

X86 instruction semantics and basic block symbolic execution

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

December 14, 2021

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

1	Bit and byte-level theorems	2
1.1	Basics	2
1.2	Take_Bits and arithmetic	4
2	Memory-related theorems	6

3	Concrete state and instructions	11
4	Instruction Semantics	20
5	Removing superfluous memory writes	52
6	A symbolic execution engine	55
7	Small examples	57
8	Parser	58
9	Example: word count program from GNU	59

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

where *take-bits* $l\ h\ w \equiv (w \gg l)\ \text{AND}\ \text{mask}\ (h-l)$

$\langle l, h \rangle w$ 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* :: $nat \Rightarrow 'a::len\ word \Rightarrow 8\ word$ — little-endian

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

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

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 h, \text{LENGTH}\('a)\rangle w0) \ll h)\ \text{OR}\ ((\langle l, h \rangle w1) \ll l)\ \text{OR}\ (\langle 0, l \rangle w0)$

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 $(\bigwedge n. n < \text{size}\ ?u \longrightarrow ?u\ !!\ n = ?v\ !!\ n) \implies ?u = ?v$.

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

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

lemma *bit-take-byte-iff* [*bit-simps*]:
 $\langle take\text{-}byte\ m\ w \rangle !! n \longleftrightarrow n < LENGTH('a) \wedge n < 8 \wedge w !! (n + m * 8) \rangle$ **for** w
 $:: \langle 'a::len\ word \rangle$
 $\langle proof \rangle$

lemma *bit-overwrite-iff* [*bit-simps*]:
 $\langle overwrite\ l\ h\ w0\ w1 \rangle !! n \longleftrightarrow n < LENGTH('a) \wedge$
 $(if\ l \leq n \wedge n < h\ then\ w1\ else\ w0) !! n \rangle$
for $w0\ w1 :: \langle 'a::len\ word \rangle$
 $\langle proof \rangle$

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

lemma *nth-takebyte*:
fixes $w :: 'a::len\ word$
shows $take\text{-}byte\ (n\ div\ 8)\ w !! (n\ mod\ 8) = (if\ n\ mod\ 8 < LENGTH('a)\ then$
 $w !! n\ else\ False)$
 $\langle proof \rangle$

lemma *nth-take-byte-overwrite*:
fixes $v\ v' :: 'a::len\ word$
shows $take\text{-}byte\ n\ (overwrite\ l\ h\ v\ v') !! i = (if\ i + n * 8 < l \vee i + n * 8 \geq h$
 $then\ take\text{-}byte\ n\ v !! i\ else\ take\text{-}byte\ n\ v' !! i)$
 $\langle proof \rangle$

lemma *nth-bitNOT*:
fixes $a :: 'a::len\ word$
shows $(NOT\ a) !! n \longleftrightarrow (if\ n < LENGTH('a)\ then\ \neg(a !! n)\ else\ False)$
 $\langle proof \rangle$

Various simplification rules

lemma *ucast-take-bits*:
fixes $w :: 'a::len\ word$
assumes $h = LENGTH('b)$
and $LENGTH('b) \leq LENGTH('a)$
shows $ucast\ (\langle 0, h \rangle w) = (ucast\ w :: 'b :: len\ word)$
 $\langle proof \rangle$

lemma *take-bits-ucast*:
fixes $w :: 'b::len\ word$
assumes $h = LENGTH('b)$

shows $\langle 0, h \rangle (\text{ucast } w :: 'a :: \text{len word}) = (\text{ucast } w :: 'a :: \text{len word})$
 $\langle \text{proof} \rangle$

lemma *take-bits-take-bits*:

fixes $w :: 'a :: \text{len word}$
shows $\langle l, h \rangle (\langle l', h' \rangle w) = (\text{if } \min \text{LENGTH}('a) \ h \geq h' - l' \text{ then } \langle l+l', h' \rangle w \text{ else } \langle l+l', l'+\min \text{LENGTH}('a) \ h \rangle w)$
 $\langle \text{proof} \rangle$

lemma *take-bits-overwrite*:

shows $\langle l, h \rangle (\text{overwrite } l \ h \ w0 \ w1) = \langle l, h \rangle w1$
 $\langle \text{proof} \rangle$

lemma *overwrite-0-take-bits-0*:

shows $\text{overwrite } 0 \ h \ (\langle 0, h \rangle w0) \ w1 = \langle 0, h \rangle w1$
 $\langle \text{proof} \rangle$

lemma *take-byte-shiftlr-256*:

fixes $v :: 256 \text{ word}$
assumes $m \leq n$
shows $\text{take-byte } n \ (v \ll m * 8) = (\text{if } (n+1) * 8 \leq 256 \text{ then } \text{take-byte } (n-m) \ v \text{ else } 0)$
 $\langle \text{proof} \rangle$

1.2 Take_Bits and arithmetic

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

definition *bitwise-add* :: $(\text{bool} \times \text{bool}) \ \text{list} \Rightarrow \text{bool} \Rightarrow \text{bool list}$

where $\text{bitwise-add } x \ c \equiv \text{foldr } (\lambda(x, y) \ \text{res } \text{car. } \text{xor3 } x \ y \ \text{car} \ \# \ \text{res } (\text{carry } x \ y \ \text{car})) \ x \ (\lambda-. \ \square) \ c$

lemma *length-foldr-bitwise-add*:

shows $\text{length } (\text{bitwise-add } x \ c) = \text{length } x$
 $\langle \text{proof} \rangle$

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 $x = \square \vee \neg \text{carry } (\text{fst } (\text{last } x)) \ (\text{snd } (\text{last } x)) \ \text{True}$
shows $\text{bitwise-add } (x \ @ \ y) \ (x \neq \square \wedge c) = \text{bitwise-add } x \ (x \neq \square \wedge c) \ @ \ \text{bitwise-add } y \ \text{False}$
 $\langle \text{proof} \rangle$

lemma *bitwise-add-take-append*:

shows $\text{take } (\text{length } x) \ (\text{bitwise-add } (x \ @ \ y) \ c) = \text{bitwise-add } x \ c$

<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-take-bits overwrite-0-take-bits-0 mask-eq-exp-minus-1
  ucast-down-ucast-id is-down take-bits-ucast ucast-up-ucast-id is-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 $64 \text{ word} \Rightarrow 8 \text{ word}$. Given an address $a::64 \text{ word}$ 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-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  $\geq$  2LENGTH('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 < 2LENGTH('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
<proof>

lemma *notin-region-addresses*:

assumes x \notin region-addresses a si

shows unat x < unat a \vee unat a + si \leq unat x

<proof>

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')

<proof>

lemma *region-addresses-eq-empty-iff*: region-addresses a si = {} \longleftrightarrow si = 0

<proof>

lemma *length-read-bytes*:

shows length (read-bytes m a si) = si

<proof>

lemma *nth-read-bytes*:

assumes n < si

shows read-bytes m a si ! n = m (a + of-nat (si - 1 - n))

<proof>

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'

<proof>

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 $\text{unat } \text{offset} + \text{si}' \leq \text{si}$
and $\text{si} \leq 2^{\text{LENGTH}('a)}$
shows $\text{read-bytes } (\text{override-on } m \ m' \ (\text{region-addresses } a \ \text{si})) \ (\text{offset}+a) \ \text{si}' =$
 $\text{read-bytes } m' \ (\text{offset}+a) \ \text{si}'$
 $\langle \text{proof} \rangle$

lemma *read-bytes-override-on-separate*:

assumes *separate* $a \ \text{si} \ a' \ \text{si}'$
shows $\text{read-bytes } (\text{override-on } m \ m' \ (\text{region-addresses } a \ \text{si})) \ a' \ \text{si}' = \text{read-bytes}$
 $m \ a' \ \text{si}'$
 $\langle \text{proof} \rangle$

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 $w :: 'b::\text{len } \text{word}$
assumes $\text{LENGTH}('b) \leq 2^{\text{LENGTH}('a)}$
and $\text{unat } \text{offset} + \text{si} \leq 2^{\text{LENGTH}('a)}$
shows $\text{word-rcat } (\text{read-bytes } (\lambda a'. \text{take-byte } (\text{unat } (a' - a)) \ w) \ (a + \text{offset}) \ \text{si})$
 $= \langle \text{unat } \text{offset} * 8, (\text{unat } \text{offset} + \text{si}) * 8 \rangle w$
 $\langle \text{proof} \rangle$

lemmas $\text{word-rcat-read-bytes} = \text{word-rcat-read-bytes-enclosed}[\mathbf{where} \ \text{offset}=0, \text{simplified}]$

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

lemma *enclosed-spec*:

assumes *enclosed*: $\text{enclosed } a' \ \text{si}' \ a \ \text{si}$
and $x\text{-in}: x \in \text{region-addresses } a' \ \text{si}'$
shows $x \in \text{region-addresses } a \ \text{si}$
 $\langle \text{proof} \rangle$

lemma *address-in-enclosed-region-as-linarith*:

assumes *enclosed* $a' \ \text{si}' \ a \ \text{si}$
and $x \in \text{region-addresses } a' \ \text{si}'$
shows $a \leq x \wedge a' \leq x \wedge x < a' + \text{of-nat } \text{si}' \wedge x < a + \text{of-nat } \text{si}$
 $\langle \text{proof} \rangle$

lemma *address-of-enclosed-region-ge*:

assumes *enclosed* $a' \ \text{si}' \ a \ \text{si}$
shows $a' \geq a$
 $\langle \text{proof} \rangle$

lemma *address-in-enclosed-region*:

assumes *enclosed* a' si' a si
and $x \in \text{region-addresses } a' si'$
shows $\text{unat } (x - a) \geq \text{unat } (a' - a) \wedge \text{unat } (a' - a) + si' > \text{unat } (x - a) \wedge$
 $\text{unat } (x - a) < si$
 $\langle \text{proof} \rangle$

lemma *enclosed-minus-minus*:
fixes $a :: 'a \text{ word}$
assumes $\text{offset} \geq \text{offset}'$
and $\text{unat } \text{offset} - si \leq \text{unat } \text{offset}' - si'$
and $\text{unat } \text{offset}' \geq si'$
and $\text{unat } \text{offset} \geq si$
and $a \geq \text{offset}$
shows *enclosed* $(a - \text{offset}')$ si' $(a - \text{offset})$ si
 $\langle \text{proof} \rangle$

lemma *enclosed-plus*:
fixes $a :: 'a \text{ word}$
assumes $si' < si$
and $\text{unat } a + si < 2^{\text{LENGTH } ('a)}$
shows *enclosed* a si' a si
 $\langle \text{proof} \rangle$

lemma *separate-symm*: *separate* a si a' si' = *separate* a' si' a si
 $\langle \text{proof} \rangle$

lemma *separate-iff*: *separate* a si a' si' $\longleftrightarrow si > 0 \wedge si' > 0 \wedge \text{unat } (a' - a) \geq$
 $si \wedge \text{unat } (a - a') \geq si'$
 $\langle \text{proof} \rangle$

lemma *separate-as-linarith*:
assumes $\neg \text{region-overflow } a$ si
and $\neg \text{region-overflow } a'$ si'
shows *separate* a si a' si' $\longleftrightarrow 0 < si \wedge 0 < si' \wedge (a + \text{of-nat } si \leq a' \vee a' +$
 $\text{of-nat } si' \leq a)$
(is $?lhs \longleftrightarrow ?rhs)$
 $\langle \text{proof} \rangle$

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

lemmas *separate-as-linarith-numeral* [*simp*] =
separate-as-linarith [*of numeral* $a :: 'a \text{ word numeral } si \text{ numeral } a' :: 'a \text{ word numeral } si'$]
for a si a' si'
lemmas *separate-as-linarith-numeral-1* [*simp*] =
separate-as-linarith [*of numeral* $a :: 'a \text{ word numeral } si \text{ numeral } a' :: 'a \text{ word } \text{Suc } 0$]
for a si a'
lemmas *separate-as-linarith-numeral1-* [*simp*] =
separate-as-linarith [*of numeral* $a :: 'a \text{ word } \text{Suc } 0 \text{ numeral } a' :: 'a \text{ word numeral } si'$]
for a a' si'
lemmas *separate-as-linarith-numeral11* [*simp*] =

separate-as-linearith [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 $si' \leq unat\ offset$
and $0 < si$
and $0 < si'$
and $unat\ offset + si \leq 2^{\wedge}LENGTH('a)$
shows *separate* ($offset + a$) $si a si'$
<proof>

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

lemma *separate-minus-minus'*:
assumes $si \neq 0$
and $si' \neq 0$
and $unat\ offset \geq si$
and $unat\ offset' \geq si'$
and $unat\ offset - si \geq unat\ offset'$
shows *separate* ($a - offset$) $si (a - offset')$ si'
<proof>

lemma *separate-minus-minus*:
assumes $si \neq 0$
and $si' \neq 0$
and $unat\ offset \geq si$
and $unat\ offset' \geq si'$
and $unat\ offset - si \geq unat\ offset' \vee unat\ offset' - si' \geq unat\ offset$
shows *separate* ($a - offset$) $si (a - offset')$ si'
<proof>

lemma *separate-minus-none*:
assumes $si \neq 0$
and $si' \neq 0$
and $unat\ offset \geq si$
and $si' \leq 2^{\wedge}LENGTH('a) - unat\ offset$
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  $\Rightarrow$  256word
  mem  :: 64 word  $\Rightarrow$  8 word
  flags :: string  $\Rightarrow$  bool
  rip  :: 64 word
```

definition *real-reg* :: string \Rightarrow bool \times string \times nat \times nat

where *real-reg* reg \equiv
— TODO: xmm, ymm, etc.

case reg of

— rip

 "rip" \Rightarrow (True, "rip", 0,64)

— rax,rbx,rcx,rdx

| "rax" \Rightarrow (True, "rax", 0,64)

| "eax" \Rightarrow (True, "rax", 0,32)

| "ax" \Rightarrow (False, "rax", 0,16)

| "ah" \Rightarrow (False, "rax", 8,16)

| "al" \Rightarrow (False, "rax", 0,8)

| "rbx" \Rightarrow (True, "rbx", 0,64)

| "ebx" \Rightarrow (True, "rbx", 0,32)

| "bx" \Rightarrow (False, "rbx", 0,16)

| "bh" \Rightarrow (False, "rbx", 8,16)

| "bl" \Rightarrow (False, "rbx", 0,8)

| "rcx" \Rightarrow (True, "rcx", 0,64)

| "ecx" \Rightarrow (True, "rcx", 0,32)

| "cx" \Rightarrow (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* ⇒ *nat* — in bytes

```
where reg-size reg ≡ let (-, -, l, h) = real-reg reg in (h - l) div 8
```

We now define functions for reading and writing from state.

definition *reg-read* :: *state* ⇒ *string* ⇒ 256 word

```
where reg-read σ reg ≡
```

```
  if reg = "rip" then ucast (rip σ) else
```

```
  if reg = "" then 0 else — happens if no base register is used in an address
```

```
  let (-, r, l, h) = real-reg reg in
```

```
    ⟨l, h⟩(regs σ r)
```

primrec *fromBool* :: *bool* \Rightarrow 'a :: *len word*

where

fromBool True = 1
| *fromBool False* = 0

definition *flag-read* :: *state* \Rightarrow *string* \Rightarrow *256 word*

where *flag-read* σ *flag* \equiv *fromBool (flags σ flag)*

definition *mem-read* :: *state* \Rightarrow *64 word* \Rightarrow *nat* \Rightarrow *256 word*

where *mem-read* σ *a si* \equiv *word-rcat (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 \Rightarrow 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) = $(\lambda \sigma . \sigma(|regs := (regs \sigma)(reg := val)|))$
| *state-update (FlagUpdate val)* = $(\lambda \sigma . \sigma(|flags := val|))$
| *state-update (RipUpdate a)* = $(\lambda \sigma . \sigma(|rip := a|))$
| *state-update (MemUpdate a si val)* = $(\lambda \sigma .$
 $let new = (\lambda a' . take-byte (unat (a' - a)) val) in$
 $\sigma(|mem := override-on (mem \sigma) new (region-addresses a si)|))$

abbreviation *RegUpdateSyntax* (*- :=_r - 30*)

where *RegUpdateSyntax reg val* \equiv *RegUpdate reg val*

abbreviation *MemUpdateSyntax* (*[[-,-]] :=_m - 30*)

where *MemUpdateSyntax a si val* \equiv *MemUpdate a si val*

abbreviation *FlagUpdateSyntax* (*setFlags*)

where *FlagUpdateSyntax val* \equiv *FlagUpdate val*

abbreviation *RipUpdateSyntax* (*setRip*)

where *RipUpdateSyntax val* \equiv *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* \equiv

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* ⇒ (*64 word* × *string* × *string* × *nat*) (*[-]1 40*)
where *mem-op-no-offset-no-index r* ≡ (*0,r,[],0*)

abbreviation *mem-op-no-index* :: *64 word* ⇒ *string* ⇒ (*64 word* × *string* × *string* × *nat*) (*[- + -]2 40*)
where *mem-op-no-index offset r* ≡ (*offset,r,[],0*)

abbreviation *mem-op* :: *64 word* ⇒ *string* ⇒ *string* ⇒ *nat* ⇒ (*64 word* × *string* × *string* × *nat*) (*[- + - + - * -]3 40*)
where *mem-op offset r index scale* ≡ (*offset,r,index,scale*)

definition *ymm-ptr* (*YMMWORD PTR -*)
where *YMMWORD PTR x* ≡ *case x of (offset,base,index,scale) ⇒ Mem 32 offset base index scale*

definition *xmm-ptr* (*XMMWORD PTR -*)
where *XMMWORD PTR x* ≡ *case x of (offset,base,index,scale) ⇒ Mem 16 offset base index scale*

definition *qword-ptr* (*QWORD PTR -*)
where *QWORD PTR x* ≡ *case x of (offset,base,index,scale) ⇒ Mem 8 offset base index scale*

definition *dword-ptr* (*DWORD PTR -*)
where *DWORD PTR x* ≡ *case x of (offset,base,index,scale) ⇒ Mem 4 offset base index scale*

definition *word-ptr* (*WORD PTR -*)
where *WORD PTR x* ≡ *case x of (offset,base,index,scale) ⇒ Mem 2 offset base index scale*

definition *byte-ptr* (*BYTE PTR -*)
where *BYTE PTR x* ≡ *case x of (offset,base,index,scale) ⇒ Mem 1 offset base index scale*

primrec (*nonexhaustive*) *operand-size* :: *Operand* ⇒ *nat* — in bytes
where

operand-size (Reg r) = reg-size r
 | *operand-size (Mem si - - -) = si*

fun *resolve-address* :: *state* ⇒ *64 word* ⇒ *char list* ⇒ *char list* ⇒ *nat* ⇒ *64 word*

where *resolve-address* σ *offset base index scale* =
 (let $i = \text{ucast } (\text{reg-read } \sigma \text{ index});$
 $b = \text{ucast } (\text{reg-read } \sigma \text{ base})$ in
 $\text{offset} + b + \text{of-nat } \text{scale} * i$)

primrec *operand-read* :: *state* \Rightarrow *Operand* \Rightarrow 256 word

where

operand-read σ (*Imm* i) = i
 | *operand-read* σ (*Reg* r) = *reg-read* σ r
 | *operand-read* σ (*Flag* f) = *flag-read* σ f
 | *operand-read* σ (*Mem* si *offset base index scale*) =
 (let $a = \text{resolve-address } \sigma \text{ offset base index scale}$ in
 $\text{mem-read } \sigma$ a si
)

primrec *state-with-updates* :: *state* \Rightarrow *StateUpdate list* \Rightarrow *state* (**infixl** with 66)

where

σ with [] = σ
 | (σ with ($f \# fs$)) = *state-update* f (σ with fs)

primrec (*nonexhaustive*) *operand-write* :: *Operand* \Rightarrow 256word \Rightarrow *state* \Rightarrow *state*

where

operand-write (*Reg* r) v σ = *reg-write* r v σ
 | *operand-write* (*Mem* si *offset base index scale*) v σ =
 (let $i = \text{ucast } (\text{reg-read } \sigma \text{ index});$
 $b = \text{ucast } (\text{reg-read } \sigma \text{ base});$
 $a = \text{offset} + b + \text{of-nat } \text{scale} * i$ in
 σ 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* (σ with ($(r :=_r w) \# \text{updates}$)) $r' =$ (if $r=r'$ then w else *regs* (σ with *updates*) r')
 <proof>

lemma *regs-mem-write*:

shows *regs* (σ with ($([a, si] :=_m v) \# \text{updates}$)) $r =$ *regs* (σ with *updates*) r
 <proof>

lemma *regs-flag-write*:

shows *regs* (σ with ($(\text{setFlags } v) \# \text{updates}$)) $r =$ *regs* (σ with *updates*) r
 <proof>

lemma *regs-rip-write*:

shows *regs* (σ with ($(\text{setRip } a) \# \text{updates}$)) $f =$ *regs* (σ with *updates*) f
 <proof>

lemma *flag-read-reg-write:*

shows $\text{flag-read } (\sigma \text{ with } ((r :=_r w) \# \text{updates})) f = \text{flag-read } (\sigma \text{ with updates}) f$
<proof>

lemma *flag-read-mem-write:*

shows $\text{flag-read } (\sigma \text{ with } ([[a, si]] :=_m v) \# \text{updates}) f = \text{flag-read } (\sigma \text{ with updates}) f$
<proof>

lemma *flag-read-flag-write:*

shows $\text{flag-read } (\sigma \text{ with } ((\text{setFlags } v) \# \text{updates})) = \text{fromBool } o v$
<proof>

lemma *flag-read-rip-write:*

shows $\text{flag-read } (\sigma \text{ with } ((\text{setRip } a) \# \text{updates})) f = \text{flag-read } (\sigma \text{ with updates}) f$
<proof>

lemma *mem-read-reg-write:*

shows $\text{mem-read } (\sigma \text{ with } ((r :=_r w) \# \text{updates})) a \text{ si} = \text{mem-read } (\sigma \text{ with updates}) a \text{ si}$
<proof>

lemma *mem-read-flag-write:*

shows $\text{mem-read } (\sigma \text{ with } ((\text{setFlags } v) \# \text{updates})) a \text{ si} = \text{mem-read } (\sigma \text{ with updates}) a \text{ si}$
<proof>

lemma *mem-read-rip-write:*

shows $\text{mem-read } (\sigma \text{ with } ((\text{setRip } a') \# \text{updates})) a \text{ si} = \text{mem-read } (\sigma \text{ with updates}) a \text{ si}$
<proof>

lemma *mem-read-mem-write-alias:*

assumes $si' \leq si$

and $si \leq 2^{64}$

shows $\text{mem-read } (\sigma \text{ with } ([[a, si]] :=_m v) \# \text{updates}) a \text{ si}' = \langle 0, si' * 8 \rangle v$
<proof>

lemma *mem-read-mem-write-separate:*

assumes *separate* $a \text{ si}' \text{ si}$

shows $\text{mem-read } (\sigma \text{ with } ([[a, si]] :=_m v) \# \text{updates}) a' \text{ si}' = \text{mem-read } (\sigma \text{ with updates}) a' \text{ si}'$
<proof>

lemma *mem-read-mem-write-enclosed-minus:*

assumes $\text{offset}' \leq \text{offset}$

and $si' \leq si$

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

lemma *mem-read-mem-write-enclosed-plus:*

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

lemma *mem-read-mem-write-enclosed-plus2:*

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

lemma *mem-read-mem-write-enclosed-numeral[simp]:*

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

lemma *mem-read-mem-write-enclosed-numeral1[simp]:*

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

lemma *mem-read-mem-write-enclosed-numeral-1[simp]:*

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

lemma *mem-read-mem-write-enclosed-numeral11[simp]:*

assumes $\text{unat } (\text{numeral } a' - \text{numeral } a::64 \text{ word}) + (\text{Suc } 0) \leq \text{Suc } 0$
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 *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 *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 *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 *rip-rip-write[simp]*:
shows $\text{rip } (\sigma \text{ with } ((\text{setRip } a) \# \text{updates})) = a$
 $\langle \text{proof} \rangle$

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

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

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
 $\text{insert-state-update } (\text{setRip } a) (\text{setRip } a' \# \text{updates}) = \text{insert-state-update } (\text{setRip } a) \text{ updates}$
 $\mid \text{insert-state-update } (\text{setRip } a) (\text{setFlags } v \# \text{updates}) = \text{setFlags } v \# (\text{insert-state-update } (\text{setRip } a) \text{ updates})$
 $\mid \text{insert-state-update } (\text{setRip } a) ((r :=_r v) \# \text{updates}) = (r :=_r v) \# (\text{insert-state-update } (\text{setRip } a) \text{ updates})$
 $\mid \text{insert-state-update } (\text{setRip } a) ((\llbracket a', si \rrbracket :=_m v) \# \text{updates}) = (\llbracket a', si \rrbracket :=_m v) \# (\text{insert-state-update } (\text{setRip } a) \text{ updates})$
 $\mid \text{insert-state-update } (\text{setFlags } v) (\text{setFlags } v' \# \text{updates}) = \text{insert-state-update } (\text{setFlags } v) \text{ updates}$

| *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-plus

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 \Rightarrow state*

A datatype for the result of floating point comparisons.

datatype $FP\text{-}Order = FP\text{-}Unordered \mid FP\text{-}GT \mid FP\text{-}LT \mid FP\text{-}EQ$

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

locale *unknowns* =

fixes *unknown-addsd* :: $64 \text{ word} \Rightarrow 64 \text{ word} \Rightarrow 64 \text{ word}$
and *unknown-subsd* :: $64 \text{ word} \Rightarrow 64 \text{ word} \Rightarrow 64 \text{ word}$
and *unknown-mulsd* :: $64 \text{ word} \Rightarrow 64 \text{ word} \Rightarrow 64 \text{ word}$
and *unknown-divsd* :: $64 \text{ word} \Rightarrow 64 \text{ word} \Rightarrow 64 \text{ word}$
and *unknown-ucomisd* :: $64 \text{ word} \Rightarrow 64 \text{ word} \Rightarrow FP\text{-}Order$
and *unknown-semantics* :: $I \Rightarrow state \Rightarrow state$
and *unknown-flags* :: $string \Rightarrow string \Rightarrow bool$

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 *f*. 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 order 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* \Rightarrow *Operand* \Rightarrow *state* \Rightarrow *state*
where *semantics-MOV* *op1 op2* $\sigma \equiv$
 let *src* = *operand-read* σ *op2* *in*
 operand-write *op1* *src* σ

abbreviation *MOV*
where *MOV* *op1 op2* \equiv *Instr* "mov" (*Some op1*) (*Some op2*) *None*

abbreviation *MOVABS*
where *MOVABS* *op1 op2* \equiv *Instr* "movabs" (*Some op1*) (*Some op2*) *None*

abbreviation *MOVAPS*
where *MOVAPS* *op1 op2* \equiv *Instr* "movaps" (*Some op1*) (*Some op2*) *None*

abbreviation *MOVZX*
where *MOVZX* *op1 op2* \equiv *Instr* "movzx" (*Some op1*) (*Some op2*) *None*

abbreviation *MOVDQU*
where *MOVDQU* *op1 op2* \equiv *Instr* "movdqu" (*Some op1*) (*Some op2*) *None*

definition *semantics-MOVD* :: *Operand* \Rightarrow *Operand* \Rightarrow *state* \Rightarrow *state*
where *semantics-MOVD* *op1 op2* $\sigma \equiv$
 let *src* = *ucast*(*operand-read* σ *op2*::32word) *in*
 operand-write *op1* (*ucast* *src*) σ

abbreviation *MOVD*
where *MOVD* *op1 op2* \equiv *Instr* "movd" (*Some op1*) (*Some op2*) *None*

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

definition *semantics-MOVSD* :: *Operand* \Rightarrow *Operand* \Rightarrow *state* \Rightarrow *state*
where *semantics-MOVSD* *op1 op2* $\sigma \equiv$
 if *isXMM* *op1* \wedge *isXMM* *op2* *then*
 let *src* = $\langle 0, 64 \rangle$ *operand-read* σ *op2*;
 dst = $\langle 64, 128 \rangle$ *operand-read* σ *op1* *in*
 operand-write *op1* (*overwrite* 0 64 *dst* *src*) σ
 else
 let *src* = $\langle 0, 64 \rangle$ *operand-read* σ *op2* *in*
 operand-write *op1* *src* σ

abbreviation *MOVSD*
where *MOVSD* *op1 op2* \equiv *Instr* "movsd" (*Some op1*) (*Some op2*) *None*

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

definition *semantics-LEA* :: *Operand* \Rightarrow *Operand* \Rightarrow *state* \Rightarrow *state*
where *semantics-LEA* *op1 op2* $\sigma \equiv$
case op2 of Mem si offset base index scale \Rightarrow
operand-write op1 (ucast (resolve-address σ offset base index scale)) σ

abbreviation *LEA*

where *LEA op1 op2* \equiv *Instr "lea" (Some op1) (Some op2) None*

definition *semantics-PUSH* :: *Operand* \Rightarrow *state* \Rightarrow *state*

where *semantics-PUSH* *op1* $\sigma \equiv$
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* \equiv *Instr "push" (Some op1) None None*

definition *semantics-POP* :: *Operand* \Rightarrow *state* \Rightarrow *state*

where *semantics-POP* *op1* $\sigma \equiv$
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* \equiv *Instr "pop" (Some op1) None None*

definition *semantics-CALL* :: *Operand* \Rightarrow *state* \Rightarrow *state*

where *semantics-CALL* *op1* $\sigma \equiv$
let src = ucast (operand-read σ op1) in
(state-update (setRip src) o semantics-PUSH (Reg "rip")) σ

definition *semantics-RET* :: *state* \Rightarrow *state*

where *semantics-RET* $\sigma \equiv$
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* \equiv *Instr "ret" None None None*

definition *semantics-LEAVE* :: *state* \Rightarrow *state*

where *semantics-LEAVE* \equiv *semantics-POP (Reg "rbp") o semantics-MOV (Reg*
"rsp") (Reg "rbp")

abbreviation *LEAVE*

where *LEAVE op1* \equiv *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 $ADD-flags\ w0\ w1\ flag \equiv case\ flag\ of$

$"zf" \Rightarrow w0 + w1 = 0$
 $| "cf" \Rightarrow \text{unat } w0 + \text{unat } w1 \geq 2^{\wedge}(\text{LENGTH}'a)$
 $| "of" \Rightarrow (w0 <_s 0 \longleftrightarrow w1 <_s 0) \wedge \neg(w0 <_s 0 \longleftrightarrow w0+w1 <_s 0)$
 $| "sf" \Rightarrow w0 + w1 <_s 0$
 $| f \Rightarrow \text{unknown-flags "ADD" } f$

definition *semantics-ADD* :: *Operand* \Rightarrow *Operand* \Rightarrow *state* \Rightarrow *state*
where *semantics-ADD* *op1* \equiv
 $\text{if operand-size } op1 = 32 \text{ then binop } ((+)::256 \text{ word } \Rightarrow - \Rightarrow -) \text{ ADD-flags}$
 $op1$
 $\text{else if operand-size } op1 = 16 \text{ then binop } ((+)::128 \text{ word } \Rightarrow - \Rightarrow -) \text{ ADD-flags}$
 $op1$
 $\text{else if operand-size } op1 = 8 \text{ then binop } ((+)::64 \text{ word } \Rightarrow - \Rightarrow -) \text{ ADD-flags}$
 $op1$
 $\text{else if operand-size } op1 = 4 \text{ then binop } ((+)::32 \text{ word } \Rightarrow - \Rightarrow -) \text{ ADD-flags}$
 $op1$
 $\text{else if operand-size } op1 = 2 \text{ then binop } ((+)::16 \text{ word } \Rightarrow - \Rightarrow -) \text{ ADD-flags}$
 $op1$
 $\text{else if operand-size } op1 = 1 \text{ then binop } ((+)::8 \text{ word } \Rightarrow - \Rightarrow -) \text{ ADD-flags}$
 $op1$
 else undefined

abbreviation *ADD*
where *ADD* *op1 op2* \equiv *Instr* "add" (*Some op1*) (*Some op2*) *None*

definition *INC-flags* :: *256 word* \Rightarrow (*'a::len word* \Rightarrow *string* \Rightarrow *bool*)
where *INC-flags* *cf w0 flag* \equiv *case flag of*
 $"zf" \Rightarrow w0 + 1 = 0$
 $| "cf" \Rightarrow cf \neq 0$
 $| "of" \Rightarrow 0 <=_s w0 \wedge w0+1 <_s 0$
 $| "sf" \Rightarrow w0 + 1 <_s 0$
 $| f \Rightarrow \text{unknown-flags "INC" } f$

definition *semantics-INC* :: *Operand* \Rightarrow *state* \Rightarrow *state*
where *semantics-INC* *op1* $\sigma \equiv$
 $\text{let } cf = \text{flag-read } \sigma \text{ "cf" in}$
 $\text{if operand-size } op1 = 32 \text{ then unop } ((+) (1::256 \text{ word})) (\text{INC-flags } cf) \text{ op1}$
 σ
 $\text{else if operand-size } op1 = 16 \text{ then unop } ((+) (1::128 \text{ word})) (\text{INC-flags } cf)$
 $op1 \sigma$
 $\text{else if operand-size } op1 = 8 \text{ then unop } ((+) (1::64 \text{ word})) (\text{INC-flags } cf) \text{ op1}$
 σ
 $\text{else if operand-size } op1 = 4 \text{ then unop } ((+) (1::32 \text{ word})) (\text{INC-flags } cf) \text{ op1}$
 σ
 $\text{else if operand-size } op1 = 2 \text{ then unop } ((+) (1::16 \text{ word})) (\text{INC-flags } cf) \text{ op1}$
 σ
 $\text{else if operand-size } op1 = 1 \text{ then unop } ((+) (1::8 \text{ word})) (\text{INC-flags } cf) \text{ op1}$
 σ

else undefined

abbreviation *INC*

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

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

where *DEC-flags cf w0 flag* \equiv *case flag of*

"zf" \Rightarrow *w0 = 1*
| *"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 ($\lambda w . w - 1::256$ word) (DEC-flags
cf) op1 σ*
*else if operand-size op1 = 16 then unop ($\lambda w . w - 1::128$ word) (DEC-flags
cf) op1 σ*
*else if operand-size op1 = 8 then unop ($\lambda w . w - 1::64$ word) (DEC-flags
cf) op1 σ*
*else if operand-size op1 = 4 then unop ($\lambda w . w - 1::32$ word) (DEC-flags
cf) op1 σ*
*else if operand-size op1 = 2 then unop ($\lambda w . w - 1::16$ word) (DEC-flags
cf) op1 σ*
*else if operand-size op1 = 1 then unop ($\lambda w . w - 1::8$ word) (DEC-flags
cf) op1 σ*
else undefined

abbreviation *DEC*

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

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

where *NEG-flags w0 flag* \equiv *case flag of*

"zf" \Rightarrow *w0 = 0*
| *"cf"* \Rightarrow *w0 \neq 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 ($\lambda w0 . - (w0::256$ word)) NEG-flags
op1 σ*
*else if operand-size op1 = 16 then unop ($\lambda w0 . - (w0::128$ word)) NEG-flags
op1 σ*
else if operand-size op1 = 8 then unop ($\lambda w0 . - (w0::64$ word)) NEG-flags

op1 σ
 else if operand-size *op1* = 4 then unop ($\lambda w0 . - (w0::32 \text{ word})$) NEG-flags
op1 σ
 else if operand-size *op1* = 2 then unop ($\lambda w0 . - (w0::16 \text{ word})$) NEG-flags
op1 σ
 else if operand-size *op1* = 1 then unop ($\lambda w0 . - (w0::8 \text{ word})$) NEG-flags
op1 σ
 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" $\Rightarrow (\text{msb } w0 \neq \text{msb } w1) \wedge (\text{msb } (w0 - w1) = \text{msb } w1)$
 | *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 ⇒ unknown-flags "SBB" f

definition semantics-SBB :: Operand ⇒ Operand ⇒ state ⇒ state
where semantics-SBB op1 op2 σ ≡
 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 ≡ Instr "sbb" (Some op1) (Some op2) None

definition adc :: 'b::len word ⇒ 'a::len word ⇒ 'a word ⇒ 'a word
where adc cf dst src ≡ dst + (src + ucast cf)

definition ADC-flags :: 'b::len word ⇒ 'a::len word ⇒ 'a::len word ⇒ string ⇒ bool

where ADC-flags cf dst src flag ≡ case flag of
 "zf" ⇒ adc cf dst src = 0
 | "cf" ⇒ unat dst + unat src + unat cf ≥ 2^{LENGTH('a)}
 | "of" ⇒ (dst < s 0 ↔ src + ucast cf < s 0) ∧ ¬(dst < s 0 ↔ adc cf dst src
 < s 0)
 | "sf" ⇒ adc cf dst src < s 0
 | f ⇒ unknown-flags "ADC" f

definition semantics-ADC :: Operand ⇒ Operand ⇒ state ⇒ state

where semantics-ADC op1 op2 σ ≡
 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
 cf) op1 op2 σ
 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 (\langle si,2*si \rangle result))
 (operand-write (Reg rl) (ucast (\langle 0,si \rangle result))
 (σ with [setFlags flgs]))

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

where MUL-flags result flag \equiv case flag of
 "cf" \Rightarrow (\langle LENGTH('a) div 2,LENGTH('a) \rangle result) \neq 0
 | "of" \Rightarrow (\langle LENGTH('a) div 2,LENGTH('a) \rangle 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 (\langle LENGTH('a) div 2,LENGTH('a) \rangle result) \neq (if result !! (LENGTH('a)
 div 2 - 1) then 2 \wedge (LENGTH('a) div 2)-1 else 0)
 | "of" \Rightarrow (\langle LENGTH('a) div 2,LENGTH('a) \rangle result) \neq (if result !! (LENGTH('a)
 div 2 - 1) then 2 \wedge (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 σ (Reg op1-reg))::'a::len word;
 src = cast (operand-read σ 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 σ
 else if LENGTH('a) = 32 then
 write-MUL-result "dx" op1-reg prod flgs σ
 else if LENGTH('a) = 64 then
 write-MUL-result "edx" op1-reg prod flgs σ
 else if LENGTH('a) = 128 then

write-MUL-result "rdx" op1-reg prod flgs σ
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 *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 $n\ dst\ flag \equiv case\ flag\ of$
 "cf" $\Rightarrow dst\ !!\ (LENGTH('a) - n)$
 | "of" $\Rightarrow dst\ !!\ (LENGTH('a) - n - 1) \neq dst\ !!\ (LENGTH('a) - n)$
 | "zf" $\Rightarrow (dst << n) = 0$
 | "sf" $\Rightarrow dst\ !!\ (LENGTH('a) - n - 1)$
 | $f \Rightarrow unknown\ flags\ "SHL"\ f$

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 $n\ dst\ flag \equiv case\ flag\ of$
 "cf" $\Rightarrow dst\ !!\ (n - 1)$
 | "of" $\Rightarrow msb\ dst$
 | "zf" $\Rightarrow (dst >> n) = 0$
 | $f \Rightarrow unknown\ flags\ "SHR"\ f$

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

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

let *src* = *unat (operand-read σ op2) in*
 if *operand-size op1 = 32* then *unop (λ w . w >> src::256 word) (SHR-flags*
src) op1 σ
 else if *operand-size op1 = 16* then *unop (λ w . w >> src::128 word) (SHR-flags*
src) op1 σ
 else if *operand-size op1 = 8* then *unop (λ w . w >> src::64 word) (SHR-flags*
src) op1 σ
 else if *operand-size op1 = 4* then *unop (λ w . w >> src::32 word) (SHR-flags*
src) op1 σ
 else if *operand-size op1 = 2* then *unop (λ w . w >> src::16 word) (SHR-flags*
src) op1 σ
 else if *operand-size op1 = 1* then *unop (λ w . w >> src::8 word) (SHR-flags*
src) op1 σ
 else *undefined*

abbreviation *SHR*

where *SHR op1 op2* ≡ *Instr "shr" (Some op1) (Some op2) None*

definition *SAR-flags* :: *nat* ⇒ (*'a::len word* ⇒ *string* ⇒ *bool*)

where *SAR-flags n dst flag* ≡ *case flag of*

"cf" ⇒ dst !! (n - 1)
 | *"of" ⇒ False*
 | *"zf" ⇒ (dst >>> n) = 0*
 | *f ⇒ unknown-flags "SAR" f*

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

where *semantics-SAR op1 op2 σ* ≡

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

abbreviation *SAR*

where *SAR op1 op2* ≡ *Instr "sar" (Some op1) (Some op2) None*

definition *shld* :: *'b::len itself* ⇒ *nat* ⇒ *'a::len word* ⇒ *'a word* ⇒ *'a word*

where *shld - n dst src* ≡


```

let dstsrc = (ucast dst << LENGTH('a)) OR (ucast src :: 'b word);
  shifted = (LENGTH('a),LENGTH('a)*2)(dstsrc << n) in
ucast shifted

```

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

```

where SHLD-flags b n src dst flag ≡ case flag of
  "cf" ⇒ dst !! (LENGTH('a) - n)
| "of" ⇒ dst !! (LENGTH('a) - n - 1) ≠ dst !! (LENGTH('a) - n)
| "zf" ⇒ shld b n dst src = 0
| "sf" ⇒ dst !! (LENGTH('a) - n - 1) — msb (shld n dst src)
| f    ⇒ unknown-flags "SHLD" f

```

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

```

where semantics-SHLD op1 op2 op3 σ ≡
  let src2 = unat (operand-read σ op3) in
    if operand-size op1 = 32 then binop (shld (TYPE(512)) src2 ::256 word
⇒ - ⇒ -) (SHLD-flags (TYPE(512)) src2) op1 op2 σ
  else if operand-size op1 = 16 then binop (shld (TYPE(256)) src2 ::128 word
⇒ - ⇒ -) (SHLD-flags (TYPE(256)) src2) op1 op2 σ
  else if operand-size op1 = 8 then binop (shld (TYPE(128)) src2 ::64 word
⇒ - ⇒ -) (SHLD-flags (TYPE(128)) src2) op1 op2 σ
  else if operand-size op1 = 4 then binop (shld (TYPE(64)) src2 ::32 word ⇒
- ⇒ -) (SHLD-flags (TYPE(64)) src2) op1 op2 σ
  else if operand-size op1 = 2 then binop (shld (TYPE(32)) src2 ::16 word
⇒ - ⇒ -) (SHLD-flags (TYPE(32)) src2) op1 op2 σ
  else if operand-size op1 = 1 then binop (shld (TYPE(16)) src2 ::8 word ⇒
- ⇒ -) (SHLD-flags (TYPE(16)) src2) op1 op2 σ
  else undefined

```

definition *ROL-flags* :: nat ⇒ ('a::len word ⇒ string ⇒ bool)

```

where ROL-flags n dst flag ≡ case flag of
  "cf" ⇒ dst !! (LENGTH('a) - n)
| "of" ⇒ dst !! (LENGTH('a) - n - 1) ≠ dst !! (LENGTH('a) - n)
| f    ⇒ unknown-flags "ROL" f

```

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

```

where semantics-ROL op1 op2 σ ≡
  let src = unat (operand-read σ op2) in
    if operand-size op1 = 32 then unop (word-rotl src::256 word⇒-) (ROL-flags
src) op1 σ
  else if operand-size op1 = 16 then unop (word-rotl src::128 word⇒-) (ROL-flags
src) op1 σ
  else if operand-size op1 = 8 then unop (word-rotl src::64 word⇒-) (ROL-flags
src) op1 σ
  else if operand-size op1 = 4 then unop (word-rotl src::32 word⇒-) (ROL-flags
src) op1 σ

```

else if operand-size op1 = 2 then unop (word-rotl src::16 word⇒-) (ROL-flags
 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 ⇒ - ⇒ -

\Rightarrow -) *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 *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 MOVSSD

where MOVSSD op1 op2 \equiv Instr "movssd" (Some op1) (Some op2) None

abbreviation MOVSSX

where MOVSSX op1 op2 \equiv Instr "movssx" (Some op1) (Some op2) None

definition semantics-CDQE :: state \Rightarrow state

where semantics-CDQE \equiv semantics-MOVSSD (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 σ (Reg "eax")) :: 32 word in
 operand-write (Reg "edx") (ucast ((32,64)(scast src::64 word))) σ

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 σ (Reg "rax")) :: 64 word in
 operand-write (Reg "rdx") (ucast ((64,128)(scast src::128 word))) σ

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

$((AND)::16 \text{ word} \Rightarrow - \Rightarrow -)$ $op1 \ op2 \ \sigma$
 else if operand-size $op1 = 1$ then binop $((AND)::8 \text{ word} \Rightarrow - \Rightarrow -)$ (logic-flags
 $((AND)::8 \text{ word} \Rightarrow - \Rightarrow -)$) $op1 \ op2 \ \sigma$
 else undefined

abbreviation AND'

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

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

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

abbreviation OR'

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

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

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

abbreviation XOR'

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

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

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

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

abbreviation XORPS

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

definition semantics-NOT :: Operand \Rightarrow state \Rightarrow state

where semantics-NOT *op1* $\sigma \equiv$

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

abbreviation NOT'

where NOT' *op1* \equiv 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 \Rightarrow state \Rightarrow bool

where

readFlagExpr (Flag *f*) $\sigma =$ (flag-read σ *f* = 1)
 | readFlagExpr (FE-NOT *fe*) $\sigma =$ (\neg readFlagExpr *fe* σ)
 | readFlagExpr (FE-AND *fe0 fe1*) $\sigma =$ (readFlagExpr *fe0* $\sigma \wedge$ readFlagExpr *fe1* σ)
 | readFlagExpr (FE-OR *fe0 fe1*) $\sigma =$ (readFlagExpr *fe0* $\sigma \vee$ readFlagExpr *fe1* σ)
 | readFlagExpr (FE-EQ *fe0 fe1*) $\sigma =$ (readFlagExpr *fe0* $\sigma \longleftrightarrow$ readFlagExpr *fe1* σ)

definition semantics-cond-jump :: FlagExpr \Rightarrow 64 word \Rightarrow state \Rightarrow state

where semantics-cond-jump *fe a* $\sigma \equiv$

let *fv* = readFlagExpr *fe* σ in

if *fv* then state-update (setRip *a*) σ else σ

definition semantics-JMP :: Operand \Rightarrow state \Rightarrow state

where *semantics-JMP* *op1* $\sigma \equiv$
 let *a* = *ucast* (*operand-read* σ *op1*) *in*
 state-update (*setRip* *a*) σ

abbreviation *JMP*

where *JMP* *op1* \equiv *Instr* "jmp" (*Some* *op1*) *None* *None*

definition *semantics-JO* :: *Operand* \Rightarrow *state* \Rightarrow *state*

where *semantics-JO* *op1* $\sigma \equiv$
 let *a* = *ucast* (*operand-read* σ *op1*) *in*
 semantics-cond-jump (*Flag* "of") *a* σ

abbreviation *JO*

where *JO* *op1* \equiv *Instr* "jo" (*Some* *op1*) *None* *None*

definition *semantics-JNO* :: *Operand* \Rightarrow *state* \Rightarrow *state*

where *semantics-JNO* *op1* $\sigma \equiv$
 let *a* = *ucast* (*operand-read* σ *op1*) *in*
 semantics-cond-jump (*FE-NOT* (*Flag* "of")) *a* σ

abbreviation *JNO*

where *JNO* *op1* \equiv *Instr* "jno" (*Some* *op1*) *None* *None*

definition *semantics-JS* :: *Operand* \Rightarrow *state* \Rightarrow *state*

where *semantics-JS* *op1* $\sigma \equiv$
 let *a* = *ucast* (*operand-read* σ *op1*) *in*
 semantics-cond-jump (*Flag* "sf") *a* σ

abbreviation *JS*

where *JS* *op1* \equiv *Instr* "js" (*Some* *op1*) *None* *None*

definition *semantics-JNS* :: *Operand* \Rightarrow *state* \Rightarrow *state*

where *semantics-JNS* *op1* $\sigma \equiv$
 let *a* = *ucast* (*operand-read* σ *op1*) *in*
 semantics-cond-jump (*FE-NOT* (*Flag* "sf")) *a* σ

abbreviation *JNS*

where *JNS* *op1* \equiv *Instr* "jns" (*Some* *op1*) *None* *None*

definition *semantics-JE* :: *Operand* \Rightarrow *state* \Rightarrow *state*

where *semantics-JE* *op1* $\sigma \equiv$
 let *a* = *ucast* (*operand-read* σ *op1*) *in*
 semantics-cond-jump (*Flag* "zf") *a* σ

abbreviation *JE*

where *JE* *op1* \equiv *Instr* "je" (*Some* *op1*) *None* *None*

abbreviation *JZ*

where *JZ* *op1* \equiv *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* σ *op1*) *in*
 semantics-cond-jump (*FE-NOT* (*Flag* "zf")) *a* σ

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* σ *op1*) *in*
 semantics-cond-jump (*Flag* "cf") *a* σ

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* σ *op1*) *in*
 semantics-cond-jump (*FE-NOT* (*Flag* "cf")) *a* σ

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* σ *op1*) *in*
 semantics-cond-jump (*FE-OR* (*Flag* "cf") (*Flag* "zf")) *a* σ

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* \equiv *Instr* "jg" (*Some* *op1*) *None* *None*

abbreviation *JNLE*

where *JNLE* *op1* \equiv *Instr* "jnle" (*Some* *op1*) *None* *None*

setXX

definition *semantics-setXX* :: *FlagExpr* \Rightarrow *Operand* \Rightarrow *state* \Rightarrow *state*

where *semantics-setXX* *fe* *op1* $\sigma \equiv$
 let *fv* = *readFlagExpr* *fe* σ *in*
 operand-write *op1* (*fromBool* *fv*) σ

abbreviation *semantics-SETO* :: *Operand* \Rightarrow *state* \Rightarrow *state*

where *semantics-SETO* \equiv *semantics-setXX* (*Flag* "of")

abbreviation *SETO*

where *SETO* *op1* \equiv *Instr* "seto" (*Some* *op1*) *None* *None*

abbreviation *semantics-SETNO* :: *Operand* \Rightarrow *state* \Rightarrow *state*

where *semantics-SETNO* \equiv *semantics-setXX* (*FE-NOT* (*Flag* "of"))

abbreviation *SETNO*

where *SETNO* *op1* \equiv *Instr* "setno" (*Some* *op1*) *None* *None*

abbreviation *semantics-SETS* :: *Operand* \Rightarrow *state* \Rightarrow *state*

where *semantics-SETS* \equiv *semantics-setXX* (*Flag* "sf")

abbreviation *SETS*

where *SETS* *op1* \equiv *Instr* "sets" (*Some* *op1*) *None* *None*

abbreviation *semantics-SETNS* :: *Operand* \Rightarrow *state* \Rightarrow *state*

where *semantics-SETNS* \equiv *semantics-setXX* (*FE-NOT* (*Flag* "sf"))

abbreviation *SETNS*

where *SETNS* *op1* \equiv *Instr* "setns" (*Some* *op1*) *None* *None*

abbreviation *semantics-SETE* :: *Operand* \Rightarrow *state* \Rightarrow *state*

where *semantics-SETE* \equiv *semantics-setXX* (*Flag* "zf")

abbreviation *SETE*

where *SETE* *op1* \equiv *Instr* "sete" (*Some* *op1*) *None* *None*

abbreviation *SETZ*

where *SETZ* *op1* \equiv *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 *cmov*

where

cmov True dst src = src
| *cmov False dst src = dst*

definition *semantics-CMOV* :: *FlagExpr* \Rightarrow *Operand* \Rightarrow *Operand* \Rightarrow *state* \Rightarrow *state*
where *semantics-CMOV* *fe op1 op2* $\sigma \equiv$
 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* \equiv *semantics-CMOV* (*FE-NOT* (*Flag* "zf"))

abbreviation *CMOVNE*
where *CMOVNE* *op1 op2* \equiv *Instr* "movne" (*Some* *op1*) (*Some* *op2*) *None*

abbreviation *semantics-CMOVNS* \equiv *semantics-CMOV* (*FE-NOT* (*Flag* "sf"))

abbreviation *CMOVNS*
where *CMOVNS* *op1 op2* \equiv *Instr* "movns" (*Some* *op1*) (*Some* *op2*) *None*

Floating Point

definition *semantics-ADDSD* :: *Operand* \Rightarrow *Operand* \Rightarrow *state* \Rightarrow *state*
where *semantics-ADDSD* \equiv *binop-XMM* *unknown-addsd*

definition *semantics-SUBSD* :: *Operand* \Rightarrow *Operand* \Rightarrow *state* \Rightarrow *state*
where *semantics-SUBSD* \equiv *binop-XMM* *unknown-subsd*

definition *semantics-MULSD* :: *Operand* \Rightarrow *Operand* \Rightarrow *state* \Rightarrow *state*
where *semantics-MULSD* \equiv *binop-XMM* *unknown-mulsd*

definition *semantics-DIVSD* :: *Operand* \Rightarrow *Operand* \Rightarrow *state* \Rightarrow *state*
where *semantics-DIVSD* \equiv *binop-XMM* *unknown-divsd*

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

definition *semantics-UCOMISD* :: *Operand* \Rightarrow *Operand* \Rightarrow *state* \Rightarrow *state*
where *semantics-UCOMISD* \equiv *binop-flags* *UCOMISD-flags*

abbreviation *ADDSD*
where *ADDSD* *op1 op2* \equiv *Instr* "addsd" (*Some* *op1*) (*Some* *op2*) *None*

abbreviation *SUBSD*
where *SUBSD* *op1 op2* \equiv *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-32-128* :: (*32 word* \Rightarrow *32 word* \Rightarrow *32 word*) \Rightarrow *128 word* \Rightarrow *128 word* \Rightarrow *128 word*

where *simd-32-128* *f dst src* \equiv

$((\text{ucast } (\langle 0, 32 \rangle) (f (\text{ucast } (\langle 96, 128 \rangle) \text{dst})) (\text{ucast } (\langle 96, 128 \rangle) \text{src})))) \ll 96$

OR

$((\text{ucast } (\langle 0, 32 \rangle) (f (\text{ucast } (\langle 64, 96 \rangle) \text{dst})) (\text{ucast } (\langle 64, 96 \rangle) \text{src})))) \ll 64$

OR

$((\text{ucast } (\langle 0, 32 \rangle) (f (\text{ucast } (\langle 32, 64 \rangle) \text{dst})) (\text{ucast } (\langle 32, 64 \rangle) \text{src})))) \ll 32$

OR

$(\text{ucast } (\langle 0, 32 \rangle) (f (\text{ucast } (\langle 0, 32 \rangle) \text{dst})) (\text{ucast } (\langle 0, 32 \rangle) \text{src}))))$

abbreviation *semantics-PADDD* :: *Operand* \Rightarrow *Operand* \Rightarrow *state* \Rightarrow *state*

where *semantics-PADDD* \equiv *binop-no-flags* (*simd-32-128* (+))

abbreviation *PADDD*

where *PADDD* *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 $((\langle 0, 32 \rangle) (\text{src} \gg (\text{unat } (\langle 6, 8 \rangle) n * 32))) \ll 96$ *OR*

$((\langle 0, 32 \rangle) (\text{src} \gg (\text{unat } (\langle 4, 6 \rangle) n * 32))) \ll 64$ *OR*

$((\langle 0, 32 \rangle) (\text{src} \gg (\text{unat } (\langle 2, 4 \rangle) n * 32))) \ll 32$ *OR*

$((\langle 0, 32 \rangle) (\text{src} \gg (\text{unat } (\langle 0, 2 \rangle) n * 32)))$

lemmas *pshufd-numeral[simp]* = *pshufd-def*[of numeral *n*] **for** *n*

lemmas *pshufd-0[simp]* = *pshufd-def*[of 0]

lemmas *pshufd-1[simp]* = *pshufd-def*[of 1]

definition *semantics-PSHUF* :: *Operand* \Rightarrow *Operand* \Rightarrow *Operand* \Rightarrow *state* \Rightarrow *state*

where *semantics-PSHUF* *op1 op2 op3* σ \equiv

let *src* = *ucast* (*operand-read* σ *op2*);

n = *ucast* (*operand-read* σ *op3*) *in*

operand-write *op1* (*ucast* (*pshufd* *src* *n*)) σ

abbreviation *PSHUF*

where *PSHUF*D op1 op2 op3 \equiv Instr "pshufd" 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 σ op2;
n = unat (operand-read σ op3) mod 4 in
operand-write op1 (ucast (((0,32)(src >> n*32)))) σ

abbreviation *PEXTRD*

where *PEXTRD* op1 op2 op3 \equiv Instr "peptrd" 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 σ op1)::128 word;
src = ucast (operand-read σ op2)::128 word;
n = unat (operand-read σ 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)) σ

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 $NOP0 \equiv Instr\ "nop"\ None\ None\ None$

abbreviation $NOP1$

where $NOP1\ op1 \equiv Instr\ "nop"\ (Some\ op1)\ None\ None$

abbreviation $NOP2$

where $NOP2\ op1\ op2 \equiv Instr\ "nop"\ (Some\ op1)\ (Some\ op2)\ None$

abbreviation $NOP3$

where $NOP3\ op1\ op2\ op3 \equiv Instr\ "nop"\ (Some\ op1)\ (Some\ op2)\ (Some\ op3)$

definition $semantics$

where $semantics\ i \equiv$

case i of

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

| (*Instr* "imul" (Some op1) (Some op2) (Some op3) -) ⇒ *semantics-IMUL3*
op1 op2 op3
 | (*Instr* "shl" (Some op1) (Some op2) None -) ⇒ *semantics-SHL op1 op2*
 | (*Instr* "sal" (Some op1) (Some op2) None -) ⇒ *semantics-SHL op1 op2*
 | (*Instr* "shr" (Some op1) (Some op2) None -) ⇒ *semantics-SHR op1*
op2
 | (*Instr* "sar" (Some op1) (Some op2) None -) ⇒ *semantics-SAR op1*
op2
 | (*Instr* "shld" (Some op1) (Some op2) (Some op3) -) ⇒ *semantics-SHLD op1*
op2 op3
 | (*Instr* "rol" (Some op1) (Some op2) None -) ⇒ *semantics-ROL op1 op2*
 | (*Instr* "ror" (Some op1) (Some op2) None -) ⇒ *semantics-ROR op1*
op2
 — flag-related
 | (*Instr* "cmp" (Some op1) (Some op2) - -) ⇒ *semantics-CMP op1 op2*
 | (*Instr* "test" (Some op1) (Some op2) - -) ⇒ *semantics-TEST op1 op2*
 — sign-extension
 | (*Instr* "movsxd" (Some op1) (Some op2) - -) ⇒ *semantics-MOV SXD*
op1 op2
 | (*Instr* "movsx" (Some op1) (Some op2) - -) ⇒ *semantics-MOV SXD*
op1 op2
 | (*Instr* "cdqe" (Some op1) (Some op2) - -) ⇒ *semantics-CDQE*
 | (*Instr* "cdq" (Some op1) (Some op2) - -) ⇒ *semantics-CDQ*
 | (*Instr* "cqo" (Some op1) (Some op2) - -) ⇒ *semantics-CQO*
 — logic
 | (*Instr* "and" (Some op1) (Some op2) - -) ⇒ *semantics-AND op1 op2*
 | (*Instr* "or" (Some op1) (Some op2) - -) ⇒ *semantics-OR op1 op2*
 | (*Instr* "xor" (Some op1) (Some op2) - -) ⇒ *semantics-XOR op1 op2*
 | (*Instr* "xorps" (Some op1) (Some op2) - -) ⇒ *semantics-XORPS op1*
op2
 | (*Instr* "not" (Some op1) - - -) ⇒ *semantics-NOT op1*
 — jumps
 | (*Instr* "jmp" (Some op1) None - -) ⇒ *semantics-JMP op1*
 | (*Instr* "jo" (Some op1) None - -) ⇒ *semantics-JO op1*
 | (*Instr* "jno" (Some op1) None - -) ⇒ *semantics-JNO op1*
 | (*Instr* "js" (Some op1) None - -) ⇒ *semantics-JS op1*
 | (*Instr* "jns" (Some op1) None - -) ⇒ *semantics-JNS op1*
 | (*Instr* "je" (Some op1) None - -) ⇒ *semantics-JE op1*
 | (*Instr* "jz" (Some op1) None - -) ⇒ *semantics-JE op1*
 | (*Instr* "jne" (Some op1) None - -) ⇒ *semantics-JNE op1*
 | (*Instr* "jnz" (Some op1) None - -) ⇒ *semantics-JNE op1*
 | (*Instr* "jb" (Some op1) None - -) ⇒ *semantics-JB op1*
 | (*Instr* "jnae" (Some op1) None - -) ⇒ *semantics-JB op1*
 | (*Instr* "jc" (Some op1) None - -) ⇒ *semantics-JB op1*
 | (*Instr* "jnb" (Some op1) None - -) ⇒ *semantics-JNB op1*
 | (*Instr* "jae" (Some op1) None - -) ⇒ *semantics-JNB op1*
 | (*Instr* "jnc" (Some op1) None - -) ⇒ *semantics-JNB op1*
 | (*Instr* "jbe" (Some op1) None - -) ⇒ *semantics-JBE op1*
 | (*Instr* "jna" (Some op1) None - -) ⇒ *semantics-JBE op1*

(Instr "ja" (Some op1) None - -)	⇒ semantics-JA op1
(Instr "jnbe" (Some op1) None - -)	⇒ semantics-JA op1
(Instr "jl" (Some op1) None - -)	⇒ semantics-JL op1
(Instr "jnge" (Some op1) None - -)	⇒ semantics-JL op1
(Instr "jge" (Some op1) None - -)	⇒ semantics-JGE op1
(Instr "jnl" (Some op1) None - -)	⇒ semantics-JGE op1
(Instr "jle" (Some op1) None - -)	⇒ semantics-JLE op1
(Instr "jng" (Some op1) None - -)	⇒ semantics-JLE op1
(Instr "jg" (Some op1) None - -)	⇒ semantics-JG op1
(Instr "jnle" (Some op1) None - -)	⇒ semantics-JG op1
— setXX	
(Instr "seto" (Some op1) None - -)	⇒ semantics-SETO op1
(Instr "setno" (Some op1) None - -)	⇒ semantics-SETNO op1
(Instr "sets" (Some op1) None - -)	⇒ semantics-SETS op1
(Instr "setns" (Some op1) None - -)	⇒ semantics-SETNS op1
(Instr "sete" (Some op1) None - -)	⇒ semantics-SETE op1
(Instr "setz" (Some op1) None - -)	⇒ semantics-SETE op1
(Instr "setne" (Some op1) None - -)	⇒ semantics-SETNE op1
(Instr "setnz" (Some op1) None - -)	⇒ semantics-SETNE op1
(Instr "setb" (Some op1) None - -)	⇒ semantics-SETB op1
(Instr "setnae" (Some op1) None - -)	⇒ semantics-SETB op1
(Instr "setc" (Some op1) None - -)	⇒ semantics-SETB op1
(Instr "setnb" (Some op1) None - -)	⇒ semantics-SETNB op1
(Instr "setae" (Some op1) None - -)	⇒ semantics-SETNB op1
(Instr "setnc" (Some op1) None - -)	⇒ semantics-SETNB op1
(Instr "setbe" (Some op1) None - -)	⇒ semantics-SETBE op1
(Instr "setna" (Some op1) None - -)	⇒ semantics-SETBE op1
(Instr "seta" (Some op1) None - -)	⇒ semantics-SETA op1
(Instr "setnbe" (Some op1) None - -)	⇒ semantics-SETA op1
(Instr "setl" (Some op1) None - -)	⇒ semantics-SETL op1
(Instr "setnge" (Some op1) None - -)	⇒ semantics-SETL op1
(Instr "setge" (Some op1) None - -)	⇒ semantics-SETGE op1
(Instr "setnl" (Some op1) None - -)	⇒ semantics-SETGE op1
(Instr "setle" (Some op1) None - -)	⇒ semantics-SETLE op1
(Instr "setng" (Some op1) None - -)	⇒ semantics-SETLE op1
(Instr "setg" (Some op1) None - -)	⇒ semantics-SETG op1
(Instr "setnle" (Some op1) None - -)	⇒ semantics-SETG op1
— conditional moves	
(Instr "cmovne" (Some op1) (Some op2) - -)	⇒ semantics-CMOVNE
op1 op2	
(Instr "cmovns" (Some op1) (Some op2) - -)	⇒ semantics-CMOVNS
op1 op2	
— floating point (double)	
(Instr "addsd" (Some op1) (Some op2) - -)	⇒ semantics-ADDSO op1
op2	
(Instr "subsd" (Some op1) (Some op2) - -)	⇒ semantics-SUBSD op1
op2	
(Instr "mulsd" (Some op1) (Some op2) - -)	⇒ semantics-MULSD op1
op2	

$$\begin{aligned}
& | (\text{Instr } \text{"divsd"} \text{ (Some op1) (Some op2) - -}) \quad \Rightarrow \text{ semantics-DIVSD op1 op2} \\
& | (\text{Instr } \text{"ucomisd"} \text{ (Some op1) (Some op2) - -}) \quad \Rightarrow \text{ semantics-UCOMISD op1 op2} \\
& \text{--- simd} \\
& | (\text{Instr } \text{"padd"} \text{ (Some op1) (Some op2) - -}) \quad \Rightarrow \text{ semantics-PADDD op1 op2} \\
& | (\text{Instr } \text{"pshufd"} \text{ (Some op1) (Some op2) (Some op3) -}) \Rightarrow \text{ semantics-PSHUFD op1 op2 op3} \\
& | (\text{Instr } \text{"pextrd"} \text{ (Some op1) (Some op2) (Some op3) -}) \Rightarrow \text{ semantics-PEXTRD op1 op2 op3} \\
& | (\text{Instr } \text{"pinsrd"} \text{ (Some op1) (Some op2) (Some op3) -}) \Rightarrow \text{ semantics-PINSRD op1 op2 op3} \\
& \text{--- remainder} \\
& | (\text{Instr } \text{"nop"} \text{ - - - -}) \quad \Rightarrow \text{ semantics-NOP} \\
& | (\text{Instr } \text{"bswap"} \text{ (Some op1) - - -}) \quad \Rightarrow \text{ semantics-BSWAP op1} \\
& \text{--- external function} \\
& | (\text{ExternalFunc } f) \Rightarrow f \\
& | i \Rightarrow \text{ unknown-semantics } i
\end{aligned}$$

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 $\text{step} :: I \Rightarrow \text{state} \Rightarrow \text{state}$

where $\text{step } i \sigma \equiv$
 $\text{let } \sigma' = \sigma \text{ with } [\text{setRip } (\text{instr-next } i)] \text{ in}$
 $\text{semantics } i \sigma'$

All simplification rules used during symbolic execution.

lemmas $\text{semantics-simps} =$

$\text{Let-def unop-def unop-no-flags-def binop-def binop-flags-def binop-no-flags-def}$
 binop-XMM-def
 $\text{semantics-def mov-sign-extension-def simd-32-128-def}$
 $\text{write-MUL-result-def unop-MUL-def ternop-IMUL-def sbb-def adc-def shld-def}$

 $\text{semantics-MOV-def semantics-MOVSD-def semantics-MOVD-def seman-}$
 tics-CMOV-def
 $\text{semantics-LEA-def semantics-PUSH-def semantics-POP-def}$
 $\text{semantics-RET-def semantics-CALL-def semantics-LEAVE-def}$
 $\text{semantics-ADD-def semantics-INC-def semantics-DEC-def semantics-NEG-def}$
 semantics-SUB-def
 $\text{semantics-SBB-def semantics-ADC-def}$
 $\text{semantics-MUL-def semantics-IMUL1-def semantics-IMUL2-def semantics-IMUL3-def}$
 $\text{semantics-SHL-def semantics-SHR-def semantics-SAR-def semantics-SHLD-def}$
 $\text{semantics-ROL-def semantics-ROR-def}$
 $\text{semantics-CMP-def semantics-TEST-def}$
 $\text{semantics-MOVSXD-def semantics-CDQE-def semantics-CDQ-def seman-}$
 tics-CQO-def
 $\text{semantics-AND-def semantics-OR-def semantics-XOR-def semantics-XORPS-def}$

```

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 seman-
tics-DIVSD-def semantics-UCOMISD-def
  semantics-NOP-def semantics-BSWAP-def semantics-PSHUF-def seman-
tics-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* $\equiv id$

We are going to make schematic theorems of the form:

$$assumptions?A \implies \dots$$

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 \Rightarrow state \Rightarrow 64 word \Rightarrow nat \Rightarrow 256 word \Rightarrow bool \Rightarrow bool

where *eq-except-for-mem* σ σ' a si v b \equiv σ with $[[a,si]] :=_m v$ = (if b then σ' else σ with $[[a,si]] :=_m v$)

lemma *efm-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 *efm-clean-mem*:

assumes $si' \leq si$

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 *efm-clean-mem* = (rule *efm-clean-mem*, (*simp* (*no-asm*);*fail*))

lemma *efm-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 $([[a,si]] :=_m v)\#updates$)
 $(\sigma$ with $([[a,si]] :=_m \text{overwrite } (8 * \text{unat } (a' - a)) (8 * \text{unat } (a' - a) + 8 * si')) v (v' << \text{unat } (a' - a) * 8)\#updates')$
 a' si' v' *True*

<proof>

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

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

method *efm-clean-mem-enclosed-plus* =

(rule *efm-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* (σ with updates) (σ with updates') *a si v b*

shows *eq-except-for-mem* (σ with ($\llbracket a',si' \rrbracket :=_m v' \rrbracket \# updates$)) (σ with ($\llbracket a',si \rrbracket :=_m v \rrbracket \# updates'$)) *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* (σ with updates) (σ with updates) *a si v False*

<proof>

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

<ML>

The following method takes a goal with state with symbolic state updates. It first applies *eq-except-for-mem* (σ with *?updates*) (σ with *?updates'*) *?a ?si ?v ?b \implies ?\sigma* with ($\llbracket ?a, ?si \rrbracket :=_m ?v \rrbracket \# ?updates$) = (*if ?b then ?\sigma* with *?updates'* else σ with ($\llbracket ?a, ?si \rrbracket :=_m ?v \rrbracket \# ?updates'$)), considering the outer-most memory update to some region $\llbracket a, si \rrbracket$. A list *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).

⟨ML⟩

```

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 ⇒ (64 word ⇒ I) ⇒ state ⇒ state ⇒ bool

```

```

  where

```

```

    rip  $\sigma$  = af ⇒ fetch (rip  $\sigma$ ) = i ⇒  $\sigma'$   $\triangleq$  step i  $\sigma$  ⇒ run af fetch  $\sigma$   $\sigma'$ 
  | rip  $\sigma$  ≠ af ⇒ fetch (rip  $\sigma$ ) = i ⇒ run af fetch (step i  $\sigma$ )  $\sigma'$  ⇒ run af
  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  $\sigma \equiv \text{regs } \sigma \text{ "rsp"}$ 
abbreviation RBP0
  where RBP0  $\sigma \equiv \text{regs } \sigma \text{ "rbp"}$ 
abbreviation RAX0
  where RAX0  $\sigma \equiv \text{regs } \sigma \text{ "rax"}$ 
abbreviation RBX0
  where RBX0  $\sigma \equiv \text{regs } \sigma \text{ "rbx"}$ 
abbreviation RCX0
  where RCX0  $\sigma \equiv \text{regs } \sigma \text{ "rcx"}$ 
abbreviation RDX0
  where RDX0  $\sigma \equiv \text{regs } \sigma \text{ "rdx"}$ 
abbreviation RDI0
  where RDI0  $\sigma \equiv \text{regs } \sigma \text{ "rdi"}$ 
abbreviation RSI0
  where RSI0  $\sigma \equiv \text{regs } \sigma \text{ "rsi"}$ 
abbreviation R150
  where R150  $\sigma \equiv \text{regs } \sigma \text{ "r15"}$ 
abbreviation R140
  where R140  $\sigma \equiv \text{regs } \sigma \text{ "r14"}$ 
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 := 0xAABBCCDDAABB

```

```

schematic-goal example2:
  assumes[simp]: fetch 0x0 = MOV (QWORD PTR ["rax"]1) (Imm 0xAABBC-
CDDEEFF) 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 (σ with [setRip 0x0]) ?σ'
    ⟨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 0xAABBCC-
DDDEEFF) 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 ==> run 0x4 fetch (σ with [setRip 0x0]) ?σ'
    ⟨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

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-insn 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* = *ucast* (*operand-read* σ (*QWORD PTR* ["rdi"]₁));
 val = *mem-read* σ *ptr* 1 in
 (*semantics-RET* *o*
 semantics-MOV (*Reg* "eax") (*Imm* (*fromBool* (*val* = 0))))
 σ

declare *EXTERNAL-FUNCTION-feof-def* [*simp*]

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

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

definition *EXTERNAL-FUNCTION-fopen* $\sigma =$
semantics-RET (σ 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,ecx
cc6: call  bd5 <isword>
ccb: test  eax,ecx
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    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]    # 202058 <lcount>
d80: mov    rdx,QWORD PTR [rip+0x2012c1]    # 202048 <wcount>
d87: mov    rsi,QWORD PTR [rip+0x2012c2]    # 202050 <ccount>
d8e: mov    rax,QWORD PTR [rbp-0x18]
d92: mov    rdi,rax
d95: call  b93 <report>
d9a: mov    rdx,QWORD PTR [rip+0x20128f]    # 202030 <total-ccount>
da1: mov    rax,QWORD PTR [rip+0x2012a8]    # 202050 <ccount>
da8: add   rax,rdx
dab: mov    QWORD PTR [rip+0x20127e],rax    # 202030 <total-ccount>
db2: mov    rdx,QWORD PTR [rip+0x20127f]    # 202038 <total-wcount>
db9: mov    rax,QWORD PTR [rip+0x201288]    # 202048 <wcount>
dc0: add   rax,rdx
dc3: mov    QWORD PTR [rip+0x20126e],rax    # 202038 <total-wcount>
dca: mov    rdx,QWORD PTR [rip+0x20126f]    # 202040 <total-lcount>
dd1: mov    rax,QWORD PTR [rip+0x201280]    # 202058 <lcount>
dd8: add   rax,rdx
ddb: mov    QWORD PTR [rip+0x20125e],rax    # 202040 <total-lcount>
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* σ_I *? σ'*

<proof>

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/>.