Formal specification of the Cap9 kernel

Mikhail Mandrykin   Ilya Shchepetkov

May 31, 2019

Contents

1 Introduction

This is an Isabelle/HOL theory that describes and proves the correctness of the Cap9 kernel specification.

2 Preliminaries

```
theory Cap9
imports
  "HOL−Word.Word"
  "HOL−Library.Adhoc_Overloading"
  "Word_Lib/Word_Lemmas"
begin
```

2.1 Type class instantiations

Instantiate *len* type class to extract lengths from word types avoiding repeated explicit numeric specification of the length e.g. $LENGTH(byte)$ or $LENGTH('a :: len\ word)$ instead of *8* or $LENGTH('a)$, where $'a$ cannot be directly extracted from a type such as $'a\ word$.

```
instantiation word :: (len) len begin
definition len_word[simp]: "len_of (_ :: 'a::len word itself) = LENGTH('a)"
instance by (standard, simp)
end
```

Instantiate *size* type class for types of the form *'a itself*. This allows us to parametrize operations by word lengths using the dummy variables of type *'a word itself*. The operations cannot be directly parametrized by numbers as there is no lifting from term numbers to type numbers due to the lack of dependent types.

```
instantiation itself :: (len) size begin
definition size_itself where [simp, code]: "size (n::'a::len itself) = LENGTH('a)"
instance ..
end

declare unat_word_ariths[simp] word_size[simp]
```

2.2 Word width

We introduce definition of the least numer of bits to hold the current value of a word. This is needed because in our specification we often word with $UCAST('a \to 'b)$'ed values (right aligned subranges of bits), largely again due to the lack of dependent types (or true type-level functions), e.g. the it's hard to specify that the length of $a \bowtie b$ (where $\bowtie$ stands for concatenation) is the sum of the length of a and b, since length is a type parameter and there's no equivalent of sum on the type level. So we instead fix the length of $a \bowtie b$ to be the maximum possible one (say, 32 bytes) and then use conditions of the form $width\ a \leq s$ to specify that the actual "size" of a is s.

```
definition "width w ≡ LEAST n. unat w < 2 ^ n" for w :: "'a::len word"

lemma widthI[intro]: "⟦⋀ u. u < n ⟹ 2 ^ u ≤ unat w; unat w < 2 ^ n⟧ ⟹ width w = n"
  unfolding width_def Least_def
  using not_le
  apply (intro the_equality, blast)
  by (meson nat_less_le)

lemma width_wf[simp]: "∃! n. (∀ u < n. 2 ^ u ≤ unat w) ∧ unat w < 2 ^ n"
  (is "?Ex1 (unat w)")
proof (induction ("unat w"))
  case 0
  show "?Ex1 0" by (intro ex1I[of _ 0], auto)
next
  case (Suc x)
  then obtain n where x:"(∀u<n. 2 ^ u ≤ x) ∧ x < 2 ^ n " by auto
  show  "?Ex1 (Suc x)"
  proof (cases "Suc x < 2 ^ n")
    case True
    thus "?Ex1 (Suc x)"
      using x
      apply (intro ex1I[of _ "n"], auto)
      by (meson Suc_lessD leD linorder_neqE_nat)
  next
    case False
    thus "?Ex1 (Suc x)"
      using x
      apply (intro ex1I[of _ "Suc n"], auto simp add: less_Suc_eq)
      apply (intro antisym)
       apply (metis One_nat_def Suc_lessI Suc_n_not_le_n leI numeral_2_eq_2 power_increasing_iff)
      by (metis Suc_lessD le_antisym not_le not_less_eq_eq)
  qed
qed
```

```
lemma width_iff[iff]: "(width w = n) = ((∀ u < n. 2 ^ u ≤ unat w) ∧ unat w < 2 ^ n)"
  using width_wf widthI by metis

lemma width_le_size: "width x ≤ size x"
proof-
  {
    assume "size x < width x"
    hence "2 ^ size x ≤ unat x" using width_iff by metis
    hence "2 ^ size x ≤ uint x" unfolding unat_def by simp
  }
  thus ?thesis using uint_range_size[of x] by (force simp del:word_size)
qed

lemma width_le_size'[simp]: "size x ≤ n ⟹ width x ≤ n" by (insert width_le_size[of x], simp)

lemma nth_width_high[simp]: "width x ≤ i ⟹ ¬ x !! i"
proof (cases "i < size x")
  case False
  thus ?thesis by (simp add: test_bit_bin')
next
  case True
  hence "(x < 2 ^ i) = (unat x < 2 ^ i)"
    unfolding unat_def
    using word_2p_lem by fastforce
  moreover assume "width x ≤ i"
  then obtain n where "unat x < 2 ^ n" and "n ≤ i" using width_iff by metis
  hence "unat x < 2 ^ i"
    by (meson le_less_trans nat_power_less_imp_less not_less zero_less_numeral)
  ultimately show ?thesis using bang_is_le by force
qed

lemma width_zero[iff]: "(width x = 0) = (x = 0)"
proof
  show "width x = 0 ⟹ x = 0" using nth_width_high[of x] word_eq_iff[of x 0] nth_0 by (metis le0)
  show "x = 0 ⟹ width x = 0" by simp
qed

lemma width_zero'[simp]: "width 0 = 0" by simp

lemma width_one[simp]: "width 1 = 1" by simp

lemma high_zeros_less: "(∀ i ≥ u. ¬ x !! i) ⟹ unat x < 2 ^ u"
  (is "?high ⟹ _") for x :: "'a::len word"
proof-
  assume ?high
  have size:"size (mask u :: 'a word) = size x" by simp
  {
    fix i
    from ‹?high› have "(x AND mask u) !! i = x !! i"
      using nth_mask[of u i] size test_bit_size[of x i]
      by (subst word_ao_nth) (elim allE[of _ i], auto)
  }
  with ‹?high› have "x AND mask u = x" using word_eq_iff by blast
  thus ?thesis unfolding unat_def using mask_eq_iff by auto
qed

lemma nth_width_msb[simp]: "x ≠ 0 ⟹ x !! (width x - 1)"
proof (rule ccontr)
  fix x :: "'a word"
  assume "x ≠ 0"
```

```
  hence width:"width x > 0" using width_zero by fastforce
  assume "¬ x !! (width x - 1)"
  with width have "∀ i ≥ width x - 1. ¬ x !! i"
    using nth_width_high[of x] antisym_conv2 by fastforce
  hence "unat x < 2 ^ (width x - 1)" using high_zeros_less[of "width x - 1" x] by simp
  moreover from width have "unat x ≥ 2 ^ (width x - 1)" using width_iff[of x "width x"] by simp
  ultimately show False by simp
qed

lemma width_iff': "((∀ i > u. ¬ x !! i) ∧ x !! u) = (width x = Suc u)"
proof (rule; (elim conjE | intro conjI))
  assume "x !! u" and "∀ i > u. ¬ x !! i"
  show "width x = Suc u"
  proof (rule antisym)
    from ‹x !! u› show "width x ≥ Suc u" using not_less nth_width_high by force
    from ‹x !! u› have "x ≠ 0" by auto
    with ‹∀ i > u. ¬ x !! i› have "width x - 1 ≤ u" using not_less nth_width_msb by metis
    thus "width x ≤ Suc u" by simp
  qed
next
  assume "width x = Suc u"
  show "∀i>u. ¬ x !! i" by (simp add:‹width x = Suc u›)
  from ‹width x = Suc u› show "x !! u" using nth_width_msb width_zero
    by (metis diff_Suc_1 old.nat.distinct(2))
qed

lemma width_word_log2: "x ≠ 0 ⟹ width x = Suc (word_log2 x)"
  using word_log2_nth_same word_log2_nth_not_set width_iff' test_bit_size
  by metis

lemma width_ucast[OF refl, simp]: "uc = ucast ⟹ is_up uc ⟹ width (uc x) = width x"
  by (metis uint_up_ucast unat_def width_def)

lemma width_ucast'[OF refl, simp]:
  "uc = ucast ⟹ width x ≤ size (uc x) ⟹ width (uc x) = width x"
proof-
  have "unat x < 2 ^ width x" unfolding width_def by (rule LeastI_ex, auto)
  moreover assume "width x ≤ size (uc x)"
  ultimately have "unat x < 2 ^ size (uc x)" by (simp add: less_le_trans)
  moreover assume "uc = ucast"
  ultimately have "unat x = unat (uc x)" by (metis unat_ucast mod_less word_size)
  thus ?thesis unfolding width_def by simp
qed

lemma width_lshift[simp]:
  "⟦x ≠ 0; n ≤ size x - width x⟧ ⟹ width (x << n) = width x + n"
  (is "⟦_; ?nbound⟧ ⟹ _")
proof-
  assume "x ≠ 0"
  hence 0:"width x = Suc (width x - 1)" using width_zero by (metis Suc_pred' neq0_conv)
  from ‹x ≠ 0› have 1:"width x > 0" by (auto intro:gr_zeroI)
  assume ?nbound
  {
    fix i
    from ‹?nbound› have "i ≥ size x ⟹ ¬ x !! (i - n)" by (auto simp add:le_diff_conv2)
    hence "(x << n) !! i = (n ≤ i ∧ x !! (i - n))" using nth_shiftl'[of x n i] by auto
  } note corr = this
  hence "∀ i > width x + n - 1. ¬ (x << n) !! i" by auto
  moreover from corr have "(x << n) !! (width x + n - 1)"
    using width_iff'[of "width x - 1" x] 1
```

```
    by auto
  ultimately have "width (x << n) = Suc (width x + n - 1)" using width_iff' by auto
  thus ?thesis using 0 by simp
qed

lemma width_lshift'[simp]: "n ≤ size x - width x ⟹ width (x << n) ≤ width x + n"
  using width_zero width_lshift shiftl_0 by (metis eq_iff le0)

lemma width_or[simp]: "width (x OR y) = max (width x) (width y)"
proof-
  {
    fix a b
    assume "width x = Suc a" and "width y = Suc b"
    hence "width (x OR y) = Suc (max a b)"
      using width_iff' word_ao_nth[of x y] max_less_iff_conj[of "a" "b"]
      by (metis (no_types) max_def)
  } note succs = this
  thus ?thesis
  proof (cases "width x = 0 ∨ width y = 0")
    case True
    thus ?thesis using width_zero word_log_esimps(3,9) by (metis max_0L max_0R)
  next
    case False
    with succs show ?thesis by (metis max_Suc_Suc not0_implies_Suc)
  qed
qed
```

2.3 Right zero-padding

Here's the first time we use *width*. If x is a value of size n right-aligned in a word of size $s = size\ x$ (note there's nowhere to keep the value n, since the size of x is some $s \geq n$, so we require it to be provided explicitly), then *rpad n x* will move the value x to the left. For the operation to be correct (no losing of significant higher bits) we need the precondition $width\ x \leq n$ in all the lemmas, hence the need for *width*.

```
definition rpad where "rpad n x ≡ x << size x - n"

lemma rpad_low[simp]: "⟦width x ≤ n; i < size x - n⟧ ⟹ ¬ (rpad n x) !! i"
  unfolding rpad_def by (simp add:nth_shiftl)

lemma rpad_high[simp]:
  "⟦width x ≤ n; n ≤ size x; size x - n ≤ i⟧ ⟹ (rpad n x) !! i = x !! (i + n - size x)"
  (is "⟦?xbound; ?nbound; i ≥ ?ibound⟧ ⟹ ?goal i")
proof-
  fix i
  assume ?xbound ?nbound and "i ≥ ?ibound"
  moreover from ‹?nbound› have "i + n - size x = i - ?ibound" by simp
  moreover from ‹?xbound› have "x !! (i + n - size x) ⟹ i < size x" by - (rule ccontr, simp)
  ultimately show "?goal i" unfolding rpad_def by (subst nth_shiftl', metis)
qed

lemma rpad_inj: "⟦width x ≤ n; width y ≤ n; n ≤ size x⟧ ⟹ rpad n x = rpad n y ⟹ x = y"
  (is "⟦?xbound; ?ybound; ?nbound; _⟧ ⟹ _")
  unfolding inj_def word_eq_iff
proof (intro allI impI)
  fix i
  let ?i' = "i + size x - n"
  assume ?xbound ?ybound ?nbound
  assume "∀j < LENGTH('a). rpad n x !! j = rpad n y !! j"
  hence "⋀ j. rpad n x !! j = rpad n y !! j" using test_bit_bin by blast
```

```
  from this[of ?i'] and ‹?xbound› ‹?ybound› ‹?nbound› show "x !! i = y !! i" by simp
qed
```

2.4 Spanning concatenation

```
abbreviation ucastl ("'(ucast')⇩_ _" [1000, 100] 100) where
  "(ucast)⇩l a ≡ ucast a :: 'b word" for l :: "'b::len0 itself"

notation (input) ucastl ("'(ucast')⇩_ _" [1000, 100] 100)

definition pad_join :: "'a::len word ⇒ nat ⇒ 'c::len itself ⇒ 'b::len word ⇒ 'c word"
  ("_ ⇩_◇⇩_ _" [60, 1000, 1000, 61] 60) where
  "x ⇩n◇⇩l y ≡ rpad n (ucast x) OR ucast y"

notation (input) pad_join ("_ ⇩_◇⇩_ _" [60, 1000, 1000, 61] 60)

lemma pad_join_high:
  "⟦width a ≤ n; n ≤ size l; width b ≤ size l − n; size l − n ≤ i⟧
   ⟹ (a ⇩n◇⇩l b) !! i = a !! (i + n − size l)"
  unfolding pad_join_def
  using nth_ucast nth_width_high by fastforce

lemma pad_join_high'[simp]:
  "⟦width a ≤ n; n ≤ size l; width b ≤ size l − n⟧ ⟹ a !! i = (a ⇩n◇⇩l b) !! (i + size l − n)"
  using pad_join_high[of a n l b "i + size l − n"] by simp

lemma pad_join_mid[simp]:
  "⟦width a ≤ n; n ≤ size l; width b ≤ size l − n; width b ≤ i; i < size l − n⟧
   ⟹ ¬ (a ⇩n◇⇩l b) !! i"
  unfolding pad_join_def by auto

lemma pad_join_low[simp]:
  "⟦width a ≤ n; n ≤ size l; width b ≤ size l − n; i < width b⟧ ⟹ (a ⇩n◇⇩l b) !! i = b !! i"
  unfolding pad_join_def by (auto simp add: nth_ucast)

lemma pad_join_inj:
  assumes eq:"a ⇩n◇⇩l b = c ⇩n◇⇩l d"
  assumes a:"width a ≤ n" and c:"width c ≤ n"
  assumes n: "n ≤ size l"
  assumes b:"width b ≤ size l − n"
  assumes d:"width d ≤ size l − n"
  shows   "a = c" and "b = d"
proof−
  from eq have eq':"⋀j. (a ⇩n◇⇩l b) !! j = (c ⇩n◇⇩l d) !! j"
    using test_bit_bin unfolding word_eq_iff by auto
  moreover from a n b
  have "⋀ i. a !! i = (a ⇩n◇⇩l b) !! (i + size l − n)" by simp
  moreover from c n d
  have "⋀ i. c !! i = (c ⇩n◇⇩l d) !! (i + size l − n)" by simp
  ultimately show "a = c" unfolding word_eq_iff by auto

  {
    fix i
    from a n b have "i < width b ⟹ b !! i = (a ⇩n◇⇩l b) !! i" by simp
    moreover from c n d have "i < width d ⟹ d !! i = (c ⇩n◇⇩l d) !! i" by simp
    moreover have "i ≥ width b ⟹ ¬ b !! i" and "i ≥ width d ⟹ ¬ d !! i" by auto
    ultimately have "b !! i = d !! i"
      using eq'[of i] b d
        pad_join_mid[of a n l b i, OF a n b]
        pad_join_mid[of c n l d i, OF c n d]
      by (meson leI less_le_trans)
```

```
  }
  thus "b = d" unfolding word_eq_iff by simp
qed

lemma pad_join_inj'[dest!]:
 "⟦a n◇l b = c n◇l d;
   width a ≤ n; width c ≤ n; n ≤ size l;
   width b ≤ size l − n;
   width d ≤ size l − n⟧ ⟹ a = c ∧ b = d"
  apply (rule conjI)
  subgoal by (frule (4) pad_join_inj(1))
  by (frule (4) pad_join_inj(2))

definition restrict :: "'a::len word ⇒ nat set ⇒ 'a word" (infixl "↾" 60) where
  "restrict x s ≡ BITS i. i ∈ s ∧ x !! i"
```

2.5 Deal with partially undefined results

```
lemma nth_restrict[iff]: "(x ↾ s) !! n = (n ∈ s ∧ x !! n)"
  unfolding restrict_def
  by (simp add: bang_conj_lt test_bit.eq_norm)

lemma restrict_inj2[dest!]:
  assumes eq:"f x1 y1 OR v1 ↾ s = f x2 y2 OR v2 ↾ s"
  assumes fi:"⋀ x y i. i ∈ s ⟹ ¬ f x y !! i"
  assumes inj:"⋀ x1 y1 x2 y2. f x1 y1 = f x2 y2 ⟹ x1 = x2 ∧ y1 = y2"
  shows   "x1 = x2 ∧ y1 = y2"
proof−
  from eq and fi have "f x1 y1 = f x2 y2" unfolding word_eq_iff by auto
  with inj show ?thesis .
qed
```

2.6 Plain concatenation

```
definition join :: "'a::len word ⇒ 'c::len itself ⇒ nat ⇒ 'b::len word ⇒ 'c word"
  ("_ _⋈_ _" [62,1000,1000,61] 61) where
  "(a l⋈n b) ≡ (ucast a << n) OR (ucast b)"

notation (input) join ("_ _⋈_ _" [62,1000,1000,61] 61)

lemma width_join:
  "⟦width a + n ≤ size l; width b ≤ n⟧ ⟹ width (a l⋈n b) ≤ width a + n"
  (is "⟦?abound; ?bbound⟧ ⟹ _")
proof−
  assume ?abound and ?bbound
  moreover hence "width b ≤ size l" by simp
  ultimately show ?thesis
    using width_lshift'[of n "(ucast)l a"]
    unfolding join_def
    by simp
qed

lemma width_join'[simp]:
  "⟦width a + n ≤ size l; width b ≤ n; width a + n ≤ q⟧ ⟹ width (a l⋈n b) ≤ q"
  by (drule (1) width_join, simp)

lemma join_high[simp]:
  "⟦width a + n ≤ size l; width b ≤ n; width a + n ≤ i⟧ ⟹ ¬ (a l⋈n b) !! i"
  by (drule (1) width_join, simp)

lemma join_mid:
```

```
  "⟦width a + n ≤ size l; width b ≤ n; n ≤ i; i < width a + n⟧ ⟹ (a l⋈n b) !! i = a !! (i − n)"
  apply (subgoal_tac "i < size ((ucast)l a) ∧ size ((ucast)l a) = size l")
  unfolding join_def
  using word_ao_nth nth_ucast nth_width_high nth_shiftl'
   apply (metis less_imp_diff_less order_trans word_size)
  by simp

lemma join_mid'[simp]:
  "⟦width a + n ≤ size l; width b ≤ n⟧ ⟹ a !! i = (a l⋈n b) !! (i + n)"
  using join_mid[of a n l b "i + n"] nth_width_high[of a i] join_high[of a n l b "i + n"]
  by force

lemma join_low[simp]:
  "⟦width a + n ≤ size l; width b ≤ n; i < n⟧ ⟹ (a l⋈n b) !! i = b !! i"
  unfolding join_def
  by (simp add: nth_shiftl nth_ucast)

lemma join_inj:
  assumes eq:"a l⋈n b = c l⋈n d"
  assumes "width a + n ≤ size l" and "width b ≤ n"
  assumes "width c + n ≤ size l" and "width d ≤ n"
  shows   "a = c" and "b = d"
proof−
  from assms show "a = c" unfolding word_eq_iff using join_mid' eq by metis
  from assms show "b = d" unfolding word_eq_iff using join_low nth_width_high
    by (metis eq less_le_trans not_le)
qed

lemma join_inj'[dest!]:
  "⟦a l⋈n b = c l⋈n d;
    width a + n ≤ size l; width b ≤ n;
    width c + n ≤ size l; width d ≤ n⟧ ⟹ a = c ∧ b = d"
  apply (rule conjI)
  subgoal by (frule (4) join_inj(1))
  by (frule (4) join_inj(2))
```

3 Data formats

3.1 Procedure keys

Procedure keys are represented as 24-byte (192 bits) machine words.

```
type_synonym word24 = "192 word" — 24 bytes
type_synonym key = word24
```

3.2 Storage state

Byte is 8-bit machine word:

```
type_synonym byte = "8 word"
```

32-byte machine words that are used to model keys and values of the storage.

```
type_synonym word32 = "256 word" — 32 bytes
```

Storage is a function that takes a 32-byte word (key) and returns another 32-byte word (value).

```
type_synonym storage = "word32 ⇒ word32"
```

3.3 Common notation

Specialize previously defined general concatenation operations for the fixed result size of 32 bytes. Thus we avoid lots of redundant type annotations for every intermediate result (note that these intermediate types cannot be inferred automatically (in a purely Hindley-Milner setting as in Isabelle), because this would require type-level functions/dependent types).

```
abbreviation "len (_ :: 'a::len word itself) ≡ TYPE('a)"

no_notation join  ("_ _⋈_ _" [62,1000,1000,61] 61)
no_notation (input) join ("_ _⋈_ _" [62,1000,1000,61] 61)

abbreviation join32 ("_ ⋈_ _" [62,1000,61] 61) where
  "a ⋈n b ≡ join a (len TYPE(word32)) (n * 8) b"
abbreviation (output) join32_out ("_ ⋈_ _" [62,1000,61] 61) where
  "join32_out a n b ≡ join a (TYPE(256)) n b"
notation (input) join32 ("_ ⋈_ _" [62,1000,61] 61)

no_notation pad_join  ("_ _◊_ _" [60,1000,1000,61] 60)
no_notation (input) pad_join ("_ _◊_ _" [60,1000,1000,61] 60)

abbreviation pad_join32 ("_ _◊ _" [60,1000,61] 60) where
  "a n◊ b ≡ pad_join a (n * 8) (len TYPE(word32)) b"
abbreviation (output) pad_join32_out ("_ _◊ _" [60,1000,61] 60) where
  "pad_join32_out a n b ≡ pad_join a n (TYPE(256)) b"
notation (input) pad_join32 ("_ _◊ _" [60,1000,61] 60)
```

Override treatment of hexidecimal numeric constants to make them monomorphic words of fixed length, mimicking the notation used in the informal specification (e.g. *1::'a*) is always a word 1 byte long and is not, say, the natural number one). Otherwise, again, lots of redundant type annotations would arise.

```
parse_ast_translation ‹
  let
    open Ast
    fun mk_numeral t = mk_appl (Constant @{syntax_const _Numeral}) t
    fun mk_word_numeral num t =
      if String.isPrefix 0x num then
        mk_appl (Constant @{syntax_const _constrain})
          [mk_numeral t,
           mk_appl (Constant @{type_syntax word})
             [mk_appl (Constant @{syntax_const _NumeralType})
             [Variable (4 * (size num − 2) |> string_of_int)]]]
      else
        mk_numeral t
    fun numeral_ast_tr ctxt (t as [Appl [Constant @{syntax_const _constrain},
                                          Constant num,
                                          _]]) =
          mk_word_numeral num t
      | numeral_ast_tr ctxt (t as [Constant num]) = mk_word_numeral num t
      | numeral_ast_tr _ t                        = mk_numeral t
      | numeral_ast_tr _ t                        = raise AST (@{syntax_const _Numeral}, t)
  in
    [(@{syntax_const _Numeral}, numeral_ast_tr)]
  end
›
```

Introduce generic notation for representation/encoding of various "logical"/abstract entities into machine words. We use adhoc overloading to use the same notation for various types of entities (indices, offsets, addresses, capabilities etc.).

```
no_notation floor ("⌊_⌋")

consts rep :: "'a ⇒ 'b" ("⌊_⌋")
```

3.4 Addresses

We don't include *Null* capability into the type. It is only handled specially inside the call delegation, otherwise it only complicates the proofs with side conditions $\neq$ *Null*. So there will be separate type *call* defined as *capability option* to respect the fact that it can be *Null*.

In general, in the following we strive to make all encoding functions injective without any preconditions. All the necessary invariants are built into the type definitions.

```
datatype capability =
    Call
  | Reg
  | Del
  | Entry
  | Write
  | Log
  | Gas

definition cap_type_rep :: "capability ⇒ byte" where
  "cap_type_rep c ≡ case c of
      Call  ⇒ 0x03
    | Reg   ⇒ 0x04
    | Del   ⇒ 0x05
    | Entry ⇒ 0x06
    | Write ⇒ 0x07
    | Log   ⇒ 0x08
    | Gas   ⇒ 0x09"

adhoc_overloading rep cap_type_rep

lemma cap_type_rng[simp]: "⌊c⌋ ∈ {0x03..0x09}" for c :: capability
  unfolding cap_type_rep_def by (simp split:capability.split)

lemma cap_type_inj[simp]: "⌊c₁⌋ = ⌊c₂⌋ ⟹ c₁ = c₂" for c₁ c₂ :: capability
  unfolding cap_type_rep_def
  by (simp split:capability.splits)

lemma width_cap_type: "width (⌊c⌋+ 1) ≤ 4" for c :: capability
proof (rule ccontr, drule not_le_imp_less)
  assume "4 < width (⌊c⌋ + 1)"
  moreover hence "(⌊c⌋ + 1) !! (width (⌊c⌋ + 1) − 1)" using nth_width_msb by force
  ultimately obtain n where "(⌊c⌋ + 1) !! n" and "n ≥ 4" by (metis le_step_down_nat nat_less_le)
  thus False unfolding cap_type_rep_def by (simp split:capability.splits)
qed

lemma width_cap_type'[simp]: "4 ≤ n ⟹ width (⌊c⌋ + 1) ≤ n" for c :: capability
  using width_cap_type[of c] by simp

lemma cap_type_nonzero[simp]: "⌊c⌋ ≠ 0" for c :: capability
  unfolding cap_type_rep_def by (simp split:capability.splits)

typedef capability_index = "{i :: byte. i < 0xff}" morphisms cap_index_rep cap_index
  by (intro exI[of _ "0"], simp)

adhoc_overloading rep cap_index_rep

lemma width_cap_index: "width (⌊i⌋+ 1) ≤ 8" for i :: capability_index by simp

lemma width_cap_index'[simp]: "8 ≤ n ⟹ width (⌊i⌋ + 1) ≤ n" for i :: capability_index by simp

lemma cap_index_nonzero[simp]: "⌊i⌋ + 1 ≠ 0" for i :: capability_index
```

```
  using less_is_non_zero_p1 cap_index_rep[of i] by auto

type_synonym capability_offset = byte

datatype data_offset =
  Addr
  | Index
  | Ncaps capability
  | Cap capability capability_index capability_offset

definition data_offset_rep :: "data_offset ⇒ word32" where
 "data_offset_rep off ≡ case off of
     Addr          ⇒ 0x00  ⋈₂ 0x00    ⋈₁  0x00
   | Index         ⇒ 0x00  ⋈₂ 0x00    ⋈₁  0x01
   | Ncaps ty      ⇒ ⌊ty⌋  ⋈₂ 0x00    ⋈₁  0x00
   | Cap ty i off  ⇒ ⌊ty⌋  ⋈₂ ⌊i⌋ + 1  ⋈₁  off"

adhoc_overloading rep data_offset_rep

lemma data_offset_inj[simp]:
  "⌊d₁⌋ = ⌊d₂⌋ ⟹ d₁ = d₂" for d₁ d₂ :: data_offset
  unfolding data_offset_rep_def
  by (auto split:data_offset.splits simp add:cap_index_rep_inject)

lemma width_data_offset: "width ⌊d⌋ ≤ 3 * 8" for d :: data_offset
  unfolding data_offset_rep_def
  by (simp split:data_offset.splits)

lemma width_data_offset'[simp]: "3 * 8 ≤ n ⟹ width ⌊d⌋ ≤ n" for d :: data_offset
  using width_data_offset[of d] by simp

typedef key_index = "{i :: nat. i < 2 ^ LENGTH(key) − 1}" morphisms key_index_rep' key_index
  by (rule exI[of _ "0"], simp)

adhoc_overloading rep key_index_rep'

datatype address =
   Heap_proc key data_offset
  | Nprocs
  | Proc_key key_index
  | Kernel
  | Curr_proc
  | Entry_proc

definition "key_index_rep i ≡ of_nat (⌊i⌋ + 1) :: key" for i :: key_index

adhoc_overloading rep key_index_rep

lemma key_index_nonzero[simp]: "⌊i⌋ ≠ (0 :: key)" for i :: key_index
  unfolding key_index_rep_def using key_index_rep'[of i]
  by (intro of_nat_neq_0, simp_all)

lemma key_index_inj[simp]: "(⌊i₁⌋ :: key) = ⌊i₂⌋ ⟹ i₁ = i₂" for i :: key_index
  unfolding key_index_rep_def using key_index_rep'[of i₁] key_index_rep'[of i₂]
  by (simp add:key_index_rep'_inject of_nat_inj)

definition addr_rep :: "address ⇒ word32" where
  "addr_rep a ≡ case a of
    Heap_proc k offs ⇒ 0xffffffff ⋈₁ 0x00 ₅◊ k          ⋈₃ ⌊offs⌋
  | Nprocs           ⇒ 0xffffffff ⋈₁ 0x01 ₅◊ (0 :: key) ⋈₃ 0x000000
```

```
  | Proc_key i      ⇒ 0xffffffff ⋈₁ 0x01 ₅◊ ⌊i⌋         ⋈₃ 0x000000
  | Kernel          ⇒ 0xffffffff ⋈₁ 0x02 ₅◊ (0 :: key) ⋈₃ 0x000000
  | Curr_proc       ⇒ 0xffffffff ⋈₁ 0x03 ₅◊ (0 :: key) ⋈₃ 0x000000
  | Entry_proc      ⇒ 0xffffffff ⋈₁ 0x04 ₅◊ (0 :: key) ⋈₃ 0x000000"

adhoc_overloading rep addr_rep

lemma address_inj[simp]: "⌊a₁⌋ = ⌊a₂⌋ ⟹ a₁ = a₂" for a₁ a₂ :: address
  unfolding addr_rep_def
  by (split address.splits) (force split:address.splits)+

definition addr ("⌈_⌉^addr") where
  "addr w ≡ if w ∈ range addr_rep then Some (the_inv addr_rep w) else None"

lemma addr_inv[simp]: "⌈⌊a⌋⌉^addr = Some a"
  unfolding addr_def
  by (auto simp add:inj_def the_inv_f_f)

lemma addr_inv'[simp]: "⌈w⌉^addr = Some a ⟹ ⌊a⌋ = w"
  unfolding addr_def
  by (auto intro:f_the_inv_into_f simp add:inj_def split:if_splits)
```

3.5 Capability formats

```
no_notation ceiling ("⌈_⌉")

locale cap_sub =
  fixes set_of :: "'a ⇒ 'b set" ("⌈_⌉")
  fixes sub :: "'a ⇒ 'a ⇒ bool" ("(_/ ⊆c _)" [51, 51] 50)
  assumes wd:"a ⊆c b = (⌈a⌉ ⊆ ⌈b⌉)" begin

lemma sub_refl: "a ⊆c a" using wd by auto

lemma sub_trans: "⟦a ⊆c b; b ⊆c c⟧ ⟹ a ⊆c c" using wd by blast
end

consts set_of :: "'a ⇒ 'b set" ("⌈_⌉")

consts sub :: "'a ⇒ 'a ⇒ bool" ("(_/ ⊆c _)" [51, 51] 50)
```

3.5.1 Prefixed capability (Call, Register, Delete)

```
typedef prefix_size = "{n :: nat. n ≤ LENGTH(key)}"
  morphisms prefix_size_rep' prefix_size
  by auto

adhoc_overloading rep prefix_size_rep'

definition "prefix_size_rep s ≡ of_nat ⌊s⌋ :: byte" for s :: prefix_size

adhoc_overloading rep prefix_size_rep

lemma prefix_size_inj[simp]: "(⌊s₁⌋ :: byte) = ⌊s₂⌋ ⟹ s₁ = s₂" for s₁ s₂ :: prefix_size
  unfolding prefix_size_rep_def using prefix_size_rep'[of s₁] prefix_size_rep'[of s₂]
  by (simp add:prefix_size_rep'_inject of_nat_inj)

lemma prefix_size_rep_less[simp]: "LENGTH(key) ≤ n ⟹ ⌊s⌋ ≤ (n :: nat)" for s :: prefix_size
  using prefix_size_rep'[of s] by simp

type_synonym prefixed_capability = "prefix_size × key"
```

```
definition
  "set_of_pref_cap sk ≡ let (s, k) = sk in {k' :: key. take ⌊s⌋ (to_bl k') = take ⌊s⌋ (to_bl k)}"
  for sk :: prefixed_capability

adhoc_overloading set_of set_of_pref_cap

definition "pref_cap_sub A B ≡
  let (s_A, k_A) = A in let (s_B, k_B) = B in
  (⌊s_A⌋ :: nat) ≥ ⌊s_B⌋ ∧ take ⌊s_B⌋ (to_bl k_A) = take ⌊s_B⌋ (to_bl k_B)"
  for A B :: prefixed_capability

adhoc_overloading sub pref_cap_sub

lemma nth_take_i[dest]: "⟦take n a = take n b; i < n⟧ ⟹ a ! i = b ! i"
  by (metis nth_take)

lemma take_less_diff:
  fixes l' l'' :: "'a list"
  assumes ex:"⋀ u :: 'a. ∃ u'. u' ≠ u"
  assumes "n < m"
  assumes "length l' = length l''"
  assumes "n ≤ length l'"
  assumes "m ≤ length l'"
  obtains l where
      "length l = length l'"
  and "take n l = take n l'"
  and "take m l ≠ take m l''"
proof–
  let ?x = "l'' ! n"
  from ex obtain y where neq:"y ≠ ?x" by auto
  let ?l = "take n l' @ y # drop (n + 1) l'"
  from assms have 0:"n = length (take n l') + 0" by simp
  from assms have "take n ?l = take n l'" by simp
  moreover from assms and neq have "take m ?l ≠ take m l''"
    using 0 nth_take_i nth_append_length
    by (metis add.right_neutral)
  moreover have "length ?l = length l'" using assms by auto
  ultimately show ?thesis using that by blast
qed

lemma pref_cap_sub_iff[iff]: "a ⊆_c b = (⌈a⌉ ⊆ ⌈b⌉)" for a b :: prefixed_capability
proof
  show "a ⊆_c b ⟹ ⌈a⌉ ⊆ ⌈b⌉"
    unfolding pref_cap_sub_def set_of_pref_cap_def
    by (force intro:nth_take_lemma)
  {
    fix n m :: prefix_size
    fix x y :: key
    assume "⌊n⌋ < (⌊m⌋ :: nat)"
    then obtain z where
      "length z = size x"
      "take ⌊n⌋ z = take ⌊n⌋ (to_bl x)" and "take ⌊m⌋ z ≠ take ⌊m⌋ (to_bl y)"
      using take_less_diff[of "⌊n⌋" "⌊m⌋" "to_bl x" "to_bl y"]
      by auto
    moreover hence "to_bl (of_bl z :: key) = z" by (intro word_bl.Abs_inverse[of z], simp)
    ultimately
    have "∃ u :: key.
          take ⌊n⌋ (to_bl u) = take ⌊n⌋ (to_bl x) ∧ take ⌊m⌋ (to_bl u) ≠ take ⌊m⌋ (to_bl y)"
      by metis
  }
```

```
  thus "⌈a⌉ ⊆ ⌈b⌉ ⟹ a ⊆c b"
    unfolding pref_cap_sub_def set_of_pref_cap_def subset_eq
    apply (auto split:prod.split)
    by (erule contrapos_pp[of "∀ x. _ x"], simp)
qed

interpretation cap_sub set_of_pref_cap pref_cap_sub unfolding cap_sub_def by auto

definition "pref_cap_rep sk r ≡
  let (s, k) = sk in ⌊s⌋ 1◊ k OR r ↾ {LENGTH(key) + 1 ..<LENGTH(word32) − LENGTH(byte)}"
  for sk :: prefixed_capability

adhoc_overloading rep pref_cap_rep

lemma pref_cap_rep_inj_helper_inj[simp]: "⌊s1⌋ 1◊ k1 = ⌊s2⌋ 1◊ k2 ⟹ s1 = s2 ∧ k1 = k2"
  for s1 s2 :: prefix_size and k1 k2 :: key
  by auto

lemma pref_cap_rep_inj_helper_zero[simplified, simp]:
  "n ∈ {LENGTH(key) + 1 ..<LENGTH(word32) − LENGTH(byte)} ⟹ ¬ (⌊s⌋ 1◊ k) !! n"
  for s :: prefix_size and k :: key
  by simp

lemma pref_cap_rep_inj[simp]: "⌊c1⌋ r1 = ⌊c2⌋ r2 ⟹ c1 = c2" for c1 c2 :: prefixed_capability
  unfolding pref_cap_rep_def
  apply (simp split:prod.splits)
  by (drule restrict_inj2, simp+)
```

4 Kernel state

Abstract state is implemented as a record with a single component labeled "procs". This component is a mapping from the set of procedure keys to the direct product of procedure indexes and procedure data.

```
record abs =
  keys    :: "key set"
  proc_id :: "key ⇒ nat"
```

4.1 Abbreviations

Here we introduce some useful abbreviations that will simplify the expression of the abstract state properties.

Number of the procedures in the abstract state:

```
abbreviation "nprocs σ ≡ card (keys σ)"
```

List of procedure indexes:

```
abbreviation "proc_ids σ ≡ {1..nprocs σ}"
```

Maximum number of procedures in the abstract state:

```
abbreviation "max_nprocs ≡ 2 ^ LENGTH(key) − 1 :: nat"
```

4.1.1 Well-formedness

For each procedure key the following must be true:

1. corresponding procedure index on the interval from 1 to the number of procedures in the state;
2. key is a valid hash of the procedure data;

3. number of procedures in the state is smaller or equal to the maximum number.

definition "$proc\_id\_rng\_wf\ \sigma \equiv$
$(\forall\ k \in keys\ \sigma.\ proc\_id\ \sigma\ k \in proc\_ids\ \sigma) \wedge$
$nprocs\ \sigma \leq max\_nprocs$"

Procedure indexes must be injective.

definition "$procs\_map\_wf\ \sigma \equiv inj\_on\ (proc\_id\ \sigma)\ (keys\ \sigma)$"

Abstract state is well-formed if the previous two properties are satisfied.

definition abs_wf :: "$abs \Rightarrow bool$" ("$\Vdash$ _" [60] 60) **where**
"$\Vdash \sigma \equiv$
$proc\_id\_rng\_wf\ \sigma$
$\wedge\ procs\_map\_wf\ \sigma$"

lemmas $procs\_rng\_wf = abs\_wf\_def\ proc\_id\_rng\_wf\_def$

lemmas $procs\_map\_wf = abs\_wf\_def\ procs\_map\_wf\_def$

end