

# X86 instruction semantics and basic block symbolic execution

Freek Verbeek      Abhijith Bharadwaj      Joshua Bockenek  
Ian Roessle      Timmy Weerwag      Binoy Ravindran

March 17, 2025

## Abstract

This AFP entry provides semantics for roughly 120 different X86-64 assembly instructions. These instructions include various moves, arithmetic/logical operations, jumps, call/return, SIMD extensions and others. External functions are supported by allowing a user to provide custom semantics for these calls. Floating-point operations are mapped to uninterpreted functions. The model provides semantics for register aliasing and a byte-level little-endian memory model. The semantics are purposefully incomplete, but overapproximative. For example, the precise effect of flags may be undefined for certain instructions, or instructions may simply have no semantics at all. In those cases, the semantics are mapped to universally quantified uninterpreted terms from a locale. Second, this entry provides a method to symbolic execution of basic blocks. The method, called "se\_step" (for: symbolic execution step) fetches an instruction and updates the current symbolic state while keeping track of assumptions made over the memory model. A key component is a set of theorems that prove how reads from memory resolve after writes have occurred. Thirdly, this entry provides a parser that allows the user to copy-paste the output of the standard disassembly tool objdump into Isabelle/HOL. Several examples are supplied: a couple small and explanatory examples, functions from the word count program, the floating-point modulo function from FDLIBM, the GLIBC strlen function and the CoreUtils SHA256 implementation.

## Contents

<b>1 Bit and byte-level theorems</b>	<b>2</b>
1.1 Basics . . . . .	2
1.2 Take_Bits and arithmetic . . . . .	4
<b>2 Memory-related theorems</b>	<b>6</b>
<b>3 Concrete state and instructions</b>	<b>11</b>

<b>4 Instruction Semantics</b>	<b>20</b>
<b>5 Removing superfluous memory writes</b>	<b>52</b>
<b>6 A symbolic execution engine</b>	<b>55</b>
<b>7 Small examples</b>	<b>57</b>
<b>8 Parser</b>	<b>58</b>
<b>9 Example: word count program from GNU</b>	<b>59</b>

## 1 Bit and byte-level theorems

```
theory BitByte
  imports Main Word-Lib.Syntax-Bundles Word-Lib.Bit-Shifts-Infix-Syntax Word-Lib.Bitwise
begin
```

### 1.1 Basics

```
unbundle bit-operations-syntax
unbundle bit-projection-infix-syntax
```

**definition** *take-bits* :: nat  $\Rightarrow$  nat  $\Rightarrow$  'a::len word  $\Rightarrow$  'a::len word ( $\langle\langle\cdot,\cdot\rangle\rangle$ - 51) — little-endian

**where** *take-bits* *l h w*  $\equiv$  (*w >> l*) AND *mask* (*h-l*)

$\langle l,h \rangle w$  takes a subset of bits from word *w*, from low (inclusive) to high (exclusive). For example,  $\langle 2::nat, 5::nat \rangle 28::8$  word =  $(7::8$  word $)$ .

**definition** *take-byte* :: nat  $\Rightarrow$  'a::len word  $\Rightarrow$  8word — little-endian

**where** *take-byte* *n w*  $\equiv$  *ucast* ( $\langle n*8, n*8+8 \rangle w$ )

*take-byte* *n w* takes the *n*th byte from word *w*. For example, *take-byte* 1 ((42::16 word) << (8::nat)) = (42::8 word).

**definition** *overwrite* :: nat  $\Rightarrow$  nat  $\Rightarrow$  'a::len word  $\Rightarrow$  'a::len word  $\Rightarrow$  'a::len word

**where** *overwrite* *l h w0 w1*  $\equiv$  (( $\langle\langle h, LENGTH('a) \rangle w0 \rangle$ ) << *h*) OR (( $\langle\langle l,h \rangle w1 \rangle$ ) << *l*) OR ( $\langle\langle 0,l \rangle w0 \rangle$ )

*overwrite* *l h w0 w1* overwrites low (inclusive) to high (exclusive) bits in word *w0* with bits from word *w1*. For example, *overwrite* (2::nat) (4::nat) (28::8 word) (227::8 word) = (16::8 word).

We prove some theorems about taking the *n*th bit/byte of an operation. These are useful to prove equality between different words, by first applying rule ( $\bigwedge n. n < size ?u \implies ?u !! n = ?v !! n \implies ?u = ?v$ ).

**lemma** *bit-take-bits-iff* [bit-simps]:

$\langle\langle l,h \rangle w \rangle !! n \longleftrightarrow n < LENGTH('a) \wedge n < h - l \wedge w !! (n + l)$  **for** *w* :: 'a::len word

$\langle proof \rangle$

```

lemma bit-take-byte-iff [bit-simps]:
  ‹take-byte m w !! n  $\longleftrightarrow$  n < LENGTH('a)  $\wedge$  n < 8  $\wedge$  w !! (n + m * 8)› for w
  :: ‹'a::len word›
  ⟨proof⟩

lemma bit-overwrite-iff [bit-simps]:
  ‹overwrite l h w0 w1 !! n  $\longleftrightarrow$  n < LENGTH('a)  $\wedge$ 
    (if l ≤ n  $\wedge$  n < h then w1 else w0) !! n›
  for w0 w1 :: ‹'a::len word›
  ⟨proof⟩

lemma nth-takebits:
  fixes w :: ‹'a::len word›
  shows (⟨l,h⟩w) !! n = (if n < LENGTH('a)  $\wedge$  n < h - l then w !! (n + l) else
  False)
  ⟨proof⟩

lemma nth-takebyte:
  fixes w :: ‹'a::len word›
  shows take-byte (n div 8) w !! (n mod 8) = (if n mod 8 < LENGTH('a) then
  w!!n else False )
  ⟨proof⟩

lemma nth-take-byte-overwrite:
  fixes v v' :: ‹'a::len word›
  shows take-byte n (overwrite l h v v') !! i = (if i + n * 8 < l  $\vee$  i + n * 8 ≥ h
  then take-byte n v !! i else take-byte n v' !! i)
  ⟨proof⟩

lemma nth-bitNOT:
  fixes a :: ‹'a::len word›
  shows (NOT a) !! n  $\longleftrightarrow$  (if n < LENGTH('a) then  $\neg(a !! n)$  else False)
  ⟨proof⟩

```

Various simplification rules

```

lemma ucast-take-bits:
  fixes w :: ‹'a::len word›
  assumes h = LENGTH('b)
  and LENGTH('b) ≤ LENGTH('a)
  shows ucast (⟨0,h⟩w) = (ucast w ::'b :: len word)
  ⟨proof⟩

lemma take-bits-ucast:
  fixes w :: ‹b::len word›
  assumes h = LENGTH('b)
  shows (⟨0,h⟩ (ucast w ::'a :: len word)) = (ucast w ::'a :: len word)
  ⟨proof⟩

```

```

lemma take-bits-take-bits:
  fixes w :: 'a::len word
  shows ⟨⟨l,h⟩(⟨l',h'⟩w)⟩ = (if min LENGTH('a) h ≥ h' - l' then ⟨l+l',h'⟩w else
  ⟨⟨l+l',l'+min LENGTH('a) h⟩w)⟩)
  ⟨proof⟩

lemma take-bits-overwrite:
  shows ⟨l,h⟩(overwrite l h w0 w1) = ⟨l,h⟩w1
  ⟨proof⟩

lemma overwrite-0-take-bits-0:
  shows overwrite 0 h (⟨0,h⟩w0) w1 = ⟨0,h⟩w1
  ⟨proof⟩

lemma take-byte-shiftlr-256:
  fixes v :: 256 word
  assumes m ≤ n
  shows take-byte n (v << m*8) = (if (n+1)*8 ≤ 256 then take-byte (n-m) v
  else 0)
  ⟨proof⟩

```

## 1.2 Take\_Bits and arithmetic

This definition is based on  $to-bl (?x + ?y) = rev (foldr (\lambda(x, y) res car. xor3 x y car \# res (carry x y car)) (rev (zip (to-bl ?x) (to-bl ?y))) (\lambda(). False))$ , which formulates addition as bitwise operations using  $xor3$  and  $carry$ .

```

definition bitwise-add :: (bool × bool) list ⇒ bool ⇒ bool list
  where bitwise-add x c ≡ foldr (\lambda(x, y) res car. xor3 x y car \# res (carry x y car)) x (\lambda(). False) c

```

```

lemma length-foldr-bitwise-add:
  shows length (bitwise-add x c) = length x
  ⟨proof⟩

```

This is the "heart" of the proof: bitwise addition of two appended zipped lists can be expressed as two consecutive bitwise additions. Here, I need to make the assumption that the final carry is False.

```

lemma bitwise-add-append:
  assumes x = [] ∨ ¬carry (fst (last x)) (snd (last x)) True
  shows bitwise-add (x @ y) (x ≠ [] ∧ c) = bitwise-add x (x ≠ [] ∧ c) @ bitwise-add
  y False
  ⟨proof⟩

```

```

lemma bitwise-add-take-append:
  shows take (length x) (bitwise-add (x @ y) c) = bitwise-add x c
  ⟨proof⟩

```

```
lemma bitwise-add-zero:
  shows bitwise-add (replicate n (False, False)) False = replicate n False
  <proof>
```

```
lemma bitwise-add-take:
  shows take n (bitwise-add x c) = bitwise-add (take n x) c
  <proof>
```

```
lemma fst-hd-drop-zip:
  assumes n < length x
    and length x = length y
  shows fst (hd (drop n (zip x y))) = hd (drop n x)
  <proof>
```

```
lemma snd-hd-drop-zip:
  assumes n < length x
    and length x = length y
  shows snd (hd (drop n (zip x y))) = hd (drop n y)
  <proof>
```

Ucasting of  $a + b$  can be rewritten to taking bits of  $a$  and  $b$ .

```
lemma uCast-plus:
  fixes a b :: 'a::len word
  assumes LENGTH('a) > LENGTH('b)
  shows (uCast (a + b) ::'b::len word) = (uCast a + uCast b::'b::len word)
<proof>
```

```
lemma uCast-uminus:
  fixes a b :: 'a::len word
  assumes LENGTH('a) > LENGTH('b)
  shows (uCast (- a)) = (- uCast a :: 'b::len word)
<proof>
```

```
lemma uCast-minus:
  fixes a b :: 'a::len word
  assumes LENGTH('a) > LENGTH('b)
  shows (uCast (a - b) ::'b::len word) = (uCast a - uCast b::'b::len word)
<proof>
```

```
lemma to-bl-takebits:
  fixes a :: 'a::len word
  shows (to-bl ((0,h)a)) = replicate (LENGTH('a) - h) False @ drop (LENGTH('a) - h) (to-bl a)
<proof>
```

All simplification rules that are used during symbolic execution.

```
lemmas BitByte-simps = uCast-plus uCast-minus uCast-uminus take-bits-overwrite take-bits-take-bits
```

```

 $ucast\text{-}take\text{-}bits\;overwrite\text{-}0\text{-}take\text{-}bits\text{-}0\;mask\text{-}eq\text{-}exp\text{-}minus\text{-}1$ 
 $ucast\text{-}down\text{-}ucast\text{-}id\;is\text{-}down\;take\text{-}bits\text{-}ucast\;ucast\text{-}up\text{-}ucast\text{-}id\;is\text{-}up$ 

```

Simplification for immediate (numeral) values.

```

lemmas take-bits-numeral[simp] = take-bits-def[of - - numeral n] for n
lemmas take-bits-num0[simp] = take-bits-def[of - - 0] for n
lemmas take-bits-num1[simp] = take-bits-def[of - - 1] for n
lemmas overwrite-numeral-numeral[simp] = overwrite-def[of - - numeral n numeral m] for n m
lemmas overwrite-num0-numeral[simp] = overwrite-def[of - - 0 numeral m] for n m
lemmas overwrite-numeral-num0[simp] = overwrite-def[of - - numeral m 0] for n m
lemmas overwrite-numeral-00[simp] = overwrite-def[of - - 0 0]

end

```

## 2 Memory-related theorems

```

theory Memory
  imports BitByte
begin

context
  fixes dummy-type :: 'a::len
begin

primrec read-bytes :: ('a word  $\Rightarrow$  8 word)  $\Rightarrow$  'a word  $\Rightarrow$  nat  $\Rightarrow$  8word list
  where read-bytes m a 0 = []
    | read-bytes m a (Suc n) = m (a + of-nat n) # read-bytes m a n

```

Read bytes from memory. Memory is represented by a term of  $64\;word \Rightarrow 8\;word$ . Given an address  $a::64\;word$  and a size  $n$ , retrieve the bytes in the order they are stored in memory.

```

definition region-addresses :: 'a word  $\Rightarrow$  nat  $\Rightarrow$  'a word set
  where region-addresses a si  $\equiv$  {a' .  $\exists i < si . a' = a + of\text{-}nat (si - i - 1)$ }

```

The set of addresses belonging to a region starting at address  $a$  of  $si$  bytes.

```

definition region-overflow :: 'a word  $\Rightarrow$  nat  $\Rightarrow$  bool
  where region-overflow a si  $\equiv$  unat a + si  $\geq 2^{LENGTH('a)}$ 

```

An overflow occurs if the address plus the size is greater equal  $(2::'b)^{64}$

```

definition enclosed :: 'a word  $\Rightarrow$  nat  $\Rightarrow$  'a word  $\Rightarrow$  nat  $\Rightarrow$  bool
  where enclosed a' si' a si  $\equiv$  unat a + si < 2 $^{LENGTH('a)}$   $\wedge$  unat a  $\leq$  unat a'
     $\wedge$  unat a' + si'  $\leq$  unat a + si

```

A region is enclosed in another if its *local.region-addresses* is a subset of the other.

```

definition separate :: 'a word  $\Rightarrow$  nat  $\Rightarrow$  'a word  $\Rightarrow$  nat  $\Rightarrow$  bool
  where separate a si a' si'  $\equiv$  si  $\neq$  0  $\wedge$  si'  $\neq$  0  $\wedge$  region-addresses a si  $\cap$  region-addresses a' si' = {}

```

A region is separate from another if they do not overlap.

```

lemma region-addresses-iff: a'  $\in$  region-addresses a si  $\longleftrightarrow$  unat (a' - a) < si
   $\langle proof \rangle$ 

```

```

lemma notin-region-addresses:
  assumes x  $\notin$  region-addresses a si
  shows unat x < unat a  $\vee$  unat a + si  $\leq$  unat x
   $\langle proof \rangle$ 

```

```

lemma notin-region-addresses-sub:
  assumes x  $\notin$  region-addresses a si
  shows unat (x - a') < unat (a - a')  $\vee$  unat (a - a') + si  $\leq$  unat (x - a')
   $\langle proof \rangle$ 

```

```

lemma region-addresses-eq-empty-iff: region-addresses a si = {}  $\longleftrightarrow$  si = 0
   $\langle proof \rangle$ 

```

```

lemma length-read-bytes:
  shows length (read-bytes m a si) = si
   $\langle proof \rangle$ 

```

```

lemma nth-read-bytes:
  assumes n < si
  shows read-bytes m a si ! n = m (a + of-nat (si - 1 - n))
   $\langle proof \rangle$ 

```

Writing to memory occurs via function *override-on*. In case of enclosure, reading bytes from memory overridden on a set of region addresses can be simplified to reading bytes from the overwritten memory only. In case of separation, reading bytes from overridden memory can be simplified to reading from the original memory.

```

lemma read-bytes-override-on-enclosed:
  assumes offset'  $\leq$  offset
  and si'  $\leq$  si
  and unat offset + si'  $\leq$  si + unat offset'
  shows read-bytes (override-on m m' (region-addresses (a - offset) si)) (a - offset') si' = read-bytes m' (a - offset') si'
   $\langle proof \rangle$ 

```

```

lemmas read-bytes-override-on = read-bytes-override-on-enclosed [where offset=0
and offset'=0, simplified]

```

```

lemma read-bytes-override-on-enclosed-plus:
  assumes unat offset + si'  $\leq$  si
  and si  $\leq$  2LENGTH('a')

```

```

shows read-bytes (override-on m m' (region-addresses a si)) (offset+a) si' =
read-bytes m' (offset+a) si'
⟨proof⟩

```

```

lemma read-bytes-override-on-separate:
assumes separate a si a' si'
shows read-bytes (override-on m m' (region-addresses a si)) a' si' = read-bytes
m a' si'
⟨proof⟩

```

Bytes are written to memory one-by-one, then read by *local.read-bytes* producing a list of bytes. That list is concatenated again using *word-rcat*. Writing *si* bytes of word *w* into memory, reading the byte-list and concatenating again produces *si* bytes of the original word.

```

lemma word-rcat-read-bytes-enclosed:
fixes w :: 'b::len word
assumes LENGTH('b) ≤ 2^LENGTH('a)
and unat offset + si ≤ 2^LENGTH('a)
shows word-rcat (read-bytes (λa'. take-byte (unat (a' - a)) w) (a + offset) si)
= ⟨unat offset * 8, (unat offset + si) * 8⟩w
⟨proof⟩

```

```
lemmas word-rcat-read-bytes = word-rcat-read-bytes-enclosed[where offset=0,simplified]
```

The following theorems allow reasoning over enclosure and separation, for example as linear arithmetic.

```

lemma enclosed-spec:
assumes enclosed: enclosed a' si' a si
and x-in: x ∈ region-addresses a' si'
shows x ∈ region-addresses a si
⟨proof⟩

```

```

lemma address-in-enclosed-region-as-linarith:
assumes enclosed a' si' a si
and x ∈ region-addresses a' si'
shows a ≤ x ∧ a' ≤ x ∧ x < a' + of-nat si' ∧ x < a + of-nat si
⟨proof⟩

```

```

lemma address-of-enclosed-region-ge:
assumes enclosed a' si' a si
shows a' ≥ a
⟨proof⟩

```

```

lemma address-in-enclosed-region:
assumes enclosed a' si' a si
and x ∈ region-addresses a' si'

```

```

shows unat (x - a) ≥ unat (a' - a) ∧ unat (a' - a) + si' > unat (x - a) ∧
unat (x - a) < si
⟨proof⟩

```

**lemma** enclosed-minus-minus:

```

fixes a :: 'a word
assumes offset ≥ offset'
and unat offset - si ≤ unat offset' - si'
and unat offset' ≥ si'
and unat offset ≥ si
and a ≥ offset
shows enclosed (a - offset') si' (a - offset) si
⟨proof⟩

```

**lemma** enclosed-plus:

```

fixes a :: 'a word
assumes si' < si
and unat a + si < 2^LENGTH('a)
shows enclosed a si' a si
⟨proof⟩

```

**lemma** separate-symm: separate a si a' si' = separate a' si' a si

⟨proof⟩

**lemma** separate-iff: separate a si a' si' ↔ si > 0 ∧ si' > 0 ∧ unat (a' - a) ≥
si ∧ unat (a - a') ≥ si'

⟨proof⟩

**lemma** separate-as-linarith:

```

assumes ¬region-overflow a si
and ¬region-overflow a' si'
shows separate a si a' si' ↔ 0 < si ∧ 0 < si' ∧ (a + of-nat si ≤ a' ∨ a' +
of-nat si' ≤ a)
(is ?lhs ↔ ?rhs)
⟨proof⟩

```

Compute separation in case the addresses and sizes are immediate values.

```

lemmas separate-as-linarith-numeral [simp] =
separate-as-linarith [of numeral a::'a word numeral si numeral a'::'a word numeral
si'] for a si a' si'
lemmas separate-as-linarith-numeral-1 [simp] =
separate-as-linarith [of numeral a::'a word numeral si numeral a'::'a word Suc 0]
for a si a'
lemmas separate-as-linarith-numeral1- [simp] =
separate-as-linarith [of numeral a::'a word Suc 0 numeral a'::'a word numeral si']
for a a' si'
lemmas separate-as-linarith-numeral11 [simp] =
separate-as-linarith [of numeral a::'a word Suc 0 numeral a'::'a word Suc 0] for
a a'

```

```

lemmas region-overflow-numeral[simp] =
  region-overflow-def [of numeral a::'a word numeral si] for a si
lemmas region-overflow-numeral1[simp] =
  region-overflow-def [of numeral a::'a word Suc 0] for a

lemma separate-plus-none:
assumes si' ≤ unat offset
  and 0 < si
  and 0 < si'
  and unat offset + si ≤ 2^LENGTH('a)
shows separate (offset + a) si a si'
⟨proof⟩

lemmas unat-minus = unat-sub-if'[of 0,simplified]

lemma separate-minus-minus':
assumes si ≠ 0
  and si' ≠ 0
  and unat offset ≥ si
  and unat offset' ≥ si'
  and unat offset - si ≥ unat offset'
shows separate (a - offset) si (a - offset') si'
⟨proof⟩

lemma separate-minus-minus:
assumes si ≠ 0
  and si' ≠ 0
  and unat offset ≥ si
  and unat offset' ≥ si'
  and unat offset - si ≥ unat offset' ∨ unat offset' - si' ≥ unat offset
shows separate (a - offset) si (a - offset') si'
⟨proof⟩

lemma separate-minus-none:
assumes si ≠ 0
  and si' ≠ 0
  and unat offset ≥ si
  and si' ≤ 2^LENGTH('a) - unat offset
shows separate (a - offset) si a si'
⟨proof⟩

```

The following theorems are used during symbolic execution to determine whether two regions are separate.

**lemmas** separate-simps = separate-plus-none separate-minus-none separate-minus-minus

```

end
end

```

### 3 Concrete state and instructions

```
theory State
  imports Main Memory
begin
```

A state consists of registers, memory, flags and a rip. Some design considerations here:

- All register values are 256 bits. We could also distinguish 64 bits registers, 128 registers etc. That would increase complexity in proofs and datastructures. The cost of using 256 everywhere is that a goal typically will have some casted 64 bits values.
- The instruction pointer RIP is a special 64-bit register outside of the normal register set.
- Strings are used for registers and flags. We would prefer an enumerative datatype, however, that would be extremely slow since there are roughly 100 register names.

```
record state =
  regs :: string ⇒ 256word
  mem  :: 64 word ⇒ 8 word
  flags :: string ⇒ bool
  rip   :: 64 word

definition real-reg :: string ⇒ bool × string × nat × nat
  where real-reg reg ≡
    — TODO: xmm, ymm, etc.
    case reg of
      — rip
      "rip"  ⇒ (True, "rip", 0,64)
      — rax,rbx,rcx,rdx
      | "rax"  ⇒ (True, "rax", 0,64)
      | "eax"  ⇒ (True, "rax", 0,32)
      | "ax"   ⇒ (False, "rax", 0,16)
      | "ah"   ⇒ (False, "rax", 8,16)
      | "al"   ⇒ (False, "rax", 0,8)
      | "rbx"  ⇒ (True, "rbx", 0,64)
      | "ebx"  ⇒ (True, "rbx", 0,32)
      | "bx"   ⇒ (False, "rbx", 0,16)
      | "bh"   ⇒ (False, "rbx", 8,16)
      | "bl"   ⇒ (False, "rbx", 0,8)
      | "rcx"  ⇒ (True, "rcx", 0,64)
      | "ecx"  ⇒ (True, "rcx", 0,32)
      | "cx"   ⇒ (False, "rcx", 0,16)
      | "ch"   ⇒ (False, "rcx", 8,16)
      | "cl"   ⇒ (False, "rcx", 0,8)
```

```

| "rdx"  => (True, "rdx", 0,64)
| "edx"  => (True, "rdx", 0,32)
| "dx"   => (False, "rdx", 0,16)
| "dh"   => (False, "rdx", 8,16)
| "dl"   => (False, "rdx", 0,8)
— RBP, RSP
| "rbp"  => (True, "rbp", 0,64)
| "ebp"  => (True, "rbp", 0,32)
| "bp"   => (False, "rbp", 0,16)
| "bpl"  => (False, "rbp", 0,8)
| "rsp"  => (True, "rsp", 0,64)
| "esp"  => (True, "rsp", 0,32)
| "sp"   => (False, "rsp", 0,16)
| "spl"  => (False, "rsp", 0,8)
— RDI, RSI, R8 to R15
| "rdi"  => (True, "rdi", 0,64)
| "edi"  => (True, "rdi", 0,32)
| "di"   => (False, "rdi", 0,16)
| "dil"  => (False, "rdi", 0,8)
| "rsi"  => (True, "rsi", 0,64)
| "esi"  => (True, "rsi", 0,32)
| "si"   => (False, "rsi", 0,16)
| "sil"  => (False, "rsi", 0,8)
| "r15"  => (True, "r15", 0,64)
| "r15d" => (True, "r15", 0,32)
| "r15w" => (False, "r15", 0,16)
| "r15b" => (False, "r15", 0,8)
| "r14"  => (True, "r14", 0,64)
| "r14d" => (True, "r14", 0,32)
| "r14w" => (False, "r14", 0,16)
| "r14b" => (False, "r14", 0,8)
| "r13"  => (True, "r13", 0,64)
| "r13d" => (True, "r13", 0,32)
| "r13w" => (False, "r13", 0,16)
| "r13b" => (False, "r13", 0,8)
| "r12"  => (True, "r12", 0,64)
| "r12d" => (True, "r12", 0,32)
| "r12w" => (False, "r12", 0,16)
| "r12b" => (False, "r12", 0,8)
| "r11"  => (True, "r11", 0,64)
| "r11d" => (True, "r11", 0,32)
| "r11w" => (False, "r11", 0,16)
| "r11b" => (False, "r11", 0,8)
| "r10"  => (True, "r10", 0,64)
| "r10d" => (True, "r10", 0,32)
| "r10w" => (False, "r10", 0,16)
| "r10b" => (False, "r10", 0,8)
| "r9"   => (True, "r9", 0,64)
| "r9d"  => (True, "r9", 0,32)

```

```

| "r9w"  => (False, "r9", 0,16)
| "r9b"  => (False, "r9", 0,8)
| "r8"   => (True,  "r8", 0,64)
| "r8d"  => (True,  "r8", 0,32)
| "r8w"  => (False, "r8", 0,16)
| "r8b"  => (False, "r8", 0,8)
— xmm
| "xmm0" => (True, "xmm0", 0,128)
| "xmm1" => (True, "xmm1", 0,128)
| "xmm2" => (True, "xmm2", 0,128)
| "xmm3" => (True, "xmm3", 0,128)
| "xmm4" => (True, "xmm4", 0,128)
| "xmm5" => (True, "xmm5", 0,128)
| "xmm6" => (True, "xmm6", 0,128)
| "xmm7" => (True, "xmm7", 0,128)
| "xmm8" => (True, "xmm8", 0,128)
| "xmm9" => (True, "xmm9", 0,128)
| "xmm10" => (True, "xmm10", 0,128)
| "xmm11" => (True, "xmm11", 0,128)
| "xmm12" => (True, "xmm12", 0,128)
| "xmm13" => (True, "xmm13", 0,128)
| "xmm14" => (True, "xmm14", 0,128)
| "xmm15" => (True, "xmm15", 0,128)

```

x86 has register aliasing. For example, register EAX is the lower 32 bits of register RAX. This function map register aliases to the “real” register. For example:

```
real-reg "ah" = (False, "rax", 8, 16).
```

This means that register AH is the second byte (bits 8 to 16) of register RAX. The bool *False* indicates that writing to AH does not overwrite the remainder of RAX.

```
real-reg "eax" = (True, "rax", 0, 32).
```

Register EAX is the lower 4 bytes of RAX. Writing to EAX means overwriting the remainder of RAX with zeroes.

```
definition reg-size :: string  $\Rightarrow$  nat — in bytes
where reg-size reg  $\equiv$  let  $(-, -, l, h)$  = real-reg reg in  $(h - l) \text{ div } 8$ 
```

We now define functions for reading and writing from state.

```
definition reg-read :: state  $\Rightarrow$  string  $\Rightarrow$  256 word
where reg-read  $\sigma$  reg  $\equiv$ 
  if reg = "rip" then ucast (rip  $\sigma$ ) else
  if reg = "" then 0 else — happens if no base register is used in an address
  let  $(-, r, l, h)$  = real-reg reg in
     $\langle l, h \rangle$ (regs  $\sigma$  r)
```

```
primrec fromBool :: bool  $\Rightarrow$  'a :: len word
where
```

```

fromBool True = 1
| fromBool False = 0

definition flag-read :: state ⇒ string ⇒ 256 word
  where flag-read σ flag ≡ fromBool (flags σ flag)

definition mem-read :: state ⇒ 64 word ⇒ nat ⇒ 256 word
  where mem-read σ a si ≡ word-read (read-bytes (mem σ) a si)

```

Doing state-updates occur through a tiny deeply embedded language of state updates. This allows us to reason over state updates through theorems.

```

datatype StateUpdate =
  RegUpdate string 256 word      — Write value to register
  | FlagUpdate string ⇒ bool     — Update all flags at once
  | RipUpdate 64 word            — Update instruction pointer with address
  | MemUpdate 64 word nat 256 word — Write a number of bytes of a value to the
address

primrec state-update
  where
    state-update (RegUpdate reg val) = (λ σ . σ(regs := (regs σ)(reg := val)))
    | state-update (FlagUpdate val)   = (λ σ . σ(flags := val))
    | state-update (RipUpdate a)     = (λ σ . σ(rip := a))
    | state-update (MemUpdate a si val) = (λ σ .
      let new = (λ a'. take-byte (unat (a' - a)) val) in
      σ(mem := override-on (mem σ) new (region-addresses a si)))

```

```

abbreviation RegUpdateSyntax (⟨- :=r -⟩ 30)
  where RegUpdateSyntax reg val ≡ RegUpdate reg val
abbreviation MemUpdateSyntax (⟨[],[] :=m -⟩ 30)
  where MemUpdateSyntax a si val ≡ MemUpdate a si val
abbreviation FlagUpdateSyntax (⟨setFlags⟩)
  where FlagUpdateSyntax val ≡ FlagUpdate val
abbreviation RipUpdateSyntax (⟨setRip⟩)
  where RipUpdateSyntax val ≡ RipUpdate val

```

Executes a write to a register in terms of the tiny deeply embedded language above.

```

definition reg-write
  where reg-write reg val σ ≡
    let (b,r,l,h) = real-reg reg;
        curr-val = reg-read σ r;
        new-val   = if b then val else overwrite l h curr-val val in
    state-update (RegUpdate r new-val) σ

```

A datatype for operands of instructions.

```

datatype Operand =
  Imm 256 word
  | Reg string

```

| *Flag string*  
 | *Mem nat 64 word string string nat*  
 — size offset base-reg index-reg scale

**abbreviation** *mem-op-no-offset-no-index* :: *string*  $\Rightarrow$  (*64 word*  $\times$  *string*  $\times$  *string*  $\times$  *nat*) ( $\langle \cdot \rangle_1$  40)  
**where** *mem-op-no-offset-no-index r*  $\equiv$  (*0,r,[]*,0)

**abbreviation** *mem-op-no-index* :: *64 word*  $\Rightarrow$  *string*  $\Rightarrow$  (*64 word*  $\times$  *string*  $\times$  *string*  $\times$  *nat*) ( $\langle \cdot + \cdot \rangle_2$  40)  
**where** *mem-op-no-index offset r*  $\equiv$  (*offset,r,[]*,0)

**abbreviation** *mem-op* :: *64 word*  $\Rightarrow$  *string*  $\Rightarrow$  *string*  $\Rightarrow$  *nat*  $\Rightarrow$  (*64 word*  $\times$  *string*  $\times$  *string*  $\times$  *nat*) ( $\langle \cdot + \cdot + \cdot * \cdot \rangle_3$  40)  
**where** *mem-op offset r index scale*  $\equiv$  (*offset,r,index,scale*)

**definition** *ymm-ptr* ( $\langle YMMWORD PTR \rightarrow \rangle$ )  
**where** *YMMWORD PTR x*  $\equiv$  *case x of (offset,base,index,scale)  $\Rightarrow$  Mem 32 offset base index scale*

**definition** *xmm-ptr* ( $\langle XMMWORD PTR \rightarrow \rangle$ )  
**where** *XMMWORD PTR x*  $\equiv$  *case x of (offset,base,index,scale)  $\Rightarrow$  Mem 16 offset base index scale*

**definition** *qword-ptr* ( $\langle QWORD PTR \rightarrow \rangle$ )  
**where** *QWORD PTR x*  $\equiv$  *case x of (offset,base,index,scale)  $\Rightarrow$  Mem 8 offset base index scale*

**definition** *dword-ptr* ( $\langle DWORD PTR \rightarrow \rangle$ )  
**where** *DWORD PTR x*  $\equiv$  *case x of (offset,base,index,scale)  $\Rightarrow$  Mem 4 offset base index scale*

**definition** *word-ptr* ( $\langle WORD PTR \rightarrow \rangle$ )  
**where** *WORD PTR x*  $\equiv$  *case x of (offset,base,index,scale)  $\Rightarrow$  Mem 2 offset base index scale*

**definition** *byte-ptr* ( $\langle BYTE PTR \rightarrow \rangle$ )  
**where** *BYTE PTR x*  $\equiv$  *case x of (offset,base,index,scale)  $\Rightarrow$  Mem 1 offset base index scale*

**primrec** (*nonexhaustive*) *operand-size* :: *Operand*  $\Rightarrow$  *nat* — in bytes  
**where**  
*operand-size (Reg r) = reg-size r*  
| *operand-size (Mem si - - -) = si*

**fun** *resolve-address* :: *state*  $\Rightarrow$  *64 word*  $\Rightarrow$  *char list*  $\Rightarrow$  *char list*  $\Rightarrow$  *nat*  $\Rightarrow$  *64 word*  
**where** *resolve-address σ offset base index scale =*  
*(let i = ucast (reg-read σ index);*

```

 $b = \text{ucast}(\text{reg-read } \sigma \text{ base}) \text{ in}$ 
 $\text{offset} + b + \text{of-nat scale}*i)$ 

primrec operand-read :: state  $\Rightarrow$  Operand  $\Rightarrow$  256 word
where
  operand-read  $\sigma$  (Imm i) = i
  | operand-read  $\sigma$  (Reg r) = reg-read  $\sigma$  r
  | operand-read  $\sigma$  (Flag f) = flag-read  $\sigma$  f
  | operand-read  $\sigma$  (Mem si offset base index scale) =
    (let a = resolve-address  $\sigma$  offset base index scale in
     mem-read  $\sigma$  a si
    )
  )

primrec state-with-updates :: state  $\Rightarrow$  StateUpdate list  $\Rightarrow$  state (infixl  $\langle\text{with}\rangle$  66)
where
   $\sigma \text{ with } [] = \sigma$ 
  |  $(\sigma \text{ with } (f \# fs)) = \text{state-update } f (\sigma \text{ with } fs)$ 

primrec (nonexhaustive) operand-write :: Operand  $\Rightarrow$  256 word  $\Rightarrow$  state  $\Rightarrow$  state
where
  operand-write (Reg r) v  $\sigma$  = reg-write r v  $\sigma$ 
  | operand-write (Mem si offset base index scale) v  $\sigma$  =
    (let i = ucast (reg-read  $\sigma$  index);
     b = ucast (reg-read  $\sigma$  base);
     a = offset + b + of-nat scale*i in
      $\sigma \text{ with } [[a, si]] :=_m v$ 
    )
  )

```

The following theorems simplify reading from state parts after doing updates to other state parts.

```

lemma regs-reg-write:
  shows regs ( $\sigma$  with  $((r :=_r w) \# \text{updates})$ ) r' = (if r=r' then w else regs ( $\sigma$  with updates) r')
   $\langle\text{proof}\rangle$ 

lemma regs-mem-write:
  shows regs ( $\sigma$  with  $(([a, si] :=_m v) \# \text{updates})$ ) r = regs ( $\sigma$  with updates) r
   $\langle\text{proof}\rangle$ 

lemma regs-flag-write:
  shows regs ( $\sigma$  with  $((\text{setFlags } v) \# \text{updates})$ ) r = regs ( $\sigma$  with updates) r
   $\langle\text{proof}\rangle$ 

lemma regs-rip-write:
  shows regs ( $\sigma$  with  $((\text{setRip } a) \# \text{updates})$ ) f = regs ( $\sigma$  with updates) f
   $\langle\text{proof}\rangle$ 

```

```

lemma flag-read-reg-write:
  shows flag-read ( $\sigma$  with  $((r :=_r w)\#updates)$ )  $f = \text{flag-read } (\sigma \text{ with updates}) f$ 
   $\langle proof \rangle$ 

lemma flag-read-mem-write:
  shows flag-read ( $\sigma$  with  $((\llbracket a, si \rrbracket :=_m v)\#updates)$ )  $f = \text{flag-read } (\sigma \text{ with updates}) f$ 
   $\langle proof \rangle$ 

lemma flag-read-flag-write:
  shows flag-read ( $\sigma$  with  $((\text{setFlags } v)\#updates)$ ) =  $\text{fromBool } o v$ 
   $\langle proof \rangle$ 

lemma flag-read-rip-write:
  shows flag-read ( $\sigma$  with  $((\text{setRip } a)\#updates)$ )  $f = \text{flag-read } (\sigma \text{ with updates}) f$ 
   $\langle proof \rangle$ 

lemma mem-read-reg-write:
  shows mem-read ( $\sigma$  with  $((r :=_r w)\#updates)$ )  $a si = \text{mem-read } (\sigma \text{ with updates}) a si$ 
   $\langle proof \rangle$ 

lemma mem-read-flag-write:
  shows mem-read ( $\sigma$  with  $((\text{setFlags } v)\#updates)$ )  $a si = \text{mem-read } (\sigma \text{ with updates}) a si$ 
   $\langle proof \rangle$ 

lemma mem-read-rip-write:
  shows mem-read ( $\sigma$  with  $((\text{setRip } a')\#updates)$ )  $a si = \text{mem-read } (\sigma \text{ with updates}) a si$ 
   $\langle proof \rangle$ 

lemma mem-read-mem-write-alias:
  assumes  $si' \leq si$ 
  and  $si \leq 2^{64}$ 
  shows mem-read ( $\sigma$  with  $((\llbracket a, si \rrbracket :=_m v)\#updates)$ )  $a si' = \langle 0, si'*8 \rangle v$ 
   $\langle proof \rangle$ 

lemma mem-read-mem-write-separate:
  assumes separate  $a si a' si'$ 
  shows mem-read ( $\sigma$  with  $((\llbracket a, si \rrbracket :=_m v)\#updates)$ )  $a' si' = \text{mem-read } (\sigma \text{ with updates}) a' si'$ 
   $\langle proof \rangle$ 

lemma mem-read-mem-write-enclosed-minus:
  assumes  $offset' \leq offset$ 
  and  $si' \leq si$ 
  and  $\text{unat } (offset - offset') + si' < 2^{64}$ 
  and  $\text{unat } offset + si' \leq si + \text{unat } offset'$ 

```

```

shows mem-read ( $\sigma$  with  $((\llbracket a - offset, si \rrbracket :=_m v) \# updates))$   $(a - offset') si' =$ 
 $(unat(offset - offset') * 8, unat(offset - offset') * 8 + si' * 8) v$ 
 $\langle proof \rangle$ 

lemma mem-read-mem-write-enclosed-plus:
assumes  $unat(offset + si') \leq si$ 
and  $si < 2^{64}$ 
shows mem-read ( $\sigma$  with  $((\llbracket a, si \rrbracket :=_m v) \# updates))$   $(offset + a) si' = \langle unat$ 
 $offset * 8, (unat(offset + si') * 8) v$ 
 $\langle proof \rangle$ 

lemma mem-read-mem-write-enclosed-plus2:
assumes  $unat(offset + si') \leq si$ 
and  $si < 2^{64}$ 
shows mem-read ( $\sigma$  with  $((\llbracket a, si \rrbracket :=_m v) \# updates))$   $(a + offset) si' = \langle unat$ 
 $offset * 8, (unat(offset + si') * 8) v$ 
 $\langle proof \rangle$ 

lemma mem-read-mem-write-enclosed-numeral[simp]:
assumes  $unat(\text{numeral } a' - \text{numeral } a :: 64 \text{ word}) + (\text{numeral } si' :: nat) \leq \text{numeral } si$ 
and  $\text{numeral } a' \geq (\text{numeral } a :: 64 \text{ word})$ 
and  $\text{numeral } si < (2^{64} :: nat)$ 
shows mem-read ( $\sigma$  with  $((\llbracket \text{numeral } a, \text{numeral } si \rrbracket :=_m v) \# updates))$   $(\text{numeral } a')$ 
 $(\text{numeral } si') = \langle unat(\text{numeral } a' - (\text{numeral } a :: 64 \text{ word})) * 8, (unat(\text{numeral } a'$ 
 $- (\text{numeral } a :: 64 \text{ word})) + (\text{numeral } si') * 8 \rangle v$ 
 $\langle proof \rangle$ 

lemma mem-read-mem-write-enclosed-numeral1-[simp]:
assumes  $unat(\text{numeral } a' - \text{numeral } a :: 64 \text{ word}) + (\text{numeral } si' :: nat) \leq Suc 0$ 
and  $\text{numeral } a' \geq (\text{numeral } a :: 64 \text{ word})$ 
shows mem-read ( $\sigma$  with  $((\llbracket \text{numeral } a, Suc 0 \rrbracket :=_m v) \# updates))$   $(\text{numeral } a')$ 
 $(\text{numeral } si') = \langle unat(\text{numeral } a' - (\text{numeral } a :: 64 \text{ word})) * 8, (unat(\text{numeral } a'$ 
 $- (\text{numeral } a :: 64 \text{ word})) + (\text{numeral } si') * 8 \rangle v$ 
 $\langle proof \rangle$ 

lemma mem-read-mem-write-enclosed-numeral-1[simp]:
assumes  $unat(\text{numeral } a' - \text{numeral } a :: 64 \text{ word}) + (Suc 0) \leq \text{numeral } si$ 
and  $\text{numeral } a' \geq (\text{numeral } a :: 64 \text{ word})$ 
and  $\text{numeral } si < (2^{64} :: nat)$ 
shows mem-read ( $\sigma$  with  $((\llbracket \text{numeral } a, \text{numeral } si \rrbracket :=_m v) \# updates))$   $(\text{numeral } a')$ 
 $(Suc 0) = \langle unat(\text{numeral } a' - (\text{numeral } a :: 64 \text{ word})) * 8, (unat(\text{numeral } a'$ 
 $- (\text{numeral } a :: 64 \text{ word})) + (Suc 0) * 8 \rangle v$ 
 $\langle proof \rangle$ 

lemma mem-read-mem-write-enclosed-numeral11[simp]:
assumes  $unat(\text{numeral } a' - \text{numeral } a :: 64 \text{ word}) + (Suc 0) \leq Suc 0$ 
and  $\text{numeral } a' \geq (\text{numeral } a :: 64 \text{ word})$ 
```

**shows**  $\text{mem-read}(\sigma \text{ with } ((\llbracket \text{numeral } a, \text{Suc } 0 \rrbracket :=_m v) \# \text{updates})) (\text{numeral } a')$   
 $(\text{Suc } 0) = \langle \text{unat } (\text{numeral } a' - (\text{numeral } a :: 64 \text{ word})) * 8, (\text{unat } (\text{numeral } a' - (\text{numeral } a :: 64 \text{ word})) + (\text{Suc } 0)) * 8 \rangle v$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{rip-reg-write}[simp]$ :  
**shows**  $\text{rip}(\sigma \text{ with } ((r :=_r v) \# \text{updates})) = \text{rip}(\sigma \text{ with updates})$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{rip-flag-write}[simp]$ :  
**shows**  $\text{rip}(\sigma \text{ with } ((\text{setFlags } v) \# \text{updates})) = \text{rip}(\sigma \text{ with updates})$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{rip-mem-write}[simp]$ :  
**shows**  $\text{rip}(\sigma \text{ with } ((\llbracket a, si \rrbracket :=_m v) \# \text{updates})) = \text{rip}(\sigma \text{ with updates})$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{rip-rip-write}[simp]$ :  
**shows**  $\text{rip}(\sigma \text{ with } ((\text{setRip } a) \# \text{updates})) = a$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{with-with}$ :  
**shows**  $(\sigma \text{ with updates}) \text{ with updates}' = \sigma \text{ with } (\text{updates}' @ \text{updates})$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{add-state-update-to-list}$ :  
**shows**  $\text{state-update upd}(\sigma \text{ with updates}) = \sigma \text{ with } (\text{upd} \# \text{updates})$   
 $\langle \text{proof} \rangle$

The updates performed to a state are ordered: memoery, registers, flags, rip. This function is basically insertion sort. Moreover, consecutive updates to the same register are removed.

```
fun insert-state-update
  where
    insert-state-update (setRip a) (setRip a' # updates) = insert-state-update (setRip a) updates
    | insert-state-update (setRip a) (setFlags v # updates) = setFlags v # (insert-state-update (setRip a) updates)
    | insert-state-update (setRip a) ((r :=_r v) # updates) = (r :=_r v) # (insert-state-update (setRip a) updates)
    | insert-state-update (setRip a) ((\llbracket a', si \rrbracket :=_m v) # updates) = (\llbracket a', si \rrbracket :=_m v) # (insert-state-update (setRip a) updates)
    | insert-state-update (setFlags v) (setFlags v' # updates) = insert-state-update (setFlags v) updates
    | insert-state-update (setFlags v) ((r :=_r v') # updates) = (r :=_r v') # insert-state-update (setFlags v) updates
```

```

| insert-state-update (setFlags v) (([a',si] :=_m v')#updates) = ([a',si] :=_m v') #
insert-state-update (setFlags v) updates

| insert-state-update ((r :=_r v)) ((r' :=_r v')#updates) = (if r = r' then insert-state-update (r :=_r v) updates else (r' :=_r v')#insert-state-update (r :=_r v) updates)
| insert-state-update ((r :=_r v)) (([a',si] :=_m v')#updates) = ([a',si] :=_m v') #
insert-state-update (r :=_r v) updates

| insert-state-update upd updates = upd # updates

fun clean
where
  clean [] = []
  | clean [upd] = [upd]
  | clean (upd#upd'#updates) = insert-state-update upd (clean (upd'#updates))

lemma insert-state-update:
  shows σ with (insert-state-update upd updates) = σ with (upd # updates)
  ⟨proof⟩

lemma clean-state-updates:
  shows σ with (clean updates) = σ with updates
  ⟨proof⟩

The set of simplification rules used during symbolic execution.

lemmas state-simps =
  qword-ptr-def dword-ptr-def word-ptr-def byte-ptr-def reg-size-def
  reg-write-def real-reg-def reg-read-def

  regs-rip-write regs-mem-write regs-reg-write regs-flag-write
  flag-read-reg-write flag-read-mem-write flag-read-rip-write flag-read-flag-write
  mem-read-reg-write mem-read-flag-write mem-read-rip-write
  mem-read-mem-write-alias mem-read-mem-write-separate
  mem-read-mem-write-enclosed-minus mem-read-mem-write-enclosed-plus mem-read-mem-write-enclosed-pl
  with-with add-state-update-to-list

declare state-with-updates.simps(2)[simp del]
declare state-update.simps[simp del]

end

```

## 4 Instruction Semantics

```

theory X86-InstructionSemantics
  imports State
  begin

```

A datatype for storing instructions. Note that we add a special kind of meta-instruction, called ExternalFunc. A call to an external function can manually be mapped to a manually supplied state transformation function.

```
datatype I =
  Instr string Operand option Operand option Operand option 64 word
  | ExternalFunc state ⇒ state
```

A datatype for the result of floating point comparisons.

```
datatype FP-Order = FP-Unordered | FP-GT | FP-LT | FP-EQ
```

**abbreviation** instr-next  $i \equiv \text{case } i \text{ of } (\text{Instr} \dots a') \Rightarrow a'$

```
locale unknowns =
  fixes unknown-addsd :: 64 word ⇒ 64 word ⇒ 64 word
  and unknown-subsd :: 64 word ⇒ 64 word ⇒ 64 word
  and unknown-mulsd :: 64 word ⇒ 64 word ⇒ 64 word
  and unknown-divsd :: 64 word ⇒ 64 word ⇒ 64 word
  and unknown-ucomisd :: 64 word ⇒ 64 word ⇒ FP-Order
  and unknown-semantics :: I ⇒ state ⇒ state
  and unknown-flags :: string ⇒ string ⇒ bool
begin
```

The semantics below are intended to be overapproximative and incomplete. This is achieved using locale “unknowns”. Any place where semantics is *not* modelled, it is mapped to a universally quantified uninterpreted function from that locale. We do not make use of *undefined*, since that could be used to prove that the semantics of two undefined behaviors are equivalent. For example:

- Only a subset of instructions has semantics. In case of an unknown instruction  $i$ , the function *semantics* below will result in *unknown-semantics*  $i$ .
- Not all flags have been defined. In case a flag is read whose semantics is not defined below, the read will resolve to *unknown-flags*  $i$ . Note that if the semantics of an instruction do not set flags, an overapproximative semantics such as below imply that the instruction indeed does not modify flags. In other words, if we were uncertain we would assign unknown values to flags.
- Not all operations have been defined. For example, floating points operations have no executable semantics, but are mapped to uninterpreted functions such as *unknown-addsd*.

Moves

```
definition semantics-MOV :: Operand ⇒ Operand ⇒ state ⇒ state
```

```

where semantics-MOV op1 op2 σ ≡
  let src = operand-read σ op2 in
    operand-write op1 src σ

abbreviation MOV
where MOV op1 op2 ≡ Instr "mov" (Some op1) (Some op2) None

abbreviation MOVABS
where MOVABS op1 op2 ≡ Instr "movabs" (Some op1) (Some op2) None

abbreviation MOVAPS
where MOVAPS op1 op2 ≡ Instr "movaps" (Some op1) (Some op2) None

abbreviation MOVZX
where MOVZX op1 op2 ≡ Instr "movzx" (Some op1) (Some op2) None

abbreviation MOVDQU
where MOVDQU op1 op2 ≡ Instr "movdqu" (Some op1) (Some op2) None

definition semantics-MOVD :: Operand ⇒ Operand ⇒ state ⇒ state
where semantics-MOVD op1 op2 σ ≡
  let src = ucast(operand-read σ op2)::32word in
    operand-write op1 (ucast src) σ

abbreviation MOVD
where MOVD op1 op2 ≡ Instr "movd" (Some op1) (Some op2) None

fun isXMM :: Operand ⇒ bool
where isXMM (Reg r) = (take 3 r = "xmm")
  | isXMM - = False

definition semantics-MOVSD :: Operand ⇒ Operand ⇒ state ⇒ state
where semantics-MOVSD op1 op2 σ ≡
  if isXMM op1 ∧ isXMM op2 then
    let src = ⟨0,64⟩ operand-read σ op2;
      dst = ⟨64,128⟩ operand-read σ op1 in
        operand-write op1 (overwrite 0 64 dst src) σ
  else
    let src = ⟨0,64⟩ operand-read σ op2 in
      operand-write op1 src σ

abbreviation MOVSD
where MOVSD op1 op2 ≡ Instr "movsd" (Some op1) (Some op2) None

abbreviation MOVQ
where MOVQ op1 op2 ≡ Instr "movq" (Some op1) (Some op2) None
  lea/push/pop/call/ret/leave

definition semantics-LEA :: Operand ⇒ Operand ⇒ state ⇒ state

```

**where** *semantics-LEA op1 op2 σ ≡*  
*case op2 of Mem si offset base index scale ⇒*  
*operand-write op1 (ucast (resolve-address σ offset base index scale)) σ*

**abbreviation** *LEA*  
**where** *LEA op1 op2 ≡ Instr "lea" (Some op1) (Some op2) None*

**definition** *semantics-PUSH :: Operand ⇒ state ⇒ state*  
**where** *semantics-PUSH op1 σ ≡*  
*let src = operand-read σ op1;*  
*si = operand-size op1;*  
*rsp = ucast (ucast (reg-read σ "rsp") - of-nat si :: 64 word) in*  
*operand-write (QWORD PTR ["rsp"]1) src (operand-write (Reg "rsp")*  
*rsp σ)*

**abbreviation** *PUSH*  
**where** *PUSH op1 ≡ Instr "push" (Some op1) None None*

**definition** *semantics-POP :: Operand ⇒ state ⇒ state*  
**where** *semantics-POP op1 σ ≡*  
*let si = operand-size op1;*  
*src = operand-read σ (QWORD PTR ["rsp"]1);*  
*rsp = ucast (ucast (reg-read σ "rsp") + of-nat si :: 64 word) in*  
*operand-write op1 src (operand-write (Reg "rsp") rsp σ)*

**abbreviation** *POP*  
**where** *POP op1 ≡ Instr "pop" (Some op1) None None*

**definition** *semantics-CALL :: Operand ⇒ state ⇒ state*  
**where** *semantics-CALL op1 σ ≡*  
*let src = ucast (operand-read σ op1) in*  
*(state-update (setRip src) o semantics-PUSH (Reg "rip")) σ*

**definition** *semantics-RET :: state ⇒ state*  
**where** *semantics-RET σ ≡*  
*let a = ucast (operand-read σ (QWORD PTR ["rsp"]1));*  
*rsp = ucast (reg-read σ "rsp") + 8 :: 64 word in*  
*(state-update (setRip a) o operand-write (Reg "rsp") (ucast rsp)) σ*

**abbreviation** *RET*  
**where** *RET ≡ Instr "ret" None None None*

**definition** *semantics-LEAVE :: state ⇒ state*  
**where** *semantics-LEAVE ≡ semantics-POP (Reg "rbp") o semantics-MOV (Reg "rsp") (Reg "rbp")*

**abbreviation** *LEAVE*  
**where** *LEAVE op1 ≡ Instr "pop" (Some op1) None None*

Generic operators

```

definition unop :: ('a ::len word  $\Rightarrow$  'a::len word)  $\Rightarrow$ 
            ('a::len word  $\Rightarrow$  string  $\Rightarrow$  bool)  $\Rightarrow$ 
            Operand  $\Rightarrow$  state  $\Rightarrow$  state
where unop f g op1  $\sigma \equiv$ 
        let si = operand-size op1;
        dst = ucast (operand-read  $\sigma$  op1)::'a::len word in
        operand-write op1 (ucast (f dst)) ( $\sigma$  with [setFlags (g dst)])
        
definition binop :: ('a::len word  $\Rightarrow$  'a ::len word  $\Rightarrow$  'a::len word)  $\Rightarrow$ 
            ('a::len word  $\Rightarrow$  'a::len word  $\Rightarrow$  string  $\Rightarrow$  bool)  $\Rightarrow$ 
            Operand  $\Rightarrow$  Operand  $\Rightarrow$  state  $\Rightarrow$  state
where binop f g op1 op2  $\sigma \equiv$ 
        let dst = ucast (operand-read  $\sigma$  op1)::'a::len word;
        src = ucast (operand-read  $\sigma$  op2)::'a::len word in
        operand-write op1 (ucast (f dst src)) ( $\sigma$  with [setFlags (g dst src)])
        
definition unop-no-flags :: ('a ::len word  $\Rightarrow$  'a::len word)  $\Rightarrow$  Operand  $\Rightarrow$  state  $\Rightarrow$ 
state
where unop-no-flags f op1  $\sigma \equiv$ 
        let dst = ucast (operand-read  $\sigma$  op1)::'a::len word in
        operand-write op1 (ucast (f dst))  $\sigma$ 
        
definition binop-flags :: ('a::len word  $\Rightarrow$  'a::len word  $\Rightarrow$  string  $\Rightarrow$  bool)  $\Rightarrow$ 
            Operand  $\Rightarrow$  Operand  $\Rightarrow$  state  $\Rightarrow$  state
where binop-flags g op1 op2  $\sigma \equiv$ 
        let si = operand-size op1;
        dst = ucast (operand-read  $\sigma$  op1)::'a::len word;
        src = ucast (operand-read  $\sigma$  op2)::'a::len word in
         $\sigma$  with [setFlags (g dst src)]
        
definition binop-no-flags :: ('a::len word  $\Rightarrow$  'a ::len word  $\Rightarrow$  'a::len word)  $\Rightarrow$ 
            Operand  $\Rightarrow$  Operand  $\Rightarrow$  state  $\Rightarrow$  state
where binop-no-flags f op1 op2  $\sigma \equiv$ 
        let si = operand-size op1;
        dst = ucast (operand-read  $\sigma$  op1)::'a::len word;
        src = ucast (operand-read  $\sigma$  op2)::'a::len word in
        operand-write op1 (ucast (f dst src))  $\sigma$ 
        
definition binop-XMM :: (64 word  $\Rightarrow$  64 word  $\Rightarrow$  64 word)  $\Rightarrow$  Operand  $\Rightarrow$  Operand
 $\Rightarrow$  state  $\Rightarrow$  state
where binop-XMM f op1 op2  $\sigma \equiv$ 
        let dst = ucast (operand-read  $\sigma$  op1)::64word;
        src = ucast (operand-read  $\sigma$  op2)::64word in
        operand-write op1 (ucast (overwrite 0 64 dst (f dst src)))  $\sigma$ 

```

### Arithmetic

```

definition ADD-flags :: 'a::len word  $\Rightarrow$  'a::len word  $\Rightarrow$  string  $\Rightarrow$  bool
where ADD-flags w0 w1 flag  $\equiv$  case flag of
    "zf"  $\Rightarrow$  w0 + w1 = 0

```

```

| "cf" ⇒ unat w0 + unat w1 ≥ 2^(LENGTH('a))
| "of" ⇒ (w0 < s 0 ↔ w1 < s 0) ∧ ¬(w0 < s 0 ↔ w0+w1 < s 0)
| "sf" ⇒ w0 + w1 < s 0
| f     ⇒ unknown-flags "ADD" f

definition semantics-ADD :: Operand ⇒ Operand ⇒ state ⇒ state
where semantics-ADD op1 ≡
    if operand-size op1 = 32 then binop ((+):256 word ⇒ - ⇒ -) ADD-flags
op1
    else if operand-size op1 = 16 then binop ((+):128 word ⇒ - ⇒ -) ADD-flags
op1
    else if operand-size op1 = 8  then binop ((+):64  word ⇒ - ⇒ -) ADD-flags
op1
    else if operand-size op1 = 4  then binop ((+):32  word ⇒ - ⇒ -) ADD-flags
op1
    else if operand-size op1 = 2  then binop ((+):16  word ⇒ - ⇒ -) ADD-flags
op1
    else if operand-size op1 = 1  then binop ((+):8   word ⇒ - ⇒ -) ADD-flags
op1
    else undefined

```

```

abbreviation ADD
where ADD op1 op2 ≡ Instr "add" (Some op1) (Some op2) None

```

```

definition INC-flags :: 256 word ⇒ ('a::len word ⇒ string ⇒ bool)
where INC-flags cf w0 flag ≡ case flag of
    "zf" ⇒ w0 + 1 = 0
    | "cf" ⇒ cf ≠ 0
    | "of" ⇒ 0 <= s w0 ∧ w0+1 < s 0
    | "sf" ⇒ w0 + 1 < s 0
    | f     ⇒ unknown-flags "INC" f

```

```

definition semantics-INC :: Operand ⇒ state ⇒ state
where semantics-INC op1 σ ≡
    let cf = flag-read σ "cf" in
        if operand-size op1 = 32 then unop ((+) (1::256 word)) (INC-flags cf)
op1 σ
        else if operand-size op1 = 16 then unop ((+) (1::128 word)) (INC-flags cf)
op1 σ
        else if operand-size op1 = 8  then unop ((+) (1::64  word)) (INC-flags cf)
op1 σ
        else if operand-size op1 = 4  then unop ((+) (1::32  word)) (INC-flags cf)
op1 σ
        else if operand-size op1 = 2  then unop ((+) (1::16  word)) (INC-flags cf)
op1 σ
        else if operand-size op1 = 1  then unop ((+) (1::8   word)) (INC-flags cf) op1 σ
        else undefined

```

**abbreviation** *INC*

**where** *INC op1*  $\equiv$  *Instr "inc" (Some op1) None None*

**definition** *DEC-flags* ::  $256 \text{ word} \Rightarrow ('a:\text{len word} \Rightarrow \text{string} \Rightarrow \text{bool})$

**where** *DEC-flags cf w0 flag*  $\equiv$  *case flag of*  
  | *"zf"*  $\Rightarrow$  *w0 = 0*  
  | *"cf"*  $\Rightarrow$  *cf*  $\neq 0  
  | *"of"*  $\Rightarrow$  *w0 <_s 0 \wedge 0 <= s w0 - 1*  
  | *"sf"*  $\Rightarrow$  *w0 - 1 <_s 0*  
  | *f*       $\Rightarrow$  *unknown-flags "DEC" f*$

**definition** *semantics-DEC* :: *Operand*  $\Rightarrow$  *state*  $\Rightarrow$  *state*

**where** *semantics-DEC op1 σ*  $\equiv$   
  *let cf = flag-read σ "cf" in*  
    *if operand-size op1 = 32 then unop (λ w . w - 1::256 word) (DEC-flags cf) op1 σ*  
    *else if operand-size op1 = 16 then unop (λ w . w - 1::128 word) (DEC-flags cf) op1 σ*  
    *else if operand-size op1 = 8 then unop (λ w . w - 1::64 word) (DEC-flags cf) op1 σ*  
    *else if operand-size op1 = 4 then unop (λ w . w - 1::32 word) (DEC-flags cf) op1 σ*  
    *else if operand-size op1 = 2 then unop (λ w . w - 1::16 word) (DEC-flags cf) op1 σ*  
    *else if operand-size op1 = 1 then unop (λ w . w - 1::8 word) (DEC-flags cf) op1 σ*  
    *else undefined*

**abbreviation** *DEC*

**where** *DEC op1*  $\equiv$  *Instr "dec" (Some op1) None None*

**definition** *NEG-flags* ::  $('a:\text{len word} \Rightarrow \text{string} \Rightarrow \text{bool})$

**where** *NEG-flags w0 flag*  $\equiv$  *case flag of*  
  | *"zf"*  $\Rightarrow$  *w0 = 0*  
  | *"cf"*  $\Rightarrow$  *w0 ≠ 0*  
  | *"sf"*  $\Rightarrow$  *- w0 <\_s 0*  
  | *"of"*  $\Rightarrow$  *msb (- w0) \wedge msb w0*  
  | *f*       $\Rightarrow$  *unknown-flags "NEG" f*

**definition** *semantics-NEG* :: *Operand*  $\Rightarrow$  *state*  $\Rightarrow$  *state*

**where** *semantics-NEG op1 σ*  $\equiv$   
  *if operand-size op1 = 32 then unop (λ w0 . - (w0::256 word)) NEG-flags op1 σ*  
  *else if operand-size op1 = 16 then unop (λ w0 . - (w0::128 word)) NEG-flags op1 σ*  
  *else if operand-size op1 = 8 then unop (λ w0 . - (w0::64 word)) NEG-flags op1 σ*

```

    else if operand-size op1 = 4 then unop ( $\lambda w0 . - (w0::32\ word)$ ) NEG-flags
op1  $\sigma$ 
    else if operand-size op1 = 2 then unop ( $\lambda w0 . - (w0::16\ word)$ ) NEG-flags
op1  $\sigma$ 
    else if operand-size op1 = 1 then unop ( $\lambda w0 . - (w0::8\ word)$ ) NEG-flags
op1  $\sigma$ 
    else undefined

```

**abbreviation** NEG

where NEG op1  $\equiv$  Instr "neg" (Some op1) None None

**definition** SUB-flags :: 'a::len word  $\Rightarrow$  'a::len word  $\Rightarrow$  string  $\Rightarrow$  bool  
**where** SUB-flags w0 w1 flag  $\equiv$  case flag of  
 "zf"  $\Rightarrow$  w0 = w1  
 | "cf"  $\Rightarrow$  w0 < w1  
 | "sf"  $\Rightarrow$  w0 - w1 < s 0  
 | "of"  $\Rightarrow$  (msb w0  $\neq$  msb w1)  $\wedge$  (msb (w0 - w1) = msb w1)  
 | f  $\Rightarrow$  unknown-flags "SUB" f

**definition** semantics-SUB :: Operand  $\Rightarrow$  Operand  $\Rightarrow$  state  $\Rightarrow$  state

**where** semantics-SUB op1  $\equiv$   
 if operand-size op1 = 32 then binop ((-)::256 word  $\Rightarrow$  -  $\Rightarrow$  -) SUB-flags  
op1  
 else if operand-size op1 = 16 then binop ((-)::128 word  $\Rightarrow$  -  $\Rightarrow$  -) SUB-flags  
op1  
 else if operand-size op1 = 8 then binop ((-)::64 word  $\Rightarrow$  -  $\Rightarrow$  -) SUB-flags  
op1  
 else if operand-size op1 = 4 then binop ((-)::32 word  $\Rightarrow$  -  $\Rightarrow$  -) SUB-flags  
op1  
 else if operand-size op1 = 2 then binop ((-)::16 word  $\Rightarrow$  -  $\Rightarrow$  -) SUB-flags  
op1  
 else if operand-size op1 = 1 then binop ((-)::8 word  $\Rightarrow$  -  $\Rightarrow$  -) SUB-flags  
op1  
 else undefined

**abbreviation** SUB

where SUB op1 op2  $\equiv$  Instr "sub" (Some op1) (Some op2) None

**definition** sbb :: 'b::len word  $\Rightarrow$  'a::len word  $\Rightarrow$  'a word  $\Rightarrow$  'a word  
**where** sbb cf dst src  $\equiv$  dst - (src + ucast cf)

**definition** SBB-flags :: 'b::len word  $\Rightarrow$  'a::len word  $\Rightarrow$  'a::len word  $\Rightarrow$  string  $\Rightarrow$  bool  
**where** SBB-flags cf dst src flag  $\equiv$  case flag of  
 "zf"  $\Rightarrow$  sbb cf dst src = 0  
 | "cf"  $\Rightarrow$  dst < src + ucast cf  
 | "sf"  $\Rightarrow$  sbb cf dst src < s 0  
 | "of"  $\Rightarrow$  (msb dst  $\neq$  msb (src + ucast cf))  $\wedge$  (msb (sbb cf dst src) = msb (src + ucast cf))

$| f \Rightarrow \text{unknown-flags } "SBB" f$

**definition** *semantics-SBB* :: *Operand*  $\Rightarrow$  *Operand*  $\Rightarrow$  *state*  $\Rightarrow$  *state*  
**where** *semantics-SBB op1 op2 σ*  $\equiv$   
  let *cf* = *flag-read σ "cf"* in  
    if *operand-size op1* = 32 then *binop (sbb cf::256 word ⇒ - ⇒ -)* (*SBB-flags cf*) *op1 op2 σ*  
    else if *operand-size op1* = 16 then *binop (sbb cf::128 word ⇒ - ⇒ -)* (*SBB-flags cf*) *op1 op2 σ*  
    else if *operand-size op1* = 8 then *binop (sbb cf::64 word ⇒ - ⇒ -)* (*SBB-flags cf*) *op1 op2 σ*  
    else if *operand-size op1* = 4 then *binop (sbb cf::32 word ⇒ - ⇒ -)* (*SBB-flags cf*) *op1 op2 σ*  
    else if *operand-size op1* = 2 then *binop (sbb cf::16 word ⇒ - ⇒ -)* (*SBB-flags cf*) *op1 op2 σ*  
    else if *operand-size op1* = 1 then *binop (sbb cf::8 word ⇒ - ⇒ -)* (*SBB-flags cf*) *op1 op2 σ*  
    else undefined

**abbreviation** *SBB*

**where** *SBB op1 op2*  $\equiv$  *Instr "sbb" (Some op1) (Some op2) None*

**definition** *adc* :: '*b::len word*  $\Rightarrow$  '*a::len word*  $\Rightarrow$  '*a word*  $\Rightarrow$  '*a word*  
**where** *adc cf dst src*  $\equiv$  *dst + (src + ucast cf)*

**definition** *ADC-flags* :: '*b::len word*  $\Rightarrow$  '*a::len word*  $\Rightarrow$  '*a::len word*  $\Rightarrow$  *string*  $\Rightarrow$  *bool*  
**where** *ADC-flags cf dst src flag*  $\equiv$  *case flag of*  
  ''zf''  $\Rightarrow$  *adc cf dst src = 0*  
  | ''cf''  $\Rightarrow$  *unat dst + unat src + unat cf ≥ 2^(LENGTH('a))*  
  | ''of''  $\Rightarrow$  *(dst < s 0 ↔ src + ucast cf < s 0) ∧ ¬(dst < s 0 ↔ adc cf dst src < s 0)*  
  | ''sf''  $\Rightarrow$  *adc cf dst src < s 0*  
  | *f*  $\Rightarrow$  *unknown-flags "ADC" f*

**definition** *semantics-ADC* :: *Operand*  $\Rightarrow$  *Operand*  $\Rightarrow$  *state*  $\Rightarrow$  *state*

**where** *semantics-ADC op1 op2 σ*  $\equiv$   
  let *cf* = *flag-read σ "cf"* in  
    if *operand-size op1* = 32 then *binop (adc cf::256 word ⇒ - ⇒ -)* (*ADC-flags cf*) *op1 op2 σ*  
    else if *operand-size op1* = 16 then *binop (adc cf::128 word ⇒ - ⇒ -)* (*ADC-flags cf*) *op1 op2 σ*  
    else if *operand-size op1* = 8 then *binop (adc cf::64 word ⇒ - ⇒ -)* (*ADC-flags cf*) *op1 op2 σ*  
    else if *operand-size op1* = 4 then *binop (adc cf::32 word ⇒ - ⇒ -)* (*ADC-flags cf*) *op1 op2 σ*  
    else if *operand-size op1* = 2 then *binop (adc cf::16 word ⇒ - ⇒ -)* (*ADC-flags cf*) *op1 op2 σ*

else if operand-size op1 = 1 then binop (adc cf::8 word  $\Rightarrow$  -  $\Rightarrow$  -) (ADC-flags  
 $c_f$ ) op1 op2  $\sigma$   
 else undefined

**abbreviation** ADC

**where** ADC op1 op2  $\equiv$  Instr "adc" (Some op1) (Some op2) None

**definition** write-MUL-result :: string  $\Rightarrow$  string  $\Rightarrow$  'a::len word  $\Rightarrow$  -  $\Rightarrow$  state  $\Rightarrow$  state

**where** write-MUL-result rh rl result flgs  $\sigma$   $\equiv$

```

let si = LENGTH('a) div 2 in
  operand-write (Reg rh) (ucast ((si, 2*si) result))
  (operand-write (Reg rl) (ucast ((0, si) result)))
  ( $\sigma$  with [setFlags flgs]))

```

**definition** MUL-flags :: 'a::len word  $\Rightarrow$  string  $\Rightarrow$  bool

**where** MUL-flags result flag  $\equiv$  case flag of

```

| "cf"  $\Rightarrow$  ((LENGTH('a) div 2, LENGTH('a)) result)  $\neq$  0
| "of"  $\Rightarrow$  ((LENGTH('a) div 2, LENGTH('a)) result)  $\neq$  0
| f  $\Rightarrow$  unknown-flags "MUL" f

```

**definition** IMUL-flags :: 'a::len word  $\Rightarrow$  string  $\Rightarrow$  bool

**where** IMUL-flags result flag  $\equiv$  case flag of

```

| "cf"  $\Rightarrow$  ((LENGTH('a) div 2, LENGTH('a)) result)  $\neq$  (if result !! (LENGTH('a)
div 2 - 1) then 2^(LENGTH('a) div 2) - 1 else 0)
| "of"  $\Rightarrow$  ((LENGTH('a) div 2, LENGTH('a)) result)  $\neq$  (if result !! (LENGTH('a)
div 2 - 1) then 2^(LENGTH('a) div 2) - 1 else 0)
| f  $\Rightarrow$  unknown-flags "IMUL" f

```

**definition** unop-MUL :: 'a::len itself  $\Rightarrow$  bool  $\Rightarrow$  string  $\Rightarrow$  Operand  $\Rightarrow$  state  $\Rightarrow$  state

**where** unop-MUL - signd op1-reg op2  $\sigma$   $\equiv$

```

let cast = (if signd then scast else ucast);
dst = cast (operand-read  $\sigma$  (Reg op1-reg))::'a::len word;
src = cast (operand-read  $\sigma$  op2)::'a::len word;
prod = dst * src;
flgs = (if signd then IMUL-flags else MUL-flags) prod in
if LENGTH('a) = 16 then
  write-MUL-result "ah" op1-reg prod flgs  $\sigma$ 
else if LENGTH('a) = 32 then
  write-MUL-result "dx" op1-reg prod flgs  $\sigma$ 
else if LENGTH('a) = 64 then
  write-MUL-result "edx" op1-reg prod flgs  $\sigma$ 
else if LENGTH('a) = 128 then
  write-MUL-result "rdx" op1-reg prod flgs  $\sigma$ 

```

*else*  
*undefined*

**definition** *semantics-MUL* :: *Operand*  $\Rightarrow$  *state*  $\Rightarrow$  *state*  
**where** *semantics-MUL op2*  $\equiv$   
*if operand-size op2 = 8 then unop-MUL TYPE(128) False "rax" op2*  
*else if operand-size op2 = 4 then unop-MUL TYPE(64) False "eax" op2*  
*else if operand-size op2 = 2 then unop-MUL TYPE(32) False "ax" op2*  
*else if operand-size op2 = 1 then unop-MUL TYPE(16) False "al" op2*  
*else undefined*

**abbreviation** *MUL*  
**where** *MUL op1*  $\equiv$  *Instr "mul" (Some op1) None None*

**definition** *semantics-IMUL1* :: *Operand*  $\Rightarrow$  *state*  $\Rightarrow$  *state*  
**where** *semantics-IMUL1 op2*  $\equiv$   
*if operand-size op2 = 8 then unop-MUL TYPE(128) True "rax" op2*  
*else if operand-size op2 = 4 then unop-MUL TYPE(64) True "eax" op2*  
*else if operand-size op2 = 2 then unop-MUL TYPE(32) True "ax" op2*  
*else if operand-size op2 = 1 then unop-MUL TYPE(16) True "al" op2*  
*else undefined*

**abbreviation** *IMUL1*  
**where** *IMUL1 op1*  $\equiv$  *Instr "imul" (Some op1) None None*

**definition** *ternop-IMUL* :: *'a::len itself*  $\Rightarrow$  *Operand*  $\Rightarrow$  *Operand*  $\Rightarrow$  *state*  $\Rightarrow$  *state*  
**where** *ternop-IMUL - op1 op2 op3 σ*  $\equiv$   
*let src1 = scast (operand-read σ op2)::'a::len word;*  
*src2 = scast (operand-read σ op3)::'a::len word;*  
*prod = src1 \* src2;*  
*flgs = IMUL-flags prod in*  
*(operand-write op1 (ucast ((0,LENGTH('a) div 2) prod))*  
*(σ with [setFlags flgs]))*

**definition** *semantics-IMUL2* :: *Operand*  $\Rightarrow$  *Operand*  $\Rightarrow$  *state*  $\Rightarrow$  *state*  
**where** *semantics-IMUL2 op1 op2*  $\equiv$   
*if operand-size op1 = 8 then ternop-IMUL TYPE(128) op1 op1 op2*  
*else if operand-size op1 = 4 then ternop-IMUL TYPE(64) op1 op1 op2*  
*else if operand-size op1 = 2 then ternop-IMUL TYPE(32) op1 op1 op2*  
*else if operand-size op1 = 1 then ternop-IMUL TYPE(16) op1 op1 op2*  
*else undefined*

**abbreviation** *IMUL2*  
**where** *IMUL2 op1 op2*  $\equiv$  *Instr "imul" (Some op1) (Some op2) None*

**definition** *semantics-IMUL3* :: *Operand*  $\Rightarrow$  *Operand*  $\Rightarrow$  *Operand*  $\Rightarrow$  *state*  $\Rightarrow$  *state*  
**where** *semantics-IMUL3 op1 op2 op3*  $\equiv$   
*if operand-size op1 = 8 then ternop-IMUL TYPE(128) op1 op2 op3*

```

else if operand-size op1 = 4 then ternop-IMUL TYPE(64) op1 op2 op3
else if operand-size op1 = 2 then ternop-IMUL TYPE(32) op1 op2 op3
else if operand-size op1 = 1 then ternop-IMUL TYPE(16) op1 op2 op3
else undefined

```

**abbreviation** IMUL3

**where** IMUL3 op1 op2 op3  $\equiv$  Instr "imul" (Some op1) (Some op2) (Some op3)

**definition** SHL-flags :: nat  $\Rightarrow$  ('a::len word  $\Rightarrow$  string  $\Rightarrow$  bool)

**where** SHL-flags n dst flag  $\equiv$  case flag of
$$\begin{array}{l} "cf" \Rightarrow dst !! (\text{LENGTH}('a) - n) \\ | "of" \Rightarrow dst !! (\text{LENGTH}('a) - n - 1) \neq dst !! (\text{LENGTH}('a) - n) \\ | "zf" \Rightarrow (dst << n) = 0 \\ | "sf" \Rightarrow dst !! (\text{LENGTH}('a) - n - 1) \\ | f \Rightarrow \text{unknown-flags} "SHL" f \end{array}$$

**definition** semantics-SHL :: Operand  $\Rightarrow$  Operand  $\Rightarrow$  state  $\Rightarrow$  state

**where** semantics-SHL op1 op2  $\sigma$   $\equiv$ 
let src = unat (operand-read  $\sigma$  op2) in
if operand-size op1 = 32 then unop ( $\lambda w . w << src::256$  word) (SHL-flags src) op1  $\sigma$ 
else if operand-size op1 = 16 then unop ( $\lambda w . w << src::128$  word) (SHL-flags src) op1  $\sigma$ 
else if operand-size op1 = 8 then unop ( $\lambda w . w << src::64$  word) (SHL-flags src) op1  $\sigma$ 
else if operand-size op1 = 4 then unop ( $\lambda w . w << src::32$  word) (SHL-flags src) op1  $\sigma$ 
else if operand-size op1 = 2 then unop ( $\lambda w . w << src::16$  word) (SHL-flags src) op1  $\sigma$ 
else if operand-size op1 = 1 then unop ( $\lambda w . w << src::8$  word) (SHL-flags src) op1  $\sigma$ 
else undefined

**abbreviation** SHL

**where** SHL op1 op2  $\equiv$  Instr "shl" (Some op1) (Some op2) None

**abbreviation** SAL

**where** SAL op1 op2  $\equiv$  Instr "sal" (Some op1) (Some op2) None

**definition** SHR-flags :: nat  $\Rightarrow$  ('a::len word  $\Rightarrow$  string  $\Rightarrow$  bool)

**where** SHR-flags n dst flag  $\equiv$  case flag of
$$\begin{array}{l} "cf" \Rightarrow dst !! (n - 1) \\ | "of" \Rightarrow msb dst \\ | "zf" \Rightarrow (dst >> n) = 0 \\ | f \Rightarrow \text{unknown-flags} "SHR" f \end{array}$$

**definition** semantics-SHR :: Operand  $\Rightarrow$  Operand  $\Rightarrow$  state  $\Rightarrow$  state

**where** semantics-SHR op1 op2  $\sigma$   $\equiv$ 
let src = unat (operand-read  $\sigma$  op2) in

```

    if operand-size op1 = 32 then unop ( $\lambda w . w >> src::256\ word$ ) (SHR-flags
src) op1  $\sigma$ 
    else if operand-size op1 = 16 then unop ( $\lambda w . w >> src::128\ word$ ) (SHR-flags
src) op1  $\sigma$ 
    else if operand-size op1 = 8 then unop ( $\lambda w . w >> src::64\ word$ ) (SHR-flags
src) op1  $\sigma$ 
    else if operand-size op1 = 4 then unop ( $\lambda w . w >> src::32\ word$ ) (SHR-flags
src) op1  $\sigma$ 
    else if operand-size op1 = 2 then unop ( $\lambda w . w >> src::16\ word$ ) (SHR-flags
src) op1  $\sigma$ 
    else if operand-size op1 = 1 then unop ( $\lambda w . w >> src::8\ word$ ) (SHR-flags
src) op1  $\sigma$ 
    else undefined

```

**abbreviation SHR**

where  $SHR\ op1\ op2 \equiv Instr\ "shr"\ (Some\ op1)\ (Some\ op2)\ None$

```

definition SAR-flags :: nat  $\Rightarrow$  ('a::len word  $\Rightarrow$  string  $\Rightarrow$  bool)
where SAR-flags n dst flag  $\equiv$  case flag of
    "cf"  $\Rightarrow$  dst !! (n - 1)
    | "of"  $\Rightarrow$  False
    | "zf"  $\Rightarrow$  (dst >>> n) = 0
    | f       $\Rightarrow$  unknown-flags "SAR" f

```

**definition** semantics-SAR :: Operand  $\Rightarrow$  Operand  $\Rightarrow$  state  $\Rightarrow$  state

where semantics-SAR op1 op2  $\sigma \equiv$

```

let src = unat (operand-read  $\sigma$  op2) in
    if operand-size op1 = 32 then unop ( $\lambda w . w >>> src::256\ word$ ) (SAR-flags
src) op1  $\sigma$ 
    else if operand-size op1 = 16 then unop ( $\lambda w . w >>> src::128\ word$ ) (SAR-flags
src) op1  $\sigma$ 
    else if operand-size op1 = 8 then unop ( $\lambda w . w >>> src::64\ word$ ) (SAR-flags
src) op1  $\sigma$ 
    else if operand-size op1 = 4 then unop ( $\lambda w . w >>> src::32\ word$ ) (SAR-flags
src) op1  $\sigma$ 
    else if operand-size op1 = 2 then unop ( $\lambda w . w >>> src::16\ word$ ) (SAR-flags
src) op1  $\sigma$ 
    else if operand-size op1 = 1 then unop ( $\lambda w . w >>> src::8\ word$ ) (SAR-flags
src) op1  $\sigma$ 
    else undefined

```

**abbreviation SAR**

where  $SAR\ op1\ op2 \equiv Instr\ "sar"\ (Some\ op1)\ (Some\ op2)\ None$

```

definition shld :: 'b::len itself  $\Rightarrow$  nat  $\Rightarrow$  'a::len word  $\Rightarrow$  'a word  $\Rightarrow$  'a word
where shld - n dst src  $\equiv$ 
    let dstsrc = (ucast dst << LENGTH('a)) OR (ucast src :: 'b word);

```

*shifted* =  $\langle \text{LENGTH}('a), \text{LENGTH}('a)*2 \rangle (\text{dstsrc} << n)$  in  
*ucast shifted*

**definition** *SHLD-flags* :: 'b::len *itself*  $\Rightarrow$  nat  $\Rightarrow$  ('a::len word  $\Rightarrow$  'a::len word  $\Rightarrow$  string  $\Rightarrow$  bool)

**where** *SHLD-flags*  $b\ n\ src\ dst\ flag \equiv \text{case } flag \text{ of}$

- $"cf" \Rightarrow dst \text{ !! } (\text{LENGTH}('a) - n)$
- $"of" \Rightarrow dst \text{ !! } (\text{LENGTH}('a) - n - 1) \neq dst \text{ !! } (\text{LENGTH}('a) - n)$
- $"zf" \Rightarrow \text{shld } b\ n\ dst\ src = 0$
- $"sf" \Rightarrow dst \text{ !! } (\text{LENGTH}('a) - n - 1) \text{ — msb } (\text{shld } n\ dst\ src)$
- $f \Rightarrow \text{unknown-flags } "SHLD" f$

**definition** *semantics-SHLD* :: Operand  $\Rightarrow$  Operand  $\Rightarrow$  Operand  $\Rightarrow$  state  $\Rightarrow$  state

**where** *semantics-SHLD*  $op1\ op2\ op3\ \sigma \equiv$

*let src2 = unat (operand-read  $\sigma$  op3) in*

- if operand-size op1 = 32 then binop (shld (TYPE(512)) src2 ::256 word  $\Rightarrow$  -  $\Rightarrow$  -) (SHLD-flags (TYPE(512)) src2) op1 op2  $\sigma$*
- else if operand-size op1 = 16 then binop (shld (TYPE(256)) src2 ::128 word  $\Rightarrow$  -  $\Rightarrow$  -) (SHLD-flags (TYPE(256)) src2) op1 op2  $\sigma$*
- else if operand-size op1 = 8 then binop (shld (TYPE(128)) src2 ::64 word  $\Rightarrow$  -  $\Rightarrow$  -) (SHLD-flags (TYPE(128)) src2) op1 op2  $\sigma$*
- else if operand-size op1 = 4 then binop (shld (TYPE(64)) src2 ::32 word  $\Rightarrow$  -  $\Rightarrow$  -) (SHLD-flags (TYPE(64)) src2) op1 op2  $\sigma$*
- else if operand-size op1 = 2 then binop (shld (TYPE(32)) src2 ::16 word  $\Rightarrow$  -  $\Rightarrow$  -) (SHLD-flags (TYPE(32)) src2) op1 op2  $\sigma$*
- else if operand-size op1 = 1 then binop (shld (TYPE(16)) src2 ::8 word  $\Rightarrow$  -  $\Rightarrow$  -) (SHLD-flags (TYPE(16)) src2) op1 op2  $\sigma$*
- else undefined*

**definition** *ROL-flags* :: nat  $\Rightarrow$  ('a::len word  $\Rightarrow$  string  $\Rightarrow$  bool)

**where** *ROL-flags*  $n\ dst\ flag \equiv \text{case } flag \text{ of}$

- $"cf" \Rightarrow dst \text{ !! } (\text{LENGTH}('a) - n)$
- $"of" \Rightarrow dst \text{ !! } (\text{LENGTH}('a) - n - 1) \neq dst \text{ !! } (\text{LENGTH}('a) - n)$
- $f \Rightarrow \text{unknown-flags } "ROL" f$

**definition** *semantics-ROL* :: Operand  $\Rightarrow$  Operand  $\Rightarrow$  state  $\Rightarrow$  state

**where** *semantics-ROL*  $op1\ op2\ \sigma \equiv$

*let src = unat (operand-read  $\sigma$  op2) in*

- if operand-size op1 = 32 then unop (word-rotl src::256 word  $\Rightarrow$  -) (ROL-flags src) op1  $\sigma$*
- else if operand-size op1 = 16 then unop (word-rotl src::128 word  $\Rightarrow$  -) (ROL-flags src) op1  $\sigma$*
- else if operand-size op1 = 8 then unop (word-rotl src::64 word  $\Rightarrow$  -) (ROL-flags src) op1  $\sigma$*
- else if operand-size op1 = 4 then unop (word-rotl src::32 word  $\Rightarrow$  -) (ROL-flags src) op1  $\sigma$*
- else if operand-size op1 = 2 then unop (word-rotl src::16 word  $\Rightarrow$  -) (ROL-flags src) op1  $\sigma$*

```

src) op1 σ
  else if operand-size op1 = 1 then unop (word-rotl src::8 word⇒-) (ROL-flags
src) op1 σ
  else undefined

```

**abbreviation** *ROL*

**where** *ROL* *op1 op2* ≡ *Instr "rol"* (*Some op1*) (*Some op2*) *None*

```

definition ROR-flags :: nat ⇒ ('a::len word ⇒ string ⇒ bool)
where ROR-flags n dst flag ≡ case flag of
  | "cf" ⇒ dst !! (n - 1)
  | "of" ⇒ msb (word-rotr n dst) ≠ (word-rotr n dst !! (LENGTH('a) - 2))
  | f ⇒ unknown-flags "ROR" f

```

**definition** *semantics-ROR* :: *Operand* ⇒ *Operand* ⇒ *state* ⇒ *state*

**where** *semantics-ROR* *op1 op2 σ* ≡

```

let src = unat (operand-read σ op2) in
  if operand-size op1 = 32 then unop (word-rotr src::256 word⇒-) (ROR-flags
src) op1 σ
  else if operand-size op1 = 16 then unop (word-rotr src::128 word⇒-) (ROR-flags
src) op1 σ
  else if operand-size op1 = 8 then unop (word-rotr src::64 word⇒-) (ROR-flags
src) op1 σ
  else if operand-size op1 = 4 then unop (word-rotr src::32 word⇒-) (ROR-flags
src) op1 σ
  else if operand-size op1 = 2 then unop (word-rotr src::16 word⇒-) (ROR-flags
src) op1 σ
  else if operand-size op1 = 1 then unop (word-rotr src::8 word⇒-) (ROR-flags
src) op1 σ
  else undefined

```

**abbreviation** *ROR*

**where** *ROR* *op1 op2* ≡ *Instr "ror"* (*Some op1*) (*Some op2*) *None*

flag-related

**definition** *semantics-CMP* :: *Operand* ⇒ *Operand* ⇒ *state* ⇒ *state*

**where** *semantics-CMP* *op1* ≡

```

  if operand-size op1 = 32 then binop-flags (SUB-flags::256 word ⇒ - ⇒ -
⇒ -) op1
  else if operand-size op1 = 16 then binop-flags (SUB-flags::128 word ⇒ - ⇒ -
⇒ -) op1
  else if operand-size op1 = 8 then binop-flags (SUB-flags::64 word ⇒ - ⇒ -
⇒ -) op1
  else if operand-size op1 = 4 then binop-flags (SUB-flags::32 word ⇒ - ⇒ -
⇒ -) op1
  else if operand-size op1 = 2 then binop-flags (SUB-flags::16 word ⇒ - ⇒ -
⇒ -) op1
  else if operand-size op1 = 1 then binop-flags (SUB-flags::8 word ⇒ - ⇒ -
⇒ -) op1

```

*else undefined*

**abbreviation** *CMP*

**where** *CMP op1 op2*  $\equiv$  *Instr "cmp" (Some op1) (Some op2) None*

**definition** *logic-flags* ::  $('a::len\ word \Rightarrow 'a::len\ word \Rightarrow 'a::len\ word) \Rightarrow 'a::len\ word \Rightarrow 'a::len\ word \Rightarrow string \Rightarrow bool$

**where** *logic-flags logic-op w0 w1 flag*  $\equiv$  *case flag of*  
  | *"zf"*  $\Rightarrow$  *logic-op w0 w1 = 0*  
  | *"cf"*  $\Rightarrow$  *False*  
  | *"of"*  $\Rightarrow$  *False*  
  | *"sf"*  $\Rightarrow$  *msb (logic-op w0 w1)*  
  | *f*       $\Rightarrow$  *unknown-flags "logic" f*

**definition** *semantics-TEST* :: *Operand*  $\Rightarrow$  *Operand*  $\Rightarrow$  *state*  $\Rightarrow$  *state*

**where** *semantics-TEST op1*  $\equiv$

*if operand-size op1 = 32 then binop-flags (logic-flags ((AND)::256 word*  
   $\Rightarrow - \Rightarrow -)) op1$   
  *else if operand-size op1 = 16 then binop-flags (logic-flags ((AND)::128 word*  
   $\Rightarrow - \Rightarrow -)) op1$   
  *else if operand-size op1 = 8 then binop-flags (logic-flags ((AND)::64 word*  
   $\Rightarrow - \Rightarrow -)) op1$   
  *else if operand-size op1 = 4 then binop-flags (logic-flags ((AND)::32 word*  
   $\Rightarrow - \Rightarrow -)) op1$   
  *else if operand-size op1 = 2 then binop-flags (logic-flags ((AND)::16 word*  
   $\Rightarrow - \Rightarrow -)) op1$   
  *else if operand-size op1 = 1 then binop-flags (logic-flags ((AND)::8 word \Rightarrow*  
   $- \Rightarrow -)) op1$   
  *else undefined*

**abbreviation** *TEST*

**where** *TEST op1 op2*  $\equiv$  *Instr "test" (Some op1) (Some op2) None*

sign extension

**definition** *mov-sign-extension* ::  $('a::len)\ itself \Rightarrow ('b::len)\ itself \Rightarrow Operand \Rightarrow state \Rightarrow state$

**where** *mov-sign-extension - - op1 op2 σ*  $\equiv$

*let src = ucast (operand-read σ op2)::'b word in*  
  *operand-write op1 (ucast (scast src::'a word)) σ*

**definition** *semantics-MOVSXD* :: *Operand*  $\Rightarrow$  *Operand*  $\Rightarrow$  *state*  $\Rightarrow$  *state*

**where** *semantics-MOVSXD op1 op2*  $\equiv$

*if (operand-size op1, operand-size op2) = (8,4) then*  
    *mov-sign-extension (TYPE(64)) (TYPE(32)) op1 op2*  
  *else if (operand-size op1, operand-size op2) = (8,2) then*  
    *mov-sign-extension (TYPE(64)) (TYPE(16)) op1 op2*  
  *else if (operand-size op1, operand-size op2) = (8,1) then*  
    *mov-sign-extension (TYPE(64)) (TYPE(8)) op1 op2*  
  *else if (operand-size op1, operand-size op2) = (4,2) then*

```

mov-sign-extension (TYPE(32)) (TYPE(16)) op1 op2
else if (operand-size op1, operand-size op2) = (4,1) then
  mov-sign-extension (TYPE(32)) (TYPE(8)) op1 op2
else if (operand-size op1, operand-size op2) = (2,1) then
  mov-sign-extension (TYPE(16)) (TYPE(8)) op1 op2
else
  undefined

```

**abbreviation** MOVSXD

**where** MOVSXD op1 op2  $\equiv$  Instr "movsxd" (Some op1) (Some op2) None

**abbreviation** MOVSX

**where** MOVSX op1 op2  $\equiv$  Instr "movsx" (Some op1) (Some op2) None

**definition** semantics-CDQE :: state  $\Rightarrow$  state

**where** semantics-CDQE  $\equiv$  semantics-MOVSXD (Reg "rax") (Reg "eax")

**abbreviation** CDQE

**where** CDQE  $\equiv$  Instr "cdqe" None None None

**definition** semantics-CDQ :: state  $\Rightarrow$  state

**where** semantics-CDQ  $\sigma \equiv$

```

let src = ucast (operand-read  $\sigma$  (Reg "eax")) :: 32 word in
  operand-write (Reg "edx") (ucast ((32,64)(scast src::64 word)))  $\sigma$ 

```

**abbreviation** CDQ

**where** CDQ  $\equiv$  Instr "cdq" None None None

**definition** semantics-CQO :: state  $\Rightarrow$  state

**where** semantics-CQO  $\sigma \equiv$

```

let src = ucast (operand-read  $\sigma$  (Reg "rax")) :: 64 word in
  operand-write (Reg "rdx") (ucast ((64,128)(scast src::128 word)))  $\sigma$ 

```

**abbreviation** CQO

**where** CQO  $\equiv$  Instr "cqo" None None None

logic

**definition** semantics-AND :: Operand  $\Rightarrow$  Operand  $\Rightarrow$  state  $\Rightarrow$  state

**where** semantics-AND op1 op2  $\sigma \equiv$

```

if operand-size op1 = 32 then binop ((AND)::256 word  $\Rightarrow$  -  $\Rightarrow$  -) (logic-flags
((AND)::256 word  $\Rightarrow$  -  $\Rightarrow$  -)) op1 op2  $\sigma$ 
else if operand-size op1 = 16 then binop ((AND)::128 word  $\Rightarrow$  -  $\Rightarrow$  -) (logic-flags
((AND)::128 word  $\Rightarrow$  -  $\Rightarrow$  -)) op1 op2  $\sigma$ 
else if operand-size op1 = 8 then binop ((AND)::64 word  $\Rightarrow$  -  $\Rightarrow$  -) (logic-flags
((AND)::64 word  $\Rightarrow$  -  $\Rightarrow$  -)) op1 op2  $\sigma$ 
else if operand-size op1 = 4 then binop ((AND)::32 word  $\Rightarrow$  -  $\Rightarrow$  -) (logic-flags
((AND)::32 word  $\Rightarrow$  -  $\Rightarrow$  -)) op1 op2  $\sigma$ 
else if operand-size op1 = 2 then binop ((AND)::16 word  $\Rightarrow$  -  $\Rightarrow$  -) (logic-flags
((AND)::16 word  $\Rightarrow$  -  $\Rightarrow$  -)) op1 op2  $\sigma$ 

```

$\text{else if } \text{operand-size } op1 = 1 \text{ then } \text{binop } ((\text{AND})::8 \text{ word} \Rightarrow - \Rightarrow -) (\text{logic-flags } ((\text{AND})::8 \text{ word} \Rightarrow - \Rightarrow -)) op1 op2 \sigma$   
 $\text{else undefined}$

**abbreviation AND'**

where  $\text{AND}' op1 op2 \equiv \text{Instr } \text{"and"} (\text{Some } op1) (\text{Some } op2) \text{ None}$

**definition semantics-OR :: Operand  $\Rightarrow$  Operand  $\Rightarrow$  state  $\Rightarrow$  state**

where  $\text{semantics-OR } op1 op2 \sigma \equiv$

$\text{if } \text{operand-size } op1 = 32 \text{ then } \text{binop } ((\text{OR})::256 \text{ word} \Rightarrow - \Rightarrow -) (\text{logic-flags } ((\text{OR})::256 \text{ word} \Rightarrow - \Rightarrow -)) op1 op2 \sigma$   
 $\text{else if } \text{operand-size } op1 = 16 \text{ then } \text{binop } ((\text{OR})::128 \text{ word} \Rightarrow - \Rightarrow -) (\text{logic-flags } ((\text{OR})::128 \text{ word} \Rightarrow - \Rightarrow -)) op1 op2 \sigma$   
 $\text{else if } \text{operand-size } op1 = 8 \text{ then } \text{binop } ((\text{OR})::64 \text{ word} \Rightarrow - \Rightarrow -) (\text{logic-flags } ((\text{OR})::64 \text{ word} \Rightarrow - \Rightarrow -)) op1 op2 \sigma$   
 $\text{else if } \text{operand-size } op1 = 4 \text{ then } \text{binop } ((\text{OR})::32 \text{ word} \Rightarrow - \Rightarrow -) (\text{logic-flags } ((\text{OR})::32 \text{ word} \Rightarrow - \Rightarrow -)) op1 op2 \sigma$   
 $\text{else if } \text{operand-size } op1 = 2 \text{ then } \text{binop } ((\text{OR})::16 \text{ word} \Rightarrow - \Rightarrow -) (\text{logic-flags } ((\text{OR})::16 \text{ word} \Rightarrow - \Rightarrow -)) op1 op2 \sigma$   
 $\text{else if } \text{operand-size } op1 = 1 \text{ then } \text{binop } ((\text{OR})::8 \text{ word} \Rightarrow - \Rightarrow -) (\text{logic-flags } ((\text{OR})::8 \text{ word} \Rightarrow - \Rightarrow -)) op1 op2 \sigma$   
 $\text{else undefined}$

**abbreviation OR'**

where  $\text{OR}' op1 op2 \equiv \text{Instr } \text{"or"} (\text{Some } op1) (\text{Some } op2) \text{ None}$

**definition semantics-XOR :: Operand  $\Rightarrow$  Operand  $\Rightarrow$  state  $\Rightarrow$  state**

where  $\text{semantics-XOR } op1 op2 \sigma \equiv$

$\text{if } \text{operand-size } op1 = 32 \text{ then } \text{binop } ((\text{XOR})::256 \text{ word} \Rightarrow - \Rightarrow -) (\text{logic-flags } ((\text{XOR})::256 \text{ word} \Rightarrow - \Rightarrow -)) op1 op2 \sigma$   
 $\text{else if } \text{operand-size } op1 = 16 \text{ then } \text{binop } ((\text{XOR})::128 \text{ word} \Rightarrow - \Rightarrow -) (\text{logic-flags } ((\text{XOR})::128 \text{ word} \Rightarrow - \Rightarrow -)) op1 op2 \sigma$   
 $\text{else if } \text{operand-size } op1 = 8 \text{ then } \text{binop } ((\text{XOR})::64 \text{ word} \Rightarrow - \Rightarrow -) (\text{logic-flags } ((\text{XOR})::64 \text{ word} \Rightarrow - \Rightarrow -)) op1 op2 \sigma$   
 $\text{else if } \text{operand-size } op1 = 4 \text{ then } \text{binop } ((\text{XOR})::32 \text{ word} \Rightarrow - \Rightarrow -) (\text{logic-flags } ((\text{XOR})::32 \text{ word} \Rightarrow - \Rightarrow -)) op1 op2 \sigma$   
 $\text{else if } \text{operand-size } op1 = 2 \text{ then } \text{binop } ((\text{XOR})::16 \text{ word} \Rightarrow - \Rightarrow -) (\text{logic-flags } ((\text{XOR})::16 \text{ word} \Rightarrow - \Rightarrow -)) op1 op2 \sigma$   
 $\text{else if } \text{operand-size } op1 = 1 \text{ then } \text{binop } ((\text{XOR})::8 \text{ word} \Rightarrow - \Rightarrow -) (\text{logic-flags } ((\text{XOR})::8 \text{ word} \Rightarrow - \Rightarrow -)) op1 op2 \sigma$   
 $\text{else undefined}$

**abbreviation XOR'**

where  $\text{XOR}' op1 op2 \equiv \text{Instr } \text{"xor"} (\text{Some } op1) (\text{Some } op2) \text{ None}$

**definition semantics-XORPS :: Operand  $\Rightarrow$  Operand  $\Rightarrow$  state  $\Rightarrow$  state**

where  $\text{semantics-XORPS } op1 \equiv$

$\text{if } \text{operand-size } op1 = 32 \text{ then } \text{binop-no-flags } ((\text{XOR})::256 \text{ word} \Rightarrow - \Rightarrow -)$   
 $op1$

```

else if operand-size op1 = 16 then binop-no-flags ((XOR)::128 word ⇒ - ⇒ -)
-) op1
  else if operand-size op1 = 8  then binop-no-flags ((XOR)::64  word ⇒ - ⇒ -)
op1
  else if operand-size op1 = 4  then binop-no-flags ((XOR)::32  word ⇒ - ⇒ -)
op1
  else if operand-size op1 = 2  then binop-no-flags ((XOR)::16  word ⇒ - ⇒ -)
op1
  else if operand-size op1 = 1  then binop-no-flags ((XOR)::8   word ⇒ - ⇒ -)
op1
  else undefined

```

**abbreviation** *XORPS*

**where** *XORPS* *op1 op2* ≡ *Instr "xorps"* (*Some op1*) (*Some op2*) *None*

**definition** *semantics-NOT* :: *Operand* ⇒ *state* ⇒ *state*

**where** *semantics-NOT* *op1 σ* ≡

```

if operand-size op1 = 32 then unop-no-flags (not::256 word⇒-) op1 σ
else if operand-size op1 = 16 then unop-no-flags (not::128 word⇒-) op1 σ
else if operand-size op1 = 8  then unop-no-flags (not::64  word⇒-) op1 σ
else if operand-size op1 = 4  then unop-no-flags (not::32  word⇒-) op1 σ
else if operand-size op1 = 2  then unop-no-flags (not::16  word⇒-) op1 σ
else if operand-size op1 = 1  then unop-no-flags (not::8   word⇒-) op1 σ
else undefined

```

**abbreviation** *NOT'*

**where** *NOT'* *op1* ≡ *Instr "not"* (*Some op1*) *None None*

jumps

**datatype** *FlagExpr* = *Flag string* | *FE-NOT FlagExpr* | *FE-AND FlagExpr FlagExpr* | *FE-OR FlagExpr FlagExpr* | *FE-EQ FlagExpr FlagExpr*

**primrec** *readFlagExpr* :: *FlagExpr* ⇒ *state* ⇒ *bool*

**where**

```

readFlagExpr (Flag f) σ = (flag-read σ f = 1)
| readFlagExpr (FE-NOT fe) σ = (¬readFlagExpr fe σ)
| readFlagExpr (FE-AND fe0 fe1) σ = (readFlagExpr fe0 σ ∧ readFlagExpr fe1 σ)
| readFlagExpr (FE-OR fe0 fe1) σ = (readFlagExpr fe0 σ ∨ readFlagExpr fe1 σ)
| readFlagExpr (FE-EQ fe0 fe1) σ = (readFlagExpr fe0 σ ↔ readFlagExpr fe1 σ)

```

**definition** *semantics-cond-jump* :: *FlagExpr* ⇒ *64 word* ⇒ *state* ⇒ *state*

**where** *semantics-cond-jump* *fe a σ* ≡

```

let fv = readFlagExpr fe σ in
  if fv then state-update (setRip a) σ else σ

```

**definition** *semantics-JMP* :: *Operand* ⇒ *state* ⇒ *state*

**where** *semantics-JMP* *op1 σ* ≡

```

let a = ucast (operand-read σ op1) in
    state-update (setRip a) σ

abbreviation JMP
where JMP op1 ≡ Instr "jmp" (Some op1) None None

definition semantics-JO :: Operand ⇒ state ⇒ state
where semantics-JO op1 σ ≡
    let a = ucast (operand-read σ op1) in
        semantics-cond-jump (Flag "of") a σ

abbreviation JO
where JO op1 ≡ Instr "jo" (Some op1) None None

definition semantics-JNO :: Operand ⇒ state ⇒ state
where semantics-JNO op1 σ ≡
    let a = ucast (operand-read σ op1) in
        semantics-cond-jump (FE-NOT (Flag "of")) a σ

abbreviation JNO
where JNO op1 ≡ Instr "jno" (Some op1) None None

definition semantics-JS :: Operand ⇒ state ⇒ state
where semantics-JS op1 σ ≡
    let a = ucast (operand-read σ op1) in
        semantics-cond-jump (Flag "sf") a σ

abbreviation JS
where JS op1 ≡ Instr "js" (Some op1) None None

definition semantics-JNS :: Operand ⇒ state ⇒ state
where semantics-JNS op1 σ ≡
    let a = ucast (operand-read σ op1) in
        semantics-cond-jump (FE-NOT (Flag "sf")) a σ

abbreviation JNS
where JNS op1 ≡ Instr "jns" (Some op1) None None

definition semantics-JE :: Operand ⇒ state ⇒ state
where semantics-JE op1 σ ≡
    let a = ucast (operand-read σ op1) in
        semantics-cond-jump (Flag "zf") a σ

abbreviation JE
where JE op1 ≡ Instr "je" (Some op1) None None

abbreviation JZ
where JZ op1 ≡ Instr "jz" (Some op1) None None

```

```

definition semantics-JNE :: Operand  $\Rightarrow$  state  $\Rightarrow$  state
where semantics-JNE op1  $\sigma \equiv$ 
    let a = ucast (operand-read  $\sigma$  op1) in
        semantics-cond-jump (FE-NOT (Flag "zf'')) a  $\sigma$ 

abbreviation JNE
where JNE op1  $\equiv$  Instr "jne" (Some op1) None None

abbreviation JNZ
where JNZ op1  $\equiv$  Instr "jnz" (Some op1) None None

definition semantics-JB :: Operand  $\Rightarrow$  state  $\Rightarrow$  state
where semantics-JB op1  $\sigma \equiv$ 
    let a = ucast (operand-read  $\sigma$  op1) in
        semantics-cond-jump (Flag "cf') a  $\sigma$ 

abbreviation JB
where JB op1  $\equiv$  Instr "jb" (Some op1) None None

abbreviation JNAE
where JNAE op1  $\equiv$  Instr "jnae" (Some op1) None None

abbreviation JC
where JC op1  $\equiv$  Instr "jc" (Some op1) None None

definition semantics-JNB :: Operand  $\Rightarrow$  state  $\Rightarrow$  state
where semantics-JNB op1  $\sigma \equiv$ 
    let a = ucast (operand-read  $\sigma$  op1) in
        semantics-cond-jump (FE-NOT (Flag "cf')) a  $\sigma$ 

abbreviation JNB
where JNB op1  $\equiv$  Instr "jnb" (Some op1) None None

abbreviation JAE
where JAE op1  $\equiv$  Instr "jae" (Some op1) None None

abbreviation JNC
where JNC op1  $\equiv$  Instr "jnc" (Some op1) None None

definition semantics-JBE :: Operand  $\Rightarrow$  state  $\Rightarrow$  state
where semantics-JBE op1  $\sigma \equiv$ 
    let a = ucast (operand-read  $\sigma$  op1) in
        semantics-cond-jump (FE-OR (Flag "cf') (Flag "zf')) a  $\sigma$ 

abbreviation JBE
where JBE op1  $\equiv$  Instr "jbe" (Some op1) None None

abbreviation JNA
where JNA op1  $\equiv$  Instr "jna" (Some op1) None None

```

```

definition semantics-JA :: Operand  $\Rightarrow$  state  $\Rightarrow$  state
where semantics-JA op1  $\sigma \equiv$ 
  let a = ucast (operand-read  $\sigma$  op1) in
    semantics-cond-jump (FE-AND (FE-NOT (Flag "cf")) (FE-NOT (Flag
  "zf"))) a  $\sigma$ 

abbreviation JA
where JA op1  $\equiv$  Instr "ja" (Some op1) None None

abbreviation JNBE
where JNBE op1  $\equiv$  Instr "jnbe" (Some op1) None None

definition semantics-JL :: Operand  $\Rightarrow$  state  $\Rightarrow$  state
where semantics-JL op1  $\sigma \equiv$ 
  let a = ucast (operand-read  $\sigma$  op1) in
    semantics-cond-jump (FE-NOT (FE-EQ (Flag "sf") (Flag "of"))) a  $\sigma$ 

abbreviation JL
where JL op1  $\equiv$  Instr "jl" (Some op1) None None

abbreviation JNGE
where JNGE op1  $\equiv$  Instr "jnge" (Some op1) None None

definition semantics-JGE :: Operand  $\Rightarrow$  state  $\Rightarrow$  state
where semantics-JGE op1  $\sigma \equiv$ 
  let a = ucast (operand-read  $\sigma$  op1) in
    semantics-cond-jump (FE-EQ (Flag "sf") (Flag "of")) a  $\sigma$ 

abbreviation JGE
where JGE op1  $\equiv$  Instr "jge" (Some op1) None None

abbreviation JNL
where JNL op1  $\equiv$  Instr "jnl" (Some op1) None None

definition semantics-JLE :: Operand  $\Rightarrow$  state  $\Rightarrow$  state
where semantics-JLE op1  $\sigma \equiv$ 
  let a = ucast (operand-read  $\sigma$  op1) in
    semantics-cond-jump (FE-OR (Flag "zf") (FE-NOT (FE-EQ (Flag
  "sf") (Flag "of")))) a  $\sigma$ 

abbreviation JLE
where JLE op1  $\equiv$  Instr "jle" (Some op1) None None

abbreviation JNG
where JNG op1  $\equiv$  Instr "jng" (Some op1) None None

definition semantics-JG :: Operand  $\Rightarrow$  state  $\Rightarrow$  state
where semantics-JG op1  $\sigma \equiv$ 

```

```

let a = ucast (operand-read σ op1) in
  semantics-cond-jump (FE-AND (FE-NOT (Flag "zf")) (FE-EQ (Flag
  "sf") (Flag "of''))) a σ

abbreviation JG
where JG op1 ≡ Instr "jg" (Some op1) None None

abbreviation JNLE
where JNLE op1 ≡ Instr "jnle" (Some op1) None None

setXX

definition semantics-setXX :: FlagExpr ⇒ Operand ⇒ state ⇒ state
where semantics-setXX fe op1 σ ≡
  let fv = readFlagExpr fe σ in
    operand-write op1 (fromBool fv) σ

abbreviation semantics-SETO :: Operand ⇒ state ⇒ state
where semantics-SETO ≡ semantics-setXX (Flag "of")

abbreviation SETO
where SETO op1 ≡ Instr "seto" (Some op1) None None

abbreviation semantics-SETNO :: Operand ⇒ state ⇒ state
where semantics-SETNO ≡ semantics-setXX (FE-NOT (Flag "of"))

abbreviation SETNO
where SETNO op1 ≡ Instr "setno" (Some op1) None None

abbreviation semantics-SETS :: Operand ⇒ state ⇒ state
where semantics-SETS ≡ semantics-setXX (Flag "sf")

abbreviation SETS
where SETS op1 ≡ Instr "sets" (Some op1) None None

abbreviation semantics-SETNS :: Operand ⇒ state ⇒ state
where semantics-SETNS ≡ semantics-setXX (FE-NOT (Flag "sf"))

abbreviation SETNS
where SETNS op1 ≡ Instr "setns" (Some op1) None None

abbreviation semantics-SETE :: Operand ⇒ state ⇒ state
where semantics-SETE ≡ semantics-setXX (Flag "zf")

abbreviation SETE
where SETE op1 ≡ Instr "sete" (Some op1) None None

abbreviation SETZ
where SETZ op1 ≡ Instr "setz" (Some op1) None None

```

**abbreviation** *semantics-SETNE* :: *Operand*  $\Rightarrow$  *state*  $\Rightarrow$  *state*  
**where** *semantics-SETNE*  $\equiv$  *semantics-setXX* (*FE-NOT* (*Flag* "zf"))

**abbreviation** *SETNE*  
**where** *SETNE op1*  $\equiv$  *Instr* "setne" (*Some op1*) *None None*

**abbreviation** *SETNZ*  
**where** *SETNZ op1*  $\equiv$  *Instr* "setnz" (*Some op1*) *None None*

**abbreviation** *semantics-SETB* :: *Operand*  $\Rightarrow$  *state*  $\Rightarrow$  *state*  
**where** *semantics-SETB*  $\equiv$  *semantics-setXX* (*Flag* "cf")

**abbreviation** *SETB*  
**where** *SETB op1*  $\equiv$  *Instr* "setb" (*Some op1*) *None None*

**abbreviation** *SETNAE*  
**where** *SETNAE op1*  $\equiv$  *Instr* "setnae" (*Some op1*) *None None*

**abbreviation** *SETC*  
**where** *SETC op1*  $\equiv$  *Instr* "setc" (*Some op1*) *None None*

**abbreviation** *semantics-SETNB* :: *Operand*  $\Rightarrow$  *state*  $\Rightarrow$  *state*  
**where** *semantics-SETNB*  $\equiv$  *semantics-setXX* (*FE-NOT* (*Flag* "cf"))

**abbreviation** *SETNB*  
**where** *SETNB op1*  $\equiv$  *Instr* "setnb" (*Some op1*) *None None*

**abbreviation** *SETAE*  
**where** *SETAE op1*  $\equiv$  *Instr* "setae" (*Some op1*) *None None*

**abbreviation** *SETNC*  
**where** *SETNC op1*  $\equiv$  *Instr* "setnc" (*Some op1*) *None None*

**abbreviation** *semantics-SETBE* :: *Operand*  $\Rightarrow$  *state*  $\Rightarrow$  *state*  
**where** *semantics-SETBE*  $\equiv$  *semantics-setXX* (*FE-OR* (*Flag* "cf") (*Flag* "zf"))

**abbreviation** *SETBE*  
**where** *SETBE op1*  $\equiv$  *Instr* "setbe" (*Some op1*) *None None*

**abbreviation** *SETNA*  
**where** *SETNA op1*  $\equiv$  *Instr* "setna" (*Some op1*) *None None*

**abbreviation** *semantics-SETA* :: *Operand*  $\Rightarrow$  *state*  $\Rightarrow$  *state*  
**where** *semantics-SETA*  $\equiv$  *semantics-setXX* (*FE-AND* (*FE-NOT* (*Flag* "cf")) (*FE-NOT* (*Flag* "zf")))

**abbreviation** *SETA*  
**where** *SETA op1*  $\equiv$  *Instr* "seta" (*Some op1*) *None None*

**abbreviation** *SETNBE*  
**where** *SETNBE op1*  $\equiv$  *Instr "setnbe" (Some op1) None None*

**abbreviation** *semantics-SETL* :: *Operand*  $\Rightarrow$  *state*  $\Rightarrow$  *state*  
**where** *semantics-SETL*  $\equiv$  *semantics-setXX (FE-NOT (FE-EQ (Flag "sf") (Flag "of")))*

**abbreviation** *SETL*  
**where** *SETL op1*  $\equiv$  *Instr "setl" (Some op1) None None*

**abbreviation** *SETNGE*  
**where** *SETNGE op1*  $\equiv$  *Instr "setnge" (Some op1) None None*

**abbreviation** *semantics-SETGE* :: *Operand*  $\Rightarrow$  *state*  $\Rightarrow$  *state*  
**where** *semantics-SETGE*  $\equiv$  *semantics-setXX (FE-EQ (Flag "sf") (Flag "of"))*

**abbreviation** *SETGE*  
**where** *SETGE op1*  $\equiv$  *Instr "setge" (Some op1) None None*

**abbreviation** *SETNL*  
**where** *SETNL op1*  $\equiv$  *Instr "setnl" (Some op1) None None*

**abbreviation** *semantics-SETLE* :: *Operand*  $\Rightarrow$  *state*  $\Rightarrow$  *state*  
**where** *semantics-SETLE*  $\equiv$  *semantics-setXX (FE-OR (Flag "zf") (FE-NOT (FE-EQ (Flag "sf") (Flag "of"))))*

**abbreviation** *SETLE*  
**where** *SETLE op1*  $\equiv$  *Instr "setle" (Some op1) None None*

**abbreviation** *SETNG*  
**where** *SETNG op1*  $\equiv$  *Instr "setng" (Some op1) None None*

**abbreviation** *semantics-SETG* :: *Operand*  $\Rightarrow$  *state*  $\Rightarrow$  *state*  
**where** *semantics-SETG*  $\equiv$  *semantics-setXX (FE-AND (FE-NOT (Flag "zf")) (FE-EQ (Flag "sf") (Flag "of"))))*

**abbreviation** *SETG*  
**where** *SETG op1*  $\equiv$  *Instr "setg" (Some op1) None None*

**abbreviation** *SETNLE*  
**where** *SETNLE op1*  $\equiv$  *Instr "setnle" (Some op1) None None*

conditional moves

**primrec** *cmove*  
**where**  
*cmove True dst src = src*  
*| cmove False dst src = dst*

**definition** *semantics-CMOV* :: *FlagExpr*  $\Rightarrow$  *Operand*  $\Rightarrow$  *Operand*  $\Rightarrow$  *state*  $\Rightarrow$  *state*

```

where semantics-CMOV fe op1 op2 σ ≡
  let fv = readFlagExpr fe σ;
  dst = operand-read σ op1;
  src = operand-read σ op2 in
  operand-write op1 (cmov fv dst src) σ

abbreviation semantics-CMOVNE ≡ semantics-CMOV (FE-NOT (Flag "zf"))

abbreviation CMOVNE
where CMOVNE op1 op2 ≡ Instr "movne" (Some op1) (Some op2) None

abbreviation semantics-CMOVNS ≡ semantics-CMOV (FE-NOT (Flag "sf"))

abbreviation CMOVNS
where CMOVNS op1 op2 ≡ Instr "movns" (Some op1) (Some op2) None

  Floating Point

definition semantics-ADDSD :: Operand ⇒ Operand ⇒ state ⇒ state
where semantics-ADDSD ≡ binop-XMM unknown-addsd

definition semantics-SUBSD :: Operand ⇒ Operand ⇒ state ⇒ state
where semantics-SUBSD ≡ binop-XMM unknown-subsd

definition semantics-MULSD :: Operand ⇒ Operand ⇒ state ⇒ state
where semantics-MULSD ≡ binop-XMM unknown-mulsd

definition semantics-DIVSD :: Operand ⇒ Operand ⇒ state ⇒ state
where semantics-DIVSD ≡ binop-XMM unknown-divsd

definition UCOMISD-flags :: 64 word ⇒ 64 word ⇒ string ⇒ bool
where UCOMISD-flags w0 w1 f ≡
  if  $f \in \{\text{"zf"}, \text{"pf"}, \text{"cf}\}$  then case unknown-ucomisd w0 w1 of
    FP-Unordered ⇒ True
  | FP-GT      ⇒ False
  | FP-LT      ⇒ f = "cf"
  | FP-EQ      ⇒ f = "zf"
  else
    unknown-flags "UCOMISD" f

definition semantics-UCOMISD :: Operand ⇒ Operand ⇒ state ⇒ state
where semantics-UCOMISD ≡ binop-flags UCOMISD-flags

abbreviation ADDSD
where ADDSD op1 op2 ≡ Instr "addsd" (Some op1) (Some op2) None

abbreviation SUBSD
where SUBSD op1 op2 ≡ Instr "subsd" (Some op1) (Some op2) None

```

**abbreviation MULSD**  
**where**  $MULSD\ op1\ op2 \equiv Instr\ "mulsd"\ (Some\ op1)\ (Some\ op2)\ None$

**abbreviation DIVSD**  
**where**  $DIVSD\ op1\ op2 \equiv Instr\ "divsd"\ (Some\ op1)\ (Some\ op2)\ None$

**abbreviation UCOMISD**  
**where**  $UCOMISD\ op1\ op2 \equiv Instr\ "ucomisd"\ (Some\ op1)\ (Some\ op2)\ None$

**definition**  $simd\text{-}32\text{-}128 :: (32\ word \Rightarrow 32\ word \Rightarrow 32\ word) \Rightarrow 128\ word \Rightarrow 128\ word \Rightarrow 128\ word$   
**where**  $simd\text{-}32\text{-}128\ f\ dst\ src \equiv$   
 $((ucast\ (\langle 0,32 \rangle(f\ (ucast\ (\langle 96,128 \rangle dst))\ (ucast\ (\langle 96,128 \rangle src)))))) << 96)$   
*OR*  
 $((ucast\ (\langle 0,32 \rangle(f\ (ucast\ (\langle 64,96 \rangle dst))\ (ucast\ (\langle 64,96 \rangle src)))))) << 64)$   
*OR*  
 $((ucast\ (\langle 0,32 \rangle(f\ (ucast\ (\langle 32,64 \rangle dst))\ (ucast\ (\langle 32,64 \rangle src)))))) << 32)$   
*OR*  
 $((ucast\ (\langle 0,32 \rangle(f\ (ucast\ (\langle 0,32 \rangle dst))\ (ucast\ (\langle 0,32 \rangle src))))))$

**abbreviation**  $semantics\text{-}PADD :: Operand \Rightarrow Operand \Rightarrow state \Rightarrow state$   
**where**  $semantics\text{-}PADD \equiv binop\text{-}no\text{-}flags\ (simd\text{-}32\text{-}128\ (+))$

**abbreviation**  $PADD :: Operand \Rightarrow Operand \Rightarrow state \Rightarrow state$   
**where**  $PADD\ op1\ op2 \equiv Instr\ "padd"\ (Some\ op1)\ (Some\ op2)\ None$

**definition**  $pshufd :: 128\ word \Rightarrow 8\ word \Rightarrow 128\ word$   
**where**  $pshufd\ src\ n \equiv (((0,32)(src >> (unat\ (\langle 6,8 \rangle n)*32))) << 96) \ OR$   
 $((\langle 0,32 \rangle(src >> (unat\ (\langle 4,6 \rangle n)*32))) << 64) \ OR$   
 $((\langle 0,32 \rangle(src >> (unat\ (\langle 2,4 \rangle n)*32))) << 32) \ OR$   
 $((\langle 0,32 \rangle(src >> (unat\ (\langle 0,2 \rangle n)*32))))$

**lemmas**  $pshufd\text{-}numeral[simp] = pshufd\text{-}def[of numeral n]$  **for**  $n$   
**lemmas**  $pshufd\text{-}0[simp] = pshufd\text{-}def[of 0]$   
**lemmas**  $pshufd\text{-}1[simp] = pshufd\text{-}def[of 1]$

**definition**  $semantics\text{-}PSHUF :: Operand \Rightarrow Operand \Rightarrow Operand \Rightarrow state \Rightarrow state$   
**where**  $semantics\text{-}PSHUF\ op1\ op2\ op3\ \sigma \equiv$   
 $let\ src = ucast\ (operand\text{-}read\ \sigma\ op2);$   
 $n = ucast\ (operand\text{-}read\ \sigma\ op3)\ in$   
 $operand\text{-}write\ op1\ (ucast\ (pshufd\ src\ n))\ \sigma$

**abbreviation**  $PSHUF :: Operand \Rightarrow Operand \Rightarrow Operand \Rightarrow state \Rightarrow state$   
**where**  $PSHUF\ op1\ op2\ op3 \equiv Instr\ "pshuf"\ op1\ op2\ op3$

```

definition semantics-PEXTRD :: Operand  $\Rightarrow$  Operand  $\Rightarrow$  Operand  $\Rightarrow$  state  $\Rightarrow$  state
where semantics-PEXTRD op1 op2 op3  $\sigma$   $\equiv$ 
  let src = operand-read  $\sigma$  op2;
  n = unat (operand-read  $\sigma$  op3) mod 4 in
  operand-write op1 (ucast (((0,32)(src >> n*32))))  $\sigma$ 

abbreviation PEXTRD
where PEXTRD op1 op2 op3  $\equiv$  Instr "pextrd" op1 op2 op3

definition semantics-PINSRD :: Operand  $\Rightarrow$  Operand  $\Rightarrow$  Operand  $\Rightarrow$  state  $\Rightarrow$  state
where semantics-PINSRD op1 op2 op3  $\sigma$   $\equiv$ 
  let dst = ucast (operand-read  $\sigma$  op1)::128 word;
  src = ucast (operand-read  $\sigma$  op2)::128 word;
  n = unat (operand-read  $\sigma$  op3) mod 4;
  m = 0xFFFFFFFF << (n * 32) :: 128 word;
  t = (src << (n * 32)) AND m in
  operand-write op1 (ucast ((dst AND NOT m) OR t))  $\sigma$ 

abbreviation PINSRD
where PINSRD op1 op2 op3  $\equiv$  Instr "pinsrd" op1 op2 op3

```

```

definition bswap :: 32 word  $\Rightarrow$  32 word
where bswap w  $\equiv$  (((0,8>w) << 24) OR (((8,16>w) << 16) OR (((16,24>w)
<< 8) OR ((24,32>w))

lemmas bswap-numeral[simp] = bswap-def[of numeral n] for n
lemmas bswap-0[simp] = bswap-def[of 0]
lemmas bswap-1[simp] = bswap-def[of 1]

definition semantics-BSWAP :: Operand  $\Rightarrow$  state  $\Rightarrow$  state
where semantics-BSWAP  $\equiv$  unop-no-flags bswap

abbreviation BSWAP
where BSWAP op1  $\equiv$  Instr "bswap" op1 None None

```

```

definition semantics-NOP :: state  $\Rightarrow$  state
where semantics-NOP  $\equiv$  id

```

```

abbreviation NOP0

```

**where**  $NOP0 \equiv \text{Instr } "nop" \text{ None None None}$

**abbreviation**  $NOP1$

**where**  $NOP1 \text{ op1} \equiv \text{Instr } "nop" (\text{Some op1}) \text{ None None}$

**abbreviation**  $NOP2$

**where**  $NOP2 \text{ op1 op2} \equiv \text{Instr } "nop" (\text{Some op1}) (\text{Some op2}) \text{ None}$

**abbreviation**  $NOP3$

**where**  $NOP3 \text{ op1 op2 op3} \equiv \text{Instr } "nop" (\text{Some op1}) (\text{Some op2}) (\text{Some op3})$

**definition**  $\text{semantics}$

**where**  $\text{semantics } i \equiv$

**case**  $i$  **of**

$(\text{Instr } "mov")$	$(\text{Some op1}) (\text{Some op2}) \text{ - -})$	$\Rightarrow \text{semantics-MOV op1}$
$  (\text{Instr } "movabs")$	$(\text{Some op1}) (\text{Some op2}) \text{ - -})$	$\Rightarrow \text{semantics-MOV op1}$
$  (\text{Instr } "movaps")$	$(\text{Some op1}) (\text{Some op2}) \text{ - -})$	$\Rightarrow \text{semantics-MOV op1}$
$  (\text{Instr } "movdqu")$	$(\text{Some op1}) (\text{Some op2}) \text{ - -})$	$\Rightarrow \text{semantics-MOV op1}$
$  (\text{Instr } "movd")$	$(\text{Some op1}) (\text{Some op2}) \text{ - -})$	$\Rightarrow \text{semantics-MOVD op1}$
$  (\text{Instr } "movzx")$	$(\text{Some op1}) (\text{Some op2}) \text{ - -})$	$\Rightarrow \text{semantics-MOV op1}$
$  (\text{Instr } "movsd")$	$(\text{Some op1}) (\text{Some op2}) \text{ - -})$	$\Rightarrow \text{semantics-MOVSD}$
$  (\text{Instr } "movq")$	$(\text{Some op1}) (\text{Some op2}) \text{ - -})$	$\Rightarrow \text{semantics-MOVSD op1}$
$  (\text{Instr } "lea")$	$(\text{Some op1}) (\text{Some op2}) \text{ - -})$	$\Rightarrow \text{semantics-LEA op1 op2}$
$  (\text{Instr } "push")$	$(\text{Some op1}) \text{ - } \text{ - -})$	$\Rightarrow \text{semantics-PUSH op1}$
$  (\text{Instr } "pop")$	$(\text{Some op1}) \text{ - } \text{ - -})$	$\Rightarrow \text{semantics-POP op1}$
$  (\text{Instr } "ret")$	$\text{ - - - -})$	$\Rightarrow \text{semantics-RET}$
$  (\text{Instr } "call")$	$(\text{Some op1}) \text{ - - -})$	$\Rightarrow \text{semantics-CALL op1}$
$  (\text{Instr } "leave")$	$\text{ - - - -})$	$\Rightarrow \text{semantics-LEAVE}$
<b>— arithmetic</b>		
$  (\text{Instr } "add")$	$(\text{Some op1}) (\text{Some op2}) \text{ - -})$	$\Rightarrow \text{semantics-ADD op1 op2}$
$  (\text{Instr } "inc")$	$(\text{Some op1}) \text{ - } \text{ - -})$	$\Rightarrow \text{semantics-INC op1}$
$  (\text{Instr } "dec")$	$(\text{Some op1}) \text{ - } \text{ - -})$	$\Rightarrow \text{semantics-DEC op1}$
$  (\text{Instr } "neg")$	$(\text{Some op1}) \text{ - } \text{ - -})$	$\Rightarrow \text{semantics-NEG op1}$
$  (\text{Instr } "sub")$	$(\text{Some op1}) (\text{Some op2}) \text{ - -})$	$\Rightarrow \text{semantics-SUB op1 op2}$
$  (\text{Instr } "sbb")$	$(\text{Some op1}) (\text{Some op2}) \text{ - -})$	$\Rightarrow \text{semantics-SBB op1 op2}$
$  (\text{Instr } "adc")$	$(\text{Some op1}) (\text{Some op2}) \text{ - -})$	$\Rightarrow \text{semantics-ADC op1 op2}$
$  (\text{Instr } "mul")$	$(\text{Some op1}) \text{ - - -})$	$\Rightarrow \text{semantics-MUL op1}$
$  (\text{Instr } "imul")$	$(\text{Some op1}) \text{ None - -})$	$\Rightarrow \text{semantics-IMUL1 op1}$
$  (\text{Instr } "imul")$	$(\text{Some op1}) (\text{Some op2}) \text{ None - })$	$\Rightarrow \text{semantics-IMUL2 op1}$

$op2$		
( $Instr\ "imul"$ )	$(Some\ op1)\ (Some\ op2)\ (Some\ op3)\ \neg)$	$\Rightarrow semantics\text{-}IMUL3$
$op1\ op2\ op3$		
( $Instr\ "shl"$ )	$(Some\ op1)\ (Some\ op2)\ None\ \neg)$	$\Rightarrow semantics\text{-}SHL\ op1$
$op2$		
( $Instr\ "sal"$ )	$(Some\ op1)\ (Some\ op2)\ None\ \neg)$	$\Rightarrow semantics\text{-}SHL\ op1$
$op2$		
( $Instr\ "shr"$ )	$(Some\ op1)\ (Some\ op2)\ None\ \neg)$	$\Rightarrow semantics\text{-}SHR\ op1$
$op2$		
( $Instr\ "sar"$ )	$(Some\ op1)\ (Some\ op2)\ None\ \neg)$	$\Rightarrow semantics\text{-}SAR\ op1$
$op2$		
( $Instr\ "shld"$ )	$(Some\ op1)\ (Some\ op2)\ (Some\ op3)\ \neg)$	$\Rightarrow semantics\text{-}SHLD$
$op1\ op2\ op3$		
( $Instr\ "rol"$ )	$(Some\ op1)\ (Some\ op2)\ None\ \neg)$	$\Rightarrow semantics\text{-}ROL\ op1$
$op2$		
( $Instr\ "ror"$ )	$(Some\ op1)\ (Some\ op2)\ None\ \neg)$	$\Rightarrow semantics\text{-}ROR\ op1$
$op2$		
— flag-related		
( $Instr\ "cmp"$ )	$(Some\ op1)\ (Some\ op2)\ \dots$	$\Rightarrow semantics\text{-}CMP\ op1$
$op2$		
( $Instr\ "test"$ )	$(Some\ op1)\ (Some\ op2)\ \dots$	$\Rightarrow semantics\text{-}TEST\ op1$
$op2$		
— sign-extension		
( $Instr\ "movsxd"$ )	$(Some\ op1)\ (Some\ op2)\ \dots$	$\Rightarrow semantics\text{-}MOVSXD$
$op1\ op2$		
( $Instr\ "movsx"$ )	$(Some\ op1)\ (Some\ op2)\ \dots$	$\Rightarrow semantics\text{-}MOVSXD$
$op1\ op2$		
( $Instr\ "cdqe"$ )	$(Some\ op1)\ (Some\ op2)\ \dots$	$\Rightarrow semantics\text{-}CDQE$
( $Instr\ "cdq"$ )	$(Some\ op1)\ (Some\ op2)\ \dots$	$\Rightarrow semantics\text{-}CDQ$
( $Instr\ "cqo"$ )	$(Some\ op1)\ (Some\ op2)\ \dots$	$\Rightarrow semantics\text{-}CQO$
— logic		
( $Instr\ "and"$ )	$(Some\ op1)\ (Some\ op2)\ \dots$	$\Rightarrow semantics\text{-}AND\ op1\ op2$
( $Instr\ "or"$ )	$(Some\ op1)\ (Some\ op2)\ \dots$	$\Rightarrow semantics\text{-}OR\ op1\ op2$
( $Instr\ "xor"$ )	$(Some\ op1)\ (Some\ op2)\ \dots$	$\Rightarrow semantics\text{-}XOR\ op1\ op2$
( $Instr\ "xorps"$ )	$(Some\ op1)\ (Some\ op2)\ \dots$	$\Rightarrow semantics\text{-}XORPS\ op1$
$op2$		
( $Instr\ "not"$ )	$(Some\ op1)\ \dots$	$\Rightarrow semantics\text{-}NOT\ op1$
— jumps		
( $Instr\ "jmp"$ )	$(Some\ op1)\ None\ \dots$	$\Rightarrow semantics\text{-}JMP\ op1$
( $Instr\ "jo"$ )	$(Some\ op1)\ None\ \dots$	$\Rightarrow semantics\text{-}JO\ op1$
( $Instr\ "jno"$ )	$(Some\ op1)\ None\ \dots$	$\Rightarrow semantics\text{-}JNO\ op1$
( $Instr\ "js"$ )	$(Some\ op1)\ None\ \dots$	$\Rightarrow semantics\text{-}JS\ op1$
( $Instr\ "jns"$ )	$(Some\ op1)\ None\ \dots$	$\Rightarrow semantics\text{-}JNS\ op1$
( $Instr\ "je"$ )	$(Some\ op1)\ None\ \dots$	$\Rightarrow semantics\text{-}JE\ op1$
( $Instr\ "jz"$ )	$(Some\ op1)\ None\ \dots$	$\Rightarrow semantics\text{-}JE\ op1$
( $Instr\ "jne"$ )	$(Some\ op1)\ None\ \dots$	$\Rightarrow semantics\text{-}JNE\ op1$
( $Instr\ "jnz"$ )	$(Some\ op1)\ None\ \dots$	$\Rightarrow semantics\text{-}JNE\ op1$
( $Instr\ "jb"$ )	$(Some\ op1)\ None\ \dots$	$\Rightarrow semantics\text{-}JB\ op1$
( $Instr\ "jnae"$ )	$(Some\ op1)\ None\ \dots$	$\Rightarrow semantics\text{-}JB\ op1$

(Instr "jc"	(Some op1) None - -)	$\Rightarrow semantics\text{-}JB\ op1$
(Instr "jnb"	(Some op1) None - -)	$\Rightarrow semantics\text{-}JNB\ op1$
(Instr "jae"	(Some op1) None - -)	$\Rightarrow semantics\text{-}JNB\ op1$
(Instr "jnc"	(Some op1) None - -)	$\Rightarrow semantics\text{-}JNB\ op1$
(Instr "jbe"	(Some op1) None - -)	$\Rightarrow semantics\text{-}JBE\ op1$
(Instr "jna"	(Some op1) None - -)	$\Rightarrow semantics\text{-}JBE\ op1$
(Instr "ja"	(Some op1) None - -)	$\Rightarrow semantics\text{-}JA\ op1$
(Instr "jnbe"	(Some op1) None - -)	$\Rightarrow semantics\text{-}JA\ op1$
(Instr "jl"	(Some op1) None - -)	$\Rightarrow semantics\text{-}JL\ op1$
(Instr "jnge"	(Some op1) None - -)	$\Rightarrow semantics\text{-}JL\ op1$
(Instr "jge"	(Some op1) None - -)	$\Rightarrow semantics\text{-}JGE\ op1$
(Instr "jnl"	(Some op1) None - -)	$\Rightarrow semantics\text{-}JGE\ op1$
(Instr "jle"	(Some op1) None - -)	$\Rightarrow semantics\text{-}JLE\ op1$
(Instr "jng"	(Some op1) None - -)	$\Rightarrow semantics\text{-}JLE\ op1$
(Instr "jg"	(Some op1) None - -)	$\Rightarrow semantics\text{-}JG\ op1$
(Instr "jnle"	(Some op1) None - -)	$\Rightarrow semantics\text{-}JG\ op1$
— setXX		
(Instr "seto"	(Some op1) None - -)	$\Rightarrow semantics\text{-}SETO\ op1$
(Instr "setno"	(Some op1) None - -)	$\Rightarrow semantics\text{-}SETNO\ op1$
(Instr "sets"	(Some op1) None - -)	$\Rightarrow semantics\text{-}SETS\ op1$
(Instr "setns"	(Some op1) None - -)	$\Rightarrow semantics\text{-}SETNS\ op1$
(Instr "sete"	(Some op1) None - -)	$\Rightarrow semantics\text{-}SETE\ op1$
(Instr "setz"	(Some op1) None - -)	$\Rightarrow semantics\text{-}SETE\ op1$
(Instr "setne"	(Some op1) None - -)	$\Rightarrow semantics\text{-}SETNE\ op1$
(Instr "setnz"	(Some op1) None - -)	$\Rightarrow semantics\text{-}SETNE\ op1$
(Instr "setb"	(Some op1) None - -)	$\Rightarrow semantics\text{-}SETB\ op1$
(Instr "setnae"	(Some op1) None - -)	$\Rightarrow semantics\text{-}SETB\ op1$
(Instr "setc"	(Some op1) None - -)	$\Rightarrow semantics\text{-}SETB\ op1$
(Instr "setnb"	(Some op1) None - -)	$\Rightarrow semantics\text{-}SETNB\ op1$
(Instr "setae"	(Some op1) None - -)	$\Rightarrow semantics\text{-}SETNB\ op1$
(Instr "setnc"	(Some op1) None - -)	$\Rightarrow semantics\text{-}SETNB\ op1$
(Instr "setbe"	(Some op1) None - -)	$\Rightarrow semantics\text{-}SETBE\ op1$
(Instr "setna"	(Some op1) None - -)	$\Rightarrow semantics\text{-}SETBE\ op1$
(Instr "seta"	(Some op1) None - -)	$\Rightarrow semantics\text{-}SETA\ op1$
(Instr "setnbe"	(Some op1) None - -)	$\Rightarrow semantics\text{-}SETA\ op1$
(Instr "setl"	(Some op1) None - -)	$\Rightarrow semantics\text{-}SETL\ op1$
(Instr "setnge"	(Some op1) None - -)	$\Rightarrow semantics\text{-}SETL\ op1$
(Instr "setge"	(Some op1) None - -)	$\Rightarrow semantics\text{-}SETGE\ op1$
(Instr "setnl"	(Some op1) None - -)	$\Rightarrow semantics\text{-}SETGE\ op1$
(Instr "setle"	(Some op1) None - -)	$\Rightarrow semantics\text{-}SETLE\ op1$
(Instr "setng"	(Some op1) None - -)	$\Rightarrow semantics\text{-}SETLE\ op1$
(Instr "setg"	(Some op1) None - -)	$\Rightarrow semantics\text{-}SETG\ op1$
(Instr "setnle"	(Some op1) None - -)	$\Rightarrow semantics\text{-}SETG\ op1$
— conditional moves		
(Instr "cmovne"	(Some op1) (Some op2) - -	$\Rightarrow semantics\text{-}CMOVNE$
op1 op2		
(Instr "cmovns"	(Some op1) (Some op2) - -	$\Rightarrow semantics\text{-}CMOVNS$
op1 op2		
— floating point (double)		

$  (Instr \text{ ``addsd''} (Some op1) (Some op2) - -)$	$\Rightarrow semantics\text{-ADDSD } op1$
$  (Instr \text{ ``subsd''} (Some op1) (Some op2) - -)$	$\Rightarrow semantics\text{-SUBSD } op1$
$  (Instr \text{ ``mulsd''} (Some op1) (Some op2) - -)$	$\Rightarrow semantics\text{-MULSD } op1$
$  (Instr \text{ ``divsd''} (Some op1) (Some op2) - -)$	$\Rightarrow semantics\text{-DIVSD } op1$
$  (Instr \text{ ``ucomisd''} (Some op1) (Some op2) - -)$	$\Rightarrow semantics\text{-UCOMISD }$
$op1 op2$	
$— \text{simd}$	
$  (Instr \text{ ``paddsd''} (Some op1) (Some op2) - -)$	$\Rightarrow semantics\text{-PADDSD } op1$
$op2$	
$  (Instr \text{ ``pshufd''} (Some op1) (Some op2) (Some op3) -)$	$\Rightarrow semantics\text{-PSHUFD }$
$op1 op2 op3$	
$  (Instr \text{ ``pextrd''} (Some op1) (Some op2) (Some op3) -)$	$\Rightarrow semantics\text{-PEXTRD }$
$op1 op2 op3$	
$  (Instr \text{ ``pinsrd''} (Some op1) (Some op2) (Some op3) -)$	$\Rightarrow semantics\text{-PINSRD }$
$op1 op2 op3$	
$— \text{remainder}$	
$  (Instr \text{ ``nop''} - - - -)$	$\Rightarrow semantics\text{-NOP}$
$  (Instr \text{ ``bswap''} (Some op1) - - -)$	$\Rightarrow semantics\text{-BSWAP } op1$
$— \text{external function}$	
$  (ExternalFunc f) \Rightarrow f$	
$  i \Rightarrow unknown\text{-semantics } i$	

A step function. In X86, the RIP register is incremented before the instruction is executed. This is important, e.g., when RIP is used in a jump address.

```
definition step :: I ⇒ state ⇒ state
  where step i σ ≡
    let σ' = σ with [setRip (instr-next i)] in
      semantics i σ'
```

All simplification rules used during symbolic execution.

```
lemmas semantics-simps =
  Let-def unop-def unop-no-flags-def binop-def binop-flags-def binop-no-flags-def
  binop-XMM-def
  semantics-def mov-sign-extension-def simd-32-128-def
  write-MUL-result-def unop-MUL-def ternop-IMUL-def sbb-def adc-def shld-def

  semantics-MOV-def semantics-MOVSD-def semantics-MOVD-def semantics-CMOV-def
  semantics-LEA-def semantics-PUSH-def semantics-POP-def
  semantics-RET-def semantics-CALL-def semantics-LEAVE-def
  semantics-ADD-def semantics-INC-def semantics-DEC-def semantics-NEG-def
  semantics-SUB-def
  semantics-SBB-def semantics-ADC-def
  semantics-MUL-def semantics-IMUL1-def semantics-IMUL2-def semantics-IMUL3-def
```

```

semantics-SHL-def semantics-SHR-def semantics-SAR-def semantics-SHLD-def
semantics-ROL-def semantics-ROR-def
semantics-CMP-def semantics-TEST-def
semantics-MOVSXD-def semantics-CDQE-def semantics-CDQ-def semantics-CQO-def
semantics-AND-def semantics-OR-def semantics-XOR-def semantics-XORPS-def
semantics-NOT-def
semantics-cond-jump-def semantics-JMP-def
semantics-JO-def semantics-JNO-def semantics-JS-def semantics-JNS-def
semantics-JE-def semantics-JNE-def semantics-JB-def semantics-JNB-def
semantics-JBE-def semantics-JA-def semantics-JL-def semantics-JGE-def
semantics-JLE-def semantics-JG-def
semantics-setXX-def
semantics-ADDSD-def semantics-SUBSD-def semantics-MULSD-def semantics-DIVSD-def semantics-UCOMISD-def
semantics-NOP-def semantics-Bswap-def semantics-PSHUFDef semantics-PEXTRD-def semantics-PINSRD-def

SUB-flags-def ADD-flags-def INC-flags-def DEC-flags-def NEG-flags-def MUL-flags-def
IMUL-flags-def
SHL-flags-def SHR-flags-def SAR-flags-def SHLD-flags-def logic-flags-def UCOMISD-flags-def

```

```

end
end

```

## 5 Removing superfluous memory writes

```

theory StateCleanUp
imports State HOL-Eisbach.Eisbach
begin

```

```

definition assumptions ≡ id

```

We are going to make schematic theorems of the form:

$$\text{assumptions?} A \implies \dots$$

The assumptions will be generated on-the-fly. The seemingly weird lemmas below achieves that.

```

lemma assumptions-impI:
assumes assumptions (P ∧ A)
shows P
⟨proof⟩

lemma assumptions-conjE:
shows assumptions (P ∧ A) ⟷ P ∧ assumptions A
⟨proof⟩

```

```

lemma assumptionsI:
  shows assumptions True
  ⟨proof⟩

```

Consider two consecutive memory updates. If they write to the same memory locations, only one of these need to be kept. We formulate an Eisbach method to that end.

Returns true if two states are equal except for a specific memory region.

```

definition eq-except-for-mem :: state ⇒ state ⇒ 64 word ⇒ nat ⇒ 256 word ⇒
  bool ⇒ bool
  where eq-except-for-mem σ σ' a si v b ≡ σ with [[a,si]] :=m v] = (if b then σ'
    else σ' with [[a,si]] :=m v])

```

```

lemma eefm-start:

```

```

  assumes eq-except-for-mem (σ with updates) (σ with updates') a si v b
  shows (σ with (([a,si]] :=m v) #updates)) = (if b then σ with updates' else σ with
    (([a,si]] :=m v) #updates'))
  ⟨proof⟩

```

```

lemma eefm-clean-mem:

```

```

  assumes si' ≤ si
    and eq-except-for-mem (σ with updates) (σ with updates') a si v b
    shows eq-except-for-mem (σ with (([a,si]] :=m v) #updates)) (σ with updates')
      a si v b
  ⟨proof⟩

```

```

method eefm-clean-mem = (rule eefm-clean-mem, (simp (no-asm);fail))

```

```

lemma eefm-clean-mem-enclosed:

```

```

  assumes si < 32
    and enclosed a' si' a si
    and eq-except-for-mem (σ with updates) (σ with updates') a' si' v' b
    shows eq-except-for-mem (σ with (([a,si]] :=m v) #updates))
      (σ with (([a,si]] :=m overwrite (8 * unat (a' - a)) (8 * unat
        (a' - a) + 8 * si') v (v' << unat (a'-a)*8)) #updates'))
      a' si' v' True
  ⟨proof⟩

```

```

lemmas eefm-clean-mem-enclosed-plus = eefm-clean-mem-enclosed[OF - enclosed-plus,
  simplified]

```

```

lemmas eefm-clean-mem-enclosed-minus-numeral = eefm-clean-mem-enclosed[OF
  - enclosed-minus-minus, of - numeral n numeral m] for n m

```

```

method eefm-clean-mem-enclosed-plus =
  (rule eefm-clean-mem-enclosed-plus, (

```

```

(simp (no-asm);fail), (simp (no-asm);fail),
(
  (simp (no-asm-simp);fail) |
    (rule assumptions-impI[of - + - < 18446744073709551616],simp
      (no-asm-simp),subst (asm) assumptions-conjE))
)
)

method eefm-clean-mem-enclosed-minus-numeral =
  (rule eefm-clean-mem-enclosed-minus-numeral, (
    (simp (no-asm);fail), (simp (no-asm);fail), (simp (no-asm);fail), (simp
      (no-asm);fail), (simp (no-asm);fail),
    (
      (simp (no-asm-simp);fail) |
        (rule assumptions-impI[of - ≤ -],simp (no-asm-simp),subst (asm)
          assumptions-conjE))
    )
  )

lemma eefm-next-mem:
assumes separate a si a' si'
and eq-except-for-mem ( $\sigma$  with updates) ( $\sigma$  with updates') a si v b
shows eq-except-for-mem ( $\sigma$  with  $((\llbracket a', si \rrbracket :=_m v') \# \text{updates})$ ) ( $\sigma$  with  $((\llbracket a', si \rrbracket :=_m v') \# \text{updates}')$ ) a si v b
  ⟨proof⟩

method eefm-next-mem =
  (rule eefm-next-mem,
   ( (simp (no-asm-simp) add: separate-simps state-simps; fail) |
     (rule assumptions-impI[of separate - - - ],simp (no-asm-simp),subst
       (asm) assumptions-conjE))
  )
)

lemma eefm-end:
shows eq-except-for-mem ( $\sigma$  with updates) ( $\sigma$  with updates) a si v False
  ⟨proof⟩

```

We need a tactic exactly like “`subst`” but that applies to the outer most term.

$\langle ML \rangle$

The following method takes a goal with state with symbolic state updates. It first applies `eq-except-for-mem` ( $\sigma$  with  $\text{updates}$ ) ( $\sigma$  with  $\text{updates}'$ )  $?a ?si ?v ?b \implies \sigma \text{ with } ((\llbracket ?a, ?si \rrbracket :=_m ?v) \# \text{updates}) = (\text{if } ?b \text{ then } \sigma \text{ with } \text{updates}' \text{ else } \sigma \text{ with } ((\llbracket ?a, ?si \rrbracket :=_m ?v) \# \text{updates}'))$ , considering the outer-most memory update to some region  $\llbracket a, si \rrbracket$ . A list  $\text{updates}'$  is generated that produces a similar state except for region  $\llbracket a, si \rrbracket$ . The list thus can have fewer updates since any update to a region that is enclosed in region  $\llbracket a, si \rrbracket$  can be removed. Consecutively, this method applies rules to

determine whether a state update can be kept, merged or removed. It may add assumptions to the goal, that assume no overflow.

```
method clean-up-mem = (
  mysubst eefm-start,
  ( eefm-clean-mem | eefm-clean-mem-enclosed-plus | eefm-clean-mem-enclosed-minus-numeral
  | eefm-next-mem)+,
  rule eefm-end,
  simp (no-asm),
  ((match premises in A[thin]: assumptions (?A ∧ ?B) ⇒ ⟨cut-tac A; subst (asm)
  assumptions-conjE, erule conjE ⟩)+)
)
```

The method above applies to one state update. This method can be repeated as long as modifies the goal, as it always makes the goal smaller. The method below repeats a given method until the goal is unchanged. In deterministic fashion (a la the REPEAT\_DETERM tactic).

$\langle ML \rangle$

```
method clean-up = repeat-until-unchanged clean-up-mem
end
```

## 6 A symbolic execution engine

```
theory SymbolicExecution
  imports X86-InstructionSemantics StateCleanUp
begin
```

```
definition eq (infixl  $\triangleq$  50)
  where ( $\triangleq$ )  $\equiv (=)$ 
```

```
context unknowns
begin
```

```
inductive run :: 64 word  $\Rightarrow$  (64 word  $\Rightarrow$  I)  $\Rightarrow$  state  $\Rightarrow$  state  $\Rightarrow$  bool
  where
    rip  $\sigma = a_f \implies$  fetch (rip  $\sigma$ ) = i  $\implies$   $\sigma' \triangleq$  step i  $\sigma \implies$  run  $a_f$  fetch  $\sigma$   $\sigma'$ 
    | rip  $\sigma \neq a_f \implies$  fetch (rip  $\sigma$ ) = i  $\implies$  run  $a_f$  fetch (step i  $\sigma$ )  $\sigma' \implies$  run  $a_f$ 
      fetch  $\sigma$   $\sigma'$ 
```

```
method fetch-and-execute = (
  ((rule run.intros(2),(simp (no-asm) add: state-simps; fail))
  | (rule run.intros(1),(simp (no-asm) add: state-simps; fail))),
```

```

  (simp (no-asm) add: state-simps), — fetch
instruction
  (simp (no-asm-simp) add: step-def semantics-simps state-simps BitByte-simps),
— simplification
  (subst clean-state-updates[symmetric],simp (no-asm)) —
cleaning up
)

method resolve-mem-reads =
  subst mem-read-mem-write-separate,
  ((simp (no-asm-simp) add: separate-simps state-simps; fail)
  | (rule assumptions-impI,simp (no-asm-simp),subst (asm) assumptions-conjE,
  erule conjE)),
  (simp (no-asm-simp) add: semantics-simps state-simps BitByte-simps separate-simps)?
)

method se-step =
  fetch-and-execute,
  ((resolve-mem-reads)+;(subst clean-state-updates[symmetric],simp (no-asm)))?,
  clean-up

method se-step-no-clean =
  fetch-and-execute,
  ((resolve-mem-reads)+;(subst clean-state-updates[symmetric],simp (no-asm)))?

end

```

```

abbreviation RSP0
  where RSP0 σ ≡ regs σ "rsp"
abbreviation RBP0
  where RBP0 σ ≡ regs σ "rbp"
abbreviation RAX0
  where RAX0 σ ≡ regs σ "rax"
abbreviation RBX0
  where RBX0 σ ≡ regs σ "rbx"
abbreviation RCX0
  where RCX0 σ ≡ regs σ "rcx"
abbreviation RDX0
  where RDX0 σ ≡ regs σ "rdx"
abbreviation RDI0
  where RDI0 σ ≡ regs σ "rdi"
abbreviation RSI0
  where RSI0 σ ≡ regs σ "rsi"
abbreviation R150
  where R150 σ ≡ regs σ "r15"
abbreviation R140
  where R140 σ ≡ regs σ "r14"
abbreviation R130

```

```

where R130  $\sigma \equiv \text{regs } \sigma \text{ "r13"}$ 
abbreviation R120
  where R120  $\sigma \equiv \text{regs } \sigma \text{ "r12"}$ 
abbreviation R110
  where R110  $\sigma \equiv \text{regs } \sigma \text{ "r11"}$ 
abbreviation R100
  where R100  $\sigma \equiv \text{regs } \sigma \text{ "r10"}$ 
abbreviation R90
  where R90  $\sigma \equiv \text{regs } \sigma \text{ "r9"}$ 
abbreviation R80
  where R80  $\sigma \equiv \text{regs } \sigma \text{ "r8"}$ 

  Repeat a command up to n times, in deterministic fashion (a la the
REPEAT_DETERM tactic).

⟨ML⟩

end

```

## 7 Small examples

```

theory Examples
  imports SymbolicExecution
begin

context unknowns
begin

```

A simple hand-crafted example showing some basic instructions.

```

schematic-goal example1:
  assumes[simp]: fetch 0x0 = PUSH (Reg "rbp") 1
  and[simp]: fetch 0x1 = SUB (Reg "rsp") (Imm 30) 2
  and[simp]: fetch 0x2 = MOV (QWORD PTR [22 + "rsp"]2) (Imm 42) 3
  and[simp]: fetch 0x3 = MOV (QWORD PTR [30 + "rsp"]2) (Imm 43) 4
  and[simp]: fetch 0x4 = ADD (Reg "rsp") (Imm 30) 5
  and[simp]: fetch 0x5 = POP (Reg "rbp") 6
  and[simp]: fetch 0x6 = RET 1
  shows run 0x6 fetch ( $\sigma$  with [setRip 0x0]) ? $\sigma'$ 
⟨proof⟩

```

**thm** example1

Demonstrates little-endian memory and register-aliasing

RAX	+	0	1	2	3	4	5	6	7									
		FF		EE		DD		CC		BB		AA		00		00		

  

```

EDI := 0xCCDDEEFF
EBX := 0xAABB
RCX := 0xAABCDDAABB

```

```

schematic-goal example2:
  assumes[simp]: fetch 0x0 = MOV (QWORD PTR ["rax'"]1) (Imm 0xAABBCCDDEEFF) 1
    and[simp]: fetch 0x1 = MOV (Reg edi') (DWORD PTR ["rax'"]1)
  2
    and[simp]: fetch 0x2 = MOV (Reg ebx') (DWORD PTR [4 + "rax'"]2) 3
      and[simp]: fetch 0x3 = MOV (Reg rcx') (QWORD PTR ["rax'"]1)
  4
    and[simp]: fetch 0x4 = MOV (Reg cx') (WORD PTR [4 + "rax'"]2) 5
    shows run 0x4 fetch ( $\sigma$  with [setRip 0x0]) ? $\sigma'$ 
    ⟨proof⟩

```

**thm** example2

This example show how assumptions over regions are generated. Since no relation over rax and rbx is known in the initial state, they will be assumed to be separate by default.

```

schematic-goal example3:
  assumes[simp]: fetch 0x0 = MOV (QWORD PTR ["rax'"]1) (Imm 0xAABBCCDDEEFF) 1
    and[simp]: fetch 0x1 = MOV (QWORD PTR ["rbx'"]1) (Imm 0x112233445566)
  2
    and[simp]: fetch 0x2 = MOV (Reg rcx') (DWORD PTR [2 + "rax'"]2) 3
      and[simp]: fetch 0x3 = MOV (Reg cx') (WORD PTR [4 + "rbx'"]2) 4
        and[simp]: fetch 0x4 = MOV (Reg cl') (BYTE PTR ["rax'"]1) 5
        shows assumptions ?A  $\implies$  run 0x4 fetch ( $\sigma$  with [setRip 0x0]) ? $\sigma'$ 
    ⟨proof⟩

```

**thm** example3

**end**

**end**

## 8 Parser

```

theory X86-Parse
  imports X86-InstructionSemantics
  keywords x86-64-parser :: thy-decl

```

```
begin
```

```
⟨ML⟩
```

```
end
```

## 9 Example: word count program from GNU

```
theory Example-WC
  imports SymbolicExecution X86-Parse
begin
```

The wordcount (wc) program, specifically, the functions getword and counter. We compiled the source code found here:

[https://www.gnu.org/software/cflow/manual/html\\_node/Source-of-wc-command.html](https://www.gnu.org/software/cflow/manual/html_node/Source-of-wc-command.html)

The source code is also found in the directory `./examples/wc`.

The assembly below has been obtained by running in `./examples/wc`:

```
gcc wc.c -o wc
objdump -M intel -d --no-show-raw-instrn wc
```

This example:

- contains a lot of memory accesses and demonstrates how a memory model is generated through assumptions;
- contains external function calls and demonstrates how to deal with that.

First, we define definitions named “EXTERNAL\_FUNCTION\_\*” for each external function. The definitions are added to the simplifier.

We model a C file (of C-type “FILE”) as a pointer to a part of memory that contains the contents. We assume the contents are 0-terminated.

Function `feof` takes as input (via `rdi`) a `FILE*`. It reads one byte from `**rdi`, i.e., the next byte of the file. It returns true iff the byte equals 0.

Function `fopen` writes into memory both 1.) the contents of a file (the string "Hello"), and 2.) a pointer to the beginning of that file. It returns a pointer to that pointer.

Function `getc` reads the next byte of the `FILE` (same as `feof`) and increments the pointer.

Function `isword` simply returns true, and functions `report` and `fclose` simply do nothing.

```

context unknowns
begin

definition EXTERNAL-FUNCTION-feof :: state  $\Rightarrow$  state
where EXTERNAL-FUNCTION-feof  $\sigma \equiv$ 
  let  $ptr = \text{ucast}(\text{operand-read } \sigma (\text{QWORD PTR } ["rdi'"]_1))$ ;
   $val = \text{mem-read } \sigma ptr 1 \text{ in}$ 
  (semantics-RET  $\sigma$ 
    semantics-MOV (Reg "eax") (Imm (fromBool (val = 0))))
   $\sigma$ 

declare EXTERNAL-FUNCTION-feof-def [simp]

definition EXTERNAL-FUNCTION--IO-getc :: state  $\Rightarrow$  state
where EXTERNAL-FUNCTION--IO-getc  $\sigma \equiv$ 
  let  $ptr = \text{ucast}(\text{operand-read } \sigma (\text{QWORD PTR } ["rdi'"]_1))$ ;
   $val = \text{mem-read } \sigma ptr 1 \text{ in}$ 
  (semantics-RET  $\sigma$ 
    semantics-MOV (Reg "rax") (Imm (if val = 0 then -1 else val)) o
    semantics-INC (QWORD PTR ["rdi'"]_1))
   $\sigma$ 

declare EXTERNAL-FUNCTION--IO-getc-def [simp]

definition EXTERNAL-FUNCTION-fopen  $\sigma =$ 
  semantics-RET ( $\sigma$  with ["rax"] :=r 100,
    [100,8] :=m 108,
    [108,6] :=m 0x006E6C6C6548)

declare EXTERNAL-FUNCTION-fopen-def [simp]

definition EXTERNAL-FUNCTION-isword :: state  $\Rightarrow$  state
where EXTERNAL-FUNCTION-isword = operand-write (Reg "rax") 1 o semantics-RET

declare EXTERNAL-FUNCTION-isword-def [simp]

definition EXTERNAL-FUNCTION-fclose :: state  $\Rightarrow$  state
where EXTERNAL-FUNCTION-fclose = semantics-RET

declare EXTERNAL-FUNCTION-fclose-def [simp]

definition EXTERNAL-FUNCTION-report :: state  $\Rightarrow$  state
where EXTERNAL-FUNCTION-report = semantics-RET

declare EXTERNAL-FUNCTION-report-def [simp]
```

```
end
```

Below, one can see that, e.g. 810 denotes an external call (see address 0xc1b). For each external call, we replace the actual .got.plt section with a special instruction *ExternalFunc* followed by a name. These special instructions are interpreted as executing the related definition above.

```
context unknowns
begin
x86-64-parser wc-objdump <
7d0: EXTERNAL-FUNCTION fclose
810: EXTERNAL-FUNCTION feof
820: EXTERNAL-FUNCTION -IO-getc
830: EXTERNAL-FUNCTION fopen
bd5: EXTERNAL-FUNCTION isword
b93: EXTERNAL-FUNCTION report

c01: push rbp
c02: mov rbp, rsp
c05: sub rsp, 0x20
c09: mov QWORD PTR [rbp-0x18], rdi
c0d: mov DWORD PTR [rbp-0x4], 0x0
c14: mov rax, QWORD PTR [rbp-0x18]
c18: mov rdi, rax
c1b: call 810 <feof@plt>
c20: test eax, eax
c22: je c7d <getword+0x7c>
c24: mov eax, 0x0
c29: jmp cf1 <getword+0xf0>
c2e: mov eax, DWORD PTR [rbp-0x8]
c31: movzx eax, al
c34: mov edi, eax
c36: call bd5 <isword>
c3b: test eax, eax
c3d: je c53 <getword+0x52>
c3f: mov rax, QWORD PTR [rip+0x201402] # 202048 <wcount>
c46: add rax, 0x1
c4a: mov QWORD PTR [rip+0x2013f7], rax # 202048 <wcount>
c51: jmp c92 <getword+0x91>
c53: mov rax, QWORD PTR [rip+0x2013f6] # 202050 <ccount>
c5a: add rax, 0x1
c5e: mov QWORD PTR [rip+0x2013eb], rax # 202050 <ccount>
c65: cmp DWORD PTR [rbp-0x8], 0xa
c69: jne c7d <getword+0x7c>
c6b: mov rax, QWORD PTR [rip+0x2013e6] # 202058 <lcount>
c72: add rax, 0x1
c76: mov QWORD PTR [rip+0x2013db], rax # 202058 <lcount>
c7d: mov rax, QWORD PTR [rbp-0x18]
c81: mov rdi, rax
```

```

c84: call 820 <-IO-getc@plt>
c89: mov DWORD PTR [rbp-0x8],eax
c8c: cmp DWORD PTR [rbp-0x8],0xffffffff
c90: jne c2e <getword+0x2d>
c92: jmp cde <getword+0xdd>
c94: mov rax,QWORD PTR [rip+0x2013b5]      # 202050 <ccount>
c9b: add rax,0x1
c9f: mov QWORD PTR [rip+0x2013aa],rax      # 202050 <ccount>
ca6: cmp DWORD PTR [rbp-0x8],0xa
caa: jne cbe <getword+0xbd>
cac: mov rax,QWORD PTR [rip+0x2013a5]      # 202058 <lcount>
cb3: add rax,0x1
cb7: mov QWORD PTR [rip+0x20139a],rax      # 202058 <lcount>
cbe: mov eax,DWORD PTR [rbp-0x8]
cc1: movzx eax,al
cc4: mov edi, eax
cc6: call bd5 <isword>
ccb: test eax, eax
ccd: je ce6 <getword+0xe5>
ccf: mov rax,QWORD PTR [rbp-0x18]
cd3: mov rdi,rax
cd6: call 820 <-IO-getc@plt>
cdb: mov DWORD PTR [rbp-0x8],eax
cde: cmp DWORD PTR [rbp-0x8],0xffffffff
ce2: jne c94 <getword+0x93>
ce4: jmp ce7 <getword+0xe6>
ce6: nop
ce7: cmp DWORD PTR [rbp-0x8],0xffffffff
ceb: setne al
cee: movzx eax,al
cf1: leave
cf2: ret

cf3: push rbp
cf4: mov rbp, rsp
cf7: sub rsp, 0x20
cfb: mov QWORD PTR [rbp-0x18],rdi
cff: mov rax,QWORD PTR [rbp-0x18]
d03: lea rsi,[rip+0x1ff]      # f09 <-IO-stdin-used+0x19>
d0a: mov rdi,rax
d0d: call 830 <fopen@plt>
d12: mov rax,QWORD PTR [rbp-0x8],rax
d16: cmp QWORD PTR [rbp-0x8],0x0
d1b: jne d35 <counter+0x42>
d1d: mov rax,QWORD PTR [rbp-0x18]
d21: mov rsi,rax
d24: lea rdi,[rip+0x1e0]      # f0b <-IO-stdin-used+0x1b>
d2b: mov eax,0x0

```

```

d30: call ac6 <perrf>
d35: mov QWORD PTR [rip+0x201318],0x0          # 202058 <lcount>
d40: mov rax,QWORD PTR [rip+0x201311]          # 202058 <lcount>
d47: mov QWORD PTR [rip+0x2012fa],rax          # 202048 <wcount>
d4e: mov rax,QWORD PTR [rip+0x2012f3]          # 202048 <wcount>
d55: mov QWORD PTR [rip+0x2012f4],rax          # 202050 <ccount>
d5c: nop
d5d: mov rax,QWORD PTR [rbp-0x8]
d61: mov rdi,rax
d64: call c01 <getword>
d69: test eax,eax
d6b: jne d5d <counter+0x6a>
d6d: mov rax,QWORD PTR [rbp-0x8]
d71: mov rdi,rax
d74: call 7d0 <fclose@plt>
d79: mov rcx,QWORD PTR [rip+0x2012d8]
d80: mov rdx,QWORD PTR [rip+0x2012c1]
d87: mov rsi,QWORD PTR [rip+0x2012c2]
d8e: mov rax,QWORD PTR [rbp-0x18]
d92: mov rdi,rax
d95: call b93 <report>
d9a: mov rdx,QWORD PTR [rip+0x20128f]
da1: mov rax,QWORD PTR [rip+0x2012a8]
da8: add rax,rdx
dab: mov QWORD PTR [rip+0x20127e],rax
db2: mov rdx,QWORD PTR [rip+0x20127f]
db9: mov rax,QWORD PTR [rip+0x201288]
dc0: add rax,rdx
dc3: mov QWORD PTR [rip+0x20126e],rax
dca: mov rdx,QWORD PTR [rip+0x20126f]
dd1: mov rax,QWORD PTR [rip+0x201280]
dd8: add rax,rdx
ddb: mov QWORD PTR [rip+0x20125e],rax
de2: nop
de3: leave
de4: ret
>
end

context wc-objdump
begin
find-theorems fetch

```

Note: this theorems takes roughly 15 minutes to prove.

**schematic-goal** counter:

**assumes**  $\sigma_I = \sigma$  with [setRip 0xcf3]  
**shows** assumptions ?A  $\implies$  run 0xde4 fetch  $\sigma_I$  ? $\sigma'$   
 $\langle proof \rangle$

### **thm counter**

The file opened by "fopen" contains the zero-terminated string "Hello": 0x006E6C6C6548. After each call of getc, register RAX contains the read characters' ASCII code.

After termination, we can see the contents of the following global variables, set by function getword:

```
Word count:      wcount = 1      (0x202048 = 2105416)
Character count: ccount = 5      (0x202050 = 2105424)
Line count:      lcount = lcount (0x202058 = 2105432)
```

The totals accumulate to:

```
Word count:      total_wcount = total_wcount + 5 (0x202038 = 2105400)
Character count: total_ccount = total_ccount + 5 (0x202030 = 2105392)
Line count:      total_lcount = total_lcount      (0x202040 = 2105408)
```

```
end
end
[1]
```

## References

- [1] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283. 2002. <http://www.in.tum.de/~nipkow/LNCS2283/>.