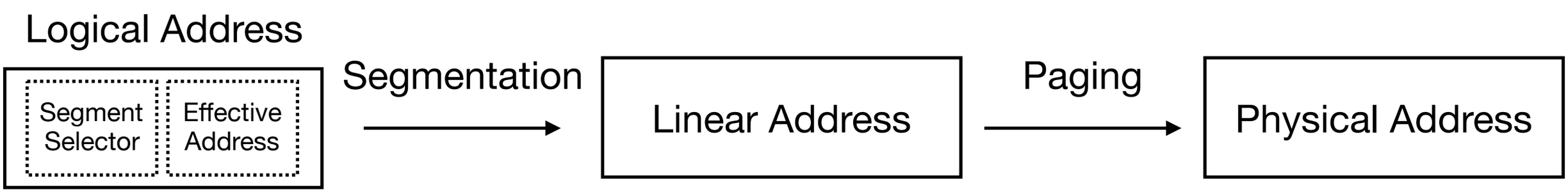
Add Translation from Logical to Linear Address

64-bit model

```
(defun la-to-pa (lin-addr r-w-x x86)
  ;; use paging (shown before)
)
```

translate linear address to physical address

64/32-bit model



Add Translation from Logical to Linear Address

64-bit model

```
(defun la-to-pa (lin-addr r-w-x x86)
  ;; use paging (shown before)
)
```

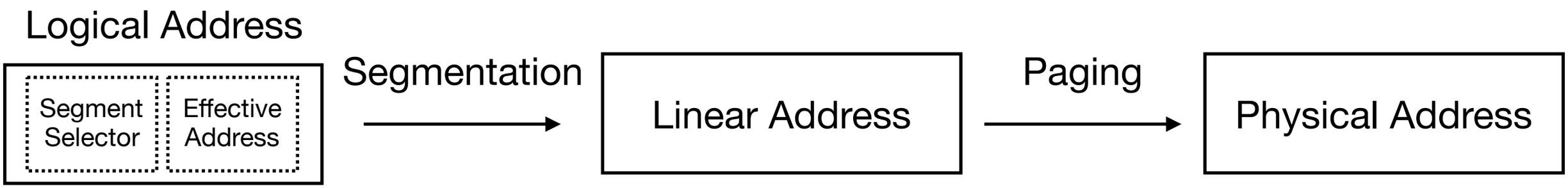
translate linear address to physical address

64/32-bit model

```
(defun la-to-pa ...) ;; unchanged

(defun ea-to-la (eff-addr seg-reg x86)
  ;; use segmentation (shown before):
  ;; retrieve segment base and bounds
  ;; (handle expand-down segments)
  ;; and add effective address to base
)
```

translate effective address, in the context of segment, to linear address



Add New Top-Level Memory Access Functions

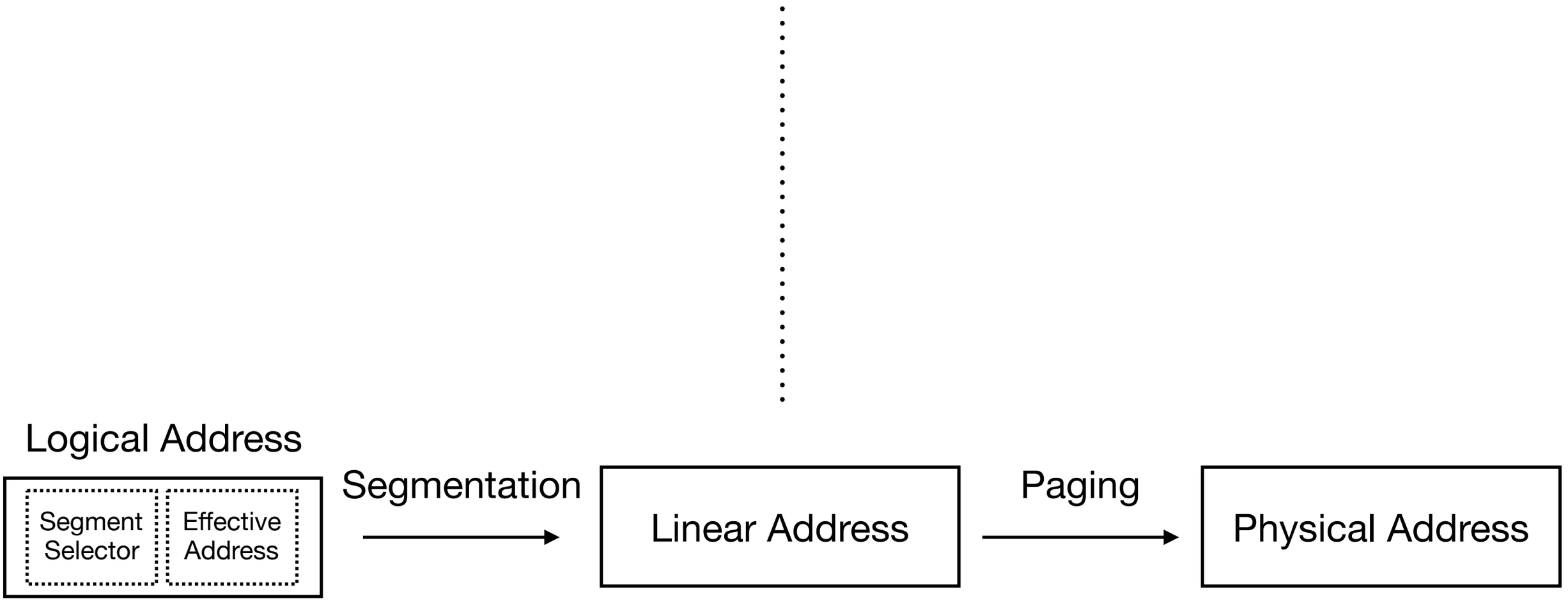
64-bit model

64/32-bit model

Add New Top-Level Memory Access Functions

64-bit model

64/32-bit model



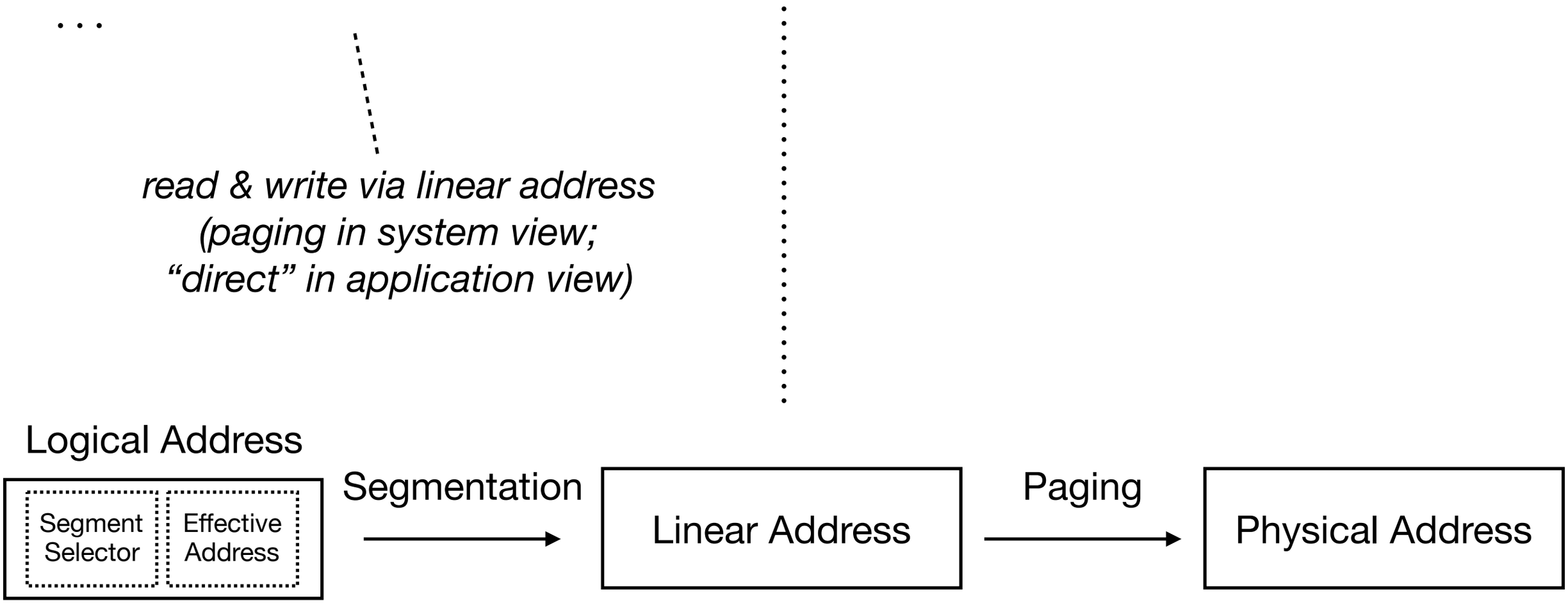
Add New Top-Level Memory Access Functions

64-bit model

```
(defun rm08 (lin-addr ...) ...)
(defun rm16 (lin-addr ...) ...)
...
(defun wm08 (lin-addr ...) ...)
(defun wm16 (lin-addr ...) ...)
...
```

read & write via linear address
(paging in system view;
“direct” in application view)

64/32-bit model



Add New Top-Level Memory Access Functions

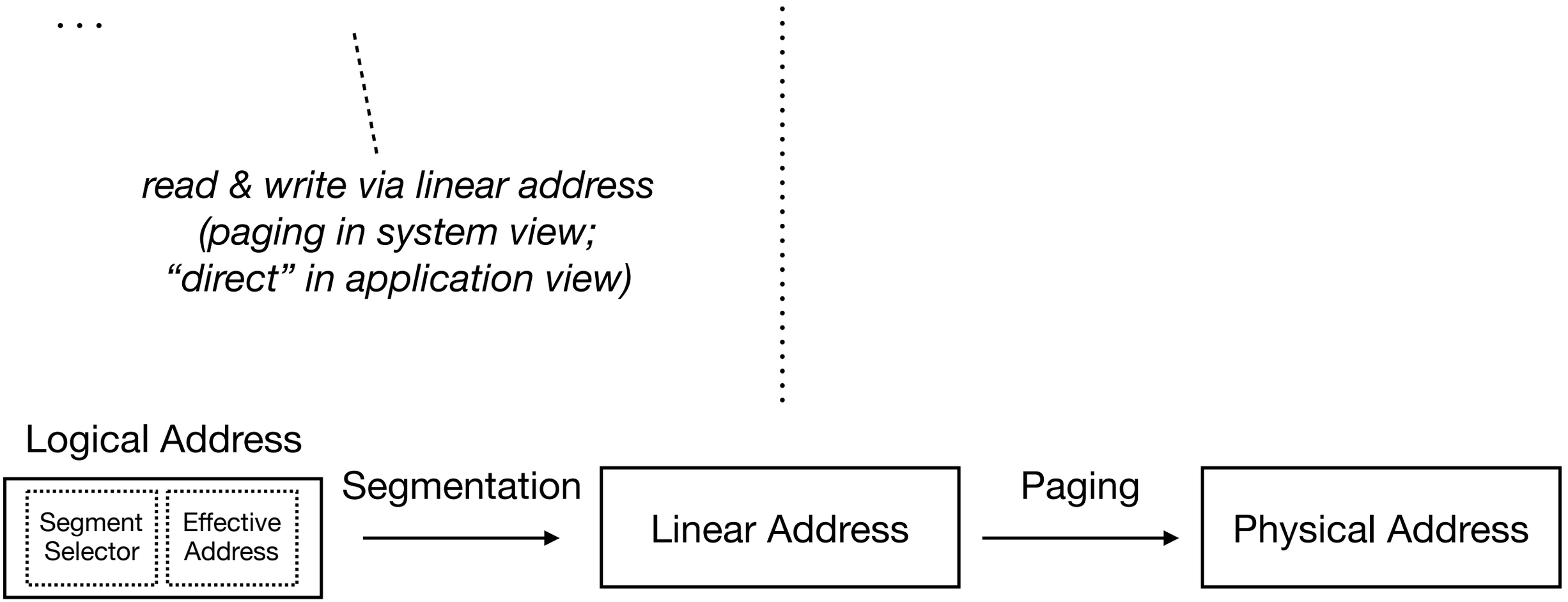
64-bit model

```
(defun rm08 (lin-addr ...) ...)
(defun rm16 (lin-addr ...) ...)
...
(defun wm08 (lin-addr ...) ...)
(defun wm16 (lin-addr ...) ...)
...
```

read & write via linear address
(paging in system view;
“direct” in application view)

64/32-bit model

```
;; unchanged but renamed:
(defun rml08 (lin-addr ...) ...)
(defun wml08 (lin-addr ...) ...)
...
```



Add New Top-Level Memory Access Functions

64-bit model

```
(defun rm08 (lin-addr ...) ...)
(defun rm16 (lin-addr ...) ...)
...
(defun wm08 (lin-addr ...) ...)
(defun wm16 (lin-addr ...) ...)
...
```

read & write via linear address
(paging in system view;
“direct” in application view)

64/32-bit model

```
;; unchanged but renamed:
(defun rml08 (lin-addr ...) ...)
(defun wml08 (lin-addr ...) ...)
...

;; new:
(defun rme08 (eff-addr ...) ...)
(defun wme08 (eff-addr ...) ...)
...
```

read & write via effective address
(call `ea-to-la` *and then*
call `rml08`, `wml08`, *...)*

Extend Instruction Fetching

64-bit model

64/32-bit model

Extend Instruction Fetching

64-bit model

```
;; read instruction pointer from RIP:
rip := (rip x86) ;; 48-bit (canonical)
```

artistic license

stobj field reader

64/32-bit model

Extend Instruction Fetching

64-bit model

```
;; read instruction pointer from RIP:
rip := (rip x86) ;; 48-bit (canonical)

;; read instruction (via lin. addr.):
opcode := (rml08 rip ...) ;; etc.
```

artistic license

stobj field reader

64/32-bit model

Extend Instruction Fetching

64-bit model

```
;; read instruction pointer from RIP:
rip := (rip x86) ;; 48-bit (canonical)
```

artistic license

stobj field reader

```
;; read instruction (via lin. addr.):
opcode := (rml08 rip ...) ;; etc.
```

```
;; increment instruction pointer:
new-rip := (+ rip delta)
;; if new-rip not canonical then fault
```

64/32-bit model

Extend Instruction Fetching

64-bit model

```
;; read instruction pointer from RIP:
rip := (rip x86) ;; 48-bit (canonical)
```

artistic license

stobj field reader

```
;; read instruction (via lin. addr.):
opcode := (rml08 rip ...) ;; etc.
```

```
;; increment instruction pointer:
new-rip := (+ rip delta)
;; if new-rip not canonical then fault
```

```
;; write instruction pointer to RIP:
x86 := (!rip new-rip x86)
```

stobj field writer

64/32-bit model

Extend Instruction Fetching

64-bit model

```
;; read instruction pointer from RIP:
rip := (rip x86) ;; 48-bit (canonical)
```

artistic license

stobj field reader

```
;; read instruction (via lin. addr.):
opcode := (rml08 rip ...) ;; etc.
```

```
;; increment instruction pointer:
new-rip := (+ rip delta)
;; if new-rip not canonical then fault
```

```
;; write instruction pointer to RIP:
x86 := (!rip new-rip x86)
```

stobj field writer

64/32-bit model

```
;; read instr. pointer from RIP/EIP/IP:
*ip := (read-*ip x86) ;; 48/32/16-bit
```

new function

Extend Instruction Fetching

64-bit model

```
;; read instruction pointer from RIP:
rip := (rip x86) ;; 48-bit (canonical)
```

artistic license

stobj field reader

```
;; read instruction (via lin. addr.):
opcode := (rml08 rip ...) ;; etc.
```

```
;; increment instruction pointer:
new-rip := (+ rip delta)
;; if new-rip not canonical then fault
```

```
;; write instruction pointer to RIP:
x86 := (!rip new-rip x86)
```

stobj field writer

64/32-bit model

```
;; read instr. pointer from RIP/EIP/IP:
*ip := (read-*ip x86) ;; 48/32/16-bit
```

new function

```
;; read instruction (via eff. addr.):
opcode := (rme08 *ip ...) ;; etc.
```

Extend Instruction Fetching

64-bit model

```
;; read instruction pointer from RIP:
rip := (rip x86) ;; 48-bit (canonical)
```

artistic license

stobj field reader

```
;; read instruction (via lin. addr.):
opcode := (rml08 rip ...) ;; etc.
```

```
;; increment instruction pointer:
new-rip := (+ rip delta)
;; if new-rip not canonical then fault
```

```
;; write instruction pointer to RIP:
x86 := (!rip new-rip x86)
```

stobj field writer

64/32-bit model

```
;; read instr. pointer from RIP/EIP/IP:
*ip := (read-*ip x86) ;; 48/32/16-bit
```

new function

```
;; read instruction (via eff. addr.):
opcode := (rme08 *ip ...) ;; etc.
```

```
;; increment instruction pointer:
new-*ip := (add-to-*ip *ip delta x86)
```

new function
(includes canonical
and segment checks)

Extend Instruction Fetching

64-bit model

```
;; read instruction pointer from RIP:
rip := (rip x86) ;; 48-bit (canonical)
```

artistic license

stobj field reader

```
;; read instruction (via lin. addr.):
opcode := (rml08 rip ...) ;; etc.
```

```
;; increment instruction pointer:
new-rip := (+ rip delta)
;; if new-rip not canonical then fault
```

```
;; write instruction pointer to RIP:
x86 := (!rip new-rip x86)
```

stobj field writer

64/32-bit model

```
;; read instr. pointer from RIP/EIP/IP:
*ip := (read-*ip x86) ;; 48/32/16-bit
```

new function

```
;; read instruction (via eff. addr.):
opcode := (rme08 *ip ...) ;; etc.
```

```
;; increment instruction pointer:
new-*ip := (add-to-*ip *ip delta x86)
```

new function
(includes canonical
and segment checks)

```
;; write instr. pointer to RIP/EIP/IP:
x86 := (write-*ip new-*ip x86)
```

new function

Other Infrastructural Extensions

Other Infrastructural Extensions

- Generalize stack manipulation analogously to instruction fetching.

Other Infrastructural Extensions

- Generalize stack manipulation analogously to instruction fetching.
- Add 16-bit addressing modes — for effective address calculation (base, index, scale, displacement, …).

Other Infrastructural Extensions

- Generalize stack manipulation analogously to instruction fetching.
- Add 16-bit addressing modes — for effective address calculation (base, index, scale, displacement, …).
- Generalize the functions to read/write memory operands.

Other Infrastructural Extensions

- Generalize stack manipulation analogously to instruction fetching.
- Add 16-bit addressing modes — for effective address calculation (base, index, scale, displacement, …).
- Generalize the functions to read/write memory operands.
  - Use effective addresses instead of linear addresses.

Other Infrastructural Extensions

- Generalize stack manipulation analogously to instruction fetching.
- Add 16-bit addressing modes — for effective address calculation (base, index, scale, displacement, …).
- Generalize the functions to read/write memory operands.
  - Use effective addresses instead of linear addresses.
  - Handle segment defaults and override prefixes.

Other Infrastructural Extensions

- Generalize stack manipulation analogously to instruction fetching.
- Add 16-bit addressing modes — for effective address calculation (base, index, scale, displacement, …).
- Generalize the functions to read/write memory operands.
  - Use effective addresses instead of linear addresses.
  - Handle segment defaults and override prefixes.
  - Use 32-bit or 16-bit addressing modes.

Other Infrastructural Extensions

- Generalize stack manipulation analogously to instruction fetching.

- Add 16-bit addressing modes — for effective address calculation (base, index, scale, displacement, …).

- Generalize the functions to read/write memory operands.
  - Use effective addresses instead of linear addresses.
  - Handle segment defaults and override prefixes.
  - Use 32-bit or 16-bit addressing modes.

- No changes to the x86 stobj were needed.

Instruction Extensions

Instruction Extensions

- Comparatively easy, after all the previous infrastructural extensions were in place.

Instruction Extensions

- Comparatively easy, after all the previous infrastructural extensions were in place.
- Extend one instruction at a time, removing each `64-bit-modep` wrapper in the top-level instruction dispatch.

Instruction Extensions

- Comparatively easy, after all the previous infrastructural extensions were in place.
- Extend one instruction at a time, removing each `64-bit-modep` wrapper in the top-level instruction dispatch.
- Generalize determination of operand, address, and stack size.

Instruction Extensions

- Comparatively easy, after all the previous infrastructural extensions were in place.

- Extend one instruction at a time, removing each `64-bit-modep` wrapper in the top-level instruction dispatch.

- Generalize determination of operand, address, and stack size.

- No changes to existing core arithmetic and logical functions, which already handled operands of different sizes.

Instruction Extensions

- Comparatively easy, after all the previous infrastructural extensions were in place.

- Extend one instruction at a time, removing each `64-bit-modep` wrapper in the top-level instruction dispatch.

- Generalize determination of operand, address, and stack size.

- No changes to existing core arithmetic and logical functions, which already handled operands of different sizes.

- Call the new or extended functions to read & write stack, immediate, and (other) memory operands.

Instruction Extensions

- Comparatively easy, after all the previous infrastructural extensions were in place.
- Extend one instruction at a time, removing each `64-bit-modep` wrapper in the top-level instruction dispatch.
- Generalize determination of operand, address, and stack size.
- No changes to existing core arithmetic and logical functions, which already handled operands of different sizes.
- Call the new or extended functions to read & write stack, immediate, and (other) memory operands.
- Slightly better code factoring as a byproduct (e.g. alignment checks).

Proof Adaptations: Add 64-bit Mode Hypotheses

64-bit model

64/32-bit model

Proof Adaptations: Add 64-bit Mode Hypotheses

64-bit model

```
(defthm program-is-correct
  formula<(run ... x86)>)
```

an existing theorem about a 64-bit program

64/32-bit model

Proof Adaptations:
Add 64-bit Mode Hypotheses

64-bit model

```
(defthm program-is-correct
  formula<(run ... x86)>)
```

an existing theorem about a 64-bit program

```
(defun run ... step ...)
```

64/32-bit model

Proof Adaptations: Add 64-bit Mode Hypotheses

64-bit model

```
(defthm program-is-correct
  formula<(run ... x86)>)
```

an existing theorem about a 64-bit program

```
(defun run ... step ...)

(defun step (x86)
  ;; fetch and decode...
  (case opcode
    (#x00 (execute-00 x86))
  ...)
```

simplified code (shown before)

64/32-bit model

Proof Adaptations: Add 64-bit Mode Hypotheses

64-bit model

```
(defthm program-is-correct
  formula<(run ... x86)>)
```

an existing theorem about a 64-bit program

```
(defun run ... step ...)

(defun step (x86)
  ;; fetch and decode...
  (case opcode
    (#x00 (execute-00 x86))
  ...)
```

simplified code (shown before)

64/32-bit model

```
(defun run ... step ...)

(defun step (x86)
  ;; fetch and decode...
  (case opcode
    (#x00 (if (64-bit-modep x86)
              (execute-00 x86)
            <throw-error>)
  ...)
```

our initial wrapping in the top-level dispatch (shown before)

Proof Adaptations: Add 64-bit Mode Hypotheses

64-bit model

```
(defthm program-is-correct
  formula<(run ... x86)>)
```

an existing theorem about a 64-bit program

```
(defun run ... step ...)

(defun step (x86)
  ;; fetch and decode...
  (case opcode
    (#x00 (execute-00 x86))
    ...)
```

simplified code (shown before)

64/32-bit model

```
(defthm program-is-correct
  (implies (64-bit-modep x86)
           formula<(run ... x86)>))
```

add 64-bit mode hypotheses to this theorem and many lemmas

```
(defun run ... step ...)

(defun step (x86)
  ;; fetch and decode...
  (case opcode
    (#x00 (if (64-bit-modep x86)
              (execute-00 x86)
            <throw-error>))
    ...)
```

our initial wrapping in the top-level dispatch (shown before)

Proof Adaptations: Add “Reduction” Rules

64-bit model

64/32-bit model

Proof Adaptations: Add “Reduction” Rules

64-bit model

same as before

64/32-bit model

```
(defthm program-is-correct
  formula<(run ... x86)>)
```

```
(defthm program-is-correct
  (implies (64-bit-modep x86)
           formula<(run ... x86)>))
```

Proof Adaptations: Add “Reduction” Rules

64-bit model

same as before

64/32-bit model

```
(defthm program-is-correct
  formula<(run ... x86)>)
```

```
(defthm program-is-correct
  (implies (64-bit-modep x86)
           formula<(run ... x86)>))
```

```
;; run -> step -> execute-XX:
(defun execute-XX (x86)
... (rgfi *rsp* x86) ...)
```

read stack pointer (for example)

Proof Adaptations: Add “Reduction” Rules

64-bit model

same as before

64/32-bit model

```
(defthm program-is-correct
  formula<(run ... x86)>)
```

```
(defthm program-is-correct
  (implies (64-bit-modep x86)
           formula<(run ... x86)>))
```

```
;; run -> step -> execute-XX:
(defun execute-XX (x86)
  ... (rgfi *rsp* x86) ...)
```

stobj field reader

read stack pointer (for example)

Proof Adaptations: Add “Reduction” Rules

64-bit model

same as before

64/32-bit model

```
(defthm program-is-correct
  formula<(run ... x86)>)
```

```
(defthm program-is-correct
  (implies (64-bit-modep x86)
           formula<(run ... x86)>))
```

```
;; run -> step -> execute-XX:
(defun execute-XX (x86)
  ... (rgfi *rsp* x86) ...)
```

```
;; run -> step -> execute-XX:
(defun execute-XX (x86)
  ... (read-*sp x86) …)
```

stobj field reader

read stack pointer (for example)

Proof Adaptations: Add “Reduction” Rules

64-bit model

```
(defthm program-is-correct
  formula<(run ... x86)>)

;; run -> step -> execute-XX:
(defun execute-XX (x86)
  ... (rgfi *rsp* x86) ...)
```

stobj field reader

64/32-bit model

same as before

```
(defthm program-is-correct
  (implies (64-bit-modep x86)
           formula<(run ... x86)>))

;; run -> step -> execute-XX:
(defun execute-XX (x86)
  ... (read-*sp x86) …)
```

read stack pointer (for example)

```
(defthm read-*sp-when-64-bit-modep
  (implies (64-bit-modep x86)
           (equal (read-*sp x86)
                  (rgfi *rsp* x86))))
```

reduce general stack pointer read to 64-bit-mode stack pointer read

Other Proof Adaptations

Other Proof Adaptations

- Add theorems asserting that `64-bit-modep` is preserved by state updates.

Other Proof Adaptations

- Add theorems asserting that `64-bit-modep` is preserved by state updates.
  - So the reduction rules keep applying.

Other Proof Adaptations

- Add theorems asserting that `64-bit-modep` is preserved by state updates.
  - So the reduction rules keep applying.
  - The model does not cover mode changes yet.

Other Proof Adaptations

- Add theorems asserting that `64-bit-modep` is preserved by state updates.
  - So the reduction rules keep applying.
  - The model does not cover mode changes yet.
- Adapt the congruence-based reasoning for 64-bit system programs.

Other Proof Adaptations

- Add theorems asserting that `64-bit-modep` is preserved by state updates.
  - So the reduction rules keep applying.
  - The model does not cover mode changes yet.
- Adapt the congruence-based reasoning for 64-bit system programs.
  - Linear-to-physical address translations may change the state — the *accessed* and *dirty* flags of paging structures.

Other Proof Adaptations

- Add theorems asserting that `64-bit-modep` is preserved by state updates.
  - So the reduction rules keep applying.
  - The model does not cover mode changes yet.
- Adapt the congruence-based reasoning for 64-bit system programs.
  - Linear-to-physical address translations may change the state — the *accessed* and *dirty* flags of paging structures.
  - These flags are “abstracted away” via an equivalence relation on x86 states — i.e. everything is the same except possibly these flags.

Other Proof Adaptations

- Add theorems asserting that `64-bit-modep` is preserved by state updates.
  - So the reduction rules keep applying.
  - The model does not cover mode changes yet.

- Adapt the congruence-based reasoning for 64-bit system programs.
  - Linear-to-physical address translations may change the state — the *accessed* and *dirty* flags of paging structures.
  - These flags are “abstracted away” via an equivalence relation on x86 states — i.e. everything is the same except possibly these flags.
  - `64-bit-modep` had to be added to this equivalence relation, as well as to other related theorems.

Other Proof Adaptations

- Add theorems asserting that `64-bit-modep` is preserved by state updates.
  - So the reduction rules keep applying.
  - The model does not cover mode changes yet.
- Adapt the congruence-based reasoning for 64-bit system programs.
  - Linear-to-physical address translations may change the state — the *accessed* and *dirty* flags of paging structures.
  - These flags are “abstracted away” via an equivalence relation on x86 states — i.e. everything is the same except possibly these flags.
  - `64-bit-modep` had to be added to this equivalence relation, as well as to other related theorems.
  - This was more laborious than all the previous proof adaptations.

Performance

simulation speed (application view), in instructions/second	before the extensions	after the extensions	after some optimizations
64-bit mode			
32-bit mode			

All measurements done on an Intel Xeon E31280 CPU @ 3.50GHz with 32GB RAM.

Performance

simulation speed (application view), in instructions/second	before the extensions	after the extensions	after some optimizations
64-bit mode	3.0M		
32-bit mode	—		

All measurements done on an Intel Xeon E31280 CPU @ 3.50GHz with 32GB RAM.

Performance

simulation speed (application view), in instructions/second	before the extensions	after the extensions	after some optimizations
64-bit mode	3.0M	1.9M	
32-bit mode	—	0.9M	

All measurements done on an Intel Xeon E31280 CPU @ 3.50GHz with 32GB RAM.

Performance

simulation speed (application view), in instructions/second	before the extensions	after the extensions	after some optimizations
64-bit mode	3.0M	1.9M	3.0M
32-bit mode	—	0.9M	2.5M

All measurements done on an Intel Xeon E31280 CPU @ 3.50GHz with 32GB RAM.

Future Work

Future Work

- Short and medium term:

Future Work

- Short and medium term:
  - Extend floating-point instructions to 32-bit mode.

Future Work

- Short and medium term:
  - Extend floating-point instructions to 32-bit mode.
  - Extend system view to 32-bit mode — add 32-bit paging.

Future Work

- Short and medium term:
  - Extend floating-point instructions to 32-bit mode.
  - Extend system view to 32-bit mode — add 32-bit paging.
  - Remaining modes — real-address, virtual-8086, system management.

Future Work

- Short and medium term:
  - Extend floating-point instructions to 32-bit mode.
  - Extend system view to 32-bit mode — add 32-bit paging.
  - Remaining modes — real-address, virtual-8086, system management.
  - More instructions, especially vector features (AVX, AVX2, AVX-512).

Future Work

- Short and medium term:
  - Extend floating-point instructions to 32-bit mode.
  - Extend system view to 32-bit mode — add 32-bit paging.
  - Remaining modes — real-address, virtual-8086, system management.
  - More instructions, especially vector features (AVX, AVX2, AVX-512).
  - Co-simulate 32-bit programs, for validation.

Future Work

- Short and medium term:
  - Extend floating-point instructions to 32-bit mode.
  - Extend system view to 32-bit mode — add 32-bit paging.
  - Remaining modes — real-address, virtual-8086, system management.
  - More instructions, especially vector features (AVX, AVX2, AVX-512).
  - Co-simulate 32-bit programs, for validation.
  - Improve performance in 32-bit mode.

Future Work

- Short and medium term:
  - Extend floating-point instructions to 32-bit mode.
  - Extend system view to 32-bit mode — add 32-bit paging.
  - Remaining modes — real-address, virtual-8086, system management.
  - More instructions, especially vector features (AVX, AVX2, AVX-512).
  - Co-simulate 32-bit programs, for validation.
  - Improve performance in 32-bit mode.
  - Verify 32-bit programs.

Future Work

- Short and medium term:
  - Extend floating-point instructions to 32-bit mode.
  - Extend system view to 32-bit mode — add 32-bit paging.
  - Remaining modes — real-address, virtual-8086, system management.
  - More instructions, especially vector features (AVX, AVX2, AVX-512).
  - Co-simulate 32-bit programs, for validation.
  - Improve performance in 32-bit mode.
  - Verify 32-bit programs.
  - Detect malware variants via semantic equivalence checking by symbolic execution — this prompted these 32-bit extensions.

Future Work

- Short and medium term:
  - Extend floating-point instructions to 32-bit mode.
  - Extend system view to 32-bit mode — add 32-bit paging.
  - Remaining modes — real-address, virtual-8086, system management.
  - More instructions, especially vector features (AVX, AVX2, AVX-512).
  - Co-simulate 32-bit programs, for validation.
  - Improve performance in 32-bit mode.
  - Verify 32-bit programs.
  - Detect malware variants via semantic equivalence checking by symbolic execution — this prompted these 32-bit extensions.

- Long term:

Future Work

- Short and medium term:
  - Extend floating-point instructions to 32-bit mode.
  - Extend system view to 32-bit mode — add 32-bit paging.
  - Remaining modes — real-address, virtual-8086, system management.
  - More instructions, especially vector features (AVX, AVX2, AVX-512).
  - Co-simulate 32-bit programs, for validation.
  - Improve performance in 32-bit mode.
  - Verify 32-bit programs.
  - Detect malware variants via semantic equivalence checking by symbolic execution — this prompted these 32-bit extensions.

- Long term:
  - Add concurrent semantics.

Future Work

- Short and medium term:
  - Extend floating-point instructions to 32-bit mode.
  - Extend system view to 32-bit mode — add 32-bit paging.
  - Remaining modes — real-address, virtual-8086, system management.
  - More instructions, especially vector features (AVX, AVX2, AVX-512).
  - Co-simulate 32-bit programs, for validation.
  - Improve performance in 32-bit mode.
  - Verify 32-bit programs.
  - Detect malware variants via semantic equivalence checking by symbolic execution — this prompted these 32-bit extensions.

- Long term:
  - Add concurrent semantics.
  - Make the specification more declarative, generating efficient code via macros, possibly APT transformations.

Future Work

- Short and medium term:
  - Extend floating-point instructions to 32-bit mode.
  - Extend system view to 32-bit mode — add 32-bit paging.
  - Remaining modes — real-address, virtual-8086, system management.
  - More instructions, especially vector features (AVX, AVX2, AVX-512).
  - Co-simulate 32-bit programs, for validation.
  - Improve performance in 32-bit mode.
  - Verify 32-bit programs.
  - Detect malware variants via semantic equivalence checking by symbolic execution — this prompted these 32-bit extensions.

- Long term:
  - Add concurrent semantics.
  - Make the specification more declarative, generating efficient code via macros, possibly APT transformations.
  - Verified compilation to binaries.

Future Work

- Short and medium term:
  - Extend floating-point instructions to 32-bit mode.
  - Extend system view to 32-bit mode — add 32-bit paging.
  - Remaining modes — real-address, virtual-8086, system management.
  - More instructions, especially vector features (AVX, AVX2, AVX-512).
  - Co-simulate 32-bit programs, for validation.
  - Improve performance in 32-bit mode.
  - Verify 32-bit programs.
  - Detect malware variants via semantic equivalence checking by symbolic execution — this prompted these 32-bit extensions.

- Long term:
  - Add concurrent semantics.
  - Make the specification more declarative, generating efficient code via macros, possibly APT transformations.
  - Verified compilation to binaries.
  - Synthesis of verified binaries.