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

power on or reset

Real-Address Mode

no memory management,
limited address space
x86 Modes of Operation

- **Real-Address Mode**
  - no memory management
  - limited address space

- **Protected Mode**
  - memory management
  - privilege levels
  - ‘32-bit mode’

*Power on or reset:* Real-Address Mode → Protected Mode
x86 Modes of Operation

- **Real-Address Mode**: no memory management, limited address space
- **Protected Mode**: memory management, privilege levels, ‘32-bit mode’
- **IA-32e Mode**: larger addresses and operands, simplified memory management

On power on or reset:
- Transition from Real-Address Mode to Protected Mode
- Transition from Protected Mode to IA-32e Mode
x86 Modes of Operation

- **Power on or reset**
  - Real-Address Mode
    - No memory management, limited address space
  - Protected Mode
    - Memory management, privilege levels, ‘32-bit mode’
  - Compatibility (Sub-)Mode
    - Emulates 32-bit mode
  - IA-32e Mode
    - Larger addresses and operands, simplified memory management
  - 64-bit (Sub-)Mode
x86 Modes of Operation

- **Real-Address Mode**
  - power on or reset
  - no memory management, limited address space

- **Virtual-8086 Mode**
  - emulates real-address mode, very legacy

- **Protected Mode**
  - memory management, privilege levels, '32-bit mode'

- **Compatibility (Sub-)Mode**
  - emulates 32-bit mode

- **IA-32e Mode**
  - larger addresses and operands, simplified memory management

- **64-bit (Sub-)Mode**
  - emulates 32-bit mode
x86 Modes of Operation

- **Real-Address Mode**: no memory management, limited address space. Power on or reset.
- **Virtual-8086 Mode**: emulates real-address mode, very legacy.
- **Protected Mode**: memory management, privilege levels, ‘32-bit mode’.
- **Compatibility (Sub-)Mode**: emulates 32-bit mode.
- **IA-32e Mode**: larger addresses and operands, simplified memory management.
- **64-bit (Sub-)Mode**: from/to all the other modes.
- **System Management Mode**: run firmware for special uses.
Memory Management

Logical Address

Segmentation

Segment Selector
Effective Address

Linear Address

Paging
Physical Address
Memory Management

instructions use logical addresses

Logical Address

Segmentation

Segment Selector
Effective Address

Linear Address

Paging

Physical Address
Memory Management

instructions use logical addresses

Logical Address

Segmentation

Segment Selector
Effective Address

Linear Address

Paging

Physical Address

the bus uses physical addresses
Memory Management

- Logical Address
  - Segment Selector
  - Effective Address

Segmentation → Linear Address

Paging → Physical Address

- Instructions use logical addresses
- The bus uses physical addresses
Segmentation

- **Logical Address**
  - **Segment Selector**
    - **Segment Descriptor**
      - **Descriptor Table**
        - **Global or Local Descriptor Table Register**
          - `table_indicator(selector)` points to the appropriate register

- **Effective Address**

- **Linear Address**

- **Segment**
  - **points to the base of the segment**

- **Hidden Part**
  - cached part of the selector holding information from the corresponding descriptor

- **Segment points to the base of the descriptor**
  - **index(selector)** points to the descriptor

- **points to the base of the descriptor table**

- **machine registers**
  - CS (default), SS, DS, ES, FS, and GS
Memory Management

Instructions use logical addresses

Logical Address

Segmentation

Segment Selector    Effective Address

Linear Address

Paging

Physical Address

The bus uses physical addresses
Memory Management

instructions use logical addresses

Logical Address

Segment Selector  Effective Address

Segmentation

Linear Address

Paging

Physical Address

the bus uses physical addresses
IA-32e Paging (4K Pages)

First few bits of linear address point to an entry, PML4E, in this structure.

PML4E points to the next structure in the hierarchy.

The last entry points to the base of a page.

Offset points to the address inside the page.

Control register CR3 points to the base of the first paging data structure.

And so on…
Memory Management

Logical Address

Segmentation → Linear Address

Segmentation

Effective Address

Paging → Physical Address
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)

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

visible to system code and to application code
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).

visible to system code and to application code

visible to system code but not to application code
X86ISA: The ACL2 Formal Model of the x86 ISA
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).
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
X86ISA: Overview

- regs
- flags

byte-addressable mem

x86 state (stobj)
X86 ISA: Overview

- model-specific fields
- env
- view
- regs
- flags
- byte-addressable mem

x86 state (stobj)
X86ISA: Overview

- model-specific fields
- env
- view
- regs
- byte-addressable mem

x86 state (stobj)

interface to the x86 state
X86ISA: Overview

- Model-specific fields
- CPU registers (regs)
- Byte-addressable memory (mem)
- Environment (env)
- Flags (flags)
- View (view)

Interface to the x86 state

- x86 read (xr)
- x86 write (xw)
- Read mem bytes (rb)
- Write mem bytes (wb)

x86 state (stobj)
X86ISA: Overview

Interface to the x86 state

- Model-specific fields
- Registers (regs)
- Byte-addressable memory
- Flags

X86 read (xr)
X86 write (xw)
Read mem bytes (rb)
Write mem bytes (wb)

Instruction semantic functions
- ADD
- SUB
- MUL
- MOV
- PUSH
- POP
**X86ISA: Overview**

**X86 read**
- ADD
- SUB
- MUL
- ... (step x86) fetch, decode, & execute one instruction

**X86 write**
- MOV
- PUSH
- POP

**Read mem bytes**
- read mem bytes

**Write mem bytes**
- write mem bytes

**Interface to the x86 state**
- x86 state (stobj)

**Instruction semantic functions**
- model-specific fields
- regs
- flags
- byte-addressable mem
- view
X86ISA: Overview

- **X86ISA**: Instruction set architecture for Intel x86 processors.

**Overview**:
- **Memory**: Byte-addressable memory.
- **Registers**: regs
- **Flags**: flags
- **Model-specific Fields**: ms, view, env

**Interface to the x86 State**:
- **x86 State (stobj)**

**Instruction Semantic Functions**:
- ADD, SUB, MUL, ..., MOV, PUSH, POP

**Steps**:
- (step x86): Fetch, decode, & execute one instruction
- (run n x86): Run n x86 instructions
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

Logical Address

Segment Selector  Effective Address

Segmentation  Linear Address

Paging  Physical Address
X86ISA: Views

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

Application View

- Lowest level of memory address: linear address.

System View

Logical Address

Segmentation  Linear Address  Paging  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.

System View

Logical Address

Segmentation

Linear Address

Paging

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.

System View

Logical Address

Segment Selector Effective Address

Segmentation

Linear Address

Paging

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

<table>
<thead>
<tr>
<th>Logical Address</th>
<th>Segmentation</th>
<th>Linear Address</th>
<th>Paging</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment Selector</td>
<td>Effective Address</td>
<td>Linear Address</td>
<td>Paging</td>
<td>Physical Address</td>
</tr>
</tbody>
</table>
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.

Logical Address

Segmentation

- Segment Selector
- Effective Address

Linear Address

Paging

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.

Logical Address

Segment Selector  Effective Address  Segmentation  Linear Address  Paging  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.
- **Necessary** level of operation for verification of system programs.
Coverage of the Model
Coverage of the Model

- Real-Address Mode
- Protected (32-bit) Mode
- Virtual-8086 Mode
- Compatibility (Sub-)Mode
- 64-bit (Sub-)Mode
- System Management Mode
Coverage of the Model

before the work in this paper
Coverage of the Model

after the work in this paper
(application view only in 32-bit mode: no paging yet)
(no floating point instructions in 32-bit more yet either)

before the work in this paper

Real-Address Mode

Protected (32-bit) Mode

Virtual-8086 Mode

Compatibility (Sub-)Mode

IA-32e Mode

64-bit (Sub-)Mode

System Management Mode
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

Logical Address

Segment Selector    Effective Address

Segmentation

Linear Address

Paging

Physical Address
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

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

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

64/32-bit model
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
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

Segmentation

Linear Address

Segment Selector  Effective Address

Segmentation

Linear Address

Paging

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

Logical Address

Segment Selector Effective Address

Segmentation Linear Address Paging Physical Address
Add Translation from Logical to Linear Address

64-bit model

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

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
)

Logical Address

Segment Selector  Effective Address

Segmentation

Linear Address

Paging

Physical 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

Logical Address

Segment Selector

Effective Address

Segmentation

Linear Address

Paging

Physical Address
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

Logical Address
Segment Selector  Effective Address
Segmentation → Linear Address → Paging → Physical Address
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 ...) ...)
...

Logical Address

Segmentation

Segment Selector
Effective Address

Linear Address

Paging

Physical Address
Add New Top-Level Memory Access Functions

### 64-bit model

```lisp
(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

```lisp
;; 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, ...)
```

---

**Logical Address**

- Segment Selector
- Effective Address

**Segmentation**

**Linear Address**

**Paging**

**Physical Address**
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)

64/32-bit model

;; read instruction (via lin. addr.):
opcode := (rml08 rip ...) ;; etc.
Extend Instruction Fetching

64-bit model

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

64/32-bit model

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

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

stobj field reader

artistic license
Extend Instruction Fetching

64-bit model

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

64/32-bit model

;; 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)

artistic license

stobj field reader

stobj field writer
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.

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

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

artistic license

**64/32-bit model**

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

new function

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

`; 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 reader

stobj field writer
Extend Instruction Fetching

**64-bit model**

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

**artistic license**

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

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

**stobj field reader**

**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.

**stobj field writer**
Extend Instruction Fetching

**64-bit model**

`; read instruction pointer from RIP:`
```plaintext`
rip := (rip x86) ;; 48-bit (canonical)
```
```
*ip := (read-*ip x86) ;; 48/32/16-bit

`; read instruction (via lin. addr.):`
```plaintext`
opcode := (rml08 rip ...) ;; etc.
```
```
opcode := (rme08 *ip ...) ;; etc.

`; increment instruction pointer:`
```plaintext`
new-rip := (+ rip delta) ;; if new-rip not canonical then fault
```
```
new-*ip := (add-to-*ip *ip delta x86) (includes canonical and segment checks)

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

**64/32-bit model**

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

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

`; increment instruction pointer:`
```plaintext`
new-*ip := (add-to-*ip *ip delta x86) (includes canonical and segment checks)
```
```
```
```

**Artistic license**
Extend Instruction Fetching

**64-bit model**

- read instruction pointer from RIP:
  \[ \text{rip} := (\text{rip} x86) \] ;; 48-bit (canonical)

  \[ \text{stobj field reader} \]

- read instruction (via lin. addr.):
  \[ \text{opcode} := (\text{rml08 rip ...}) \] ;; etc.

- increment instruction pointer:
  \[ \text{new-rip} := (+ \text{rip} \text{delta}) \]
  ;; if new-rip not canonical then fault

- write instruction pointer to RIP:
  \[ x86 := (\text{!rip new-rip} x86) \]

  \[ \text{stobj field writer} \]

**64/32-bit model**

- read instr. pointer from RIP/EIP/IP:
  \[ \text{*ip} := (\text{read-*ip} x86) \] ;; 48/32/16-bit

  \[ \text{new function} \]

- read instruction (via eff. addr.):
  \[ \text{opcode} := (\text{rme08 *ip ...}) \] ;; etc.

- increment instruction pointer:
  \[ \text{new-*ip} := (\text{add-to-*ip} \text{ip delta} x86) \]

  \[ \text{new function} \]
  \[ \text{(includes canonical and segment checks)} \]

- write instr. pointer to RIP/EIP/IP:
  \[ x86 := (\text{write-*ip new-*ip} x86) \]

  \[ \text{new function} \]

**Artistic license**
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-mode 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

<table>
<thead>
<tr>
<th>64-bit model</th>
<th>64/32-bit model</th>
</tr>
</thead>
</table>
| (defthm program-is-correct
  formula<(run ... x86)>)
  /
  /
  an existing theorem
  about a 64-bit program |
Proof Adaptations:
Add 64-bit Mode Hypotheses

64-bit model

(defun run ... step ...)

(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

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

an existing theorem
about a 64-bit program

64/32-bit model

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

simplified code
(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

(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

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

64/32-bit model

(defthm program-is-correct
  (implies (64-bit-modep x86)
    formula<(run ... x86)>))
Proof Adaptations:
Add “Reduction” Rules

64-bit model

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

;; run -&gt; step -&gt; execute-XX:
(defun execute-XX (x86)
  ... (rgfi *rsp* x86) ...)

64/32-bit model

(same as before)

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

(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

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

same as before

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

(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)
Proof Adaptations: Add “Reduction” Rules

64-bit model

(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)

(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) ...)

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

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

<table>
<thead>
<tr>
<th>simulation speed (application view), in instructions/second</th>
<th>before the extensions</th>
<th>after the extensions</th>
<th>after some optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-bit mode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit mode</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Simulation speed (application view), in instructions/second</th>
<th>Before the extensions</th>
<th>After the extensions</th>
<th>After some optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-bit mode</td>
<td>3.0M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit mode</td>
<td>—</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Simulation speed (application view), in instructions/second</th>
<th>before the extensions</th>
<th>after the extensions</th>
<th>after some optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-bit mode</td>
<td>3.0M</td>
<td>1.9M</td>
<td></td>
</tr>
<tr>
<td>32-bit mode</td>
<td>—</td>
<td>0.9M</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>simulation speed (application view), in instructions/second</th>
<th>before the extensions</th>
<th>after the extensions</th>
<th>after some optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>64-bit mode</strong></td>
<td></td>
<td>3.0M</td>
<td>1.9M</td>
<td>3.0M</td>
</tr>
<tr>
<td><strong>32-bit mode</strong></td>
<td></td>
<td>—</td>
<td>0.9M</td>
<td>2.5M</td>
</tr>
</tbody>
</table>

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.