Finite Machine Word Library

Joel Beeren, Sascha Böhme, Matthew Fernandez, Xin Gao, Gerwin Klein, Rafal Kolanski, Japheth Lim, Corey Lewis, Daniel Matichuk, Thomas Sewell

March 19, 2025

Abstract

This entry contains an extension to the Isabelle library for fixed-width machine words. In particular, the entry adds printing as hexadecimals, additional operations, reasoning about alignment, signed words, enumerations of words, normalisation of word numerals, and an extensive library of properties about generic fixed-width words, as well as an instantiation of many of these to the commonly used 32 and 64-bit bases.

In addition to the listed authors, the entry contains contributions by Nelson Billing, Andrew Boyton, Matthew Brecknell, Cornelius Diekmann, Peter Gammie, Gianpaolo Gioiosa, David Greenaway, Lars Noschinski, Sean Seefried, and Simon Winwood.

Contents

1 Comprehension syntax for bit expressions

theory Bit_Comprehension
  imports
    "HOL-Library.Word"
begin

class bit_comprehension = ring_bit_operations +
  fixes set_bits :: ‹(nat ⇒ bool) ⇒ 'a› (**binder** ‹BITS › 10)
  assumes set_bits_bit_eq: ‹set_bits (bit a) = a›
begin

lemma set_bits_False_eq [simp]:
  ‹(BITS _. False) = 0›
  ⟨*proof*⟩

end

instantiation word :: (len) bit_comprehension
begin

definition word_set_bits_def:
  ‹(BITS n. P n) = (horner_sum of_bool 2 (map P [0..<LENGTH('a)]) :: 'a word)›

instance ⟨*proof*⟩

end

lemma bit_set_bits_word_iff [bit_simps]:
  ‹bit (set_bits P :: 'a::len word) n ⟷ n < LENGTH('a) ∧ P n›
  ⟨*proof*⟩

lemma word_of_int_conv_set_bits: "word_of_int i = (BITS n. bit i n)"
  ⟨*proof*⟩

lemma set_bits_K_False:
  ‹set_bits (λ_. False) = (0 :: 'a :: len word)›
  ⟨*proof*⟩

lemma word_test_bit_set_bits: "bit (BITS n. f n :: 'a :: len word) n
⟷ n < LENGTH('a) ∧ f n"
⟨*proof*⟩

context
includes bit_operations_syntax
fixes f :: ‹nat ⇒ bool›
begin

definition set_bits_aux :: ‹nat ⇒ 'a word ⇒ 'a::len word›
where ‹set_bits_aux n w = push_bit n w OR take_bit n (set_bits f)›

lemma bit_set_bit_aux [bit_simps]:
‹bit (set_bits_aux n w) m ⟷ m < LENGTH('a) ∧
(if m < n then f m else bit w (m - n))› **for** w :: ‹'a::len word›
⟨*proof*⟩

corollary set_bits_conv_set_bits_aux:
‹set_bits f = (set_bits_aux LENGTH('a) 0 :: 'a :: len word)›
⟨*proof*⟩

lemma set_bits_aux_0 [simp]:
‹set_bits_aux 0 w = w›
⟨*proof*⟩

lemma set_bits_aux_Suc [simp]:
‹set_bits_aux (Suc n) w = set_bits_aux n (push_bit 1 w OR (if f n then
1 else 0))›
⟨*proof*⟩

lemma set_bits_aux_simps [code]:
‹set_bits_aux 0 w = w›
‹set_bits_aux (Suc n) w = set_bits_aux n (push_bit 1 w OR (if f n then
1 else 0))›
⟨*proof*⟩

lemma set_bits_aux_rec:
‹set_bits_aux n w =
(if n = 0 then w
else let n' = n - 1 in set_bits_aux n' (push_bit 1 w OR (if f n' then
1 else 0)))›
⟨*proof*⟩

end

end

2 More on bitwise operations on integers

theory More_Int
  imports Main
begin

lemma bin_nth_minus_Bit0[simp]:
  "0 < n ⟹ bit (numeral (num.Bit0 w) :: int) n = bit (numeral w :: int)
(n - 1)"
  ⟨*proof*⟩

lemma bin_nth_minus_Bit1[simp]:
  "0 < n ⟹ bit (numeral (num.Bit1 w) :: int) n = bit (numeral w :: int)
(n - 1)"
  ⟨*proof*⟩

lemma bin_cat_eq_push_bit_add_take_bit:
  ‹concat_bit n l k = push_bit n k + take_bit n l›
  ⟨*proof*⟩

lemma bin_cat_assoc: "(λk n l. concat_bit n l k) ((λk n l. concat_bit
n l k) x m y) n z = (λk n l. concat_bit n l k) x (m + n) ((λk n l. concat_bit
n l k) y n z)"
  ⟨*proof*⟩

lemma bin_cat_assoc_sym: "(λk n l. concat_bit n l k) x m ((λk n l. concat_bit
n l k) y n z) = (λk n l. concat_bit n l k) ((λk n l. concat_bit n l k)
x (m - n) y) (min m n) z"
  ⟨*proof*⟩

lemma bin_nth_cat:
  "(bit :: int ⇒ nat ⇒ bool) ((λk n l. concat_bit n l k) x k y) n =
    (if n < k then (bit :: int ⇒ nat ⇒ bool) y n else (bit :: int ⇒
nat ⇒ bool) x (n - k))"
  ⟨*proof*⟩

lemma bin_nth_drop_bit_iff:
  ‹(bit :: int ⇒ nat ⇒ bool) (drop_bit n c) k ⟷ (bit :: int ⇒ nat
⇒ bool) c (n + k)›
  ⟨*proof*⟩

lemma bin_nth_take_bit_iff:
  ‹(bit :: int ⇒ nat ⇒ bool) (take_bit n c) k ⟷ k < n ∧ (bit :: int
⇒ nat ⇒ bool) c k›
  ⟨*proof*⟩

lemma bin_cat_zero [simp]: "(λk n l. concat_bit n l k) 0 n w = (take_bit
:: nat ⇒ int ⇒ int) n w"

⟨*proof*⟩

lemma bintr_cat1: "(take_bit :: nat ⇒ int ⇒ int) (k + n) ((λk n l. concat_bit n l k) a n b) = (λk n l. concat_bit n l k) ((take_bit :: nat ⇒ int ⇒ int) k a) n b"
⟨*proof*⟩

lemma bintr_cat: "(take_bit :: nat ⇒ int ⇒ int) m ((λk n l. concat_bit n l k) a n b) =
(λk n l. concat_bit n l k) ((take_bit :: nat ⇒ int ⇒ int) (m - n) a) n ((take_bit :: nat ⇒ int ⇒ int) (min m n) b)"
⟨*proof*⟩

lemma bintr_cat_same [simp]: "(take_bit :: nat ⇒ int ⇒ int) n ((λk n l. concat_bit n l k) a n b) = (take_bit :: nat ⇒ int ⇒ int) n b"
⟨*proof*⟩

lemma cat_bintr [simp]: "(λk n l. concat_bit n l k) a n ((take_bit :: nat ⇒ int ⇒ int) n b) = (λk n l. concat_bit n l k) a n b"
⟨*proof*⟩

lemma drop_bit_bin_cat_eq:
‹drop_bit n ((λk n l. concat_bit n l k) v n w) = v›
⟨*proof*⟩

lemma take_bit_bin_cat_eq:
‹take_bit n ((λk n l. concat_bit n l k) v n w) = take_bit n w›
⟨*proof*⟩

lemma bin_cat_num: "(λk n l. concat_bit n l k) a n b = a * 2 ^ n + (take_bit :: nat ⇒ int ⇒ int) n b"
⟨*proof*⟩

lemma bin_cat_cong: "concat_bit n b a = concat_bit m d c"
if "n = m" "a = c" "take_bit m b = take_bit m d"
⟨*proof*⟩

lemma bin_cat_eqD1: "concat_bit n b a = concat_bit n d c ⟹ a = c"
⟨*proof*⟩

lemma bin_cat_eqD2: "concat_bit n b a = concat_bit n d c ⟹ take_bit n b = take_bit n d"
⟨*proof*⟩

lemma bin_cat_inj: "(concat_bit n b a) = concat_bit n d c ⟷ a = c ∧ take_bit n b = take_bit n d"
⟨*proof*⟩

lemma bin_last_def:

"(odd :: int ⇒ bool) w ⟷ w mod 2 = 1"
⟨*proof*⟩

lemma bin_last_numeral_simps [simp]:
"¬ odd (0 :: int)"
"odd (1 :: int)"
"odd (- 1 :: int)"
"odd (Numeral1 :: int)"
"¬ odd (numeral (Num.Bit0 w) :: int)"
"odd (numeral (Num.Bit1 w) :: int)"
"¬ odd (- numeral (Num.Bit0 w) :: int)"
"odd (- numeral (Num.Bit1 w) :: int)"
⟨*proof*⟩

lemma bin_rest_numeral_simps [simp]:
"(λk::int. k div 2) 0 = 0"
"(λk::int. k div 2) 1 = 0"
"(λk::int. k div 2) (- 1) = - 1"
"(λk::int. k div 2) Numeral1 = 0"
"(λk::int. k div 2) (numeral (Num.Bit0 w)) = numeral w"
"(λk::int. k div 2) (numeral (Num.Bit1 w)) = numeral w"
"(λk::int. k div 2) (- numeral (Num.Bit0 w)) = - numeral w"
"(λk::int. k div 2) (- numeral (Num.Bit1 w)) = - numeral (w + Num.One)"
⟨*proof*⟩

lemma bin_rl_eqI: "⟦(λk::int. k div 2) x = (λk::int. k div 2) y; odd x = odd y⟧ ⟹ x = y"
⟨*proof*⟩

lemma [simp]:
shows bin_rest_lt0: "(λk::int. k div 2) i < 0 ⟷ i < 0"
and bin_rest_ge_0: "(λk::int. k div 2) i ≥ 0 ⟷ i ≥ 0"
⟨*proof*⟩

lemma bin_rest_gt_0 [simp]: "(λk::int. k div 2) x > 0 ⟷ x > 1"
⟨*proof*⟩

lemma bin_nth_eq_iff: "(bit :: int ⇒ nat ⇒ bool) x = (bit :: int ⇒ nat ⇒ bool) y ⟷ x = y"
⟨*proof*⟩

lemma bin_eqI:
"x = y" **if** "⋀n. (bit :: int ⇒ nat ⇒ bool) x n ⟷ (bit :: int ⇒ nat ⇒ bool) y n"
⟨*proof*⟩

lemma bin_eq_iff: "x = y ⟷ (∀n. (bit :: int ⇒ nat ⇒ bool) x n = (bit :: int ⇒ nat ⇒ bool) y n)"
⟨*proof*⟩

lemma bin_nth_zero [simp]: "¬ (bit :: int ⇒ nat ⇒ bool) 0 n"
⟨*proof*⟩

lemma bin_nth_1 [simp]: "(bit :: int ⇒ nat ⇒ bool) 1 n ⟷ n = 0"
⟨*proof*⟩

lemma bin_nth_minus1 [simp]: "(bit :: int ⇒ nat ⇒ bool) (- 1) n"
⟨*proof*⟩

lemma bin_nth_numeral: "(λk::int. k div 2) x = y ⟹ (bit :: int ⇒ nat ⇒ bool) x (numeral n) = (bit :: int ⇒ nat ⇒ bool) y (pred_numeral n)"
⟨*proof*⟩

lemmas bin_nth_numeral_simps [simp] =
bin_nth_numeral [OF bin_rest_numeral_simps(8)]

lemmas bin_nth_simps =
bit_0 bit_Suc bin_nth_zero bin_nth_minus1
bin_nth_numeral_simps

lemma nth_2p_bin: "(bit :: int ⇒ nat ⇒ bool) (2 ^ n) m = (m = n)" — for use when simplifying with bin_nth_Bit
⟨*proof*⟩

lemma nth_rest_power_bin: "(bit :: int ⇒ nat ⇒ bool) (((λk::int. k div 2) ^^ k) w) n = (bit :: int ⇒ nat ⇒ bool) w (n + k)"
⟨*proof*⟩

lemma bin_nth_numeral_unfold:
"(bit :: int ⇒ nat ⇒ bool) (numeral (num.Bit0 x)) n ⟷ n > 0 ∧ (bit :: int ⇒ nat ⇒ bool) (numeral x) (n - 1)"
"(bit :: int ⇒ nat ⇒ bool) (numeral (num.Bit1 x)) n ⟷ (n > 0 ⟶ (bit :: int ⇒ nat ⇒ bool) (numeral x) (n - 1))"
⟨*proof*⟩

lemma bintrunc_mod2p: "(take_bit :: nat ⇒ int ⇒ int) n w = w mod 2 ^ n"
⟨*proof*⟩

lemma sbintrunc_mod2p: "(signed_take_bit :: nat ⇒ int ⇒ int) n w = (w + 2 ^ n) mod 2 ^ Suc n - 2 ^ n"
⟨*proof*⟩

lemma sbintrunc_eq_take_bit:
‹(signed_take_bit :: nat ⇒ int ⇒ int) n k = take_bit (Suc n) (k + 2 ^ n) - 2 ^ n›
⟨*proof*⟩

lemma bintrunc_n_0: "(take_bit :: nat ⇒ int ⇒ int) n 0 = 0"
⟨*proof*⟩

lemma sbintrunc_n_0: "(signed_take_bit :: nat ⇒ int ⇒ int) n 0 = 0"
⟨*proof*⟩

lemma sbintrunc_n_minus1: "(signed_take_bit :: nat ⇒ int ⇒ int) n (-1) = -1"
⟨*proof*⟩

lemma bintrunc_Suc_numeral:
"(take_bit :: nat ⇒ int ⇒ int) (Suc n) 1 = 1"
"(take_bit :: nat ⇒ int ⇒ int) (Suc n) (- 1) = 1 + 2 * (take_bit :: nat ⇒ int ⇒ int) n (- 1)"
"(take_bit :: nat ⇒ int ⇒ int) (Suc n) (numeral (Num.Bit0 w)) = 2 * (take_bit :: nat ⇒ int ⇒ int) n (numeral w)"
"(take_bit :: nat ⇒ int ⇒ int) (Suc n) (numeral (Num.Bit1 w)) = 1 + 2 * (take_bit :: nat ⇒ int ⇒ int) n (numeral w)"
"(take_bit :: nat ⇒ int ⇒ int) (Suc n) (- numeral (Num.Bit0 w)) = 2 * (take_bit :: nat ⇒ int ⇒ int) n (- numeral w)"
"(take_bit :: nat ⇒ int ⇒ int) (Suc n) (- numeral (Num.Bit1 w)) = 1 + 2 * (take_bit :: nat ⇒ int ⇒ int) n (- numeral (w + Num.One))"
⟨*proof*⟩

lemma sbintrunc_0_numeral [simp]:
"(signed_take_bit :: nat ⇒ int ⇒ int) 0 1 = -1"
"(signed_take_bit :: nat ⇒ int ⇒ int) 0 (numeral (Num.Bit0 w)) = 0"
"(signed_take_bit :: nat ⇒ int ⇒ int) 0 (numeral (Num.Bit1 w)) = -1"
"(signed_take_bit :: nat ⇒ int ⇒ int) 0 (- numeral (Num.Bit0 w)) = 0"
"(signed_take_bit :: nat ⇒ int ⇒ int) 0 (- numeral (Num.Bit1 w)) = -1"
⟨*proof*⟩

lemma sbintrunc_Suc_numeral:
"(signed_take_bit :: nat ⇒ int ⇒ int) (Suc n) 1 = 1"
"(signed_take_bit :: nat ⇒ int ⇒ int) (Suc n) (numeral (Num.Bit0 w)) = 2 * (signed_take_bit :: nat ⇒ int ⇒ int) n (numeral w)"
"(signed_take_bit :: nat ⇒ int ⇒ int) (Suc n) (numeral (Num.Bit1 w)) = 1 + 2 * (signed_take_bit :: nat ⇒ int ⇒ int) n (numeral w)"
"(signed_take_bit :: nat ⇒ int ⇒ int) (Suc n) (- numeral (Num.Bit0 w)) = 2 * (signed_take_bit :: nat ⇒ int ⇒ int) n (- numeral w)"
"(signed_take_bit :: nat ⇒ int ⇒ int) (Suc n) (- numeral (Num.Bit1 w)) = 1 + 2 * (signed_take_bit :: nat ⇒ int ⇒ int) n (- numeral (w + Num.One))"
⟨*proof*⟩

lemma nth_bintr: "(bit :: int ⇒ nat ⇒ bool) ((take_bit :: nat ⇒ int ⇒ int) m w) n ⟷ n < m ∧ (bit :: int ⇒ nat ⇒ bool) w n"

⟨*proof*⟩

lemma nth_sbintr: "(bit :: int ⇒ nat ⇒ bool) ((signed_take_bit :: nat ⇒ int ⇒ int) m w) n = (if n < m then (bit :: int ⇒ nat ⇒ bool) w n else (bit :: int ⇒ nat ⇒ bool) w m)"
⟨*proof*⟩

lemma bin_nth_Bit0:
"(bit :: int ⇒ nat ⇒ bool) (numeral (Num.Bit0 w)) n ⟷
(∃m. n = Suc m ∧ (bit :: int ⇒ nat ⇒ bool) (numeral w) m)"
⟨*proof*⟩

lemma bin_nth_Bit1:
"(bit :: int ⇒ nat ⇒ bool) (numeral (Num.Bit1 w)) n ⟷
n = 0 ∨ (∃m. n = Suc m ∧ (bit :: int ⇒ nat ⇒ bool) (numeral w) m)"
⟨*proof*⟩

lemma bintrunc_bintrunc_l: "n ≤ m ⟹ (take_bit :: nat ⇒ int ⇒ int) m ((take_bit :: nat ⇒ int ⇒ int) n w) = (take_bit :: nat ⇒ int ⇒ int) n w"
⟨*proof*⟩

lemma sbintrunc_sbintrunc_l: "n ≤ m ⟹ (signed_take_bit :: nat ⇒ int ⇒ int) m ((signed_take_bit :: nat ⇒ int ⇒ int) n w) = (signed_take_bit :: nat ⇒ int ⇒ int) n w"
⟨*proof*⟩

lemma bintrunc_bintrunc_ge: "n ≤ m ⟹ (take_bit :: nat ⇒ int ⇒ int) n ((take_bit :: nat ⇒ int ⇒ int) m w) = (take_bit :: nat ⇒ int ⇒ int) n w"
⟨*proof*⟩

lemma bintrunc_bintrunc_min [simp]: "(take_bit :: nat ⇒ int ⇒ int) m ((take_bit :: nat ⇒ int ⇒ int) n w) = (take_bit :: nat ⇒ int ⇒ int) (min m n) w"
⟨*proof*⟩

lemma sbintrunc_sbintrunc_min [simp]: "(signed_take_bit :: nat ⇒ int ⇒ int) m ((signed_take_bit :: nat ⇒ int ⇒ int) n w) = (signed_take_bit :: nat ⇒ int ⇒ int) (min m n) w"
⟨*proof*⟩

lemmas sbintrunc_Suc_Pls =
signed_take_bit_Suc [**where** a="0::int", simplified bin_last_numeral_simps bin_rest_numeral_simps]

lemmas sbintrunc_Suc_Min =
signed_take_bit_Suc [**where** a="-1::int", simplified bin_last_numeral_simps

```
bin_rest_numeral_simps]

lemmas sbintrunc_Sucs = sbintrunc_Suc_Pls sbintrunc_Suc_Min
  sbintrunc_Suc_numeral

lemmas sbintrunc_Pls =
  signed_take_bit_0 [where a="0::int", simplified bin_last_numeral_simps
bin_rest_numeral_simps]

lemmas sbintrunc_Min =
  signed_take_bit_0 [where a="-1::int", simplified bin_last_numeral_simps
bin_rest_numeral_simps]

lemmas sbintrunc_0_simps =
  sbintrunc_Pls sbintrunc_Min

lemmas sbintrunc_simps = sbintrunc_0_simps sbintrunc_Sucs

lemma bintrunc_minus: "0 < n ⟹ (take_bit :: nat ⇒ int ⇒ int) (Suc
(n - 1)) w = (take_bit :: nat ⇒ int ⇒ int) n w"
  ⟨proof⟩

lemma sbintrunc_minus: "0 < n ⟹ (signed_take_bit :: nat ⇒ int ⇒ int)
(Suc (n - 1)) w = (signed_take_bit :: nat ⇒ int ⇒ int) n w"
  ⟨proof⟩

lemmas sbintrunc_minus_simps =
  sbintrunc_Sucs [THEN [2] sbintrunc_minus [symmetric, THEN trans]]

lemma sbintrunc_BIT_I:
  ‹0 < n ⟹
  (signed_take_bit :: nat ⇒ int ⇒ int) (n - 1) 0 = y ⟹
  (signed_take_bit :: nat ⇒ int ⇒ int) n 0 = 2 * y›
  ⟨proof⟩

lemma sbintrunc_Suc_Is:
  ‹(signed_take_bit :: nat ⇒ int ⇒ int) n (- 1) = y ⟹
  (signed_take_bit :: nat ⇒ int ⇒ int) (Suc n) (- 1) = 1 + 2 * y›
  ⟨proof⟩

lemma sbintrunc_Suc_lem: "(signed_take_bit :: nat ⇒ int ⇒ int) (Suc
n) x = y ⟹ m = Suc n ⟹ (signed_take_bit :: nat ⇒ int ⇒ int) m x
= y"
  ⟨proof⟩

lemmas sbintrunc_Suc_Ialts =
  sbintrunc_Suc_Is [THEN sbintrunc_Suc_lem]

lemma sbintrunc_bintrunc_lt: "m > n ⟹ (signed_take_bit :: nat ⇒ int
```

⇒ int) n ((take_bit :: nat ⇒ int ⇒ int) m w) = (signed_take_bit ::
nat ⇒ int ⇒ int) n w"
⟨*proof*⟩

lemma bintrunc_sbintrunc_le: "m ≤ Suc n ⟹ (take_bit :: nat ⇒ int
⇒ int) m ((signed_take_bit :: nat ⇒ int ⇒ int) n w) = (take_bit ::
nat ⇒ int ⇒ int) m w"
⟨*proof*⟩

lemmas bintrunc_sbintrunc [simp] = order_refl [THEN bintrunc_sbintrunc_le]
lemmas sbintrunc_bintrunc [simp] = lessI [THEN sbintrunc_bintrunc_lt]
lemmas bintrunc_bintrunc [simp] = order_refl [THEN bintrunc_bintrunc_l]
lemmas sbintrunc_sbintrunc [simp] = order_refl [THEN sbintrunc_sbintrunc_l]

lemma bintrunc_sbintrunc' [simp]: "0 < n ⟹ (take_bit :: nat ⇒ int
⇒ int) n ((signed_take_bit :: nat ⇒ int ⇒ int) (n - 1) w) = (take_bit
:: nat ⇒ int ⇒ int) n w"
⟨*proof*⟩

lemma sbintrunc_bintrunc' [simp]: "0 < n ⟹ (signed_take_bit :: nat
⇒ int ⇒ int) (n - 1) ((take_bit :: nat ⇒ int ⇒ int) n w) = (signed_take_bit
:: nat ⇒ int ⇒ int) (n - 1) w"
⟨*proof*⟩

lemma bin_sbin_eq_iff: "(take_bit :: nat ⇒ int ⇒ int) (Suc n) x = (take_bit
:: nat ⇒ int ⇒ int) (Suc n) y ⟷ (signed_take_bit :: nat ⇒ int ⇒
int) n x = (signed_take_bit :: nat ⇒ int ⇒ int) n y"
⟨*proof*⟩

lemma bin_sbin_eq_iff':
"0 < n ⟹ (take_bit :: nat ⇒ int ⇒ int) n x = (take_bit :: nat ⇒
int ⇒ int) n y ⟷ (signed_take_bit :: nat ⇒ int ⇒ int) (n - 1) x =
(signed_take_bit :: nat ⇒ int ⇒ int) (n - 1) y"
⟨*proof*⟩

lemmas bintrunc_sbintruncS0 [simp] = bintrunc_sbintrunc' [unfolded One_nat_def]
lemmas sbintrunc_bintruncS0 [simp] = sbintrunc_bintrunc' [unfolded One_nat_def]

lemmas bintrunc_bintrunc_l' = le_add1 [THEN bintrunc_bintrunc_l]
lemmas sbintrunc_sbintrunc_l' = le_add1 [THEN sbintrunc_sbintrunc_l]

lemmas nat_non0_gr =
trans [OF iszero_def [THEN Not_eq_iff [THEN iffD2]] refl]

lemma bintrunc_numeral:
"(take_bit :: nat ⇒ int ⇒ int) (numeral k) x = of_bool (odd x) + 2
* (take_bit :: nat ⇒ int ⇒ int) (pred_numeral k) (x div 2)"

⟨*proof*⟩

lemma sbintrunc_numeral:
```
  "(signed_take_bit :: nat ⇒ int ⇒ int) (numeral k) x = of_bool (odd
x) + 2 * (signed_take_bit :: nat ⇒ int ⇒ int) (pred_numeral k) (x div
2)"
```
⟨*proof*⟩

lemma bintrunc_numeral_simps [simp]:
```
  "(take_bit :: nat ⇒ int ⇒ int) (numeral k) (numeral (Num.Bit0 w))
=
    2 * (take_bit :: nat ⇒ int ⇒ int) (pred_numeral k) (numeral w)"
  "(take_bit :: nat ⇒ int ⇒ int) (numeral k) (numeral (Num.Bit1 w))
=
    1 + 2 * (take_bit :: nat ⇒ int ⇒ int) (pred_numeral k) (numeral
w)"
  "(take_bit :: nat ⇒ int ⇒ int) (numeral k) (- numeral (Num.Bit0 w))
=
    2 * (take_bit :: nat ⇒ int ⇒ int) (pred_numeral k) (- numeral w)"
  "(take_bit :: nat ⇒ int ⇒ int) (numeral k) (- numeral (Num.Bit1 w))
=
    1 + 2 * (take_bit :: nat ⇒ int ⇒ int) (pred_numeral k) (- numeral
(w + Num.One))"
  "(take_bit :: nat ⇒ int ⇒ int) (numeral k) 1 = 1"
```
⟨*proof*⟩

lemma sbintrunc_numeral_simps [simp]:
```
  "(signed_take_bit :: nat ⇒ int ⇒ int) (numeral k) (numeral (Num.Bit0
w)) =
    2 * (signed_take_bit :: nat ⇒ int ⇒ int) (pred_numeral k) (numeral
w)"
  "(signed_take_bit :: nat ⇒ int ⇒ int) (numeral k) (numeral (Num.Bit1
w)) =
    1 + 2 * (signed_take_bit :: nat ⇒ int ⇒ int) (pred_numeral k) (numeral
w)"
  "(signed_take_bit :: nat ⇒ int ⇒ int) (numeral k) (- numeral (Num.Bit0
w)) =
    2 * (signed_take_bit :: nat ⇒ int ⇒ int) (pred_numeral k) (- numeral
w)"
  "(signed_take_bit :: nat ⇒ int ⇒ int) (numeral k) (- numeral (Num.Bit1
w)) =
    1 + 2 * (signed_take_bit :: nat ⇒ int ⇒ int) (pred_numeral k) (-
numeral (w + Num.One))"
  "(signed_take_bit :: nat ⇒ int ⇒ int) (numeral k) 1 = 1"
```
⟨*proof*⟩

lemma no_bintr_alt1: "(take_bit :: nat ⇒ int ⇒ int) n = (λw. w mod
2 ^ n :: int)"
⟨*proof*⟩

lemma range_bintrunc: "range ((take_bit :: nat ⇒ int ⇒ int) n) = {i. 0 ≤ i ∧ i < 2 ^ n}"
⟨*proof*⟩

lemma no_sbintr_alt2: "signed_take_bit n = (λw. (w + 2 ^ n) mod 2 ^ Suc n - 2 ^ n :: int)"
⟨*proof*⟩

lemma range_sbintrunc: "range ((signed_take_bit :: nat ⇒ int ⇒ int) n) = {i. - (2 ^ n) ≤ i ∧ i < 2 ^ n}"
⟨*proof*⟩

lemma sbintrunc_inc:
‹k + 2 ^ Suc n ≤ (signed_take_bit :: nat ⇒ int ⇒ int) n k› **if** ‹k < - (2 ^ n)›
⟨*proof*⟩

lemma sbintrunc_dec:
‹(signed_take_bit :: nat ⇒ int ⇒ int) n k ≤ k - 2 ^ (Suc n)› **if** ‹k ≥ 2 ^ n›
⟨*proof*⟩

lemma bintr_ge0: "0 ≤ (take_bit :: nat ⇒ int ⇒ int) n w"
⟨*proof*⟩

lemma bintr_lt2p: "(take_bit :: nat ⇒ int ⇒ int) n w < 2 ^ n"
⟨*proof*⟩

lemma bintr_Min: "(take_bit :: nat ⇒ int ⇒ int) n (- 1) = 2 ^ n - 1"
⟨*proof*⟩

lemma sbintr_ge: "- (2 ^ n) ≤ (signed_take_bit :: nat ⇒ int ⇒ int) n w"
⟨*proof*⟩

lemma sbintr_lt: "(signed_take_bit :: nat ⇒ int ⇒ int) n w < 2 ^ n"
⟨*proof*⟩

lemma bin_rest_trunc: "(λk::int. k div 2) ((take_bit :: nat ⇒ int ⇒ int) n bin) = (take_bit :: nat ⇒ int ⇒ int) (n - 1) ((λk::int. k div 2) bin)"
⟨*proof*⟩

lemma bin_rest_power_trunc:
"((λk::int. k div 2) ^^ k) ((take_bit :: nat ⇒ int ⇒ int) n bin) = (take_bit :: nat ⇒ int ⇒ int) (n - k) (((λk::int. k div 2) ^^ k) bin)"
⟨*proof*⟩

lemma bin_rest_trunc_i: "(take_bit :: nat ⇒ int ⇒ int) n ((λk::int. k div 2) bin) = (λk::int. k div 2) ((take_bit :: nat ⇒ int ⇒ int) (Suc n) bin)"
⟨*proof*⟩

lemma bin_rest_strunc: "(λk::int. k div 2) ((signed_take_bit :: nat ⇒ int ⇒ int) (Suc n) bin) = (signed_take_bit :: nat ⇒ int ⇒ int) n ((λk::int. k div 2) bin)"
⟨*proof*⟩

lemma bintrunc_rest [simp]: "(take_bit :: nat ⇒ int ⇒ int) n ((λk::int. k div 2) ((take_bit :: nat ⇒ int ⇒ int) n bin)) = (λk::int. k div 2) ((take_bit :: nat ⇒ int ⇒ int) n bin)"
⟨*proof*⟩

lemma sbintrunc_rest [simp]: "(signed_take_bit :: nat ⇒ int ⇒ int) n ((λk::int. k div 2) ((signed_take_bit :: nat ⇒ int ⇒ int) n bin)) = (λk::int. k div 2) ((signed_take_bit :: nat ⇒ int ⇒ int) n bin)"
⟨*proof*⟩

lemma bintrunc_rest': "(take_bit :: nat ⇒ int ⇒ int) n ∘ (λk::int. k div 2) ∘ (take_bit :: nat ⇒ int ⇒ int) n = (λk::int. k div 2) ∘ (take_bit :: nat ⇒ int ⇒ int) n"
⟨*proof*⟩

lemma sbintrunc_rest': "(signed_take_bit :: nat ⇒ int ⇒ int) n ∘ (λk::int. k div 2) ∘ (signed_take_bit :: nat ⇒ int ⇒ int) n = (λk::int. k div 2) ∘ (signed_take_bit :: nat ⇒ int ⇒ int) n"
⟨*proof*⟩

lemma rco_lem:
assumes "f ∘ g ∘ f = g ∘ f"
shows "f ∘ (g ∘ f) ^^ n = g ^^ n ∘ f"
⟨*proof*⟩

lemmas rco_bintr = bintrunc_rest'
[THEN rco_lem [THEN fun_cong], unfolded o_def]
lemmas rco_sbintr = sbintrunc_rest'
[THEN rco_lem [THEN fun_cong], unfolded o_def]

context
includes bit_operations_syntax
begin

lemmas int_not_def = not_int_def

lemma int_not_simps:
"NOT (0::int) = -1"
"NOT (1::int) = -2"

```
  "NOT (- 1::int) = 0"
  "NOT (numeral w::int) = - numeral (w + Num.One)"
  "NOT (- numeral (Num.Bit0 w)::int) = numeral (Num.BitM w)"
  "NOT (- numeral (Num.Bit1 w)::int) = numeral (Num.Bit0 w)"
```
⟨*proof*⟩

lemma `int_not_not: "NOT (NOT x) = x"`
for `x :: int`
⟨*proof*⟩

lemma `int_and_0 [simp]: "0 AND x = 0"`
for `x :: int`
⟨*proof*⟩

lemma `int_and_m1 [simp]: "-1 AND x = x"`
for `x :: int`
⟨*proof*⟩

lemma `int_or_zero [simp]: "0 OR x = x"`
for `x :: int`
⟨*proof*⟩

lemma `int_or_minus1 [simp]: "-1 OR x = -1"`
for `x :: int`
⟨*proof*⟩

lemma `int_xor_zero [simp]: "0 XOR x = x"`
for `x :: int`
⟨*proof*⟩

lemma `bin_rest_NOT [simp]: "(λk::int. k div 2) (NOT x) = NOT ((λk::int. k div 2) x)"`
⟨*proof*⟩

lemma `bin_last_NOT [simp]: "(odd :: int ⇒ bool) (NOT x) ⟷ ¬ (odd :: int ⇒ bool) x"`
⟨*proof*⟩

lemma `bin_rest_AND [simp]: "(λk::int. k div 2) (x AND y) = (λk::int. k div 2) x AND (λk::int. k div 2) y"`
⟨*proof*⟩

lemma `bin_last_AND [simp]: "(odd :: int ⇒ bool) (x AND y) ⟷ (odd :: int ⇒ bool) x ∧ (odd :: int ⇒ bool) y"`
⟨*proof*⟩

lemma `bin_rest_OR [simp]: "(λk::int. k div 2) (x OR y) = (λk::int. k div 2) x OR (λk::int. k div 2) y"`
⟨*proof*⟩

lemma bin_last_OR [simp]: "(odd :: int ⇒ bool) (x OR y) ⟷ (odd :: int ⇒ bool) x ∨ (odd :: int ⇒ bool) y"
⟨*proof*⟩

lemma bin_rest_XOR [simp]: "(λk::int. k div 2) (x XOR y) = (λk::int. k div 2) x XOR (λk::int. k div 2) y"
⟨*proof*⟩

lemma bin_last_XOR [simp]: "(odd :: int ⇒ bool) (x XOR y) ⟷ ((odd :: int ⇒ bool) x ∨ (odd :: int ⇒ bool) y) ∧ ¬ ((odd :: int ⇒ bool) x ∧ (odd :: int ⇒ bool) y)"
⟨*proof*⟩

lemma bin_nth_ops:
"⋀x y. (bit :: int ⇒ nat ⇒ bool) (x AND y) n ⟷ (bit :: int ⇒ nat ⇒ bool) x n ∧ (bit :: int ⇒ nat ⇒ bool) y n"
"⋀x y. (bit :: int ⇒ nat ⇒ bool) (x OR y) n ⟷ (bit :: int ⇒ nat ⇒ bool) x n ∨ (bit :: int ⇒ nat ⇒ bool) y n"
"⋀x y. (bit :: int ⇒ nat ⇒ bool) (x XOR y) n ⟷ (bit :: int ⇒ nat ⇒ bool) x n ≠ (bit :: int ⇒ nat ⇒ bool) y n"
"⋀x. (bit :: int ⇒ nat ⇒ bool) (NOT x) n ⟷ ¬ (bit :: int ⇒ nat ⇒ bool) x n"
⟨*proof*⟩

lemma int_xor_minus1 [simp]: "-1 XOR x = NOT x"
for x :: int
⟨*proof*⟩

lemma int_xor_extra_simps [simp]:
"w XOR 0 = w"
"w XOR -1 = NOT w"
for w :: int
⟨*proof*⟩

lemma int_or_extra_simps [simp]:
"w OR 0 = w"
"w OR -1 = -1"
for w :: int
⟨*proof*⟩

lemma int_and_extra_simps [simp]:
"w AND 0 = 0"
"w AND -1 = w"
for w :: int
⟨*proof*⟩

Commutativity of the above.

lemma bin_ops_comm:

fixes x y :: int
shows int_and_comm: "x AND y = y AND x"
and int_or_comm: "x OR y = y OR x"
and int_xor_comm: "x XOR y = y XOR x"
⟨*proof*⟩

lemma bin_ops_same [simp]:
"x AND x = x"
"x OR x = x"
"x XOR x = 0"
for x :: int
⟨*proof*⟩

lemmas bin_log_esimps =
int_and_extra_simps int_or_extra_simps int_xor_extra_simps
int_and_0 int_and_m1 int_or_zero int_or_minus1 int_xor_zero int_xor_minus1

lemma bbw_ao_absorb: "x AND (y OR x) = x ∧ x OR (y AND x) = x"
for x y :: int
⟨*proof*⟩

lemma bbw_ao_absorbs_other:
"x AND (x OR y) = x ∧ (y AND x) OR x = x"
"(y OR x) AND x = x ∧ x OR (x AND y) = x"
"(x OR y) AND x = x ∧ (x AND y) OR x = x"
for x y :: int
⟨*proof*⟩

lemmas bbw_ao_absorbs [simp] = bbw_ao_absorb bbw_ao_absorbs_other

lemma int_xor_not: "(NOT x) XOR y = NOT (x XOR y) ∧ x XOR (NOT y) =
NOT (x XOR y)"
for x y :: int
⟨*proof*⟩

lemma int_and_assoc: "(x AND y) AND z = x AND (y AND z)"
for x y z :: int
⟨*proof*⟩

lemma int_or_assoc: "(x OR y) OR z = x OR (y OR z)"
for x y z :: int
⟨*proof*⟩

lemma int_xor_assoc: "(x XOR y) XOR z = x XOR (y XOR z)"
for x y z :: int
⟨*proof*⟩

lemmas bbw_assocs = int_and_assoc int_or_assoc int_xor_assoc

lemma bbw_lcs [simp]:
"y AND (x AND z) = x AND (y AND z)"
"y OR (x OR z) = x OR (y OR z)"
"y XOR (x XOR z) = x XOR (y XOR z)"
for x y :: int
⟨*proof*⟩

lemma bbw_not_dist:
"NOT (x OR y) = (NOT x) AND (NOT y)"
"NOT (x AND y) = (NOT x) OR (NOT y)"
for x y :: int
⟨*proof*⟩

lemma bbw_oa_dist: "(x AND y) OR z = (x OR z) AND (y OR z)"
for x y z :: int
⟨*proof*⟩

lemma bbw_ao_dist: "(x OR y) AND z = (x AND z) OR (y AND z)"
for x y z :: int
⟨*proof*⟩

Cases for 0 and -1 are already covered by other simp rules.

lemma bin_rest_neg_numeral_BitM [simp]:
"(λk::int. k div 2) (- numeral (Num.BitM w)) = - numeral w"
⟨*proof*⟩

lemma bin_last_neg_numeral_BitM [simp]:
"(odd :: int ⇒ bool) (- numeral (Num.BitM w))"
⟨*proof*⟩

Interaction between bit-wise and arithmetic: good example of bin_induction.

lemma bin_add_not: "x + NOT x = (-1::int)"
⟨*proof*⟩

lemma AND_mod: "x AND (2 ^ n - 1) = x mod 2 ^ n"
for x :: int
⟨*proof*⟩

lemma bin_trunc_ao:
"(take_bit :: nat ⇒ int ⇒ int) n x AND (take_bit :: nat ⇒ int ⇒ int) n y = (take_bit :: nat ⇒ int ⇒ int) n (x AND y)"
"(take_bit :: nat ⇒ int ⇒ int) n x OR (take_bit :: nat ⇒ int ⇒ int) n y = (take_bit :: nat ⇒ int ⇒ int) n (x OR y)"
⟨*proof*⟩

lemma bin_trunc_xor: "(take_bit :: nat ⇒ int ⇒ int) n ((take_bit :: nat ⇒ int ⇒ int) n x XOR (take_bit :: nat ⇒ int ⇒ int) n y) = (take_bit :: nat ⇒ int ⇒ int) n (x XOR y)"

⟨*proof*⟩

lemma bin_trunc_not: "(take_bit :: nat ⇒ int ⇒ int) n (NOT ((take_bit :: nat ⇒ int ⇒ int) n x)) = (take_bit :: nat ⇒ int ⇒ int) n (NOT x)"
⟨*proof*⟩

Want theorems of the form of bin_trunc_xor.

lemma bintr_bintr_i: "x = (take_bit :: nat ⇒ int ⇒ int) n y ⟹ (take_bit :: nat ⇒ int ⇒ int) n x = (take_bit :: nat ⇒ int ⇒ int) n y"
⟨*proof*⟩

lemmas bin_trunc_and = bin_trunc_ao(1) [THEN bintr_bintr_i]
lemmas bin_trunc_or = bin_trunc_ao(2) [THEN bintr_bintr_i]

lemma not_int_cmp_0 [simp]:
fixes i :: int **shows**
"0 < NOT i ⟷ i < -1"
"0 ≤ NOT i ⟷ i < 0"
"NOT i < 0 ⟷ i ≥ 0"
"NOT i ≤ 0 ⟷ i ≥ -1"
⟨*proof*⟩

lemma bbw_ao_dist2: "(x :: int) AND (y OR z) = x AND y OR x AND z"
⟨*proof*⟩

lemmas int_and_ac = bbw_lcs(1) int_and_comm int_and_assoc

lemma int_nand_same [simp]: **fixes** x :: int **shows** "x AND NOT x = 0"
⟨*proof*⟩

lemma int_nand_same_middle: **fixes** x :: int **shows** "x AND y AND NOT x = 0"
⟨*proof*⟩

lemma and_xor_dist: **fixes** x :: int **shows**
"x AND (y XOR z) = (x AND y) XOR (x AND z)"
⟨*proof*⟩

lemma int_and_lt0 [simp]:
‹x AND y < 0 ⟷ x < 0 ∧ y < 0› **for** x y :: int
⟨*proof*⟩

lemma int_and_ge0 [simp]:
‹x AND y ≥ 0 ⟷ x ≥ 0 ∨ y ≥ 0› **for** x y :: int
⟨*proof*⟩

lemma int_and_1: **fixes** x :: int **shows** "x AND 1 = x mod 2"
⟨*proof*⟩

lemma int_1_and: **fixes** x :: int **shows** "1 AND x = x mod 2"
⟨*proof*⟩

lemma int_or_lt0 [simp]:
‹x OR y < 0 ⟷ x < 0 ∨ y < 0› **for** x y :: int
⟨*proof*⟩

lemma int_or_ge0 [simp]:
‹x OR y ≥ 0 ⟷ x ≥ 0 ∧ y ≥ 0› **for** x y :: int
⟨*proof*⟩

lemma int_xor_lt0 [simp]:
‹x XOR y < 0 ⟷ (x < 0) ≠ (y < 0)› **for** x y :: int
⟨*proof*⟩

lemma int_xor_ge0 [simp]:
‹x XOR y ≥ 0 ⟷ (x ≥ 0 ⟷ y ≥ 0)› **for** x y :: int
⟨*proof*⟩

lemma even_conv_AND:
‹even i ⟷ i AND 1 = 0› **for** i :: int
⟨*proof*⟩

lemma bin_last_conv_AND:
"(odd :: int ⇒ bool) i ⟷ i AND 1 ≠ 0"
⟨*proof*⟩

lemma bitval_bin_last:
"of_bool ((odd :: int ⇒ bool) i) = i AND 1"
⟨*proof*⟩

lemma int_not_neg_numeral: "NOT (- numeral n) = (Num.sub n num.One :: int)"
⟨*proof*⟩

lemma int_neg_numeral_pOne_conv_not: "- numeral (n + num.One) = (NOT (numeral n) :: int)"
⟨*proof*⟩

lemma int_0_shiftl: "push_bit n 0 = (0 :: int)"
⟨*proof*⟩

lemma bin_last_shiftl: "odd (push_bit n x) ⟷ n = 0 ∧ (odd :: int ⇒ bool) x"
⟨*proof*⟩

lemma bin_rest_shiftl: "(λk::int. k div 2) (push_bit n x) = (if n > 0 then push_bit (n - 1) x else (λk::int. k div 2) x)"
⟨*proof*⟩

lemma bin_nth_shiftl: "(bit :: int ⇒ nat ⇒ bool) (push_bit n x) m ⟷ n ≤ m ∧ (bit :: int ⇒ nat ⇒ bool) x (m - n)"
⟨*proof*⟩

lemma bin_last_shiftr: "odd (drop_bit n x) ⟷ bit x n" **for** x :: int
⟨*proof*⟩

lemma bin_rest_shiftr: "(λk::int. k div 2) (drop_bit n x) = drop_bit (Suc n) x"
⟨*proof*⟩

lemma bin_nth_shiftr: "(bit :: int ⇒ nat ⇒ bool) (drop_bit n x) m = (bit :: int ⇒ nat ⇒ bool) x (n + m)"
⟨*proof*⟩

lemma bin_nth_conv_AND:
fixes x :: int **shows**
"(bit :: int ⇒ nat ⇒ bool) x n ⟷ x AND (push_bit n 1) ≠ 0"
⟨*proof*⟩

lemma int_shiftl_numeral [simp]:
"push_bit (numeral w') (numeral w :: int) = push_bit (pred_numeral w') (numeral (num.Bit0 w))"
"push_bit (numeral w') (- numeral w :: int) = push_bit (pred_numeral w') (- numeral (num.Bit0 w))"
⟨*proof*⟩

lemma int_shiftl_One_numeral [simp]:
"push_bit (numeral w) (1::int) = push_bit (pred_numeral w) 2"
⟨*proof*⟩

lemma shiftl_ge_0: **fixes** i :: int **shows** "push_bit n i ≥ 0 ⟷ i ≥ 0"
⟨*proof*⟩

lemma shiftl_lt_0: **fixes** i :: int **shows** "push_bit n i < 0 ⟷ i < 0"
⟨*proof*⟩

lemma int_shiftl_test_bit: "bit (push_bit i n :: int) m ⟷ m ≥ i ∧ bit n (m - i)"
⟨*proof*⟩

lemma int_0shiftr: "drop_bit x (0 :: int) = 0"
⟨*proof*⟩

lemma int_minus1_shiftr: "drop_bit x (-1 :: int) = -1"
⟨*proof*⟩

lemma int_shiftr_ge_0: **fixes** i :: int **shows** "drop_bit n i ≥ 0 ⟷ i

```
≥ 0"
```
⟨*proof*⟩

lemma int_shiftr_lt_0 [simp]: **fixes** i :: int **shows** "drop_bit n i < 0
⟷ i < 0"
⟨*proof*⟩

lemma int_shiftr_numeral [simp]:
"drop_bit (numeral w') (1 :: int) = 0"
"drop_bit (numeral w') (numeral num.One :: int) = 0"
"drop_bit (numeral w') (numeral (num.Bit0 w) :: int) = drop_bit (pred_numeral
w') (numeral w)"
"drop_bit (numeral w') (numeral (num.Bit1 w) :: int) = drop_bit (pred_numeral
w') (numeral w)"
"drop_bit (numeral w') (- numeral (num.Bit0 w) :: int) = drop_bit (pred_numeral
w') (- numeral w)"
"drop_bit (numeral w') (- numeral (num.Bit1 w) :: int) = drop_bit (pred_numeral
w') (- numeral (Num.inc w))"
⟨*proof*⟩

lemma int_shiftr_numeral_Suc0 [simp]:
"drop_bit (Suc 0) (1 :: int) = 0"
"drop_bit (Suc 0) (numeral num.One :: int) = 0"
"drop_bit (Suc 0) (numeral (num.Bit0 w) :: int) = numeral w"
"drop_bit (Suc 0) (numeral (num.Bit1 w) :: int) = numeral w"
"drop_bit (Suc 0) (- numeral (num.Bit0 w) :: int) = - numeral w"
"drop_bit (Suc 0) (- numeral (num.Bit1 w) :: int) = - numeral (Num.inc
w)"
⟨*proof*⟩

lemmas bin_log_bintrs = bin_trunc_not bin_trunc_xor bin_trunc_and bin_trunc_or

lemma bintrunc_shiftl:
"take_bit n (push_bit i m) = push_bit i (take_bit (n - i) m)"
for m :: int
⟨*proof*⟩

lemma bin_mask_conv_pow2:
"mask n = 2 ^ n - (1 :: int)"
⟨*proof*⟩

lemma bin_mask_ge0: "mask n ≥ (0 :: int)"
⟨*proof*⟩

context
includes bit_operations_syntax
begin

lemma and_bin_mask_conv_mod: "x AND mask n = x mod 2 ^ n"

for x :: int
⟨*proof*⟩

end

end

lemma bin_mask_numeral:
"mask (numeral n) = (1 :: int) + 2 * mask (pred_numeral n)"
⟨*proof*⟩

lemma bin_nth_mask: "bit (mask n :: int) i ⟷ i < n"
⟨*proof*⟩

lemma bin_mask_p1_conv_shift: "mask n + 1 = push_bit n (1 :: int)"
⟨*proof*⟩

lemma sbintrunc_eq_in_range:
"((signed_take_bit :: nat ⇒ int ⇒ int) n x = x) = (x ∈ range ((signed_take_bit
:: nat ⇒ int ⇒ int) n))"
"(x = (signed_take_bit :: nat ⇒ int ⇒ int) n x) = (x ∈ range ((signed_take_bit
:: nat ⇒ int ⇒ int) n))"
⟨*proof*⟩

lemma sbintrunc_If:
"- 3 * (2 ^ n) ≤ x ∧ x < 3 * (2 ^ n)
⟹ signed_take_bit n x = (if x < - (2 ^ n) then x + 2 * (2 ^ n)
else if x ≥ 2 ^ n then x - 2 * (2 ^ n) else x)" **for** x :: int
⟨*proof*⟩

lemma bintrunc_id:
"⟦m ≤ int n; 0 < m⟧ ⟹ take_bit n m = m"
⟨*proof*⟩

end

theory More_Bit_Ring
imports Main
begin

context semiring_bit_operations
begin

context
includes bit_operations_syntax

begin

lemma disjunctive_add2:
"(x AND y) = 0 ⟹ x + y = x OR y"
⟨*proof*⟩

end
end

context ring_bit_operations
begin

context
includes bit_operations_syntax
begin

lemma not_xor_is_eqv:
"NOT (x XOR y) = (x AND y) OR (NOT x AND NOT y)"
⟨*proof*⟩

lemma not_xor_eq_xor_not:
"(NOT x) XOR y = x XOR (NOT y)"
⟨*proof*⟩

lemma not_minus:
"NOT (x - y) = y - x - 1"
⟨*proof*⟩

lemma minus_not_eq_plus_1:
"- NOT x = x + 1"
⟨*proof*⟩

lemma not_minus_eq_minus_1:
"NOT (- x) = x - 1"
⟨*proof*⟩

lemma and_plus_not_and:
"(x AND y) + (x AND NOT y) = x"
⟨*proof*⟩

lemma or_eq_and_not_plus:
"x OR y = (x AND NOT y) + y"
⟨*proof*⟩

lemma and_eq_not_or_minus:
"x AND y = (NOT x OR y) - NOT x"
⟨*proof*⟩

lemma and_not_eq_or_minus:

```
  "x AND NOT y = (x OR y) - y"
```
⟨*proof*⟩

lemma and_not_eq_minus_and:
```
  "x AND NOT y = x - (x AND y)"
```
⟨*proof*⟩

lemma or_minus_eq_minus_and:
```
  "(x OR y) - y = x - (x AND y)"
```
⟨*proof*⟩

lemma plus_eq_and_or:
```
  "x + y = (x OR y) + (x AND y)"
```
⟨*proof*⟩

lemma xor_eq_or_minus_and:
```
  "x XOR y = (x OR y) - (x AND y)"
```
⟨*proof*⟩

lemma not_xor_eq_and_plus_not_or:
```
  "NOT (x XOR y) = (x AND y) + NOT (x OR y)"
```
⟨*proof*⟩

lemma not_xor_eq_and_minus_or:
```
  "NOT (x XOR y) = (x AND y) - (x OR y) - 1"
```
⟨*proof*⟩

lemma plus_eq_xor_plus_carry:
```
  "x + y = (x XOR y) + 2 * (x AND y)"
```
⟨*proof*⟩

lemma plus_eq_or_minus_xor:
```
  "x + y = 2 * (x OR y) - (x XOR y)"
```
⟨*proof*⟩

lemma plus_eq_minus_neg:
```
  "x + y = x - NOT y - 1"
```
⟨*proof*⟩

lemma minus_eq_plus_neg:
```
  "x - y = x + NOT y + 1"
```
⟨*proof*⟩

lemma minus_eq_and_not_minus_not_and:
```
  "x - y = (x AND NOT y) - (NOT x AND y)"
```
⟨*proof*⟩

lemma minus_eq_xor_minus_not_and:
```
  "x - y = (x XOR y) - 2 * (NOT x AND y)"
```

⟨*proof*⟩

lemma minus_eq_and_not_minus_xor:
"x - y = 2 * (x AND NOT y) - (x XOR y)"
⟨*proof*⟩

lemma and_one_neq_simps[simp]:
"x AND 1 ≠ 0 ⟷ x AND 1 = 1"
"x AND 1 ≠ 1 ⟷ x AND 1 = 0"
⟨*proof*⟩

end
end

end

3 Lemmas on words

theory More_Word
imports
"HOL-Library.Word" More_Arithmetic More_Divides More_Bit_Ring
begin

context
includes bit_operations_syntax
begin

— problem posed by TPHOLs referee: criterion for overflow of addition of signed integers

lemma sofl_test:
‹sint x + sint y = sint (x + y) ⟷
drop_bit (size x - 1) ((x + y XOR x) AND (x + y XOR y)) = 0›
for x y :: ‹'a::len word›
⟨*proof*⟩

lemma unat_power_lower [simp]:
"unat ((2::'a::len word) ^ n) = 2 ^ n" **if** "n < LENGTH('a::len)"
⟨*proof*⟩

lemma unat_p2: "n < LENGTH('a :: len) ⟹ unat (2 ^ n :: 'a word) = 2 ^ n"
⟨*proof*⟩

lemma word_div_lt_eq_0:
"x < y ⟹ x div y = 0" **for** x :: "'a :: len word"
⟨*proof*⟩

lemma word_div_eq_1_iff: "n div m = 1 ⟷ n ≥ m ∧ unat n < 2 * unat

```
(m :: 'a :: len word)"
```
⟨*proof*⟩

lemma AND_twice [simp]:
```
  "(w AND m) AND m = w AND m"
```
⟨*proof*⟩

lemma word_combine_masks:
```
  "w AND m = z ⟹ w AND m' = z' ⟹ w AND (m OR m') = (z OR z')"
  for w m m' z z' :: ‹'a::len word›
```
⟨*proof*⟩

lemma p2_gt_0:
```
  "(0 < (2 ^ n :: 'a :: len word)) = (n < LENGTH('a))"
```
⟨*proof*⟩

lemma uint_2p_alt:
```
  ‹n < LENGTH('a::len) ⟹ uint ((2::'a::len word) ^ n) = 2 ^ n›
```
⟨*proof*⟩

lemma p2_eq_0:
```
  ‹(2::'a::len word) ^ n = 0 ⟷ LENGTH('a::len) ≤ n›
```
⟨*proof*⟩

lemma p2len:
```
  ‹(2 :: 'a word) ^ LENGTH('a::len) = 0›
```
⟨*proof*⟩

lemma neg_mask_is_div:
```
  "w AND NOT (mask n) = (w div 2^n) * 2^n"
  for w :: ‹'a::len word›
```
⟨*proof*⟩

lemma neg_mask_is_div':
```
  "n < size w ⟹ w AND NOT (mask n) = ((w div (2 ^ n)) * (2 ^ n))"
  for w :: ‹'a::len word›
```
⟨*proof*⟩

lemma and_mask_arith:
```
  "w AND mask n = (w * 2^(size w - n)) div 2^(size w - n)"
  for w :: ‹'a::len word›
```
⟨*proof*⟩

lemma and_mask_arith':
```
  "0 < n ⟹ w AND mask n = (w * 2^(size w - n)) div 2^(size w - n)"
  for w :: ‹'a::len word›
```
⟨*proof*⟩

lemma mask_2pm1: "mask n = 2 ^ n - (1 :: 'a::len word)"

⟨*proof*⟩

lemma add_mask_fold:
"x + 2 ^ n - 1 = x + mask n"
for x :: ‹'a::len word›
⟨*proof*⟩

lemma word_and_mask_le_2pm1: "w AND mask n ≤ 2 ^ n - 1"
for w :: ‹'a::len word›
⟨*proof*⟩

lemma is_aligned_AND_less_0:
"u AND mask n = 0 ⟹ v < 2^n ⟹ u AND v = 0"
for u v :: ‹'a::len word›
⟨*proof*⟩

lemma and_mask_eq_iff_le_mask:
‹w AND mask n = w ⟷ w ≤ mask n›
for w :: ‹'a::len word›
⟨*proof*⟩

lemma less_eq_mask_iff_take_bit_eq_self:
‹w ≤ mask n ⟷ take_bit n w = w›
for w :: ‹'a::len word›
⟨*proof*⟩

lemma NOT_eq:
"NOT (x :: 'a :: len word) = - x - 1"
⟨*proof*⟩

lemma NOT_mask: "NOT (mask n :: 'a::len word) = - (2 ^ n)"
⟨*proof*⟩

lemma le_m1_iff_lt: "(x > (0 :: 'a :: len word)) = ((y ≤ x - 1) = (y < x))"
⟨*proof*⟩

lemma gt0_iff_gem1: ‹0 < x ⟷ x - 1 < x› **for** x :: ‹'a::len word›
⟨*proof*⟩

lemma power_2_ge_iff:
‹2 ^ n - (1 :: 'a::len word) < 2 ^ n ⟷ n < LENGTH('a)›
⟨*proof*⟩

lemma le_mask_iff_lt_2n:
"n < len_of TYPE ('a) = (((w :: 'a :: len word) ≤ mask n) = (w < 2 ^ n))"
⟨*proof*⟩

lemma mask_lt_2pn:
  ‹n < LENGTH('a) ⟹ mask n < (2 :: 'a::len word) ^ n›
  ⟨*proof*⟩

lemma word_unat_power:
  "(2 :: 'a :: len word) ^ n = of_nat (2 ^ n)"
  ⟨*proof*⟩

lemma of_nat_mono_maybe:
  assumes xlt: "x < 2 ^ len_of TYPE ('a)"
  shows "y < x ⟹ of_nat y < (of_nat x :: 'a :: len word)"
  ⟨*proof*⟩

lemma word_and_max_word:
  fixes a:: "'a::len word"
  shows "x = - 1 ⟹ a AND x = a"
  ⟨*proof*⟩

lemma word_and_full_mask_simp [simp]:
  ‹x AND mask LENGTH('a) = x› **for** x :: ‹'a::len word›
  ⟨*proof*⟩

lemma of_int_uint [simp]: "of_int (uint x) = x"
  ⟨*proof*⟩

corollary word_plus_and_or_coroll:
  "x AND y = 0 ⟹ x + y = x OR y"
  for x y :: ‹'a::len word›
  ⟨*proof*⟩

corollary word_plus_and_or_coroll2:
  "(x AND w) + (x AND NOT w) = x"
  for x w :: ‹'a::len word›
  ⟨*proof*⟩

lemma unat_mask_eq:
  ‹unat (mask n :: 'a::len word) = mask (min LENGTH('a) n)›
  ⟨*proof*⟩

lemma word_plus_mono_left:
  fixes x :: "'a :: len word"
  shows "⟦y ≤ z; x ≤ x + z⟧ ⟹ y + x ≤ z + x"
  ⟨*proof*⟩

lemma less_Suc_unat_less_bound:
  "n < Suc (unat (x :: 'a :: len word)) ⟹ n < 2 ^ LENGTH('a)"
  ⟨*proof*⟩

lemma up_ucast_inj:

"⟦ ucast x = (ucast y::'b::len word); LENGTH('a) ≤ len_of TYPE ('b)
⟧ ⟹ x = (y::'a::len word)"
⟨*proof*⟩

lemmas ucast_up_inj = up_ucast_inj

lemma up_ucast_inj_eq:
"LENGTH('a) ≤ len_of TYPE ('b) ⟹ (ucast x = (ucast y::'b::len word))
= (x = (y::'a::len word))"
⟨*proof*⟩

lemma no_plus_overflow_neg:
"(x :: 'a :: len word) < -y ⟹ x ≤ x + y"
⟨*proof*⟩

lemma ucast_ucast_eq:
"⟦ ucast x = (ucast (ucast y::'a word)::'c::len word); LENGTH('a) ≤
LENGTH('b);
LENGTH('b) ≤ LENGTH('c) ⟧ ⟹
x = ucast y" **for** x :: "'a::len word" **and** y :: "'b::len word"
⟨*proof*⟩

lemma ucast_0_I:
"x = 0 ⟹ ucast x = 0"
⟨*proof*⟩

lemma word_add_offset_less:
fixes x :: "'a :: len word"
assumes yv: "y < 2 ^ n"
and xv: "x < 2 ^ m"
and mnv: "sz < LENGTH('a :: len)"
and xv': "x < 2 ^ (LENGTH('a :: len) - n)"
and mn: "sz = m + n"
shows "x * 2 ^ n + y < 2 ^ sz"
⟨*proof*⟩

lemma word_less_power_trans:
fixes n :: "'a :: len word"
assumes "n < 2 ^ (m - k)" "k ≤ m" "m < len_of TYPE ('a)"
shows "2 ^ k * n < 2 ^ m"
⟨*proof*⟩

lemma word_less_power_trans2:
fixes n :: "'a::len word"
shows "⟦n < 2 ^ (m - k); k ≤ m; m < LENGTH('a)⟧ ⟹ n * 2 ^ k < 2 ^
m"
⟨*proof*⟩

lemma Suc_unat_diff_1:

fixes x :: "'a :: len word"
assumes lt: "1 ≤ x"
shows "Suc (unat (x - 1)) = unat x"
⟨*proof*⟩

lemma word_eq_unatI:
‹v = w› **if** ‹unat v = unat w›
⟨*proof*⟩

lemma word_div_sub:
fixes x :: "'a :: len word"
assumes "y ≤ x" "0 < y"
shows "(x - y) div y = x div y - 1"
⟨*proof*⟩

lemma word_mult_less_mono1:
fixes i :: "'a :: len word"
assumes "i < j" **and** "0 < k"
and "unat j * unat k < 2 ^ len_of TYPE ('a)"
shows "i * k < j * k"
⟨*proof*⟩

lemma word_mult_less_dest:
fixes i :: "'a :: len word"
assumes ij: "i * k < j * k"
and uik: "unat i * unat k < 2 ^ len_of TYPE ('a)"
and ujk: "unat j * unat k < 2 ^ len_of TYPE ('a)"
shows "i < j"
⟨*proof*⟩

lemma word_mult_less_cancel:
fixes k :: "'a :: len word"
assumes knz: "0 < k"
and uik: "unat i * unat k < 2 ^ len_of TYPE ('a)"
and ujk: "unat j * unat k < 2 ^ len_of TYPE ('a)"
shows "(i * k < j * k) = (i < j)"
⟨*proof*⟩

lemma Suc_div_unat_helper:
assumes szv: "sz < LENGTH('a :: len)"
and usszv: "us ≤ sz"
shows "2 ^ (sz - us) = Suc (unat (((2::'a :: len word) ^ sz - 1) div
2 ^ us))"
⟨*proof*⟩

lemma enum_word_nth_eq:
‹(Enum.enum :: 'a::len word list) ! n = word_of_nat n›
if ‹n < 2 ^ LENGTH('a)›
for n

⟨*proof*⟩

lemma length_enum_word_eq:
  ‹length (Enum.enum :: 'a::len word list) = 2 ^ LENGTH('a)›
  ⟨*proof*⟩

lemma unat_lt2p [iff]:
  ‹unat x < 2 ^ LENGTH('a)› **for** x :: ‹'a::len word›
  ⟨*proof*⟩

lemma of_nat_unat [simp]:
  "of_nat ∘ unat = id"
  ⟨*proof*⟩

lemma Suc_unat_minus_one [simp]:
  "x ≠ 0 ⟹ Suc (unat (x - 1)) = unat x"
  ⟨*proof*⟩

lemma word_add_le_dest:
  fixes i :: "'a :: len word"
  assumes le: "i + k ≤ j + k"
  and uik: "unat i + unat k < 2 ^ len_of TYPE ('a)"
  and ujk: "unat j + unat k < 2 ^ len_of TYPE ('a)"
  shows "i ≤ j"
  ⟨*proof*⟩

lemma word_add_le_mono1:
  fixes i :: "'a :: len word"
  assumes "i ≤ j" **and** "unat j + unat k < 2 ^ len_of TYPE ('a)"
  shows "i + k ≤ j + k"
  ⟨*proof*⟩

lemma word_add_le_mono2:
  fixes i :: "'a :: len word"
  shows "⟦i ≤ j; unat j + unat k < 2 ^ LENGTH('a)⟧ ⟹ k + i ≤ k + j"
  ⟨*proof*⟩

lemma word_add_le_iff:
  fixes i :: "'a :: len word"
  assumes uik: "unat i + unat k < 2 ^ len_of TYPE ('a)"
  and ujk: "unat j + unat k < 2 ^ len_of TYPE ('a)"
  shows "(i + k ≤ j + k) = (i ≤ j)"
  ⟨*proof*⟩

lemma word_add_less_mono1:
  fixes i :: "'a :: len word"
  assumes "i < j" **and** "unat j + unat k < 2 ^ len_of TYPE ('a)"
  shows "i + k < j + k"
  ⟨*proof*⟩

lemma word_add_less_dest:
fixes i :: "'a :: len word"
assumes le: "i + k < j + k"
and uik: "unat i + unat k < 2 ^ len_of TYPE ('a)"
and ujk: "unat j + unat k < 2 ^ len_of TYPE ('a)"
shows "i < j"
⟨*proof*⟩

lemma word_add_less_iff:
fixes i :: "'a :: len word"
assumes uik: "unat i + unat k < 2 ^ len_of TYPE ('a)"
and ujk: "unat j + unat k < 2 ^ len_of TYPE ('a)"
shows "(i + k < j + k) = (i < j)"
⟨*proof*⟩

lemma word_mult_less_iff:
fixes i :: "'a :: len word"
assumes knz: "0 < k"
and uik: "unat i * unat k < 2 ^ len_of TYPE ('a)"
and ujk: "unat j * unat k < 2 ^ len_of TYPE ('a)"
shows "(i * k < j * k) = (i < j)"
⟨*proof*⟩

lemma word_le_imp_diff_le:
fixes n :: "'a::len word"
shows "⟦k ≤ n; n ≤ m⟧ ⟹ n - k ≤ m"
⟨*proof*⟩

lemma word_less_imp_diff_less:
fixes n :: "'a::len word"
shows "⟦k ≤ n; n < m⟧ ⟹ n - k < m"
⟨*proof*⟩

lemma word_mult_le_mono1:
fixes i :: "'a :: len word"
assumes ij: "i ≤ j" "0 < k"
and "unat j * unat k < 2 ^ len_of TYPE ('a)"
shows "i * k ≤ j * k"
⟨*proof*⟩

lemma word_mult_le_iff:
fixes i :: "'a :: len word"
assumes "0 < k"
and "unat i * unat k < 2 ^ len_of TYPE ('a)"
and "unat j * unat k < 2 ^ len_of TYPE ('a)"
shows "(i * k ≤ j * k) = (i ≤ j)"
⟨*proof*⟩

lemma word_diff_less:
  fixes n :: "'a :: len word"
  shows "⟦0 < n; 0 < m; n ≤ m⟧ ⟹ m - n < m"
  ⟨*proof*⟩

lemma word_add_increasing:
  fixes x :: "'a :: len word"
  shows "⟦ p + w ≤ x; p ≤ p + w ⟧ ⟹ p ≤ x"
  ⟨*proof*⟩

lemma word_random:
  fixes x :: "'a :: len word"
  shows "⟦ p ≤ p + x'; x ≤ x' ⟧ ⟹ p ≤ p + x"
  ⟨*proof*⟩

lemma word_sub_mono:
  "⟦ a ≤ c; d ≤ b; a - b ≤ a; c - d ≤ c ⟧
    ⟹ (a - b) ≤ (c - d :: 'a :: len word)"
  ⟨*proof*⟩

lemma power_not_zero:
  "n < LENGTH('a::len) ⟹ (2 :: 'a word) ^ n ≠ 0"
  ⟨*proof*⟩

lemma word_gt_a_gt_0:
  "a < n ⟹ (0 :: 'a::len word) < n"
  ⟨*proof*⟩

lemma word_power_less_1 [simp]:
  "sz < LENGTH('a::len) ⟹ (2::'a word) ^ sz - 1 < 2 ^ sz"
  ⟨*proof*⟩

lemma word_sub_1_le:
  "x ≠ 0 ⟹ x - 1 ≤ (x :: 'a :: len word)"
  ⟨*proof*⟩

lemma unat_less_power:
  fixes k :: "'a::len word"
  assumes szv: "sz < LENGTH('a)"
  and   kv: "k < 2 ^ sz"
  shows   "unat k < 2 ^ sz"
  ⟨*proof*⟩

lemma unat_mult_power_lem:
  assumes kv: "k < 2 ^ (LENGTH('a::len) - sz)"
  shows "unat (2 ^ sz * of_nat k :: (('a::len) word)) = 2 ^ sz * k"
⟨*proof*⟩

lemma word_plus_mcs_4:
"⟦v + x ≤ w + x; x ≤ v + x⟧ ⟹ v ≤ (w::'a::len word)"
⟨*proof*⟩

lemma word_plus_mcs_3:
"⟦v ≤ w; x ≤ w + x⟧ ⟹ v + x ≤ w + (x::'a::len word)"
⟨*proof*⟩

lemma word_le_minus_one_leq:
"x < y ⟹ x ≤ y - 1" **for** x :: "'a :: len word"
⟨*proof*⟩

lemma word_less_sub_le[simp]:
fixes x :: "'a :: len word"
assumes nv: "n < LENGTH('a)"
shows "(x ≤ 2 ^ n - 1) = (x < 2 ^ n)"
⟨*proof*⟩

lemma unat_of_nat_len:
"x < 2 ^ LENGTH('a) ⟹ unat (of_nat x :: 'a::len word) = x"
⟨*proof*⟩

lemma unat_of_nat_eq:
"x < 2 ^ LENGTH('a) ⟹ unat (of_nat x ::'a::len word) = x"
⟨*proof*⟩

lemma unat_eq_of_nat:
"n < 2 ^ LENGTH('a) ⟹ (unat (x :: 'a::len word) = n) = (x = of_nat
n)"
⟨*proof*⟩

lemma alignUp_div_helper:
fixes a :: "'a::len word"
assumes kv: "k < 2 ^ (LENGTH('a) - n)"
and xk: "x = 2 ^ n * of_nat k"
and le: "a ≤ x"
and sz: "n < LENGTH('a)"
and anz: "a mod 2 ^ n ≠ 0"
shows "a div 2 ^ n < of_nat k"
⟨*proof*⟩

lemma mask_out_sub_mask:
"(x AND NOT (mask n)) = x - (x AND (mask n))"
for x :: ‹'a::len word›
⟨*proof*⟩

lemma subtract_mask:
"p - (p AND mask n) = (p AND NOT (mask n))"
"p - (p AND NOT (mask n)) = (p AND mask n)"

for p :: ‹'a::len word›
⟨*proof*⟩

lemma take_bit_word_eq_self_iff:
‹take_bit n w = w ⟷ n ≥ LENGTH('a) ∨ w < 2 ^ n›
for w :: ‹'a::len word›
⟨*proof*⟩

lemma word_power_increasing:
assumes x: "2 ^ x < (2 ^ y::'a::len word)" "x < LENGTH('a)" "y < LENGTH('a)"
shows "x < y" ⟨*proof*⟩

lemma mask_twice:
"(x AND mask n) AND mask m = x AND mask (min m n)"
for x :: ‹'a::len word›
⟨*proof*⟩

lemma plus_one_helper[elim!]:
"x < n + (1 :: 'a :: len word) ⟹ x ≤ n"
⟨*proof*⟩

lemma plus_one_helper2:
"⟦ x ≤ n; n + 1 ≠ 0 ⟧ ⟹ x < n + (1 :: 'a :: len word)"
⟨*proof*⟩

lemma less_x_plus_1:
"x ≠ - 1 ⟹ (y < x + 1) = (y < x ∨ y = x)" **for** x :: "'a::len word"
⟨*proof*⟩

lemma word_Suc_leq:
fixes k:: "'a::len word" **shows** "k ≠ - 1 ⟹ x < k + 1 ⟷ x ≤ k"
⟨*proof*⟩

lemma word_Suc_le:
fixes k:: "'a::len word" **shows** "x ≠ - 1 ⟹ x + 1 ≤ k ⟷ x < k"
⟨*proof*⟩

lemma word_lessThan_Suc_atMost:
‹{..< k + 1} = {..k}› **if** ‹k ≠ - 1› **for** k :: ‹'a::len word›
⟨*proof*⟩

lemma word_atLeastLessThan_Suc_atLeastAtMost:
‹{l ..< u + 1} = {l..u}› **if** ‹u ≠ - 1› **for** l :: ‹'a::len word›
⟨*proof*⟩

lemma word_atLeastAtMost_Suc_greaterThanAtMost:
‹{m<..u} = {m + 1..u}› **if** ‹m ≠ - 1› **for** m :: ‹'a::len word›
⟨*proof*⟩

lemma word_atLeastLessThan_Suc_atLeastAtMost_union:
fixes l:: "'a::len word"
assumes "m ≠ - 1" **and** "l ≤ m" **and** "m ≤ u"
shows "{l..m} ∪ {m+1..u} = {l..u}"
⟨*proof*⟩

lemma max_word_less_eq_iff [simp]:
‹- 1 ≤ w ⟷ w = - 1› **for** w :: ‹'a::len word›
⟨*proof*⟩

lemma word_or_zero:
"(a OR b = 0) = (a = 0 ∧ b = 0)"
for a b :: ‹'a::len word›
⟨*proof*⟩

lemma word_2p_mult_inc:
assumes x: "2 * 2 ^ n < (2::'a::len word) * 2 ^ m"
assumes suc_n: "Suc n < LENGTH('a::len)"
shows "2^n < (2::'a::len word)^m"
⟨*proof*⟩

lemma power_overflow:
"n ≥ LENGTH('a) ⟹ 2 ^ n = (0 :: 'a::len word)"
⟨*proof*⟩

lemmas extra_sle_sless_unfolds [simp] =
word_sle_eq[**where** a=0 **and** b=1]
word_sle_eq[**where** a=0 **and** b="numeral n"]
word_sle_eq[**where** a=1 **and** b=0]
word_sle_eq[**where** a=1 **and** b="numeral n"]
word_sle_eq[**where** a="numeral n" **and** b=0]
word_sle_eq[**where** a="numeral n" **and** b=1]
word_sless_alt[**where** a=0 **and** b=1]
word_sless_alt[**where** a=0 **and** b="numeral n"]
word_sless_alt[**where** a=1 **and** b=0]
word_sless_alt[**where** a=1 **and** b="numeral n"]
word_sless_alt[**where** a="numeral n" **and** b=0]
word_sless_alt[**where** a="numeral n" **and** b=1]
for n

lemma word_sint_1:
"sint (1::'a::len word) = (if LENGTH('a) = 1 then -1 else 1)"
⟨*proof*⟩

lemma ucast_of_nat:
"is_down (ucast :: 'a :: len word ⇒ 'b :: len word)
⟹ ucast (of_nat n :: 'a word) = (of_nat n :: 'b word)"
⟨*proof*⟩

lemma scast_1':
"(scast (1::'a::len word) :: 'b::len word) =
(word_of_int (signed_take_bit (LENGTH('a::len) - Suc 0) (1::int)))"
⟨*proof*⟩

lemma scast_1:
"(scast (1::'a::len word) :: 'b::len word) = (if LENGTH('a) = 1 then
-1 else 1)"
⟨*proof*⟩

lemma unat_minus_one_word:
"unat (-1 :: 'a :: len word) = 2 ^ LENGTH('a) - 1"
⟨*proof*⟩

lemmas word_diff_ls'' = word_diff_ls [**where** xa=x **and** x=x **for** x]
lemmas word_diff_ls' = word_diff_ls'' [simplified]

lemmas word_l_diffs' = word_l_diffs [**where** xa=x **and** x=x **for** x]
lemmas word_l_diffs = word_l_diffs' [simplified]

lemma two_power_increasing:
"⟦ n ≤ m; m < LENGTH('a) ⟧ ⟹ (2 :: 'a :: len word) ^ n ≤ 2 ^ m"
⟨*proof*⟩

lemma word_leq_le_minus_one:
"⟦ x ≤ y; x ≠ 0 ⟧ ⟹ x - 1 < (y :: 'a :: len word)"
⟨*proof*⟩

lemma neg_mask_combine:
"NOT(mask a) AND NOT(mask b) = NOT(mask (max a b) :: 'a::len word)"
⟨*proof*⟩

lemma neg_mask_twice:
"x AND NOT(mask n) AND NOT(mask m) = x AND NOT(mask (max n m))"
for x :: ‹'a::len word›
⟨*proof*⟩

lemma multiple_mask_trivia:
"n ≥ m ⟹ (x AND NOT(mask n)) + (x AND mask n AND NOT(mask m)) = x
AND NOT(mask m)"
for x :: ‹'a::len word›
⟨*proof*⟩

lemma word_of_nat_less: "n < unat x ⟹ of_nat n < x"
⟨*proof*⟩

lemma unat_mask:
"unat (mask n :: 'a :: len word) = 2 ^ (min n (LENGTH('a))) - 1"
⟨*proof*⟩

lemma mask_over_length:
"LENGTH('a) ≤ n ⟹ mask n = (-1::'a::len word)"
⟨*proof*⟩

lemma Suc_2p_unat_mask:
"n < LENGTH('a) ⟹ Suc (2 ^ n * k + unat (mask n :: 'a::len word))
= 2 ^ n * (k+1)"
⟨*proof*⟩

lemma sint_of_nat_ge_zero:
"x < 2 ^ (LENGTH('a) - 1) ⟹ sint (of_nat x :: 'a :: len word) ≥ 0"
⟨*proof*⟩

lemma int_eq_sint:
assumes "x < 2 ^ (LENGTH('a) - 1)"
shows "sint (of_nat x :: 'a :: len word) = int x"
⟨*proof*⟩

lemma sint_of_nat_le:
"⟦ b < 2 ^ (LENGTH('a) - 1); a ≤ b ⟧
⟹ sint (of_nat a :: 'a :: len word) ≤ sint (of_nat b :: 'a :: len
word)"
⟨*proof*⟩

lemma word_le_not_less:
fixes b :: "'a::len word"
shows "b ≤ a ⟷ ¬ a < b"
⟨*proof*⟩

lemma less_is_non_zero_p1:
fixes a :: "'a :: len word"
shows "a < k ⟹ a + 1 ≠ 0"
⟨*proof*⟩

lemma unat_add_lem':
fixes y :: "'a::len word"
shows "(unat x + unat y < 2 ^ LENGTH('a)) ⟹ (unat (x + y) = unat
x + unat y)"
⟨*proof*⟩

lemma word_less_two_pow_divI:
"⟦ (x :: 'a::len word) < 2 ^ (n - m); m ≤ n; n < LENGTH('a) ⟧ ⟹ x
< 2 ^ n div 2 ^ m"
⟨*proof*⟩

lemma word_less_two_pow_divD:
fixes x :: "'a::len word"
assumes "x < 2 ^ n div 2 ^ m"

shows "n ≥ m ∧ (x < 2 ^ (n - m))"
⟨*proof*⟩

lemma of_nat_less_two_pow_div_set:
assumes "n < LENGTH('a)"
shows "{x. x < (2 ^ n div 2 ^ m :: 'a::len word)} = of_nat ` {k. k < 2 ^ n div 2 ^ m}"
⟨*proof*⟩

lemma ucast_less:
"LENGTH('b) < LENGTH('a) ⟹
(ucast (x :: 'b :: len word) :: ('a :: len word)) < 2 ^ LENGTH('b)"
⟨*proof*⟩

lemma ucast_range_less:
"LENGTH('a :: len) < LENGTH('b :: len) ⟹
range (ucast :: 'a word ⇒ 'b word) = {x. x < 2 ^ len_of TYPE ('a)}"
⟨*proof*⟩

lemma word_power_less_diff:
fixes q :: "'a::len word"
assumes "2 ^ n * q < 2 ^ m" **and** "q < 2 ^ (LENGTH('a) - n)"
shows "q < 2 ^ (m - n)"
⟨*proof*⟩

lemma word_less_sub_1:
"x < (y :: 'a :: len word) ⟹ x ≤ y - 1"
⟨*proof*⟩

lemma word_sub_mono2:
"⟦ a + b ≤ c + d; c ≤ a; b ≤ a + b; d ≤ c + d ⟧ ⟹ b ≤ (d :: 'a :: len word)"
⟨*proof*⟩

lemma word_not_le:
"(¬ x ≤ (y :: 'a :: len word)) = (y < x)"
⟨*proof*⟩

lemma word_subset_less:
fixes s :: "'a :: len word"
assumes "{x..x + r - 1} ⊆ {y..y + s - 1}"
and xy: "x ≤ x + r - 1" "y ≤ y + s - 1"
and "s ≠ 0"
shows "r ≤ s"
⟨*proof*⟩

lemma uint_power_lower:
"n < LENGTH('a) ⟹ uint (2 ^ n :: 'a :: len word) = (2 ^ n :: int)"
⟨*proof*⟩

lemma power_le_mono:
"⟦2 ^ n ≤ (2::'a::len word) ^ m; n < LENGTH('a); m < LENGTH('a)⟧ ⟹
n ≤ m"
⟨*proof*⟩

lemma two_power_eq:
"⟦n < LENGTH('a); m < LENGTH('a)⟧
⟹ ((2::'a::len word) ^ n = 2 ^ m) = (n = m)"
⟨*proof*⟩

lemma unat_less_helper:
"x < of_nat n ⟹ unat x < n"
⟨*proof*⟩

lemma nat_uint_less_helper:
"nat (uint y) = z ⟹ x < y ⟹ nat (uint x) < z"
⟨*proof*⟩

lemma of_nat_0:
"⟦of_nat n = (0::'a::len word); n < 2 ^ LENGTH('a)⟧ ⟹ n = 0"
⟨*proof*⟩

lemma of_nat_inj:
"⟦x < 2 ^ LENGTH('a); y < 2 ^ LENGTH('a)⟧ ⟹
(of_nat x = (of_nat y :: 'a :: len word)) = (x = y)"
⟨*proof*⟩

lemma div_to_mult_word_lt:
"⟦ (x :: 'a :: len word) ≤ y div z ⟧ ⟹ x * z ≤ y"
⟨*proof*⟩

lemma ucast_ucast_mask:
"(ucast :: 'a :: len word ⇒ 'b :: len word) (ucast x) = x AND mask
(len_of TYPE ('a))"
⟨*proof*⟩

lemma ucast_ucast_len:
"⟦ x < 2 ^ LENGTH('b) ⟧ ⟹ ucast (ucast x::'b::len word) = (x::'a::len
word)"
⟨*proof*⟩

lemma ucast_ucast_id:
"LENGTH('a) < LENGTH('b) ⟹ ucast (ucast (x::'a::len word)::'b::len
word) = x"
⟨*proof*⟩

lemma unat_ucast:
"unat (ucast x :: ('a :: len) word) = unat x mod 2 ^ (LENGTH('a))"

⟨*proof*⟩

lemma ucast_less_ucast:
"LENGTH('a) ≤ LENGTH('b) ⟹
(ucast x < ((ucast (y :: 'a::len word)) :: 'b::len word)) = (x < y)"
⟨*proof*⟩
lemmas ucast_less_ucast_weak = ucast_less_ucast[OF order.strict_implies_order]

lemma unat_Suc2:
fixes n :: "'a :: len word"
shows "n ≠ -1 ⟹ unat (n + 1) = Suc (unat n)"
⟨*proof*⟩

lemma word_div_1:
"(n :: 'a :: len word) div 1 = n"
⟨*proof*⟩

lemma word_minus_one_le:
"-1 ≤ (x :: 'a :: len word) = (x = -1)"
⟨*proof*⟩

lemma up_scast_inj:
"⟦ scast x = (scast y :: 'b :: len word); size x ≤ LENGTH('b) ⟧ ⟹
x = y"
⟨*proof*⟩

lemma up_scast_inj_eq:
"LENGTH('a) ≤ len_of TYPE ('b) ⟹
(scast x = (scast y::'b::len word)) = (x = (y::'a::len word))"
⟨*proof*⟩

lemma word_le_add:
fixes x :: "'a :: len word"
shows "x ≤ y ⟹ ∃n. y = x + of_nat n"
⟨*proof*⟩

lemma word_plus_mcs_4':
"⟦x + v ≤ x + w; x ≤ x + v⟧ ⟹ v ≤ w" **for** x :: "'a::len word"
⟨*proof*⟩

lemma unat_eq_1:
‹unat x = Suc 0 ⟷ x = 1›
⟨*proof*⟩

lemma word_unat_Rep_inject1:
‹unat x = unat 1 ⟷ x = 1›
⟨*proof*⟩

lemma and_not_mask_twice:

"(w AND NOT (mask n)) AND NOT (mask m) = w AND NOT (mask (max m n))"
for w :: ‹'a::len word›
⟨*proof*⟩

lemma word_less_cases:
"x < y ⟹ x = y - 1 ∨ x < y - (1 ::'a::len word)"
⟨*proof*⟩

lemma mask_and_mask:
"mask a AND mask b = (mask (min a b) :: 'a::len word)"
⟨*proof*⟩

lemma mask_eq_0_eq_x:
"(x AND w = 0) = (x AND NOT w = x)"
for x w :: ‹'a::len word›
⟨*proof*⟩

lemma mask_eq_x_eq_0:
"(x AND w = x) = (x AND NOT w = 0)"
for x w :: ‹'a::len word›
⟨*proof*⟩

lemma compl_of_1: "NOT 1 = (-2 :: 'a :: len word)"
⟨*proof*⟩

lemma split_word_eq_on_mask:
"(x = y) = (x AND m = y AND m ∧ x AND NOT m = y AND NOT m)"
for x y m :: ‹'a::len word›
⟨*proof*⟩

lemma word_FF_is_mask:
"0xFF = (mask 8 :: 'a::len word)"
⟨*proof*⟩

lemma word_1FF_is_mask:
"0x1FF = (mask 9 :: 'a::len word)"
⟨*proof*⟩

lemma ucast_of_nat_small:
"x < 2 ^ LENGTH('a) ⟹ ucast (of_nat x :: 'a :: len word) = (of_nat
x :: 'b :: len word)"
⟨*proof*⟩

lemma word_le_make_less:
fixes x :: "'a :: len word"
shows "y ≠ -1 ⟹ (x ≤ y) = (x < (y + 1))"
⟨*proof*⟩

lemmas finite_word = finite [**where** 'a="'a::len word"]

lemma word_to_1_set:
"{0 ..< (1 :: 'a :: len word)} = {0}"
⟨*proof*⟩

lemma word_leq_minus_one_le:
fixes x :: "'a::len word"
shows "⟦y ≠ 0; x ≤ y - 1 ⟧ ⟹ x < y"
⟨*proof*⟩

lemma word_count_from_top:
"n ≠ 0 ⟹ {0 ..< n :: 'a :: len word} = {0 ..< n - 1} ∪ {n - 1}"
⟨*proof*⟩

lemma word_minus_one_le_leq:
"⟦ x - 1 < y ⟧ ⟹ x ≤ (y :: 'a :: len word)"
⟨*proof*⟩

lemma word_must_wrap:
"⟦ x ≤ n - 1; n ≤ x ⟧ ⟹ n = (0 :: 'a :: len word)"
⟨*proof*⟩

lemma range_subset_card:
"⟦ {a :: 'a :: len word .. b} ⊆ {c .. d}; b ≥ a ⟧ ⟹ d ≥ c ∧ d - c
≥ b - a"
⟨*proof*⟩

lemma less_1_simp:
"n - 1 < m = (n ≤ (m :: 'a :: len word) ∧ n ≠ 0)"
⟨*proof*⟩

lemma word_power_mod_div:
fixes x :: "'a::len word"
shows "⟦ n < LENGTH('a); m < LENGTH('a)⟧
⟹ x mod 2 ^ n div 2 ^ m = x div 2 ^ m mod 2 ^ (n - m)"
⟨*proof*⟩

lemma word_range_minus_1':
fixes a :: "'a :: len word"
shows "a ≠ 0 ⟹ {a-1<..b} = {a..b}"
⟨*proof*⟩

lemma word_range_minus_1:
fixes a :: "'a :: len word"
shows "b ≠ 0 ⟹ {a..b - 1} = {a..<b}"
⟨*proof*⟩

lemma ucast_nat_def:
"of_nat (unat x) = (ucast :: 'a :: len word ⇒ 'b :: len word) x"

⟨*proof*⟩

lemma overflow_plus_one_self:
"(1 + p ≤ p) = (p = (-1 :: 'a :: len word))"
⟨*proof*⟩

lemma plus_1_less:
"(x + 1 ≤ (x :: 'a :: len word)) = (x = -1)"
⟨*proof*⟩

lemma pos_mult_pos_ge:
"[|x > (0::int); n>=0 |] ==> n * x >= n*1"
⟨*proof*⟩

lemma word_plus_strict_mono_right:
fixes x :: "'a :: len word"
shows "⟦y < z; x ≤ x + z⟧ ⟹ x + y < x + z"
⟨*proof*⟩

lemma word_div_mult:
"0 < c ⟹ a < b * c ⟹ a div c < b" **for** a b c :: "'a::len word"
⟨*proof*⟩

lemma word_less_power_trans_ofnat:
"⟦n < 2 ^ (m - k); k ≤ m; m < LENGTH('a)⟧
⟹ of_nat n * 2 ^ k < (2::'a::len word) ^ m"
⟨*proof*⟩

lemma word_1_le_power:
"n < LENGTH('a) ⟹ (1 :: 'a :: len word) ≤ 2 ^ n"
⟨*proof*⟩

lemma unat_1_0:
"1 ≤ (x::'a::len word) = (0 < unat x)"
⟨*proof*⟩

lemma x_less_2_0_1':
fixes x :: "'a::len word"
shows "⟦LENGTH('a) ≠ 1; x < 2⟧ ⟹ x = 0 ∨ x = 1"
⟨*proof*⟩

lemmas word_add_le_iff2 = word_add_le_iff [folded no_olen_add_nat]

lemma of_nat_power:
shows "⟦ p < 2 ^ x; x < len_of TYPE ('a) ⟧ ⟹ of_nat p < (2 :: 'a ::
len word) ^ x"
⟨*proof*⟩

lemma of_nat_n_less_equal_power_2:

"n < LENGTH('a::len) ⟹ ((of_nat n)::'a word) < 2 ^ n"
⟨*proof*⟩

lemma eq_mask_less:
fixes w :: "'a::len word"
assumes eqm: "w = w AND mask n"
and sz: "n < len_of TYPE ('a)"
shows "w < (2::'a word) ^ n"
⟨*proof*⟩

lemma of_nat_mono_maybe':
fixes Y :: "nat"
assumes xlt: "x < 2 ^ len_of TYPE ('a)"
assumes ylt: "y < 2 ^ len_of TYPE ('a)"
shows "(y < x) = (of_nat y < (of_nat x :: 'a :: len word))"
⟨*proof*⟩

lemma of_nat_mono_maybe_le:
"⟦x < 2 ^ LENGTH('a); y < 2 ^ LENGTH('a)⟧ ⟹
(y ≤ x) = ((of_nat y :: 'a :: len word) ≤ of_nat x)"
⟨*proof*⟩

lemma mask_AND_NOT_mask:
"(w AND NOT (mask n)) AND mask n = 0"
for w :: ‹'a::len word›
⟨*proof*⟩

lemma AND_NOT_mask_plus_AND_mask_eq:
"(w AND NOT (mask n)) + (w AND mask n) = w"
for w :: ‹'a::len word›
⟨*proof*⟩

lemma mask_eqI:
fixes x :: "'a :: len word"
assumes m1: "x AND mask n = y AND mask n"
and m2: "x AND NOT (mask n) = y AND NOT (mask n)"
shows "x = y"
⟨*proof*⟩

lemma neq_0_no_wrap:
fixes x :: "'a :: len word"
shows "⟦ x ≤ x + y; x ≠ 0 ⟧ ⟹ x + y ≠ 0"
⟨*proof*⟩

lemma unatSuc2:
fixes n :: "'a :: len word"
shows "n + 1 ≠ 0 ⟹ unat (n + 1) = Suc (unat n)"
⟨*proof*⟩

lemma word_of_nat_le:
"n ≤ unat x ⟹ of_nat n ≤ x"
⟨*proof*⟩

lemma word_unat_less_le:
"a ≤ of_nat b ⟹ unat a ≤ b"
⟨*proof*⟩

lemma mask_Suc_0 : "mask (Suc 0) = (1 :: 'a::len word)"
⟨*proof*⟩

lemma bool_mask':
fixes x :: "'a :: len word"
shows "2 < LENGTH('a) ⟹ (0 < x AND 1) = (x AND 1 = 1)"
⟨*proof*⟩

lemma ucast_ucast_add:
fixes x :: "'a :: len word"
fixes y :: "'b :: len word"
shows "LENGTH('b) ≥ LENGTH('a) ⟹ ucast (ucast x + y) = x + ucast
y"
⟨*proof*⟩

lemma lt1_neq0:
fixes x :: "'a :: len word"
shows "(1 ≤ x) = (x ≠ 0)" ⟨*proof*⟩

lemma word_plus_one_nonzero:
fixes x :: "'a :: len word"
shows "⟦x ≤ x + y; y ≠ 0⟧ ⟹ x + 1 ≠ 0"
⟨*proof*⟩

lemma word_sub_plus_one_nonzero:
fixes n :: "'a :: len word"
shows "⟦n' ≤ n; n' ≠ 0⟧ ⟹ (n - n') + 1 ≠ 0"
⟨*proof*⟩

lemma word_le_minus_mono_right:
fixes x :: "'a :: len word"
shows "⟦ z ≤ y; y ≤ x; z ≤ x ⟧ ⟹ x - y ≤ x - z"
⟨*proof*⟩

lemma word_0_sle_from_less:
‹0 ≤s x› **if** ‹x < 2 ^ (LENGTH('a) - 1)› **for** x :: ‹'a::len word›
⟨*proof*⟩

lemma ucast_sub_ucast:
fixes x :: "'a::len word"
assumes "y ≤ x"

assumes T: "LENGTH('a) ≤ LENGTH('b)"
shows "ucast (x - y) = (ucast x - ucast y :: 'b::len word)"
⟨*proof*⟩

lemma word_1_0:
"⟦a + (1::('a::len) word) ≤ b; a < of_nat x⟧ ⟹ a < b"
⟨*proof*⟩

lemma unat_of_nat_less: "⟦ a < b; unat b = c ⟧ ⟹ a < of_nat c"
⟨*proof*⟩

lemma word_le_plus_1: "⟦ (y::('a::len) word) < y + n; a < n ⟧ ⟹ y + a ≤ y + a + 1"
⟨*proof*⟩

lemma word_le_plus: "⟦(a::('a::len) word) < a + b; c < b⟧ ⟹ a ≤ a + c"
⟨*proof*⟩

lemma sint_minus1 [simp]: "(sint x = -1) = (x = -1)"
⟨*proof*⟩

lemma sint_0 [simp]: "(sint x = 0) = (x = 0)"
⟨*proof*⟩

lemma sint_1_cases:
P **if** ‹⟦ len_of TYPE ('a::len) = 1; (a::'a word) = 0; sint a = 0 ⟧ ⟹ P›
‹⟦ len_of TYPE ('a) = 1; a = 1; sint (1 :: 'a word) = -1 ⟧ ⟹ P›
‹⟦ len_of TYPE ('a) > 1; sint (1 :: 'a word) = 1 ⟧ ⟹ P›
⟨*proof*⟩

lemma sint_int_min:
"sint (- (2 ^ (LENGTH('a) - Suc 0)) :: ('a::len) word) = - (2 ^ (LENGTH('a) - Suc 0))"
⟨*proof*⟩

lemma sint_int_max_plus_1:
"sint (2 ^ (LENGTH('a) - Suc 0) :: ('a::len) word) = - (2 ^ (LENGTH('a) - Suc 0))"
⟨*proof*⟩

lemma uint_range': ‹0 ≤ uint x ∧ uint x < 2 ^ LENGTH('a)› **for** x :: ‹'a::len word›
⟨*proof*⟩

lemma sint_of_int_eq:
"⟦ - (2 ^ (LENGTH('a) - 1)) ≤ x; x < 2 ^ (LENGTH('a) - 1) ⟧ ⟹ sint

```
(of_int x :: ('a::len) word) = x"
```
⟨*proof*⟩

lemma of_int_sint:
```
  "of_int (sint a) = a"
```
⟨*proof*⟩

lemma sint_ucast_eq_uint:
```
    "⟦ ¬ is_down (ucast :: ('a::len word ⇒ 'b::len word)) ⟧
           ⟹ sint ((ucast :: ('a::len word ⇒ 'b::len word)) x) = uint
x"
```
⟨*proof*⟩

lemma word_less_nowrapI':
```
  "(x :: 'a :: len word) ≤ z - k ⟹ k ≤ z ⟹ 0 < k ⟹ x < x + k"
```
⟨*proof*⟩

lemma mask_plus_1:
```
  "mask n + 1 = (2 ^ n :: 'a::len word)"
```
⟨*proof*⟩

lemma unat_inj: "inj unat"
⟨*proof*⟩

lemma unat_ucast_upcast:
```
  "is_up (ucast :: 'b word ⇒ 'a word)
      ⟹ unat (ucast x :: ('a::len) word) = unat (x :: ('b::len) word)"
```
⟨*proof*⟩

lemma ucast_mono:
```
  "⟦ (x :: 'b :: len word) < y; y < 2 ^ LENGTH('a) ⟧
   ⟹ ucast x < ((ucast y) :: 'a :: len word)"
```
⟨*proof*⟩

lemma ucast_mono_le:
```
  "⟦x ≤ y; y < 2 ^ LENGTH('b)⟧ ⟹ (ucast (x :: 'a :: len word) :: 'b
:: len word) ≤ ucast y"
```
⟨*proof*⟩

lemma ucast_mono_le':
```
  "⟦ unat y < 2 ^ LENGTH('b); LENGTH('b::len) < LENGTH('a::len); x ≤ y
⟧
   ⟹ ucast x ≤ (ucast y :: 'b word)" for x y :: <'a::len word>
```
⟨*proof*⟩

lemma neg_mask_add_mask:
```
  "((x:: 'a :: len word) AND NOT (mask n)) + (2 ^ n - 1) = x OR mask n"
```
⟨*proof*⟩

lemma le_step_down_word: "⟦(i::('a::len) word) ≤ n; i = n ⟶ P; i ≤ n - 1 ⟶ P⟧ ⟹ P"
⟨*proof*⟩

lemma le_step_down_word_2:
fixes x :: "'a::len word"
shows "⟦x ≤ y; x ≠ y⟧ ⟹ x ≤ y - 1"
⟨*proof*⟩

lemma NOT_mask_AND_mask[simp]: "(w AND mask n) AND NOT (mask n) = 0"
⟨*proof*⟩

lemma and_and_not[simp]: "(a AND b) AND NOT b = 0"
for a b :: ‹'a::len word›
⟨*proof*⟩

lemma ex_mask_1[simp]: "(∃x. mask x = (1 :: 'a::len word))"
⟨*proof*⟩

lemma not_switch: "NOT a = x ⟹ a = NOT x"
⟨*proof*⟩

lemma test_bit_eq_iff: "bit u = bit v ⟷ u = v"
for u v :: "'a::len word"
⟨*proof*⟩

lemma test_bit_size: "bit w n ⟹ n < size w"
for w :: "'a::len word"
⟨*proof*⟩

lemma word_eq_iff: "x = y ⟷ (∀n<LENGTH('a). bit x n = bit y n)"
for x y :: "'a::len word"
⟨*proof*⟩

lemma word_eqI: "(⋀n. n < size u ⟹ bit u n = bit v n) ⟹ u = v"
for u :: "'a::len word"
⟨*proof*⟩

lemma word_eqD: "u = v ⟹ bit u x = bit v x"
for u v :: "'a::len word"
⟨*proof*⟩

lemma test_bit_bin': "bit w n ⟷ n < size w ∧ bit (uint w) n"
⟨*proof*⟩

lemmas test_bit_bin = test_bit_bin' [unfolded word_size]

lemma word_test_bit_def: ‹bit a = bit (uint a)›
⟨*proof*⟩

lemmas test_bit_def' = word_test_bit_def [THEN fun_cong]

lemma word_test_bit_transfer [transfer_rule]:
"(rel_fun pcr_word (rel_fun (=) (=)))
(λx n. n < LENGTH('a) ∧ bit x n) (bit :: 'a::len word ⇒ _)"
⟨*proof*⟩

lemma word_ops_nth_size:
"n < size x ⟹
bit (x OR y) n = (bit x n | bit y n) ∧
bit (x AND y) n = (bit x n ∧ bit y n) ∧
bit (x XOR y) n = (bit x n ≠ bit y n) ∧
bit (NOT x) n = (¬ bit x n)"
for x :: "'a::len word"
⟨*proof*⟩

lemma word_ao_nth:
"bit (x OR y) n = (bit x n | bit y n) ∧
bit (x AND y) n = (bit x n ∧ bit y n)"
for x :: "'a::len word"
⟨*proof*⟩

lemmas lsb0 = len_gt_0 [THEN word_ops_nth_size [unfolded word_size]]

lemma nth_sint:
fixes w :: "'a::len word"
defines "l ≡ LENGTH('a)"
shows "bit (sint w) n = (if n < l - 1 then bit w n else bit w (l - 1))"
⟨*proof*⟩

lemma test_bit_2p: "bit (word_of_int (2 ^ n)::'a::len word) m ⟷ m
= n ∧ m < LENGTH('a)"
⟨*proof*⟩

lemma nth_w2p: "bit ((2::'a::len word) ^ n) m ⟷ m = n ∧ m < LENGTH('a::len)"
⟨*proof*⟩

lemma bang_is_le: "bit x m ⟹ 2 ^ m ≤ x"
for x :: "'a::len word"
⟨*proof*⟩

lemmas msb0 = len_gt_0 [THEN diff_Suc_less, THEN word_ops_nth_size [unfolded
word_size]]
lemmas msb1 = msb0 [**where** i = 0]

lemma nth_0: "¬ bit (0 :: 'a::len word) n"
⟨*proof*⟩

lemma nth_minus1: "bit (-1 :: 'a::len word) n ⟷ n < LENGTH('a)"
⟨*proof*⟩

lemma nth_ucast_weak:
"bit (ucast w::'a::len word) n = (bit w n ∧ n < LENGTH('a))"
⟨*proof*⟩

lemma nth_ucast:
"bit (ucast (w::'a::len word)::'b::len word) n =
(bit w n ∧ n < min LENGTH('a) LENGTH('b))"
⟨*proof*⟩

lemma nth_mask:
‹bit (mask n :: 'a::len word) i ⟷ i < n ∧ i < size (mask n :: 'a word)›
⟨*proof*⟩

lemma nth_slice: "bit (slice n w :: 'a::len word) m = (bit w (m + n) ∧ m < LENGTH('a))"
⟨*proof*⟩
lemma test_bit_split':
"word_split c = (a, b) ⟶
(∀n m.
bit b n = (n < size b ∧ bit c n) ∧
bit a m = (m < size a ∧ bit c (m + size b)))"
⟨*proof*⟩

lemma test_bit_split:
"word_split c = (a, b) ⟹
(∀n::nat. bit b n ⟷ n < size b ∧ bit c n) ∧
(∀m::nat. bit a m ⟷ m < size a ∧ bit c (m + size b))"
⟨*proof*⟩

lemma test_bit_rcat:
"sw = size (hd wl) ⟹ rc = word_rcat wl ⟹ bit rc n =
(n < size rc ∧ n div sw < size wl ∧ bit ((rev wl) ! (n div sw)) (n mod sw))"
for wl :: "'a::len word list"
⟨*proof*⟩

lemmas test_bit_cong = arg_cong [**where** f = "bit", THEN fun_cong]

lemma max_test_bit: "bit (- 1::'a::len word) n ⟷ n < LENGTH('a)"
⟨*proof*⟩

lemma map_nth_0 [simp]: "map (bit (0::'a::len word)) xs = replicate (length xs) False"
⟨*proof*⟩

lemma word_and_1:
"n AND 1 = (if bit n 0 then 1 else 0)" **for** n :: "_ word"
⟨*proof*⟩

lemma nth_w2p_same:
"bit (2^n :: 'a :: len word) n = (n < LENGTH('a))"
⟨*proof*⟩

lemma word_leI:
fixes u :: "'a::len word"
assumes "⋀n. ⟦n < size u; bit u n ⟧ ⟹ bit (v::'a::len word) n"
shows "u ≤ v"
⟨*proof*⟩

lemma bang_eq:
fixes x :: "'a::len word"
shows "(x = y) = (∀n. bit x n = bit y n)"
⟨*proof*⟩

lemma neg_mask_test_bit:
"bit (NOT(mask n) :: 'a :: len word) m = (n ≤ m ∧ m < LENGTH('a))"
⟨*proof*⟩

lemma upper_bits_unset_is_l2p:
‹(∀n'. n ≤ n' ⟶ ¬ bit p n') ⟷ (p < 2 ^ n)› (**is** ‹?P ⟷ ?Q›)
if ‹n < LENGTH('a)›
for p :: "'a :: len word"
⟨*proof*⟩

lemma less_2p_is_upper_bits_unset:
"p < 2 ^ n ⟷ n < LENGTH('a) ∧ (∀n'. n ≤ n' ⟶ ¬ bit p n')" **for** p :: "'a :: len word"
⟨*proof*⟩

lemma test_bit_over:
"n ≥ size (x::'a::len word) ⟹ (bit x n) = False"
⟨*proof*⟩

lemma le_mask_high_bits:
"w ≤ mask n ⟷ (∀i ∈ {n ..< size w}. ¬ bit w i)"
for w :: ‹'a::len word›
⟨*proof*⟩

lemma test_bit_conj_lt:
"(bit x m ∧ m < LENGTH('a)) = bit x m" **for** x :: "'a :: len word"
⟨*proof*⟩

lemma neg_test_bit:

"bit (NOT x) n = (¬ bit x n ∧ n < LENGTH('a))" **for** x :: "'a::len word"
⟨*proof*⟩

lemma nth_bounded:
"⟦bit (x :: 'a :: len word) n; x < 2 ^ m; m ≤ len_of TYPE ('a)⟧ ⟹
n < m"
⟨*proof*⟩

lemma and_neq_0_is_nth:
‹x AND y ≠ 0 ⟷ bit x n› **if** ‹y = 2 ^ n› **for** x y :: ‹'a::len word›
⟨*proof*⟩

lemma nth_is_and_neq_0:
"bit (x::'a::len word) n = (x AND 2 ^ n ≠ 0)"
⟨*proof*⟩

lemma max_word_not_less [simp]:
"¬ - 1 < x" **for** x :: ‹'a::len word›
⟨*proof*⟩

lemma bit_twiddle_min:
"(y::'a::len word) XOR (((x::'a::len word) XOR y) AND (if x < y then
-1 else 0)) = min x y"
⟨*proof*⟩

lemma bit_twiddle_max:
"(x::'a::len word) XOR (((x::'a::len word) XOR y) AND (if x < y then
-1 else 0)) = max x y"
⟨*proof*⟩

lemma swap_with_xor:
"⟦(x::'a::len word) = a XOR b; y = b XOR x; z = x XOR y⟧ ⟹ z = b ∧
y = a"
⟨*proof*⟩

lemma le_mask_imp_and_mask:
"(x::'a::len word) ≤ mask n ⟹ x AND mask n = x"
⟨*proof*⟩

lemma or_not_mask_nop:
"((x::'a::len word) OR NOT (mask n)) AND mask n = x AND mask n"
⟨*proof*⟩

lemma mask_subsume:
"⟦n ≤ m⟧ ⟹ ((x::'a::len word) OR y AND mask n) AND NOT (mask m) =
x AND NOT (mask m)"
⟨*proof*⟩

lemma and_mask_0_iff_le_mask:

fixes w :: "'a::len word"
shows "(w AND NOT(mask n) = 0) = (w ≤ mask n)"
⟨*proof*⟩

lemma mask_twice2:
"n ≤ m ⟹ ((x::'a::len word) AND mask m) AND mask n = x AND mask n"
⟨*proof*⟩

lemma uint_2_id:
"LENGTH('a) ≥ 2 ⟹ uint (2::('a::len) word) = 2"
⟨*proof*⟩

lemma div_of_0_id[simp]: "(0::('a::len) word) div n = 0"
⟨*proof*⟩

lemma degenerate_word: "LENGTH('a) = 1 ⟹ (x::('a::len) word) = 0 ∨
x = 1"
⟨*proof*⟩

lemma div_less_dividend_word:
fixes x :: "('a::len) word"
assumes "x ≠ 0" "n ≠ 1"
shows "x div n < x"
⟨*proof*⟩

lemma word_less_div:
fixes x :: "('a::len) word"
and y :: "('a::len) word"
shows "x div y = 0 ⟹ y = 0 ∨ x < y"
⟨*proof*⟩

lemma not_degenerate_imp_2_neq_0: "LENGTH('a) > 1 ⟹ (2::('a::len) word)
≠ 0"
⟨*proof*⟩

lemma word_overflow: "(x::('a::len) word) + 1 > x ∨ x + 1 = 0"
⟨*proof*⟩

lemma word_overflow_unat: "unat ((x::('a::len) word) + 1) = unat x +
1 ∨ x + 1 = 0"
⟨*proof*⟩

lemma even_word_imp_odd_next:
"even (unat (x::('a::len) word)) ⟹ x + 1 = 0 ∨ odd (unat (x + 1))"
⟨*proof*⟩

lemma odd_word_imp_even_next: "odd (unat (x::('a::len) word)) ⟹ x +
1 = 0 ∨ even (unat (x + 1))"
⟨*proof*⟩

lemma overflow_imp_lsb: "(x::('a::len) word) + 1 = 0 ⟹ bit x 0"
⟨*proof*⟩

lemma odd_iff_lsb: "odd (unat (x::('a::len) word)) = bit x 0"
⟨*proof*⟩

lemma of_nat_neq_iff_word:
"x mod 2 ^ LENGTH('a) ≠ y mod 2 ^ LENGTH('a) ⟹
(((of_nat x)::('a::len) word) ≠ of_nat y) = (x ≠ y)"
⟨*proof*⟩

lemma lsb_this_or_next: "¬ (bit ((x::('a::len) word) + 1) 0) ⟹ bit
x 0"
⟨*proof*⟩

lemma mask_or_not_mask:
"x AND mask n OR x AND NOT (mask n) = x"
for x :: ‹'a::len word›
⟨*proof*⟩

lemma word_gr0_conv_Suc: "(m::'a::len word) > 0 ⟹ ∃n. m = n + 1"
⟨*proof*⟩

lemma revcast_down_us [OF refl]:
"rc = revcast ⟹ source_size rc = target_size rc + n ⟹ rc w = ucast
(signed_drop_bit n w)"
for w :: "'a::len word"
⟨*proof*⟩

lemma revcast_down_ss [OF refl]:
"rc = revcast ⟹ source_size rc = target_size rc + n ⟹ rc w = scast
(signed_drop_bit n w)"
for w :: "'a::len word"
⟨*proof*⟩

lemma revcast_down_uu [OF refl]:
"rc = revcast ⟹ source_size rc = target_size rc + n ⟹ rc w = ucast
(drop_bit n w)"
for w :: "'a::len word"
⟨*proof*⟩

lemma revcast_down_su [OF refl]:
"rc = revcast ⟹ source_size rc = target_size rc + n ⟹ rc w = scast
(drop_bit n w)"
for w :: "'a::len word"
⟨*proof*⟩

lemma cast_down_rev [OF refl]:

```
  "uc = ucast ⟹ source_size uc = target_size uc + n ⟹ uc w = revcast
(push_bit n w)"
  for w :: "'a::len word"
  ⟨proof⟩

lemma revcast_up [OF refl]:
  "rc = revcast ⟹ source_size rc + n = target_size rc ⟹
    rc w = push_bit n (ucast w :: 'a::len word)"
  ⟨proof⟩

lemmas rc1 = revcast_up [THEN
  revcast_rev_ucast [symmetric, THEN trans, THEN word_rev_gal, symmetric]]
lemmas rc2 = revcast_down_uu [THEN
  revcast_rev_ucast [symmetric, THEN trans, THEN word_rev_gal, symmetric]]

lemma word_ops_nth:
  fixes x y :: ‹'a::len word›
  shows
  word_or_nth:  "bit (x OR y) n = (bit x n ∨ bit y n)" and
  word_and_nth: "bit (x AND y) n = (bit x n ∧ bit y n)" and
  word_xor_nth: "bit (x XOR y) n = (bit x n ≠ bit y n)"
  ⟨proof⟩

lemma word_power_nonzero:
  "⟦ (x :: 'a::len word) < 2 ^ (LENGTH('a) - n); n < LENGTH('a); x ≠ 0
⟧
  ⟹ x * 2 ^ n ≠ 0"
  ⟨proof⟩

lemma less_1_helper:
  "n ≤ m ⟹ (n - 1 :: int) < m"
  ⟨proof⟩

lemma div_power_helper:
  "⟦ x ≤ y; y < LENGTH('a) ⟧ ⟹ (2 ^ y - 1) div (2 ^ x :: 'a::len word)
= 2 ^ (y - x) - 1"
  ⟨proof⟩

lemma max_word_mask:
  "(- 1 :: 'a::len word) = mask LENGTH('a)"
  ⟨proof⟩

lemmas mask_len_max = max_word_mask[symmetric]

lemma mask_out_first_mask_some:
  "⟦ x AND NOT (mask n) = y; n ≤ m ⟧ ⟹ x AND NOT (mask m) = y AND NOT
(mask m)"
  for x y :: ‹'a::len word›
  ⟨proof⟩
```

lemma mask_lower_twice:
"n ≤ m ⟹ (x AND NOT (mask n)) AND NOT (mask m) = x AND NOT (mask m)"
for x :: ‹'a::len word›
⟨*proof*⟩

lemma mask_lower_twice2:
"(a AND NOT (mask n)) AND NOT (mask m) = a AND NOT (mask (max n m))"
for a :: ‹'a::len word›
⟨*proof*⟩

lemma ucast_and_neg_mask:
"ucast (x AND NOT (mask n)) = ucast x AND NOT (mask n)"
⟨*proof*⟩

lemma ucast_and_mask:
"ucast (x AND mask n) = ucast x AND mask n"
⟨*proof*⟩

lemma ucast_mask_drop:
"LENGTH('a :: len) ≤ n ⟹ (ucast (x AND mask n) :: 'a word) = ucast
x"
⟨*proof*⟩

lemma mask_exceed:
"n ≥ LENGTH('a) ⟹ (x::'a::len word) AND NOT (mask n) = 0"
⟨*proof*⟩

lemma word_add_no_overflow: "(x::'a::len word) < - 1 ⟹ x < x + 1"
⟨*proof*⟩

lemma lt_plus_1_le_word:
fixes x :: "'a::len word"
assumes bound: "n < unat (maxBound::'a word)"
shows "x < 1 + of_nat n = (x ≤ of_nat n)"
⟨*proof*⟩

lemma unat_ucast_up_simp:
fixes x :: "'a::len word"
assumes "LENGTH('a) ≤ LENGTH('b)"
shows "unat (ucast x :: 'b::len word) = unat x"
⟨*proof*⟩

lemma unat_ucast_less_no_overflow:
"⟦n < 2 ^ LENGTH('a); unat f < n⟧ ⟹ (f::('a::len) word) < of_nat n"
⟨*proof*⟩

lemma unat_ucast_less_no_overflow_simp:
"n < 2 ^ LENGTH('a) ⟹ (unat f < n) = ((f::('a::len) word) < of_nat

n)"
⟨*proof*⟩

lemma unat_ucast_no_overflow_le:
fixes f :: "'a::len word" **and** b :: "'b :: len word"
assumes no_overflow: "unat b < (2 :: nat) ^ LENGTH('a)"
and upward_cast: "LENGTH('a) < LENGTH('b)"
shows "(ucast f < b) = (unat f < unat b)"
⟨*proof*⟩

lemmas ucast_up_mono = ucast_less_ucast[THEN iffD2]

lemma minus_one_word:
"(-1 :: 'a :: len word) = 2 ^ LENGTH('a) - 1"
⟨*proof*⟩

lemma le_2p_upper_bits:
"⟦ (p::'a::len word) ≤ 2^n - 1; n < LENGTH('a) ⟧ ⟹
∀n'≥n. n' < LENGTH('a) ⟶ ¬ bit p n'"
⟨*proof*⟩

lemma le2p_bits_unset:
"p ≤ 2 ^ n - 1 ⟹ ∀n'≥n. n' < LENGTH('a) ⟶ ¬ bit (p::'a::len word)
n'"
⟨*proof*⟩

lemma complement_nth_w2p:
shows "n' < LENGTH('a) ⟹ bit (NOT (2 ^ n :: 'a::len word)) n' = (n'
≠ n)"
⟨*proof*⟩

lemma word_unat_and_lt:
"unat x < n ∨ unat y < n ⟹ unat (x AND y) < n"
⟨*proof*⟩

lemma word_unat_mask_lt:
"m ≤ size w ⟹ unat ((w::'a::len word) AND mask m) < 2 ^ m"
⟨*proof*⟩

lemma word_sless_sint_le: "x <s y ⟹ sint x ≤ sint y - 1"
⟨*proof*⟩

lemma upper_trivial:
fixes x :: "'a::len word"
shows "x ≠ 2 ^ LENGTH('a) - 1 ⟹ x < 2 ^ LENGTH('a) - 1"
⟨*proof*⟩

lemma constraint_expand:
fixes x :: "'a::len word"

shows "x ∈ {y. lower ≤ y ∧ y ≤ upper} = (lower ≤ x ∧ x ≤ upper)"
⟨*proof*⟩

lemma card_map_elide:
"card ((of_nat :: nat ⇒ 'a::len word) ` {0..<n}) = card {0..<n}"
if "n ≤ CARD('a::len word)"
⟨*proof*⟩

lemma card_map_elide2:
"n ≤ CARD('a::len word) ⟹ card ((of_nat::nat ⇒ 'a::len word) ` {0..<n}) = n"
⟨*proof*⟩

lemma eq_ucast_ucast_eq:
"LENGTH('b) ≤ LENGTH('a) ⟹ x = ucast y ⟹ ucast x = y"
for x :: "'a::len word" **and** y :: "'b::len word"
⟨*proof*⟩

lemma le_ucast_ucast_le:
"x ≤ ucast y ⟹ ucast x ≤ y"
for x :: "'a::len word" **and** y :: "'b::len word"
⟨*proof*⟩

lemma less_ucast_ucast_less:
"LENGTH('b) ≤ LENGTH('a) ⟹ x < ucast y ⟹ ucast x < y"
for x :: "'a::len word" **and** y :: "'b::len word"
⟨*proof*⟩

lemma ucast_le_ucast:
"LENGTH('a) ≤ LENGTH('b) ⟹ (ucast x ≤ (ucast y::'b::len word)) = (x ≤ y)"
for x :: "'a::len word"
⟨*proof*⟩

lemmas ucast_up_mono_le = ucast_le_ucast[THEN iffD2]

lemma ucast_or_distrib:
fixes x :: "'a::len word"
fixes y :: "'a::len word"
shows "(ucast (x OR y) :: ('b::len) word) = ucast x OR ucast y"
⟨*proof*⟩

lemma word_exists_nth:
"(w::'a::len word) ≠ 0 ⟹ ∃i. bit w i"
⟨*proof*⟩

lemma max_word_not_0 [simp]:
"- 1 ≠ (0 :: 'a::len word)"
⟨*proof*⟩

lemma unat_max_word_pos[simp]: "0 < unat (- 1 :: 'a::len word)"
⟨*proof*⟩

lemma mult_pow2_inj:
assumes ws: "m + n ≤ LENGTH('a)"
assumes le: "x ≤ mask m" "y ≤ mask m"
assumes eq: "x * 2 ^ n = y * (2 ^ n::'a::len word)"
shows "x = y"
⟨*proof*⟩

lemma word_of_nat_inj:
assumes "x < 2 ^ LENGTH('a)" "y < 2 ^ LENGTH('a)"
assumes "of_nat x = (of_nat y :: 'a::len word)"
shows "x = y"
⟨*proof*⟩

lemma word_of_int_bin_cat_eq_iff:
"(word_of_int (concat_bit LENGTH('b) (uint b) (uint a))::'c::len word)
=
word_of_int (concat_bit LENGTH('b) (uint d) (uint c)) ⟷ b = d ∧
a = c" (**is** "?L=?R")
if "LENGTH('a) + LENGTH('b) ≤ LENGTH('c)"
for a:: "'a::len word" **and** b:: "'b::len word"
⟨*proof*⟩

lemma word_cat_inj: "(word_cat a b::'c::len word) = word_cat c d ⟷
a = c ∧ b = d"
if "LENGTH('a) + LENGTH('b) ≤ LENGTH('c)"
for a:: "'a::len word" **and** b:: "'b::len word"
⟨*proof*⟩

lemma p2_eq_1: "2 ^ n = (1::'a::len word) ⟷ n = 0"
⟨*proof*⟩

end

lemmas word_div_less = div_word_less

lemma word_mod_by_0: "k mod (0::'a::len word) = k"
⟨*proof*⟩
lemmas word_log_binary_defs =
word_and_def word_or_def word_xor_def

— limited hom result
lemma word_cat_hom:

```
  "LENGTH('a::len) ≤ LENGTH('b::len) + LENGTH('c::len) ⟹
    (word_cat (word_of_int w :: 'b word) (b :: 'c word) :: 'a word) =
    word_of_int ((λk n l. concat_bit n l k) w (size b) (uint b))"
```
⟨*proof*⟩

lemma uint_shiftl:
```
  "uint (push_bit i n) = take_bit (size n) (push_bit i (uint n))"
```
⟨*proof*⟩

lemma sint_range':
```
  <- (2 ^ (LENGTH('a) - Suc 0)) ≤ sint x ∧ sint x < 2 ^ (LENGTH('a) -
Suc 0)>
  for x :: <'a::len word>
```
⟨*proof*⟩

lemma signed_arith_eq_checks_to_ord:
```
  <sint a + sint b = sint (a + b) ⟷ (a ≤s a + b ⟷ 0 ≤s b)> (is ?P)
  <sint a - sint b = sint (a - b) ⟷ (0 ≤s a - b ⟷ b ≤s a)> (is ?Q)
  <- sint a = sint (- a) ⟷ (0 ≤s - a ⟷ a ≤s 0)> (is ?R)
```
⟨*proof*⟩

lemma signed_mult_eq_checks_double_size:
```
  assumes mult_le: "(2 ^ (LENGTH('a) - 1) + 1) ^ 2 ≤ (2 :: int) ^ (LENGTH('b)
- 1)"
          and le: "2 ^ (LENGTH('a) - 1) ≤ (2 :: int) ^ (LENGTH('b) -
1)"
  shows "(sint (a :: 'a :: len word) * sint b = sint (a * b))
       = (scast a * scast b = (scast (a * b) :: 'b :: len word))"
```
⟨*proof*⟩

context
```
  includes bit_operations_syntax
```
begin

lemma wils1:
```
  <word_of_int (NOT (uint (word_of_int x :: 'a word))) = (word_of_int
(NOT x) :: 'a::len word)>
  <word_of_int (uint (word_of_int x :: 'a word) XOR uint (word_of_int
y :: 'a word)) = (word_of_int (x XOR y) :: 'a::len word)>
  <word_of_int (uint (word_of_int x :: 'a word) AND uint (word_of_int
y :: 'a word)) = (word_of_int (x AND y) :: 'a::len word)>
  <word_of_int (uint (word_of_int x :: 'a word) OR uint (word_of_int y
:: 'a word)) = (word_of_int (x OR y) :: 'a::len word)>
```
⟨*proof*⟩

end

end

4 Shift operations with infix syntax

theory Bit_Shifts_Infix_Syntax
  imports "HOL-Library.Word" More_Word
begin

context semiring_bit_operations
begin

definition shiftl :: ‹'a ⇒ nat ⇒ 'a› (**infixl** ‹<<› 55)
  where [code_unfold]: ‹a << n = push_bit n a›

lemma bit_shiftl_iff [bit_simps]:
  ‹bit (a << m) n ⟷ m ≤ n ∧ possible_bit TYPE('a) n ∧ bit a (n - m)›
  ⟨*proof*⟩

definition shiftr :: ‹'a ⇒ nat ⇒ 'a› (**infixl** ‹>>› 55)
  where [code_unfold]: ‹a >> n = drop_bit n a›

lemma bit_shiftr_eq [bit_simps]:
  ‹bit (a >> n) = bit a ∘ (+) n›
  ⟨*proof*⟩

end

definition sshiftr :: ‹'a::len word ⇒ nat ⇒ 'a word› (**infixl** ‹>>>› 55)
  where [code_unfold]: ‹w >>> n = signed_drop_bit n w›

lemma bit_sshiftr_iff [bit_simps]:
  ‹bit (w >>> m) n ⟷ bit w (if LENGTH('a) - m ≤ n ∧ n < LENGTH('a)
then LENGTH('a) - 1 else m + n)›
  for w :: ‹'a::len word›
  ⟨*proof*⟩

context
  includes lifting_syntax
begin

lemma shiftl_word_transfer [transfer_rule]:
  ‹(pcr_word ===> (=) ===> pcr_word) (λk n. push_bit n k) (<<)›
  ⟨*proof*⟩

lemma shiftr_word_transfer [transfer_rule]:
  ‹((pcr_word :: int ⇒ 'a::len word ⇒ _) ===> (=) ===> pcr_word)
    (λk n. drop_bit n (take_bit LENGTH('a) k))
    (>>)›
⟨*proof*⟩

lemma sshiftr_transfer [transfer_rule]:

```
  ‹((pcr_word :: int ⇒ 'a::len word ⇒ _) ===> (=) ===> pcr_word)
    (λk n. drop_bit n (signed_take_bit (LENGTH('a) - Suc 0) k))
    (>>>)›
```
⟨*proof*⟩

end

context semiring_bit_operations
begin

lemma shiftl_0 [simp]:
```
  ‹0 << n = 0›
```
⟨*proof*⟩

lemma shiftl_of_0 [simp]:
```
  ‹a << 0 = a›
```
⟨*proof*⟩

lemma shiftl_of_Suc [simp]:
```
  ‹a << Suc n = (a * 2) << n›
```
⟨*proof*⟩

lemma shiftl_1 [simp]:
```
  ‹1 << n = 2 ^ n›
```
⟨*proof*⟩

lemma shiftl_numeral_Suc [simp]:
```
  ‹numeral m << Suc n = push_bit (Suc n) (numeral m)›
```
⟨*proof*⟩

lemma shiftl_numeral_numeral [simp]:
```
  ‹numeral m << numeral n = push_bit (numeral n) (numeral m)›
```
⟨*proof*⟩

lemma shiftr_0 [simp]:
```
  ‹0 >> n = 0›
```
⟨*proof*⟩

lemma shiftr_of_0 [simp]:
```
  ‹a >> 0 = a›
```
⟨*proof*⟩

lemma shiftr_1 [simp]:
```
  ‹1 >> n = of_bool (n = 0)›
```
⟨*proof*⟩

lemma shiftr_numeral_Suc [simp]:
```
  ‹numeral m >> Suc n = drop_bit (Suc n) (numeral m)›
```
⟨*proof*⟩

lemma shiftr_numeral_numeral [simp]:
‹numeral m >> numeral n = drop_bit (numeral n) (numeral m)›
⟨*proof*⟩

lemma shiftl_eq_mult:
‹x << n = x * 2 ^ n›
⟨*proof*⟩

lemma shiftr_eq_div:
‹x >> n = x div 2 ^ n›
⟨*proof*⟩

end

lemmas shiftl_int_def = shiftl_eq_mult[**of** x **for** x::int]
lemmas shiftr_int_def = shiftr_eq_div[**of** x **for** x::int]

lemma int_shiftl_BIT: **fixes** x :: int
shows int_shiftl0: "x << 0 = x"
and int_shiftl_Suc: "x << Suc n = 2 * x << n"
⟨*proof*⟩

context ring_bit_operations
begin

context
includes bit_operations_syntax
begin

lemma shiftl_minus_1_numeral [simp]:
‹- 1 << numeral n = NOT (mask (numeral n))›
⟨*proof*⟩

end

end

lemma shiftl_Suc_0 [simp]:
‹Suc 0 << n = 2 ^ n›
⟨*proof*⟩

lemma shiftr_Suc_0 [simp]:
‹Suc 0 >> n = of_bool (n = 0)›
⟨*proof*⟩

lemma sshiftr_numeral_Suc [simp]:
‹numeral m >>> Suc n = signed_drop_bit (Suc n) (numeral m)›
⟨*proof*⟩

lemma sshiftr_numeral_numeral [simp]:
‹numeral m >>> numeral n = signed_drop_bit (numeral n) (numeral m)›
⟨*proof*⟩

context ring_bit_operations
begin

lemma shiftl_minus_numeral_Suc [simp]:
‹- numeral m << Suc n = push_bit (Suc n) (- numeral m)›
⟨*proof*⟩

lemma shiftl_minus_numeral_numeral [simp]:
‹- numeral m << numeral n = push_bit (numeral n) (- numeral m)›
⟨*proof*⟩

lemma shiftr_minus_numeral_Suc [simp]:
‹- numeral m >> Suc n = drop_bit (Suc n) (- numeral m)›
⟨*proof*⟩

lemma shiftr_minus_numeral_numeral [simp]:
‹- numeral m >> numeral n = drop_bit (numeral n) (- numeral m)›
⟨*proof*⟩

end

lemma sshiftr_0 [simp]:
‹0 >>> n = 0›
⟨*proof*⟩

lemma sshiftr_of_0 [simp]:
‹w >>> 0 = w›
⟨*proof*⟩

lemma sshiftr_1 [simp]:
‹(1 :: 'a::len word) >>> n = of_bool (LENGTH('a) = 1 ∨ n = 0)›
⟨*proof*⟩

lemma sshiftr_minus_numeral_Suc [simp]:
‹- numeral m >>> Suc n = signed_drop_bit (Suc n) (- numeral m)›
⟨*proof*⟩

lemma sshiftr_minus_numeral_numeral [simp]:
‹- numeral m >>> numeral n = signed_drop_bit (numeral n) (- numeral
m)›
⟨*proof*⟩

end

5 Word Alignment

theory Aligned
  imports
    "HOL-Library.Word"
    More_Word
    Bit_Shifts_Infix_Syntax

begin

context
  includes bit_operations_syntax
begin

lift_definition is_aligned :: ‹'a::len word ⇒ nat ⇒ bool›
  is ‹λk n. 2 ^ n dvd take_bit LENGTH('a) k›
  ⟨*proof*⟩

lemma is_aligned_iff_udvd:
  ‹is_aligned w n ⟷ 2 ^ n udvd w›
  ⟨*proof*⟩

lemma is_aligned_iff_take_bit_eq_0:
  ‹is_aligned w n ⟷ take_bit n w = 0›
  ⟨*proof*⟩

lemma is_aligned_iff_dvd_int:
  ‹is_aligned ptr n ⟷ 2 ^ n dvd uint ptr›
  ⟨*proof*⟩

lemma is_aligned_iff_dvd_nat:
  ‹is_aligned ptr n ⟷ 2 ^ n dvd unat ptr›
⟨*proof*⟩

lemma is_aligned_0 [simp]:
  ‹is_aligned 0 n›
  ⟨*proof*⟩

lemma is_aligned_at_0 [simp]:
  ‹is_aligned w 0›
  ⟨*proof*⟩

lemma is_aligned_beyond_length:
  ‹is_aligned w n ⟷ w = 0› **if** ‹LENGTH('a) ≤ n› **for** w :: ‹'a::len word›
  ⟨*proof*⟩

lemma is_alignedI [intro?]:
  ‹is_aligned x n› **if** ‹x = 2 ^ n * k› **for** x :: ‹'a::len word›
⟨*proof*⟩

lemma is_alignedE:
fixes w :: ‹'a::len word›
assumes ‹is_aligned w n›
obtains q **where** ‹w = 2 ^ n * word_of_nat q› ‹q < 2 ^ (LENGTH('a) - n)›
⟨*proof*⟩

lemma is_alignedE' [elim?]:
fixes w :: ‹'a::len word›
assumes ‹is_aligned w n›
obtains q **where** ‹w = push_bit n (word_of_nat q)› ‹q < 2 ^ (LENGTH('a) - n)›
⟨*proof*⟩

lemma is_aligned_mask:
‹is_aligned w n ⟷ w AND mask n = 0›
⟨*proof*⟩

lemma is_aligned_imp_not_bit:
‹¬ bit w m› **if** ‹is_aligned w n› **and** ‹m < n›
for w :: ‹'a::len word›
⟨*proof*⟩

lemma is_aligned_weaken:
"⟦ is_aligned w x; x ≥ y ⟧ ⟹ is_aligned w y"
⟨*proof*⟩

lemma is_alignedE_pre:
fixes w::"'a::len word"
assumes aligned: "is_aligned w n"
shows rl: "∃q. w = 2 ^ n * (of_nat q) ∧ q < 2 ^ (LENGTH('a) - n)"
⟨*proof*⟩

lemma aligned_add_aligned:
fixes x::"'a::len word"
assumes aligned1: "is_aligned x n"
and aligned2: "is_aligned y m"
and lt: "m ≤ n"
shows "is_aligned (x + y) m"
⟨*proof*⟩

corollary aligned_sub_aligned:
"⟦is_aligned (x::'a::len word) n; is_aligned y m; m ≤ n⟧
⟹ is_aligned (x - y) m"
⟨*proof*⟩

lemma is_aligned_shift:

fixes k::"'a::len word"
shows "is_aligned (k << m) m"
⟨*proof*⟩

lemma aligned_mod_eq_0:
fixes p::"'a::len word"
assumes al: "is_aligned p sz"
shows "p mod 2 ^ sz = 0"
⟨*proof*⟩

lemma is_aligned_triv: "is_aligned (2 ^ n ::'a::len word) n"
⟨*proof*⟩

lemma is_aligned_mult_triv1: "is_aligned (2 ^ n * x ::'a::len word) n"
⟨*proof*⟩

lemma is_aligned_mult_triv2: "is_aligned (x * 2 ^ n ::'a::len word) n"
⟨*proof*⟩

lemma word_power_less_0_is_0:
fixes x :: "'a::len word"
shows "x < a ^ 0 ⟹ x = 0" ⟨*proof*⟩

lemma is_aligned_no_wrap:
fixes off :: "'a::len word"
fixes ptr :: "'a::len word"
assumes al: "is_aligned ptr sz"
and off: "off < 2 ^ sz"
shows "unat ptr + unat off < 2 ^ LENGTH('a)"
⟨*proof*⟩

lemma is_aligned_no_wrap':
fixes ptr :: "'a::len word"
assumes al: "is_aligned ptr sz"
and off: "off < 2 ^ sz"
shows "ptr ≤ ptr + off"
⟨*proof*⟩

lemma is_aligned_no_overflow':
fixes p :: "'a::len word"
assumes al: "is_aligned p n"
shows "p ≤ p + (2 ^ n - 1)"
⟨*proof*⟩

lemma is_aligned_no_overflow:
"is_aligned ptr sz ⟹ ptr ≤ ptr + 2^sz - 1"
⟨*proof*⟩

lemma replicate_not_True:
"⋀n. xs = replicate n False ⟹ True ∉ set xs"
⟨*proof*⟩

lemma map_zip_replicate_False_xor:
"n = length xs ⟹ map (λ(x, y). x = (¬ y)) (zip xs (replicate n False)) = xs"
⟨*proof*⟩

lemma drop_minus_lem:
"⟦ n ≤ length xs; 0 < n; n' = length xs ⟧ ⟹ drop (n' - n) xs = rev xs ! (n - 1) # drop (Suc (n' - n)) xs"
⟨*proof*⟩

lemma drop_minus:
"⟦ n < length xs; n' = length xs ⟧ ⟹ drop (n' - Suc n) xs = rev xs ! n # drop (n' - n) xs"
⟨*proof*⟩

lemma aligned_add_xor:
‹(x + 2 ^ n) XOR 2 ^ n = x›
if al: ‹is_aligned (x::'a::len word) n'› **and** le: ‹n < n'›
⟨*proof*⟩

lemma is_aligned_add_mult_multI:
fixes p :: "'a::len word"
shows "⟦is_aligned p m; n ≤ m; n' = n⟧ ⟹ is_aligned (p + x * 2 ^ n * z) n'"
⟨*proof*⟩

lemma is_aligned_add_multI:
fixes p :: "'a::len word"
shows "⟦is_aligned p m; n ≤ m; n' = n⟧ ⟹ is_aligned (p + x * 2 ^ n) n'"
⟨*proof*⟩

lemma is_aligned_no_wrap''':
fixes ptr :: "'a::len word"
shows "⟦ is_aligned ptr sz; sz < LENGTH('a); off < 2 ^ sz ⟧
⟹ unat ptr + off < 2 ^ LENGTH('a)"
⟨*proof*⟩

lemma is_aligned_get_word_bits:
fixes p :: "'a::len word"
assumes "is_aligned p n"
obtains "is_aligned p n" "n < LENGTH('a)" | "is_aligned p n" "p = 0" "n ≥ LENGTH('a)"

⟨*proof*⟩

lemma aligned_small_is_0:
"⟦ is_aligned x n; x < 2 ^ n ⟧ ⟹ x = 0"
⟨*proof*⟩

corollary is_aligned_less_sz:
"⟦is_aligned a sz; a ≠ 0⟧ ⟹ ¬ a < 2 ^ sz"
⟨*proof*⟩

lemma aligned_at_least_t2n_diff:
assumes x: "is_aligned x n"
and y: "is_aligned y n"
and "x < y"
shows "x ≤ y - 2 ^ n"
⟨*proof*⟩

lemma is_aligned_no_overflow'':
"⟦is_aligned x n; x + 2 ^ n ≠ 0⟧ ⟹ x ≤ x + 2 ^ n"
⟨*proof*⟩

lemma is_aligned_bitI:
‹is_aligned p m› **if** ‹⋀n. n < m ⟹ ¬ bit p n›
⟨*proof*⟩

lemma is_aligned_nth:
"is_aligned p m = (∀n < m. ¬ bit p n)"
⟨*proof*⟩

lemma range_inter:
"({a..b} ∩ {c..d} = {}) = (∀x. ¬(a ≤ x ∧ x ≤ b ∧ c ≤ x ∧ x ≤ d))"
⟨*proof*⟩

lemma aligned_inter_non_empty:
"⟦ {p..p + (2 ^ n - 1)} ∩ {p..p + 2 ^ m - 1} = {};
is_aligned p n; is_aligned p m⟧ ⟹ False"
⟨*proof*⟩

lemma not_aligned_mod_nz:
assumes al: "¬ is_aligned a n"
shows "a mod 2 ^ n ≠ 0"
⟨*proof*⟩

lemma nat_add_offset_le:
fixes x :: nat
assumes yv: "y ≤ 2 ^ n"
and xv: "x < 2 ^ m"
and mn: "sz = m + n"
shows "x * 2 ^ n + y ≤ 2 ^ sz"

⟨*proof*⟩

lemma is_aligned_no_wrap_le:
  fixes ptr::"'a::len word"
  assumes al: "is_aligned ptr sz"
  and szv: "sz < LENGTH('a)"
  and off: "off ≤ 2 ^ sz"
  shows "unat ptr + off ≤ 2 ^ LENGTH('a)"
⟨*proof*⟩

lemma is_aligned_neg_mask:
  "m ≤ n ⟹ is_aligned (x AND NOT (mask n)) m"
  ⟨*proof*⟩

lemma unat_minus:
  "unat (- (x :: 'a :: len word)) = (if x = 0 then 0 else 2 ^ size x -
unat x)"
  ⟨*proof*⟩

lemma is_aligned_minus:
  ‹is_aligned (- p) n› **if** ‹is_aligned p n› **for** p :: ‹'a::len word›
  ⟨*proof*⟩

lemma add_mask_lower_bits:
  fixes x :: "'a :: len word"
  assumes "is_aligned x n"
    and "∀n'≥n. n' < LENGTH('a) ⟶ ¬ bit p n'"
  shows "x + p AND NOT (mask n) = x"
⟨*proof*⟩

lemma is_aligned_andI1:
  "is_aligned x n ⟹ is_aligned (x AND y) n"
  ⟨*proof*⟩

lemma is_aligned_andI2:
  "is_aligned y n ⟹ is_aligned (x AND y) n"
  ⟨*proof*⟩

lemma is_aligned_shiftl:
  "is_aligned w (n - m) ⟹ is_aligned (w << m) n"
  ⟨*proof*⟩

lemma is_aligned_shiftr:
  "is_aligned w (n + m) ⟹ is_aligned (w >> m) n"
  ⟨*proof*⟩

lemma is_aligned_shiftl_self:
  "is_aligned (p << n) n"
  ⟨*proof*⟩

lemma is_aligned_neg_mask_eq:
"is_aligned p n ⟹ p AND NOT (mask n) = p"
⟨*proof*⟩

lemma is_aligned_shiftr_shiftl:
"is_aligned w n ⟹ w >> n << n = w"
⟨*proof*⟩

lemma aligned_shiftr_mask_shiftl:
assumes "is_aligned x n"
shows "((x >> n) AND mask v) << n = x AND mask (v + n)"
⟨*proof*⟩

lemma mask_zero:
"is_aligned x a ⟹ x AND mask a = 0"
⟨*proof*⟩

lemma is_aligned_neg_mask_eq_concrete:
"⟦ is_aligned p n; msk AND NOT (mask n) = NOT (mask n) ⟧
⟹ p AND msk = p"
⟨*proof*⟩

lemma is_aligned_and_not_zero:
"⟦ is_aligned n k; n ≠ 0 ⟧ ⟹ 2 ^ k ≤ n"
⟨*proof*⟩

lemma is_aligned_and_2_to_k:
"(n AND 2 ^ k - 1) = 0 ⟹ is_aligned (n :: 'a :: len word) k"
⟨*proof*⟩

lemma is_aligned_power2:
"b ≤ a ⟹ is_aligned (2 ^ a) b"
⟨*proof*⟩

lemma aligned_sub_aligned':
"⟦ is_aligned (a :: 'a :: len word) n; is_aligned b n; n < LENGTH('a)
⟧
⟹ is_aligned (a - b) n"
⟨*proof*⟩

lemma is_aligned_neg_mask_weaken:
"⟦ is_aligned p n; m ≤ n ⟧ ⟹ p AND NOT (mask m) = p"
⟨*proof*⟩

lemma is_aligned_neg_mask2 [simp]:
"is_aligned (a AND NOT (mask n)) n"
⟨*proof*⟩

lemma is_aligned_0':
"is_aligned 0 n"
⟨*proof*⟩

lemma aligned_add_offset_no_wrap:
fixes off :: "('a::len) word"
and x :: "'a word"
assumes al: "is_aligned x sz"
and offv: "off < 2 ^ sz"
shows "unat x + unat off < 2 ^ LENGTH('a)"
⟨*proof*⟩

lemma aligned_add_offset_mod:
fixes x :: "('a::len) word"
assumes al: "is_aligned x sz"
and kv: "k < 2 ^ sz"
shows "(x + k) mod 2 ^ sz = k"
⟨*proof*⟩

lemma aligned_neq_into_no_overlap:
fixes x :: "'a::len word"
assumes neq: "x ≠ y"
and alx: "is_aligned x sz"
and aly: "is_aligned y sz"
shows "{x .. x + (2 ^ sz - 1)} ∩ {y .. y + (2 ^ sz - 1)} = {}"
⟨*proof*⟩

lemma is_aligned_add_helper:
"⟦ is_aligned p n; d < 2 ^ n ⟧
⟹ (p + d AND mask n = d) ∧ (p + d AND (NOT (mask n)) = p)"
⟨*proof*⟩

lemmas mask_inner_mask = mask_eqs(1)

lemma mask_add_aligned:
"is_aligned p n ⟹ (p + q) AND mask n = q AND mask n"
⟨*proof*⟩

lemma mask_out_add_aligned:
assumes al: "is_aligned p n"
shows "p + (q AND NOT (mask n)) = (p + q) AND NOT (mask n)"
⟨*proof*⟩

lemma is_aligned_add_or:
"⟦is_aligned p n; d < 2 ^ n⟧ ⟹ p + d = p OR d"
⟨*proof*⟩

lemma not_greatest_aligned:
"⟦ x < y; is_aligned x n; is_aligned y n ⟧ ⟹ x + 2 ^ n ≠ 0"

⟨*proof*⟩

lemma neg_mask_mono_le:
"x ≤ y ⟹ x AND NOT(mask n) ≤ y AND NOT(mask n)" **for** x :: "'a :: len word"
⟨*proof*⟩

lemma and_neg_mask_eq_iff_not_mask_le:
"w AND NOT(mask n) = NOT(mask n) ⟷ NOT(mask n) ≤ w"
for w :: ‹'a::len word›
⟨*proof*⟩

lemma neg_mask_le_high_bits:
‹NOT (mask n) ≤ w ⟷ (∀i ∈ {n ..< size w}. bit w i)› (**is** ‹?P ⟷ ?Q›)
for w :: ‹'a::len word›
⟨*proof*⟩

lemma is_aligned_add_less_t2n:
fixes p :: "'a::len word"
assumes "is_aligned p n"
and "d < 2 ^ n"
and "n ≤ m"
and "p < 2 ^ m"
shows "p + d < 2^m"
⟨*proof*⟩

lemma aligned_offset_non_zero:
"⟦ is_aligned x n; y < 2 ^ n; x ≠ 0 ⟧ ⟹ x + y ≠ 0"
⟨*proof*⟩

lemma is_aligned_over_length:
"⟦ is_aligned p n; LENGTH('a) ≤ n ⟧ ⟹ (p::'a::len word) = 0"
⟨*proof*⟩

lemma is_aligned_no_overflow_mask:
"is_aligned x n ⟹ x ≤ x + mask n"
⟨*proof*⟩

lemma aligned_mask_step:
fixes p' :: "'a::len word"
assumes "n' ≤ n"
and "p' ≤ p + mask n"
and "is_aligned p n"
and "is_aligned p' n'"
shows "p' + mask n' ≤ p + mask n"
⟨*proof*⟩

lemma is_aligned_mask_offset_unat:

```
  fixes off :: "('a::len) word"
  and     x :: "'a word"
  assumes al: "is_aligned x sz"
  and   offv: "off ≤ mask sz"
  shows  "unat x + unat off < 2 ^ LENGTH('a)"
  ⟨proof⟩

lemma aligned_less_plus_1:
  assumes "is_aligned x n" and "0 < n"
  shows "x < x + 1"
⟨proof⟩

lemma aligned_add_offset_less:
  assumes x: "is_aligned x n"
      and y: "is_aligned y n"
      and "x < y"
      and z: "z < 2 ^ n"
    shows "x + z < y"
⟨proof⟩

lemma gap_between_aligned:
  "⟦a < (b :: 'a ::len word); is_aligned a n; is_aligned b n; n < LENGTH('a)
⟧
  ⟹ a + (2^n - 1) < b"
  ⟨proof⟩

lemma is_aligned_add_step_le:
  "⟦ is_aligned (a::'a::len word) n; is_aligned b n; a < b; b ≤ a + mask
n ⟧ ⟹ False"
  ⟨proof⟩

lemma aligned_add_mask_lessD:
  "⟦ x + mask n < y; is_aligned x n ⟧ ⟹ x < y" for y::"'a::len word"
  ⟨proof⟩

lemma aligned_add_mask_less_eq:
  "⟦ is_aligned x n; is_aligned y n;  n < LENGTH('a) ⟧ ⟹ (x + mask n
< y) = (x < y)"
  for y::"'a::len word"
  ⟨proof⟩

lemma is_aligned_diff:
  fixes m :: "'a::len word"
  assumes alm: "is_aligned m s1"
  and     aln: "is_aligned n s2"
  and    s2wb: "s2 < LENGTH('a)"
  and      nm: "m ∈ {n .. n + (2 ^ s2 - 1)}"
  and    s1s2: "s1 ≤ s2"
shows  "∃q. m - n = of_nat q * 2 ^ s1 ∧ q < 2 ^ (s2 - s1)"
```

⟨*proof*⟩

lemma is_aligned_addD1:
  assumes al1: "is_aligned (x + y) n"
  and al2: "is_aligned (x::'a::len word) n"
  shows "is_aligned y n"
  ⟨*proof*⟩

lemmas is_aligned_addD2 =
      is_aligned_addD1[OF subst[OF add.commute,
                                of "λx. is_aligned x n" **for** n]]

lemma is_aligned_add:
  "⟦is_aligned p n; is_aligned q n⟧ ⟹ is_aligned (p + q) n"
  ⟨*proof*⟩

lemma aligned_shift:
  fixes x :: "'a::len word"
  assumes x: "x < 2 ^ n"
      and "is_aligned y n"
      and "n ≤ LENGTH('a::len)"
    shows "(x + y) >> n = y >> n"
⟨*proof*⟩

lemma aligned_shift':
  "⟦x < 2 ^ n; is_aligned (y :: 'a :: len word) n;n ≤ LENGTH('a)⟧
   ⟹ (y + x) >> n = y >> n"
  ⟨*proof*⟩

lemma and_neg_mask_plus_mask_mono: "(p AND NOT (mask n)) + mask n ≥
p"
  for p :: ‹'a::len word›
  ⟨*proof*⟩

lemma word_neg_and_le:
  "ptr ≤ (ptr AND NOT (mask n)) + (2 ^ n - 1)"
  for ptr :: ‹'a::len word›
  ⟨*proof*⟩

lemma is_aligned_sub_helper:
  "⟦ is_aligned (p - d) n; d < 2 ^ n ⟧
     ⟹ (p AND mask n = d) ∧ (p AND (NOT (mask n)) = p - d)"
  ⟨*proof*⟩

lemma is_aligned_after_mask:
  "⟦is_aligned k m;m≤ n⟧ ⟹ is_aligned (k AND mask n) m"
  ⟨*proof*⟩

lemma and_mask_plus:

assumes "is_aligned ptr m"
and "m ≤ n"
and "a < 2 ^ m"
shows "ptr + a AND mask n = (ptr AND mask n) + a"
⟨*proof*⟩

lemma is_aligned_add_not_aligned:
"⟦is_aligned (p::'a::len word) n; ¬ is_aligned (q::'a::len word) n⟧
⟹ ¬ is_aligned (p + q) n"
⟨*proof*⟩

lemma neg_mask_add_aligned:
"⟦ is_aligned p n; q < 2 ^ n ⟧ ⟹ (p + q) AND NOT (mask n) = p AND NOT
(mask n)"
⟨*proof*⟩

lemma word_add_power_off:
fixes a :: "'a :: len word"
assumes ak: "a < k"
and kw: "k < 2 ^ (LENGTH('a) - m)"
and mw: "m < LENGTH('a)"
and off: "off < 2 ^ m"
shows "(a * 2 ^ m) + off < k * 2 ^ m"
⟨*proof*⟩

lemma offset_not_aligned:
"⟦ is_aligned (p::'a::len word) n; i > 0; i < 2 ^ n; n < LENGTH('a)⟧
⟹
¬ is_aligned (p + of_nat i) n"
⟨*proof*⟩

lemma le_or_mask:
"w ≤ w' ⟹ w OR mask x ≤ w' OR mask x"
for w w' :: ‹'a::len word›
⟨*proof*⟩

end

end

6 Increment and Decrement Machine Words Without Wrap-Around

theory Next_and_Prev
imports
Aligned
begin

Previous and next words addresses, without wrap around.

lift_definition word_next :: ‹'a::len word ⇒ 'a word›
  is ‹λk. if 2 ^ LENGTH('a) dvd k + 1 then - 1 else k + 1›
  ⟨*proof*⟩

lift_definition word_prev :: ‹'a::len word ⇒ 'a word›
  is ‹λk. if 2 ^ LENGTH('a) dvd k then 0 else k - 1›
  ⟨*proof*⟩

lemma word_next_unfold:
  ‹word_next w = (if w = - 1 then - 1 else w + 1)›
  ⟨*proof*⟩

lemma word_prev_unfold:
  ‹word_prev w = (if w = 0 then 0 else w - 1)›
  ⟨*proof*⟩

lemma [code]:
  ‹Word.the_int (word_next w :: 'a::len word) =
    (if w = - 1 then Word.the_int w else Word.the_int w + 1)›
  ⟨*proof*⟩

lemma [code]:
  ‹Word.the_int (word_prev w :: 'a::len word) =
    (if w = 0 then Word.the_int w else Word.the_int w - 1)›
  ⟨*proof*⟩

lemma word_adjacent_union:
  "word_next e = s' ⟹ s ≤ e ⟹ s' ≤ e' ⟹ {s..e} ∪ {s'..e'} = {s
.. e'}"
  ⟨*proof*⟩

end

7 Signed division on word

theory Signed_Division_Word
  imports "HOL-Library.Signed_Division" "HOL-Library.Word"
begin

The following specification of division follows ISO C99, which in turn adopted the typical behavior of hardware modern in the beginning of the 1990ies. The underlying integer division is named "T-division" in [1].

instantiation word :: (len) signed_division
begin

lift_definition signed_divide_word :: ‹'a::len word ⇒ 'a word ⇒ 'a word›
  is ‹λk l. signed_take_bit (LENGTH('a) - Suc 0) k sdiv signed_take_bit
(LENGTH('a) - Suc 0) l›

⟨*proof*⟩

lift_definition signed_modulo_word :: ‹'a::len word ⇒ 'a word ⇒ 'a word›
  is ‹λk l. signed_take_bit (LENGTH('a) - Suc 0) k smod signed_take_bit
(LENGTH('a) - Suc 0) l›
  ⟨*proof*⟩

lemma sdiv_word_def:
  ‹v sdiv w = word_of_int (sint v sdiv sint w)›
  for v w :: ‹'a::len word›
  ⟨*proof*⟩

lemma smod_word_def:
  ‹v smod w = word_of_int (sint v smod sint w)›
  for v w :: ‹'a::len word›
  ⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma signed_divide_word_code [code]:
  ‹v sdiv w =
   (let v' = sint v; w' = sint w;
        negative = (v' < 0) ≠ (w' < 0);
        result = |v'| div |w'|
    in word_of_int (if negative then - result else result))›
  for v w :: ‹'a::len word›
  ⟨*proof*⟩

lemma signed_modulo_word_code [code]:
  ‹v smod w =
   (let v' = sint v; w' = sint w;
        negative = (v' < 0);
        result = |v'| mod |w'|
    in word_of_int (if negative then - result else result))›
  for v w :: ‹'a::len word›
  ⟨*proof*⟩

lemma sdiv_smod_id:
  ‹(a sdiv b) * b + (a smod b) = a›
  for a b :: ‹'a::len word›
  ⟨*proof*⟩

lemma signed_div_arith:
    "sint ((a::('a::len) word) sdiv b) = signed_take_bit (LENGTH('a) -
1) (sint a sdiv sint b)"
  ⟨*proof*⟩

lemma signed_mod_arith:
"sint ((a::('a::len) word) smod b) = signed_take_bit (LENGTH('a) - 1) (sint a smod sint b)"
⟨*proof*⟩

lemma word_sdiv_div0 [simp]:
"(a :: ('a::len) word) sdiv 0 = 0"
⟨*proof*⟩

lemma smod_word_zero [simp]:
‹w smod 0 = w› **for** w :: ‹'a::len word›
⟨*proof*⟩

lemma word_sdiv_div1 [simp]:
"(a :: ('a::len) word) sdiv 1 = a"
⟨*proof*⟩

lemma smod_word_one [simp]:
‹w smod 1 = 0› **for** w :: ‹'a::len word›
⟨*proof*⟩

lemma word_sdiv_div_minus1 [simp]:
"(a :: ('a::len) word) sdiv -1 = -a"
⟨*proof*⟩

lemma smod_word_minus_one [simp]:
‹w smod - 1 = 0› **for** w :: ‹'a::len word›
⟨*proof*⟩

lemma one_sdiv_word_eq [simp]:
‹1 sdiv w = of_bool (w = 1 ∨ w = - 1) * w› **for** w :: ‹'a::len word›
⟨*proof*⟩

lemma one_smod_word_eq [simp]:
‹1 smod w = 1 - of_bool (w = 1 ∨ w = - 1)› **for** w :: ‹'a::len word›
⟨*proof*⟩

lemma minus_one_sdiv_word_eq [simp]:
‹- 1 sdiv w = - (1 sdiv w)› **for** w :: ‹'a::len word›
⟨*proof*⟩

lemma minus_one_smod_word_eq [simp]:
‹- 1 smod w = - (1 smod w)› **for** w :: ‹'a::len word›
⟨*proof*⟩

lemma smod_word_alt_def:
"(a :: ('a::len) word) smod b = a - (a sdiv b) * b"
⟨*proof*⟩

lemmas sdiv_word_numeral_numeral [simp] =
  sdiv_word_def [of ‹numeral a› ‹numeral b›, simplified sint_sbintrunc sint_sbintrunc_neg]
  for a b
lemmas sdiv_word_minus_numeral_numeral [simp] =
  sdiv_word_def [of ‹- numeral a› ‹numeral b›, simplified sint_sbintrunc sint_sbintrunc_neg]
  for a b
lemmas sdiv_word_numeral_minus_numeral [simp] =
  sdiv_word_def [of ‹numeral a› ‹- numeral b›, simplified sint_sbintrunc sint_sbintrunc_neg]
  for a b
lemmas sdiv_word_minus_numeral_minus_numeral [simp] =
  sdiv_word_def [of ‹- numeral a› ‹- numeral b›, simplified sint_sbintrunc sint_sbintrunc_neg]
  for a b

lemmas smod_word_numeral_numeral [simp] =
  smod_word_def [of ‹numeral a› ‹numeral b›, simplified sint_sbintrunc sint_sbintrunc_neg]
  for a b
lemmas smod_word_minus_numeral_numeral [simp] =
  smod_word_def [of ‹- numeral a› ‹numeral b›, simplified sint_sbintrunc sint_sbintrunc_neg]
  for a b
lemmas smod_word_numeral_minus_numeral [simp] =
  smod_word_def [of ‹numeral a› ‹- numeral b›, simplified sint_sbintrunc sint_sbintrunc_neg]
  for a b
lemmas smod_word_minus_numeral_minus_numeral [simp] =
  smod_word_def [of ‹- numeral a› ‹- numeral b›, simplified sint_sbintrunc sint_sbintrunc_neg]
  for a b

lemmas word_sdiv_0 = word_sdiv_div0

lemma sdiv_word_min:
    "- (2 ^ (size a - 1)) ≤ sint (a :: ('a::len) word) sdiv sint (b :: ('a::len) word)"
  ⟨*proof*⟩

lemma sdiv_word_max:
  ‹sint a sdiv sint b ≤ 2 ^ (size a - Suc 0)›
  for a b :: ‹'a::len word›
⟨*proof*⟩

lemmas word_sdiv_numerals_lhs = sdiv_word_def[**where** v="numeral x" **for** x]
    sdiv_word_def[**where** v=0] sdiv_word_def[**where** v=1]

lemmas word_sdiv_numerals = word_sdiv_numerals_lhs[**where** w="numeral y" **for** y]
    word_sdiv_numerals_lhs[**where** w=0] word_sdiv_numerals_lhs[**where** w=1]

lemma smod_word_mod_0:
  "x smod (0 :: ('a::len) word) = x"
  ⟨*proof*⟩

lemma smod_word_0_mod [simp]:
  "0 smod (x :: ('a::len) word) = 0"
  ⟨*proof*⟩

lemma smod_word_max:
  "sint (a::'a word) smod sint (b::'a word) < 2 ^ (LENGTH('a::len) - Suc 0)"
⟨*proof*⟩

lemma smod_word_min:
  "- (2 ^ (LENGTH('a::len) - Suc 0)) ≤ sint (a::'a word) smod sint (b::'a word)"
  ⟨*proof*⟩

lemmas word_smod_numerals_lhs = smod_word_def[**where** v="numeral x" **for** x]
    smod_word_def[**where** v=0] smod_word_def[**where** v=1]

lemmas word_smod_numerals = word_smod_numerals_lhs[**where** w="numeral y" **for** y]
    word_smod_numerals_lhs[**where** w=0] word_smod_numerals_lhs[**where** w=1]

end

theory Bitwise
  imports
    "HOL-Library.Word"
    More_Arithmetic
    Reversed_Bit_Lists
    Bit_Shifts_Infix_Syntax

begin

Helper constants used in defining addition

definition xor3 :: "bool ⇒ bool ⇒ bool ⇒ bool"
  where "xor3 a b c = (a = (b = c))"

definition carry :: "bool ⇒ bool ⇒ bool ⇒ bool"
  where "carry a b c = ((a ∧ (b ∨ c)) ∨ (b ∧ c))"

lemma carry_simps:
"carry True a b = (a ∨ b)"
"carry a True b = (a ∨ b)"
"carry a b True = (a ∨ b)"
"carry False a b = (a ∧ b)"
"carry a False b = (a ∧ b)"
"carry a b False = (a ∧ b)"
⟨*proof*⟩

lemma xor3_simps:
"xor3 True a b = (a = b)"
"xor3 a True b = (a = b)"
"xor3 a b True = (a = b)"
"xor3 False a b = (a ≠ b)"
"xor3 a False b = (a ≠ b)"
"xor3 a b False = (a ≠ b)"
⟨*proof*⟩

Breaking up word equalities into equalities on their bit lists. Equalities are generated and manipulated in the reverse order to to_bl.

lemma bl_word_sub: "to_bl (x - y) = to_bl (x + (- y))"
⟨*proof*⟩

lemma rbl_word_1: "rev (to_bl (1 :: 'a::len word)) = takefill False (LENGTH('a)) [True]"
⟨*proof*⟩

lemma rbl_word_if: "rev (to_bl (if P then x else y)) = map2 (If P) (rev (to_bl x)) (rev (to_bl y))"
⟨*proof*⟩

lemma rbl_add_carry_Cons:
"(if car then rbl_succ else id) (rbl_add (x # xs) (y # ys)) =
xor3 x y car # (if carry x y car then rbl_succ else id) (rbl_add xs ys)"
⟨*proof*⟩

lemma rbl_add_suc_carry_fold:
"length xs = length ys ⟹
∀car. (if car then rbl_succ else id) (rbl_add xs ys) =
(foldr (λ(x, y) res car. xor3 x y car # res (carry x y car)) (zip xs ys) (λ_. [])) car"
⟨*proof*⟩

lemma to_bl_plus_carry:
"to_bl (x + y) =
rev (foldr (λ(x, y) res car. xor3 x y car # res (carry x y car))
(rev (zip (to_bl x) (to_bl y))) (λ_. []) False)"

⟨*proof*⟩

definition "rbl_plus cin xs ys =
  foldr (λ(x, y) res car. xor3 x y car # res (carry x y car)) (zip xs ys) (λ_. []) cin"

lemma rbl_plus_simps:
  "rbl_plus cin (x # xs) (y # ys) = xor3 x y cin # rbl_plus (carry x y cin) xs ys"
  "rbl_plus cin [] ys = []"
  "rbl_plus cin xs [] = []"
  ⟨*proof*⟩

lemma rbl_word_plus: "rev (to_bl (x + y)) = rbl_plus False (rev (to_bl x)) (rev (to_bl y))"
  ⟨*proof*⟩

definition "rbl_succ2 b xs = (if b then rbl_succ xs else xs)"

lemma rbl_succ2_simps:
  "rbl_succ2 b [] = []"
  "rbl_succ2 b (x # xs) = (b ≠ x) # rbl_succ2 (x ∧ b) xs"
  ⟨*proof*⟩

lemma twos_complement: "- x = word_succ (not x)"
  ⟨*proof*⟩

lemma rbl_word_neg: "rev (to_bl (- x)) = rbl_succ2 True (map Not (rev (to_bl x)))"
  for x :: ‹'a::len word›
  ⟨*proof*⟩

lemma rbl_word_cat:
  "rev (to_bl (word_cat x y :: 'a::len word)) =
    takefill False (LENGTH('a)) (rev (to_bl y) @ rev (to_bl x))"
  ⟨*proof*⟩

lemma rbl_word_slice:
  "rev (to_bl (slice n w :: 'a::len word)) =
    takefill False (LENGTH('a)) (drop n (rev (to_bl w)))"
  ⟨*proof*⟩

lemma rbl_word_ucast:
  "rev (to_bl (ucast x :: 'a::len word)) = takefill False (LENGTH('a)) (rev (to_bl x))"
  ⟨*proof*⟩

lemma rbl_shiftl:
  "rev (to_bl (w << n)) = takefill False (size w) (replicate n False @

```
rev (to_bl w))"
```
⟨*proof*⟩

lemma rbl_shiftr:
```
  "rev (to_bl (w >> n)) = takefill False (size w) (drop n (rev (to_bl
w)))"
```
⟨*proof*⟩

definition "drop_nonempty v n xs = (if n < length xs then drop n xs else
[last (v # xs)])"

lemma drop_nonempty_simps:
```
  "drop_nonempty v (Suc n) (x # xs) = drop_nonempty x n xs"
  "drop_nonempty v 0 (x # xs) = (x # xs)"
  "drop_nonempty v n [] = [v]"
```
⟨*proof*⟩

definition "takefill_last x n xs = takefill (last (x # xs)) n xs"

lemma takefill_last_simps:
```
  "takefill_last z (Suc n) (x # xs) = x # takefill_last x n xs"
  "takefill_last z 0 xs = []"
  "takefill_last z n [] = replicate n z"
```
⟨*proof*⟩

lemma rbl_sshiftr:
```
  "rev (to_bl (w >>> n)) = takefill_last False (size w) (drop_nonempty
False n (rev (to_bl w)))"
```
⟨*proof*⟩

lemma nth_word_of_int:
```
  "bit (word_of_int x :: 'a::len word) n = (n < LENGTH('a) ∧ bit x n)"
```
⟨*proof*⟩

lemma nth_scast:
```
  "bit (scast (x :: 'a::len word) :: 'b::len word) n =
    (n < LENGTH('b) ∧
    (if n < LENGTH('a) - 1 then bit x n
     else bit x (LENGTH('a) - 1)))"
```
⟨*proof*⟩

lemma rbl_word_scast:
```
  "rev (to_bl (scast x :: 'a::len word)) = takefill_last False (LENGTH('a))
(rev (to_bl x))"
```
⟨*proof*⟩

definition rbl_mul :: "bool list ⇒ bool list ⇒ bool list"
where "rbl_mul xs ys = foldr (λx sm. rbl_plus False (map ((∧) x) ys)
(False # sm)) xs []"

lemma rbl_mul_simps:
"rbl_mul (x # xs) ys = rbl_plus False (map ((∧) x) ys) (False # rbl_mul xs ys)"
"rbl_mul [] ys = []"
⟨*proof*⟩

lemma takefill_le2: "length xs ≤ n ⟹ takefill x m (takefill x n xs) = takefill x m xs"
⟨*proof*⟩

lemma take_rbl_plus: "∀n b. take n (rbl_plus b xs ys) = rbl_plus b (take n xs) (take n ys)"
⟨*proof*⟩

lemma word_rbl_mul_induct:
"length xs ≤ size y ⟹
rbl_mul xs (rev (to_bl y)) = take (length xs) (rev (to_bl (of_bl (rev xs) * y)))"
for y :: "'a::len word"
⟨*proof*⟩

lemma rbl_word_mul: "rev (to_bl (x * y)) = rbl_mul (rev (to_bl x)) (rev (to_bl y))"
for x :: "'a::len word"
⟨*proof*⟩

Breaking up inequalities into bitlist properties.

definition
"rev_bl_order F xs ys =
(length xs = length ys ∧
((xs = ys ∧ F)
∨ (∃n < length xs. drop (Suc n) xs = drop (Suc n) ys
∧ ¬ xs ! n ∧ ys ! n)))"

lemma rev_bl_order_simps:
"rev_bl_order F [] [] = F"
"rev_bl_order F (x # xs) (y # ys) = rev_bl_order ((y ∧ ¬ x) ∨ ((y ∨ ¬ x) ∧ F)) xs ys"
⟨*proof*⟩

lemma rev_bl_order_rev_simp:
"length xs = length ys ⟹
rev_bl_order F (xs @ [x]) (ys @ [y]) = ((y ∧ ¬ x) ∨ ((y ∨ ¬ x) ∧ rev_bl_order F xs ys))"
⟨*proof*⟩

lemma rev_bl_order_bl_to_bin:
"length xs = length ys ⟹

```
    rev_bl_order True xs ys = (bl_to_bin (rev xs) ≤ bl_to_bin (rev ys))
∧
    rev_bl_order False xs ys = (bl_to_bin (rev xs) < bl_to_bin (rev ys))"
```
⟨*proof*⟩

lemma `word_le_rbl: "x ≤ y ⟷ rev_bl_order True (rev (to_bl x)) (rev (to_bl y))"`
  for `x y :: "'a::len word"`
  ⟨*proof*⟩

lemma `word_less_rbl: "x < y ⟷ rev_bl_order False (rev (to_bl x)) (rev (to_bl y))"`
  for `x y :: "'a::len word"`
  ⟨*proof*⟩

definition `"map_last f xs = (if xs = [] then [] else butlast xs @ [f (last xs)])"`

lemma `map_last_simps:`
```
  "map_last f [] = []"
  "map_last f [x] = [f x]"
  "map_last f (x # y # zs) = x # map_last f (y # zs)"
```
  ⟨*proof*⟩

lemma `word_sle_rbl:`
```
  "x <=s y ⟷ rev_bl_order True (map_last Not (rev (to_bl x))) (map_last Not (rev (to_bl y)))"
```
⟨*proof*⟩

lemma `word_sless_rbl:`
```
  "x <s y ⟷ rev_bl_order False (map_last Not (rev (to_bl x))) (map_last Not (rev (to_bl y)))"
```
  ⟨*proof*⟩

Lemmas for unpacking `rev (to_bl n)` for numerals n and also for irreducible values and expressions.

lemma `rev_bin_to_bl_simps:`
```
  "rev (bin_to_bl 0 x) = []"
  "rev (bin_to_bl (Suc n) (numeral (num.Bit0 nm))) = False # rev (bin_to_bl n (numeral nm))"
  "rev (bin_to_bl (Suc n) (numeral (num.Bit1 nm))) = True # rev (bin_to_bl n (numeral nm))"
  "rev (bin_to_bl (Suc n) (numeral (num.One))) = True # replicate n False"
  "rev (bin_to_bl (Suc n) (- numeral (num.Bit0 nm))) = False # rev (bin_to_bl n (- numeral nm))"
  "rev (bin_to_bl (Suc n) (- numeral (num.Bit1 nm))) =
    True # rev (bin_to_bl n (- numeral (nm + num.One)))"
  "rev (bin_to_bl (Suc n) (- numeral (num.One))) = True # replicate n True"
```

```
  "rev (bin_to_bl (Suc n) (- numeral (num.Bit0 nm + num.One))) =
    True # rev (bin_to_bl n (- numeral (nm + num.One)))"
  "rev (bin_to_bl (Suc n) (- numeral (num.Bit1 nm + num.One))) =
    False # rev (bin_to_bl n (- numeral (nm + num.One)))"
  "rev (bin_to_bl (Suc n) (- numeral (num.One + num.One))) =
    False # rev (bin_to_bl n (- numeral num.One))"
```
⟨*proof*⟩

lemma `to_bl_upt: "to_bl x = rev (map (bit x) [0 ..< size x])"`
⟨*proof*⟩

lemma `rev_to_bl_upt: "rev (to_bl x) = map (bit x) [0 ..< size x]"`
⟨*proof*⟩

lemma `upt_eq_list_intros:`
```
  "j ≤ i ⟹ [i ..< j] = []"
  "i = x ⟹ x < j ⟹ [x + 1 ..< j] = xs ⟹ [i ..< j] = (x # xs)"
```
⟨*proof*⟩

Tactic definition

lemma `if_bool_simps:`
```
  "If p True y = (p ∨ y) ∧ If p False y = (¬ p ∧ y) ∧
    If p y True = (p ⟶ y) ∧ If p y False = (p ∧ y)"
```
⟨*proof*⟩

⟨*ML*⟩

end

8 Comprehension syntax for `int`

theory `Bit_Comprehension_Int`
imports
`Bit_Comprehension`
begin

instantiation `int :: bit_comprehension`
begin

definition
```
  ‹set_bits f = (
      if ∃n. ∀m≥n. f m = f n then
      let n = LEAST n. ∀m≥n. f m = f n
      in signed_take_bit n (horner_sum of_bool 2 (map f [0..<Suc n]))
     else 0 :: int)›
```

instance ⟨*proof*⟩

end

lemma int_set_bits_K_False [simp]: "(BITS _. False) = (0 :: int)"
⟨*proof*⟩

lemma int_set_bits_K_True [simp]: "(BITS _. True) = (-1 :: int)"
⟨*proof*⟩

lemma set_bits_code [code]:
"set_bits = Code.abort (STR ''set_bits is unsupported on type int'')
(λ_. set_bits :: _ ⇒ int)"
⟨*proof*⟩

lemma set_bits_int_unfold':
‹set_bits f =
(if ∃n. ∀n'≥n. ¬ f n' then
let n = LEAST n. ∀n'≥n. ¬ f n'
in horner_sum of_bool 2 (map f [0..<n])
else if ∃n. ∀n'≥n. f n' then
let n = LEAST n. ∀n'≥n. f n'
in signed_take_bit n (horner_sum of_bool 2 (map f [0..<n] @ [True]))
else 0 :: int)›
⟨*proof*⟩

inductive wf_set_bits_int :: "(nat ⇒ bool) ⇒ bool"
for f :: "nat ⇒ bool"
where
zeros: "∀n' ≥ n. ¬ f n' ⟹ wf_set_bits_int f"
| ones: "∀n' ≥ n. f n' ⟹ wf_set_bits_int f"

lemma wf_set_bits_int_simps: "wf_set_bits_int f ⟷ (∃n. (∀n'≥n. ¬
f n') ∨ (∀n'≥n. f n'))"
⟨*proof*⟩

lemma wf_set_bits_int_const [simp]: "wf_set_bits_int (λ_. b)"
⟨*proof*⟩

lemma wf_set_bits_int_fun_upd [simp]:
"wf_set_bits_int (f(n := b)) ⟷ wf_set_bits_int f" (**is** "?lhs ⟷ ?rhs")
⟨*proof*⟩

lemma wf_set_bits_int_Suc [simp]:
"wf_set_bits_int (λn. f (Suc n)) ⟷ wf_set_bits_int f" (**is** "?lhs ⟷
?rhs")
⟨*proof*⟩

context
fixes f
assumes wff: "wf_set_bits_int f"
begin

lemma int_set_bits_unfold_BIT:
  "set_bits f = of_bool (f 0) + (2 :: int) * set_bits (f ∘ Suc)"
⟨*proof*⟩

lemma bin_last_set_bits [simp]:
  "odd (set_bits f :: int) = f 0"
  ⟨*proof*⟩

lemma bin_rest_set_bits [simp]:
  "set_bits f div (2 :: int) = set_bits (f ∘ Suc)"
  ⟨*proof*⟩

lemma bin_nth_set_bits [simp]:
  "bit (set_bits f :: int) m ⟷ f m"
⟨*proof*⟩

end

end

9 Signed Words

theory Signed_Words
  imports "HOL-Library.Word"
begin

Signed words as separate (isomorphic) word length class. Useful for tagging words in C.

typedef ('a::len0) signed = "UNIV :: 'a set" ⟨*proof*⟩

lemma card_signed [simp]: "CARD (('a::len0) signed) = CARD('a)"
  ⟨*proof*⟩

instantiation signed :: (len0) len0
begin

definition
  len_signed [simp]: "len_of (x::'a::len0 signed itself) = LENGTH('a)"

instance ⟨*proof*⟩

end

instance signed :: (len) len
  ⟨*proof*⟩

lemma scast_scast_id [simp]:
  "scast (scast x :: ('a::len) signed word) = (x :: 'a word)"

```
  "scast (scast y :: ('a::len) word) = (y :: 'a signed word)"
```
⟨*proof*⟩

lemma ucast_scast_id [simp]:
```
  "ucast (scast (x :: 'a::len signed word) :: 'a word) = x"
```
⟨*proof*⟩

lemma scast_of_nat [simp]:
```
  "scast (of_nat x :: 'a::len signed word) = (of_nat x :: 'a word)"
```
⟨*proof*⟩

lemma scast_ucast_id [simp]:
```
  "scast (ucast (x :: 'a::len word) :: 'a signed word) = x"
```
⟨*proof*⟩

lemma scast_eq_scast_id [simp]:
```
  "((scast (a :: 'a::len signed word) :: 'a word) = scast b) = (a = b)"
```
⟨*proof*⟩

lemma ucast_eq_ucast_id [simp]:
```
  "((ucast (a :: 'a::len word) :: 'a signed word) = ucast b) = (a = b)"
```
⟨*proof*⟩

lemma scast_ucast_norm [simp]:
```
  "(ucast (a :: 'a::len word) = (b :: 'a signed word)) = (a = scast b)"
  "((b :: 'a signed word) = ucast (a :: 'a::len word)) = (a = scast b)"
```
⟨*proof*⟩

lemma scast_2_power [simp]: "scast ((2 :: 'a::len signed word) ^ x) =
((2 :: 'a word) ^ x)"
⟨*proof*⟩

lemma ucast_nat_def':
```
  "of_nat (unat x) = (ucast :: 'a :: len word ⇒ ('b :: len) signed word)
x"
```
⟨*proof*⟩

lemma zero_sle_ucast_up:
```
  "¬ is_down (ucast :: 'a word ⇒ 'b signed word) ⟹
          (0 <=s ((ucast (b::('a::len) word)) :: ('b::len) signed word))"
```
⟨*proof*⟩

lemma word_le_ucast_sless:
```
  "⟦ x ≤ y; y ≠ -1; LENGTH('a) < LENGTH('b) ⟧ ⟹
    (ucast x :: ('b :: len) signed word) <s ucast (y + 1)"
  for x y :: ‹'a::len word›
```
⟨*proof*⟩

lemma zero_sle_ucast:

```
  "(0 <=s ((ucast (b::('a::len) word)) :: ('a::len) signed word))
               = (uint b < 2 ^ (LENGTH('a) - 1))"
```
⟨*proof*⟩

lemma nth_w2p_scast:
```
  "(bit (scast ((2::'a::len signed word) ^ n) :: 'a word) m)
        ⟷ (bit (((2::'a::len  word) ^ n) :: 'a word) m)"
```
⟨*proof*⟩

lemma scast_nop_1 [simp]:
```
  "((scast ((of_int x)::('a::len) word))::'a signed word) = of_int x"
```
⟨*proof*⟩

lemma scast_nop_2 [simp]:
```
  "((scast ((of_int x)::('a::len) signed word))::'a word) = of_int x"
```
⟨*proof*⟩

lemmas scast_nop = scast_nop_1 scast_nop_2 scast_id

type_synonym 'a sword = "'a signed word"

end

10 Bitwise tactic for Signed Words

theory Bitwise_Signed
imports
```
  "HOL-Library.Word"
  Bitwise
  Signed_Words
```
begin

⟨*ML*⟩

end

11 Enumeration Instances for Words

theory Enumeration_Word
  imports More_Word Enumeration Even_More_List
begin

lemma length_word_enum: "length (enum :: 'a :: len word list) = 2 ^ LENGTH('a)"
⟨*proof*⟩

lemma fromEnum_unat[simp]: "fromEnum (x :: 'a::len word) = unat x"
⟨*proof*⟩

lemma toEnum_of_nat[simp]: "n < 2 ^ LENGTH('a) ⟹ (toEnum n :: 'a :: len word) = of_nat n"
⟨*proof*⟩

instantiation word :: (len) enumeration_both
begin

definition
enum_alt_word_def: "enum_alt ≡ alt_from_ord (enum :: ('a :: len) word list)"

instance
⟨*proof*⟩

end

definition
upto_enum_step :: "('a :: len) word ⇒ 'a word ⇒ 'a word ⇒ 'a word list"
(‹(‹notation=‹mixfix upto_enum_step››[_ , _ .e. _])›)
where
"[a , b .e. c] ≡ if c < a then [] else map (λx. a + x * (b - a)) [0 .e. (c - a) div (b - a)]"

lemma maxBound_word:
"(maxBound::'a::len word) = -1"
⟨*proof*⟩

lemma minBound_word:
"(minBound::'a::len word) = 0"
⟨*proof*⟩

lemma maxBound_max_word:
"(maxBound::'a::len word) = - 1"
⟨*proof*⟩

lemma leq_maxBound [simp]:
"(x::'a::len word) ≤ maxBound"
⟨*proof*⟩

lemma upto_enum_red':
assumes lt: "1 ≤ X"
shows "[(0::'a :: len word) .e. X - 1] = map of_nat [0 ..< unat X]"
⟨*proof*⟩

lemma upto_enum_red2:
assumes szv: "sz < LENGTH('a :: len)"
shows "[(0:: 'a :: len word) .e. 2 ^ sz - 1] =

```
  map of_nat [0 ..< 2 ^ sz]" ⟨proof⟩

lemma upto_enum_step_red:
  assumes szv: "sz < LENGTH('a)"
  and   usszv: "us ≤ sz"
  shows "[0 :: 'a :: len word , 2 ^ us .e. 2 ^ sz - 1]
       = map (λx. of_nat x * 2 ^ us) [0 ..< 2 ^ (sz - us)]"
⟨proof⟩

lemma upto_enum_word: "[x .e. y] = map of_nat [unat x ..< Suc (unat y)]"
  ⟨proof⟩

lemma word_upto_Cons_eq:
  "x < y ⟹ [x::'a::len word .e. y] = x # [x + 1 .e. y]"
  ⟨proof⟩

lemma distinct_enum_upto:
  "distinct [(0 :: 'a::len word) .e. b]"
⟨proof⟩

lemma upto_enum_set_conv [simp]:
  fixes a :: "'a :: len word"
  shows "set [a .e. b] = {x. a ≤ x ∧ x ≤ b}"
⟨proof⟩

lemma upto_enum_less:
  assumes "x ∈ set [(a::'a::len word).e.2 ^ n - 1]" and "n < LENGTH('a::len)"
  shows   "x < 2 ^ n"
  ⟨proof⟩

lemma upto_enum_len_less:
  "⟦ n ≤ length [a, b .e. c]; n ≠ 0 ⟧ ⟹ a ≤ c"
  ⟨proof⟩

lemma length_upto_enum_step:
  fixes x :: "'a :: len word"
  shows "x ≤ z ⟹ length [x , y .e. z] = (unat ((z - x) div (y - x)))
+ 1"
  ⟨proof⟩

lemma map_length_unfold_one:
  fixes x :: "'a::len word"
  assumes xv: "Suc (unat x) < 2 ^ LENGTH('a)"
  and     ax: "a < x"
  shows   "map f [a .e. x] = f a # map f [a + 1 .e. x]"
  ⟨proof⟩

lemma upto_enum_set_conv2:
  fixes a :: "'a::len word"
```

shows "set [a .e. b] = {a .. b}"
⟨*proof*⟩

lemma length_upto_enum [simp]:
fixes a :: "'a :: len word"
shows "length [a .e. b] = Suc (unat b) - unat a"
⟨*proof*⟩

lemma length_upto_enum_cases:
fixes a :: "'a::len word"
shows "length [a .e. b] = (if a ≤ b then Suc (unat b) - unat a else 0)"
⟨*proof*⟩

lemma length_upto_enum_less_one:
"⟦a ≤ b; b ≠ 0⟧ ⟹ length [a .e. b - 1] = unat (b - a)"
⟨*proof*⟩

lemma drop_upto_enum: "drop (unat n) [0 .e. m] = [n .e. m]"
⟨*proof*⟩

lemma distinct_enum_upto' [simp]:
"distinct [a::'a::len word .e. b]"
⟨*proof*⟩

lemma length_interval:
"⟦set xs = {x. (a::'a::len word) ≤ x ∧ x ≤ b}; distinct xs⟧
⟹ length xs = Suc (unat b) - unat a"
⟨*proof*⟩

lemma enum_word_div:
fixes v :: "'a :: len word"
shows "∃xs ys. enum = xs @ [v] @ ys ∧ (∀x ∈ set xs. x < v) ∧ (∀y ∈ set ys. v < y)"
⟨*proof*⟩

lemma remdups_enum_upto:
fixes s::"'a::len word"
shows "remdups [s .e. e] = [s .e. e]"
⟨*proof*⟩

lemma card_enum_upto:
fixes s::"'a::len word"
shows "card (set [s .e. e]) = Suc (unat e) - unat s"
⟨*proof*⟩

lemma length_upto_enum_one:
fixes x :: "'a :: len word"
assumes lt1: "x < y" **and** lt2: "z < y" **and** lt3: "x ≤ z"

shows "[x , y .e. z] = [x]"
⟨*proof*⟩

end

12 Print Words in Hex

theory Hex_Words
imports
"HOL-Library.Word"
begin

Print words in hex.

⟨*ML*⟩

end

13 Normalising Word Numerals

theory Norm_Words
imports Signed_Words
begin

Normalise word numerals, including negative ones apart from - 1, to the interval [0..2^len_of 'a). Only for concrete word lengths.

lemma bintrunc_numeral:
"(take_bit :: nat ⇒ int ⇒ int) (numeral k) x = of_bool (odd x) + 2 * (take_bit :: nat ⇒ int ⇒ int) (pred_numeral k) (x div 2)"
⟨*proof*⟩

lemma neg_num_bintr:
"(- numeral x :: 'a::len word) = word_of_int (take_bit LENGTH('a) (- numeral x))"
⟨*proof*⟩

⟨*ML*⟩

lemma minus_one_norm:
"(-1 :: 'a :: len word) = of_nat (2 ^ LENGTH('a) - 1)"
⟨*proof*⟩

lemmas minus_one_norm_num =
minus_one_norm [**where** 'a="'b::len bit0"] minus_one_norm [**where** 'a="'b::len0 bit1"]

context
begin

private lemma "f (7 :: 2 word) = f 3" ⟨*proof*⟩ **lemma** "f 7 = f (3 :: 2 word)" ⟨*proof*⟩ **lemma** "f (-2) = f (21 + 1 :: 3 word)" ⟨*proof*⟩ **lemma** "f (-2) = f (13 + 1 :: 'a::len word)"
⟨*proof*⟩ **lemma** "f (-2) = f (0xFFFFFFFE :: 32 word)" ⟨*proof*⟩ **lemma** "(-1 :: 2 word) = 3" ⟨*proof*⟩ **lemma** "f (-2) = f (0xFFFFFFFE :: 32 signed word)" ⟨*proof*⟩

We leave - 1 untouched by default, because it is often useful and its normal form can be large. To include it in the normalisation, add minus_one_norm_num. The additional normalisation is restricted to concrete numeral word lengths, like the rest.

context
notes minus_one_norm_num [simp]
begin

private lemma "f (-1) = f (15 :: 4 word)" ⟨*proof*⟩ **lemma** "f (-1) = f (7 :: 3 word)" ⟨*proof*⟩ **lemma** "f (-1) = f (0xFFFF :: 16 word)" ⟨*proof*⟩ **lemma** "f (-1) = f (0xFFFF + 1 :: 'a::len word)"
⟨*proof*⟩

end

end

end

14 Syntax bundles for traditional infix syntax

theory Syntax_Bundles
imports "HOL-Library.Word"
begin

bundle bit_projection_infix_syntax
begin

notation bit (**infixl** <!!> 100)

end

end

theory Sgn_Abs
imports Most_significant_bit
begin

15 sgn and abs for 'a word

15.1 Instances

sgn on words returns -1, 0, or 1.

instantiation word :: (len) sgn
begin

definition sgn_word :: ‹'a word ⇒ 'a word› **where**
  ‹sgn w = (if w = 0 then 0 else if 0 <s w then 1 else -1)›

instance ⟨*proof*⟩

end

lemma word_sgn_0[simp]:
  "sgn 0 = (0::'a::len word)"
  ⟨*proof*⟩

lemma word_sgn_1[simp]:
  "sgn 1 = (1::'a::len word)"
  ⟨*proof*⟩

lemma word_sgn_max_word[simp]:
  "sgn (- 1) = (-1::'a::len word)"
  ⟨*proof*⟩

lemmas word_sgn_numeral[simp] = sgn_word_def[**where** w="numeral w" **for** w]

abs on words is the usual definition.

instantiation word :: (len) abs
begin

definition abs_word :: ‹'a word ⇒ 'a word› **where**
  ‹abs w = (if w ≤s 0 then -w else w)›

instance ⟨*proof*⟩

end

lemma word_abs_0[simp]:
  "|0| = (0::'a::len word)"

⟨*proof*⟩

lemma word_abs_1[simp]:
"|1| = (1::'a::len word)"
⟨*proof*⟩

lemma word_abs_max_word[simp]:
"|- 1| = (1::'a::len word)"
⟨*proof*⟩

lemma word_abs_msb:
"|w| = (if msb w then -w else w)" **for** w::"'a::len word"
⟨*proof*⟩

lemmas word_abs_numeral[simp] = word_abs_msb[**where** w="numeral w" **for** w]

15.2 Properties

lemma word_sgn_0_0:
"sgn a = 0 ⟷ a = 0" **for** a::"'a::len word"
⟨*proof*⟩

lemma word_sgn_1_pos:
"1 < LENGTH('a) ⟹ sgn a = 1 ⟷ 0 <s a" **for** a::"'a::len word"
⟨*proof*⟩

lemma word_sgn_1_neg:
"sgn a = - 1 ⟷ a <s 0"
⟨*proof*⟩

lemma word_sgn_pos[simp]:
"0 <s a ⟹ sgn a = 1"
⟨*proof*⟩

lemma word_sgn_neg[simp]:
"a <s 0 ⟹ sgn a = - 1"
⟨*proof*⟩

lemma word_abs_sgn:
"|k| = k * sgn k" **for** k :: "'a::len word"
⟨*proof*⟩

lemma word_sgn_greater[simp]:
"0 <s sgn a ⟷ 0 <s a" **for** a::"'a::len word"
⟨*proof*⟩

lemma word_sgn_less[simp]:
"sgn a <s 0 ⟷ a <s 0" **for** a::"'a::len word"

⟨*proof*⟩

lemma word_abs_sgn_eq_1[simp]:
"a ≠ 0 ⟹ |sgn a| = 1" **for** a::"'a::len word"
⟨*proof*⟩

lemma word_abs_sgn_eq:
"|sgn a| = (if a = 0 then 0 else 1)" **for** a::"'a::len word"
⟨*proof*⟩

lemma word_sgn_mult_self_eq[simp]:
"sgn a * sgn a = of_bool (a ≠ 0)" **for** a::"'a::len word"
⟨*proof*⟩

end

16 Displaying Phantom Types for Word Operations

theory Type_Syntax
imports "HOL-Library.Word"
begin

⟨*ML*⟩

syntax
"_Ucast" :: "type ⇒ type ⇒ logic"
(<(<indent=1 notation=<mixfix UCAST>>UCAST/(<indent=1 notation=<infix cast>>'(_ → _')))>)
syntax_consts
"_Ucast" == ucast
translations
"UCAST('s → 't)" => "CONST ucast :: ('s word ⇒ 't word)"
⟨*ML*⟩

syntax
"_Scast" :: "type ⇒ type ⇒ logic"
(<(<indent=1 notation=<mixfix SCAST>>SCAST/(<indent=1 notation=<infix cast>>'(_ → _')))>)
syntax_consts
"_Scast" == scast
translations
"SCAST('s → 't)" => "CONST scast :: ('s word ⇒ 't word)"
⟨*ML*⟩

syntax
  "_Revcast" :: "type ⇒ type ⇒ logic"
    (<(<indent=1 notation=<mixfix REVCAST>>REVCAST/(<indent=1 notation=<infix cast>>'(_ → _')))>)
syntax_consts
  "_Revcast" == revcast
translations
  "REVCAST('s → 't)" => "CONST revcast :: ('s word ⇒ 't word)"
⟨*ML*⟩

end

17 Solving Word Equalities

theory Word_EqI
  imports
    More_Word
    Aligned
    "HOL-Eisbach.Eisbach_Tools"
begin

Some word equalities can be solved by considering the problem bitwise for all `n < LENGTH('a)`. This is similar to the existing method `word_bitwise` and expanding into an explicit list of bits. The `word_bitwise` only works on concrete word lengths, but can treat a wider number of operators (in particular a mix of arithmetic, order, and bit operations). The `word_eqI` method below works on words of abstract size (`'a word`) and produces smaller, more abstract goals, but does not deal with arithmetic operations.

lemmas le_mask_high_bits_len = le_mask_high_bits[unfolded word_size]
lemmas neg_mask_le_high_bits_len = neg_mask_le_high_bits[unfolded word_size]

named_theorems word_eqI_simps

lemmas [word_eqI_simps] =
  word_or_zero
  neg_mask_test_bit
  nth_ucast
  less_2p_is_upper_bits_unset
  le_mask_high_bits_len
  neg_mask_le_high_bits_len
  bang_eq
  is_up
  is_down
  is_aligned_nth
  word_size

```
lemmas word_eqI_folds [symmetric] =
  push_bit_eq_mult
  drop_bit_eq_div
  take_bit_eq_mod


lemmas word_eqI_rules = word_eqI [rule_format, unfolded word_size] bit_eqI

lemma test_bit_lenD:
  "bit x n ⟹ n < LENGTH('a) ∧ bit x n" for x :: "'a :: len word"
  ⟨proof⟩
method word_eqI uses simp simp_del split split_del cong flip =
  (
   rule word_eqI_rules,

   (simp only: word_eqI_folds)?,

   (clarsimp simp: simp simp del: simp_del simp flip: flip split: split
split del: split_del cong: cong)?,

   ((drule less_mask_eq)+)?,

   (simp only: bit_simps word_eqI_simps)?,

   (clarsimp simp: simp not_less not_le simp del: simp_del simp flip:
flip
             split: split split del: split_del cong: cong)?,

   ((drule test_bit_lenD)+)?,

   (simp only: bit_simps word_eqI_simps)?,
   (clarsimp simp: simp simp del: simp_del simp flip: flip
             split: split split del: split_del cong: cong)?,

   (simp add: simp test_bit_conj_lt del: simp_del flip: flip split: split
split del: split_del cong: cong)?)
```

— Method to reduce goals of the form `P ⟹ x = y` for words of abstract length to reasoning on bits of the words. Fails if goal unsolved, but tries harder than `word_eqI`.

```
method word_eqI_solve uses simp simp_del split split_del cong flip dest
=
  solves ‹word_eqI simp: simp simp_del: simp_del split: split split_del:
split_del
                    cong: cong simp flip: flip;
          (fastforce dest: dest simp: simp flip: flip
                     simp: simp simp del: simp_del split: split split
del: split_del cong: cong)?›
```

end

theory Boolean_Inequalities
  imports Word_EqI
begin

18 All inequalities between binary Boolean operations on 'a word

Enumerates all binary functions resulting from Boolean operations on 'a word and derives all inequalities of the form f x y ≤ g x y between them. We leave out the trivial 0 ≤ g x y, f x y ≤ - 1, and f x y ≤ f x y, because these are already readily available to the simplifier and other methods.

This leaves 36 inequalities. Some of these are subsumed by each other, but we generate the full list to avoid too much manual processing.

All inequalities produced here are in simp normal form.

context
  includes bit_operations_syntax
begin

definition all_bool_word_funs :: "('a::len word ⇒ 'a word ⇒ 'a word) list" **where**
  "all_bool_word_funs ≡ [
     λx y. 0,
     λx y. x AND y,
     λx y. x AND NOT y,
     λx y. x,
     λx y. NOT x AND y,
     λx y. y,
     λx y. x XOR y,
     λx y. x OR y,
     λx y. NOT (x OR y),
     λx y. NOT (x XOR y),
     λx y. NOT y,
     λx y. x OR NOT y,
     λx y. NOT x,
     λx y. NOT x OR y,
     λx y. NOT (x AND y),
     λx y. -1
   ]"

The inequalities on 'a word follow directly from implications on propositional Boolean logic, which simp can solve automatically. This means, we can simply enumerate all combinations, reduce from 'a word to bool, and

attempt to solve by `simp` to get the complete list.

⟨*ML*⟩

lemma
  "x AND y ≤ x" **for** x :: "'a::len word"
  ⟨*proof*⟩

lemma
  "NOT x ≤ NOT x OR NOT y" **for** x :: "'a::len word"
  ⟨*proof*⟩

lemma
  "x XOR y ≤ NOT x OR NOT y" **for** x :: "'a::len word"
  ⟨*proof*⟩

lemma word_xor_le_nand:
  "x XOR y ≤ NOT (x AND y)" **for** x :: "'a::len word"
  ⟨*proof*⟩

end

end

19 Lemmas with Generic Word Length

theory Word_Lemmas
  imports
    Type_Syntax
    Signed_Division_Word
    Signed_Words
    More_Word
    Most_significant_bit
    Enumeration_Word
    Aligned
    Bit_Shifts_Infix_Syntax
    Boolean_Inequalities
    Word_EqI

begin

context
  includes bit_operations_syntax
begin

lemma word_max_le_or:
  "max x y ≤ x OR y" **for** x :: "'a::len word"

⟨*proof*⟩

lemma word_and_le_min:
"x AND y ≤ min x y" **for** x :: "'a::len word"
⟨*proof*⟩

lemma word_not_le_eq:
"(NOT x ≤ y) = (NOT y ≤ x)" **for** x :: "'a::len word"
⟨*proof*⟩

lemma word_not_le_not_eq[simp]:
"(NOT y ≤ NOT x) = (x ≤ y)" **for** x :: "'a::len word"
⟨*proof*⟩

lemma not_min_eq:
"NOT (min x y) = max (NOT x) (NOT y)" **for** x :: "'a::len word"
⟨*proof*⟩

lemma not_max_eq:
"NOT (max x y) = min (NOT x) (NOT y)" **for** x :: "'a::len word"
⟨*proof*⟩

lemma ucast_le_ucast_eq:
fixes x y :: "'a::len word"
assumes x: "x < 2 ^ n"
assumes y: "y < 2 ^ n"
assumes n: "n = LENGTH('b::len)"
shows "(UCAST('a → 'b) x ≤ UCAST('a → 'b) y) ⟷ (x ≤ y)" (**is** "?L=_")
⟨*proof*⟩

lemma ucast_zero_is_aligned:
‹is_aligned w n› **if** ‹UCAST('a::len → 'b::len) w = 0› ‹n ≤ LENGTH('b)›
⟨*proof*⟩

lemma unat_ucast_eq_unat_and_mask:
"unat (UCAST('b::len → 'a::len) w) = unat (w AND mask LENGTH('a))"
⟨*proof*⟩

lemma le_max_word_ucast_id:
‹UCAST('b → 'a) (UCAST('a → 'b) x) = x›
if ‹x ≤ UCAST('b::len → 'a) (- 1)›
for x :: ‹'a::len word›
⟨*proof*⟩

lemma uint_shiftr_eq:
‹uint (w >> n) = uint w div 2 ^ n›
⟨*proof*⟩

lemma bit_shiftl_word_iff [bit_simps]:

```
  ‹bit (w << m) n ⟷ m ≤ n ∧ n < LENGTH('a) ∧ bit w (n - m)›
  for w :: ‹'a::len word›
```
⟨*proof*⟩

lemma bit_shiftr_word_iff:
```
  ‹bit (w >> m) n ⟷ bit w (m + n)›
  for w :: ‹'a::len word›
```
⟨*proof*⟩

lemma uint_sshiftr_eq:
```
  ‹uint (w >>> n) = take_bit LENGTH('a) (sint w div 2 ^  n)›
  for w :: ‹'a::len word›
```
⟨*proof*⟩

lemma sshiftr_n1: "-1 >>> n = -1"
⟨*proof*⟩

lemma nth_sshiftr:
```
  "bit (w >>> m) n = (n < size w ∧ (if n + m ≥ size w then bit w (size
w - 1) else bit w (n + m)))"
```
⟨*proof*⟩

lemma sshiftr_numeral:
```
  ‹(numeral k >>> numeral n :: 'a::len word) =
    word_of_int (signed_take_bit (LENGTH('a) - 1) (numeral k) >> numeral
n)›
```
⟨*proof*⟩

lemma sshiftr_div_2n: "sint (w >>> n) = sint w div 2 ^ n"
⟨*proof*⟩

lemma mask_eq:
```
  ‹mask n = (1 << n) - (1 :: 'a::len word)›
```
⟨*proof*⟩

lemma nth_shiftl': "bit (w << m) n ⟷ n < size w ∧ n >= m ∧ bit w (n
- m)"
```
  for w :: "'a::len word"
```
⟨*proof*⟩

lemmas nth_shiftl = nth_shiftl' [unfolded word_size]

lemma nth_shiftr: "bit (w >> m) n = bit w (n + m)"
```
  for w :: "'a::len word"
```
⟨*proof*⟩

lemma shiftr_div_2n: "uint (shiftr w n) = uint w div 2 ^ n"
⟨*proof*⟩

lemma shiftl_rev: "shiftl w n = word_reverse (shiftr (word_reverse w) n)"
⟨*proof*⟩

lemma rev_shiftl: "word_reverse w << n = word_reverse (w >> n)"
⟨*proof*⟩

lemma shiftr_rev: "w >> n = word_reverse (word_reverse w << n)"
⟨*proof*⟩

lemma rev_shiftr: "word_reverse w >> n = word_reverse (w << n)"
⟨*proof*⟩

lemmas ucast_up =
rc1 [simplified rev_shiftr [symmetric] revcast_ucast [symmetric]]
lemmas ucast_down =
rc2 [simplified rev_shiftr revcast_ucast [symmetric]]

lemma shiftl_zero_size: "size x ≤ n ⟹ x << n = 0"
for x :: "'a::len word"
⟨*proof*⟩

lemma shiftl_t2n: "shiftl w n = 2 ^ n * w"
for w :: "'a::len word"
⟨*proof*⟩

lemma word_shift_by_2:
"x * 4 = (x::'a::len word) << 2"
⟨*proof*⟩

lemma word_shift_by_3:
"x * 8 = (x::'a::len word) << 3"
⟨*proof*⟩

lemma slice_shiftr: "slice n w = ucast (w >> n)"
⟨*proof*⟩

lemma shiftr_zero_size: "size x ≤ n ⟹ x >> n = 0"
for x :: "'a :: len word"
⟨*proof*⟩

lemma shiftr_x_0 [simp]: "x >> 0 = x"
for x :: "'a::len word"
⟨*proof*⟩

lemma shiftl_x_0 [simp]: "x << 0 = x"
for x :: "'a::len word"
⟨*proof*⟩

lemmas shiftl0 = shiftl_x_0

lemma shiftr_1 [simp]: "(1::'a::len word) >> n = (if n = 0 then 1 else 0)"
⟨*proof*⟩

lemma and_not_mask:
"w AND NOT (mask n) = (w >> n) << n"
for w :: ‹'a::len word›
⟨*proof*⟩

lemma and_mask:
"w AND mask n = (w << (size w - n)) >> (size w - n)"
for w :: ‹'a::len word›
⟨*proof*⟩

lemma shiftr_div_2n_w: "w >> n = w div (2^n :: 'a :: len word)"
⟨*proof*⟩

lemma le_shiftr:
"u ≤ v ⟹ u >> (n :: nat) ≤ (v :: 'a :: len word) >> n"
⟨*proof*⟩

lemma le_shiftr':
"⟦ u >> n ≤ v >> n ; u >> n ≠ v >> n ⟧ ⟹ (u::'a::len word) ≤ v"
⟨*proof*⟩

lemma shiftr_mask_le:
"n ≤ m ⟹ mask n >> m = (0 :: 'a::len word)"
⟨*proof*⟩

lemma shiftr_mask [simp]:
‹mask m >> m = (0::'a::len word)›
⟨*proof*⟩

lemma le_mask_iff:
"(w ≤ mask n) = (w >> n = 0)"
for w :: ‹'a::len word›
⟨*proof*⟩

lemma and_mask_eq_iff_shiftr_0:
"(w AND mask n = w) = (w >> n = 0)"
for w :: ‹'a::len word›
⟨*proof*⟩

lemma mask_shiftl_decompose:
"mask m << n = mask (m + n) AND NOT (mask n :: 'a::len word)"
⟨*proof*⟩

lemma shiftl_over_and_dist:
fixes a::"'a::len word"
shows "(a AND b) << c = (a << c) AND (b << c)"
⟨*proof*⟩

lemma shiftr_over_and_dist:
fixes a::"'a::len word"
shows "a AND b >> c = (a >> c) AND (b >> c)"
⟨*proof*⟩

lemma sshiftr_over_and_dist:
fixes a::"'a::len word"
shows "a AND b >>> c = (a >>> c) AND (b >>> c)"
⟨*proof*⟩

lemma shiftl_over_or_dist:
fixes a::"'a::len word"
shows "a OR b << c = (a << c) OR (b << c)"
⟨*proof*⟩

lemma shiftr_over_or_dist:
fixes a::"'a::len word"
shows "a OR b >> c = (a >> c) OR (b >> c)"
⟨*proof*⟩

lemma sshiftr_over_or_dist:
fixes a::"'a::len word"
shows "a OR b >>> c = (a >>> c) OR (b >>> c)"
⟨*proof*⟩

lemmas shift_over_ao_dists =
shiftl_over_or_dist shiftr_over_or_dist
sshiftr_over_or_dist shiftl_over_and_dist
shiftr_over_and_dist sshiftr_over_and_dist

lemma shiftl_shiftl:
fixes a::"'a::len word"
shows "a << b << c = a << (b + c)"
⟨*proof*⟩

lemma shiftr_shiftr:
fixes a::"'a::len word"
shows "a >> b >> c = a >> (b + c)"
⟨*proof*⟩

lemma shiftl_shiftr1:
fixes a::"'a::len word"
shows "c ≤ b ⟹ a << b >> c = a AND (mask (size a - b)) << (b - c)"
⟨*proof*⟩

lemma shiftl_shiftr2:
fixes a::"'a::len word"
shows "b < c ⟹ a << b >> c = (a >> (c - b)) AND (mask (size a - c))"
⟨*proof*⟩

lemma shiftr_shiftl1:
fixes a::"'a::len word"
shows "c ≤ b ⟹ a >> b << c = (a >> (b - c)) AND (NOT (mask c))"
⟨*proof*⟩

lemma shiftr_shiftl2:
fixes a::"'a::len word"
shows "b < c ⟹ a >> b << c = (a << (c - b)) AND (NOT (mask c))"
⟨*proof*⟩

lemmas multi_shift_simps =
shiftl_shiftl shiftr_shiftr
shiftl_shiftr1 shiftl_shiftr2
shiftr_shiftl1 shiftr_shiftl2

lemma shiftr_mask2:
"n ≤ LENGTH('a) ⟹ (mask n >> m :: ('a :: len) word) = mask (n - m)"
⟨*proof*⟩

lemma word_shiftl_add_distrib:
fixes x :: "'a :: len word"
shows "(x + y) << n = (x << n) + (y << n)"
⟨*proof*⟩

lemma mask_shift:
"(x AND NOT (mask y)) >> y = x >> y"
for x :: ‹'a::len word›
⟨*proof*⟩

lemma shiftr_div_2n':
"unat (w >> n) = unat w div 2 ^ n"
⟨*proof*⟩

lemma shiftl_shiftr_id:
"⟦ n < LENGTH('a); x < 2 ^ (LENGTH('a) - n) ⟧ ⟹ x << n >> n = (x::'a::len word)"
⟨*proof*⟩

lemma ucast_shiftl_eq_0:
fixes w :: "'a :: len word"
shows "⟦ n ≥ LENGTH('b) ⟧ ⟹ ucast (w << n) = (0 :: 'b :: len word)"
⟨*proof*⟩

lemma word_shift_nonzero:
  fixes x:: "'a::len word"
  assumes "x ≤ 2 ^ m"
      and mn: "m + n < LENGTH('a::len)"
      and "x ≠ 0"
    shows "x << n ≠ 0"
⟨*proof*⟩

lemma word_shiftr_lt:
  fixes w :: "'a::len word"
  shows "unat (w >> n) < (2 ^ (LENGTH('a) - n))"
  ⟨*proof*⟩

lemma shiftr_less_t2n':
  "⟦ x AND mask (n + m) = x; m < LENGTH('a) ⟧ ⟹ x >> n < 2 ^ m" **for** x
:: "'a :: len word"
  ⟨*proof*⟩

lemma shiftr_less_t2n:
  "x < 2 ^ (n + m) ⟹ x >> n < 2 ^ m" **for** x :: "'a :: len word"
  ⟨*proof*⟩

lemma shiftr_eq_0: "n ≥ LENGTH('a) ⟹ ((w::'a::len word) >> n) = 0"
  ⟨*proof*⟩

lemma shiftl_less_t2n:
  fixes x :: "'a :: len word"
  shows "⟦ x < (2 ^ (m - n)); m < LENGTH('a) ⟧ ⟹ (x << n) < 2 ^ m"
  ⟨*proof*⟩

lemma shiftl_less_t2n':
  "(x::'a::len word) < 2 ^ m ⟹ m+n < LENGTH('a) ⟹ x << n < 2 ^ (m
+ n)"
  ⟨*proof*⟩

lemma scast_bit_test [simp]:
  "scast ((1 :: 'a::len signed word) << n) = (1 :: 'a word) << n"
  ⟨*proof*⟩

lemma signed_shift_guard_to_word:
  ‹unat x * 2 ^ y < 2 ^ n ⟷ x = 0 ∨ x < 1 << n >> y›
    if ‹n < LENGTH('a)› ‹0 < n›
    for x :: ‹'a::len word›
⟨*proof*⟩

lemma shiftr_not_mask_0:
  "n+m ≥ LENGTH('a :: len) ⟹ ((w::'a::len word) >> n) AND NOT (mask
m) = 0"
  ⟨*proof*⟩

lemma shiftl_mask_is_0[simp]:
"(x << n) AND mask n = 0"
for x :: ‹'a::len word›
⟨*proof*⟩

lemma rshift_sub_mask_eq:
"(a >> (size a - b)) AND mask b = a >> (size a - b)"
for a :: ‹'a::len word›
⟨*proof*⟩

lemma shiftl_shiftr3:
"b ≤ c ⟹ a << b >> c = (a >> c - b) AND mask (size a - c)"
for a :: ‹'a::len word›
⟨*proof*⟩

lemma and_mask_shiftr_comm:
"m ≤ size w ⟹ (w AND mask m) >> n = (w >> n) AND mask (m-n)"
for w :: ‹'a::len word›
⟨*proof*⟩

lemma and_mask_shiftl_comm:
"m+n ≤ size w ⟹ (w AND mask m) << n = (w << n) AND mask (m+n)"
for w :: ‹'a::len word›
⟨*proof*⟩

lemma le_mask_shiftl_le_mask: "s = m + n ⟹ x ≤ mask n ⟹ x << m ≤
mask s"
for x :: ‹'a::len word›
⟨*proof*⟩

lemma word_and_1_shiftl:
"x AND (1 << n) = (if bit x n then (1 << n) else 0)" **for** x :: "'a ::
len word"
⟨*proof*⟩

lemmas word_and_1_shiftls'
= word_and_1_shiftl[**where** n=0]
word_and_1_shiftl[**where** n=1]
word_and_1_shiftl[**where** n=2]

lemmas word_and_1_shiftls = word_and_1_shiftls' [simplified]

lemma word_and_mask_shiftl:
"x AND (mask n << m) = ((x >> m) AND mask n) << m"
for x :: ‹'a::len word›
⟨*proof*⟩

lemma shift_times_fold:

```
  "(x :: 'a :: len word) * (2 ^ n) << m = x << (m + n)"
```
⟨*proof*⟩

lemma of_bool_nth:
```
  "of_bool (bit x v) = (x >> v) AND 1"
  for x :: ‹'a::len word›
```
⟨*proof*⟩

lemma shiftr_mask_eq:
```
  "(x >> n) AND mask (size x - n) = x >> n" for x :: "'a :: len word"
```
⟨*proof*⟩

lemma shiftr_mask_eq':
```
  "m = (size x - n) ⟹ (x >> n) AND mask m = x >> n" for x :: "'a ::
len word"
```
⟨*proof*⟩

lemma and_eq_0_is_nth:
```
  fixes x :: "'a :: len word"
  shows "y = 1 << n ⟹ ((x AND y) = 0) = (¬ (bit x n))"
```
⟨*proof*⟩

lemma word_shift_zero:
```
  "⟦ x << n = 0; x ≤ 2^m; m + n < LENGTH('a)⟧ ⟹ (x::'a::len word) =
0"
```
⟨*proof*⟩

lemma mask_shift_and_negate[simp]:"(w AND mask n << m) AND NOT (mask
n << m) = 0"
```
  for w :: ‹'a::len word›
```
⟨*proof*⟩

lemma bitfield_op_twice:
```
  "(x AND NOT (mask n << m) OR ((y AND mask n) << m)) AND NOT (mask n
<< m) = x AND NOT (mask n << m)"
  for x :: ‹'a::len word›
```
⟨*proof*⟩

lemma bitfield_op_twice'':
```
  "⟦NOT a = b << c; ∃x. b = mask x⟧ ⟹ (x AND a OR (y AND b << c)) AND
a = x AND a"
  for a b :: ‹'a::len word›
```
⟨*proof*⟩

lemma shiftr1_unfold: "x div 2 = x >> 1"
⟨*proof*⟩

lemma shiftr1_is_div_2: "(x::('a::len) word) >> 1 = x div 2"

⟨*proof*⟩

lemma shiftl1_is_mult: "(x << 1) = (x :: 'a::len word) * 2"
⟨*proof*⟩

lemma shiftr1_lt:"x ≠ 0 ⟹ (x::('a::len) word) >> 1 < x"
⟨*proof*⟩

lemma shiftr1_0_or_1:"(x::('a::len) word) >> 1 = 0 ⟹ x = 0 ∨ x = 1"
⟨*proof*⟩

lemma shiftr1_irrelevant_lsb: "bit (x::('a::len) word) 0 ∨ x >> 1 =
(x + 1) >> 1"
⟨*proof*⟩

lemma shiftr1_0_imp_only_lsb:"((x::('a::len) word) + 1) >> 1 = 0 ⟹
x = 0 ∨ x + 1 = 0"
⟨*proof*⟩

lemma shiftr1_irrelevant_lsb': "¬ (bit (x::('a::len) word) 0) ⟹ x
>> 1 = (x + 1) >> 1"
⟨*proof*⟩

lemma cast_chunk_assemble_id:
assumes n: "n = LENGTH('a::len)"
and "m = LENGTH('b::len)"
and "n * 2 = m"
shows "UCAST('a → 'b) (UCAST('b → 'a) x::'a word) OR (UCAST('a →
'b) (UCAST('b → 'a) (x >> n)::'a word) << n) = x"
⟨*proof*⟩

lemma cast_chunk_scast_assemble_id:
fixes x:: "'b::len word"
assumes "n = LENGTH('a::len)"
and "m = LENGTH('b)"
and "n * 2 = m"
shows "UCAST('a → 'b) (SCAST('b → 'a) x) OR (UCAST('a → 'b) (SCAST('b
→ 'a) (x >> n)) << n) = x"
⟨*proof*⟩

lemma unat_shiftr_less_t2n:
fixes x :: "'a :: len word"
shows "unat x < 2 ^ (n + m) ⟹ unat (x >> n) < 2 ^ m"
⟨*proof*⟩

lemma ucast_less_shiftl_helper:
"⟦ LENGTH('b) + 2 < LENGTH('a); 2 ^ (LENGTH('b) + 2) ≤ n⟧
⟹ (ucast (x :: 'b::len word) << 2) < (n :: 'a::len word)"

⟨*proof*⟩

lemma NOT_mask_shifted_lenword:
"NOT (mask len << (LENGTH('a) - len) ::'a::len word) = mask (LENGTH('a) - len)"
⟨*proof*⟩

lemma shiftr_less:
"(w::'a::len word) < k ⟹ w >> n < k"
⟨*proof*⟩

lemma word_and_notzeroD:
"w AND w' ≠ 0 ⟹ w ≠ 0 ∧ w' ≠ 0"
⟨*proof*⟩

lemma shiftr_le_0:
"unat (w::'a::len word) < 2 ^ n ⟹ w >> n = (0::'a::len word)"
⟨*proof*⟩

lemma of_nat_shiftl:
"(of_nat x << n) = (of_nat (x * 2 ^ n) :: ('a::len) word)"
⟨*proof*⟩

lemma shiftl_1_not_0:
"n < LENGTH('a) ⟹ (1::'a::len word) << n ≠ 0"
⟨*proof*⟩

lemma bitmagic_zeroLast_leq_or1Last:
"(a::('a::len) word) AND (mask len << x - len) ≤ a OR mask (y - len)"
⟨*proof*⟩

lemma zero_base_lsb_imp_set_eq_as_bit_operation:
fixes base ::"'a::len word"
assumes valid_prefix: "mask (LENGTH('a) - len) AND base = 0"
shows "(base = NOT (mask (LENGTH('a) - len)) AND a) ⟷
(a ∈ {base .. base OR mask (LENGTH('a) - len)})"
⟨*proof*⟩

lemma of_nat_eq_signed_scast:
"(of_nat x = (y :: ('a::len) signed word))
= (of_nat x = (scast y :: 'a word))"
⟨*proof*⟩

lemma word_aligned_add_no_wrap_bounded:
"⟦ w + 2^n ≤ x; w + 2^n ≠ 0; is_aligned w n ⟧ ⟹ (w::'a::len word)
< x"
⟨*proof*⟩

lemma mask_Suc:
"mask (Suc n) = (2 :: 'a::len word) ^ n + mask n"
⟨*proof*⟩

lemma mask_mono:
"sz' ≤ sz ⟹ mask sz' ≤ (mask sz :: 'a::len word)"
⟨*proof*⟩

lemma aligned_mask_disjoint:
"⟦ is_aligned (a :: 'a :: len word) n; b ≤ mask n ⟧ ⟹ a AND b = 0"
⟨*proof*⟩

lemma word_and_or_mask_aligned:
"⟦ is_aligned a n; b ≤ mask n ⟧ ⟹ a + b = a OR b"
⟨*proof*⟩

lemma word_and_or_mask_aligned2:
‹is_aligned b n ⟹ a ≤ mask n ⟹ a + b = a OR b›
⟨*proof*⟩

lemma is_aligned_ucastI:
"is_aligned w n ⟹ is_aligned (ucast w) n"
⟨*proof*⟩

lemma ucast_le_maskI:
"a ≤ mask n ⟹ UCAST('a::len → 'b::len) a ≤ mask n"
⟨*proof*⟩

lemma ucast_add_mask_aligned:
"⟦ a ≤ mask n; is_aligned b n ⟧ ⟹ UCAST ('a::len → 'b::len) (a +
b) = ucast a + ucast b"
⟨*proof*⟩

lemma ucast_shiftl:
"LENGTH('b) ≤ LENGTH ('a) ⟹ UCAST ('a::len → 'b::len) x << n = ucast
(x << n)"
⟨*proof*⟩

lemma ucast_leq_mask:
"LENGTH('a) ≤ n ⟹ ucast (x::'a::len word) ≤ mask n"
⟨*proof*⟩

lemma shiftl_inj:
‹x = y›

if ‹x << n = y << n› ‹x ≤ mask (LENGTH('a) - n)› ‹y ≤ mask (LENGTH('a) - n)›
for x y :: ‹'a::len word›
⟨*proof*⟩

lemma distinct_word_add_ucast_shift_inj:
‹p' = p ∧ off' = off›
if *: ‹p + (UCAST('a::len → 'b::len) off << n) = p' + (ucast off' << n)›
and ‹is_aligned p n'› ‹is_aligned p' n'› ‹n' = n + LENGTH('a)› ‹n' < LENGTH('b)›
⟨*proof*⟩

lemma word_upto_Nil:
"y < x ⟹ [x .e. y ::'a::len word] = []"
⟨*proof*⟩

lemma word_enum_decomp_elem:
assumes "[x .e. (y ::'a::len word)] = as @ a # bs"
shows "x ≤ a ∧ a ≤ y"
⟨*proof*⟩

lemma word_enum_prefix:
"[x .e. (y ::'a::len word)] = as @ a # bs ⟹ as = (if x < a then [x .e. a - 1] else [])"
⟨*proof*⟩

lemma word_enum_decomp_set:
"[x .e. (y ::'a::len word)] = as @ a # bs ⟹ a ∉ set as"
⟨*proof*⟩

lemma word_enum_decomp:
assumes "[x .e. (y ::'a::len word)] = as @ a # bs"
shows "x ≤ a ∧ a ≤ y ∧ a ∉ set as ∧ (∀z ∈ set as. x ≤ z ∧ z ≤ y)"
⟨*proof*⟩

lemma of_nat_unat_le_mask_ucast:
"⟦of_nat (unat t) = w; t ≤ mask LENGTH('a)⟧ ⟹ t = UCAST('a::len → 'b::len) w"
⟨*proof*⟩

lemma less_diff_gt0:
"a < b ⟹ (0 :: 'a :: len word) < b - a"
⟨*proof*⟩

lemma unat_plus_gt:
"unat ((a :: 'a :: len word) + b) ≤ unat a + unat b"
⟨*proof*⟩

lemma const_less:
"⟦ (a :: 'a :: len word) - 1 < b; a ≠ b ⟧ ⟹ a < b"
⟨*proof*⟩

lemma add_mult_aligned_neg_mask:
‹(x + y * m) AND NOT(mask n) = (x AND NOT(mask n)) + y * m›
if ‹m AND (2 ^ n - 1) = 0›
for x y m :: ‹'a::len word›
⟨*proof*⟩

lemma unat_of_nat_minus_1:
"⟦ n < 2 ^ LENGTH('a); n ≠ 0 ⟧ ⟹ unat ((of_nat n:: 'a :: len word)
- 1) = n - 1"
⟨*proof*⟩

lemma word_eq_zeroI:
"a ≤ a - 1 ⟹ a = 0" **for** a :: "'a :: len word"
⟨*proof*⟩

lemma word_add_format:
"(-1 :: 'a :: len word) + b + c = b + (c - 1)"
⟨*proof*⟩

lemma upto_enum_word_nth:
assumes "i ≤ j" **and** "k ≤ unat (j - i)"
shows "[i .e. j] ! k = i + of_nat k"
⟨*proof*⟩

lemma upto_enum_step_nth:
"⟦ a ≤ c; n ≤ unat ((c - a) div (b - a)) ⟧
⟹ [a, b .e. c] ! n = a + of_nat n * (b - a)"
⟨*proof*⟩

lemma upto_enum_inc_1_len:
fixes a :: "'a::len word"
assumes "a < - 1"
shows "[(0 :: 'a :: len word) .e. 1 + a] = [0 .e. a] @ [1 + a]"
⟨*proof*⟩

lemma neg_mask_add:
"y AND mask n = 0 ⟹ x + y AND NOT(mask n) = (x AND NOT(mask n)) +
y"
for x y :: ‹'a::len word›
⟨*proof*⟩

lemma shiftr_shiftl_shiftr[simp]:
"(x :: 'a :: len word) >> a << a >> a = x >> a"
⟨*proof*⟩

lemma add_right_shift:
fixes x y :: ‹'a::len word›
assumes "x AND mask n = 0" **and** "y AND mask n = 0" **and** "x ≤ x + y"
shows "(x + y) >> n = (x >> n) + (y >> n)"
⟨*proof*⟩

lemma sub_right_shift:
"⟦ x AND mask n = 0; y AND mask n = 0; y ≤ x ⟧
⟹ (x - y) >> n = (x >> n :: 'a :: len word) - (y >> n)"
⟨*proof*⟩

lemma and_and_mask_simple:
"y AND mask n = mask n ⟹ (x AND y) AND mask n = x AND mask n"
⟨*proof*⟩

lemma and_and_mask_simple_not:
"y AND mask n = 0 ⟹ (x AND y) AND mask n = 0"
⟨*proof*⟩

lemma word_and_le':
"b ≤ c ⟹ (a :: 'a :: len word) AND b ≤ c"
⟨*proof*⟩

lemma word_and_less':
"b < c ⟹ (a :: 'a :: len word) AND b < c"
⟨*proof*⟩

lemma shiftr_w2p:
"x < LENGTH('a) ⟹ 2 ^ x = (2 ^ (LENGTH('a) - 1) >> (LENGTH('a) - 1
- x) :: 'a :: len word)"
⟨*proof*⟩

lemma t2p_shiftr:
"⟦ b ≤ a; a < LENGTH('a) ⟧ ⟹ (2 :: 'a :: len word) ^ a >> b = 2 ^
(a - b)"
⟨*proof*⟩

lemma scast_1[simp]:
"scast (1 :: 'a :: len signed word) = (1 :: 'a word)"
⟨*proof*⟩

lemma unsigned_uminus1 [simp]:
‹(unsigned (-1::'b::len word)::'c::len word) = mask LENGTH('b)›
⟨*proof*⟩

lemma ucast_ucast_mask_eq:
"⟦ UCAST('a::len → 'b::len) x = y; x AND mask LENGTH('b) = x ⟧ ⟹ x
= ucast y"
⟨*proof*⟩

lemma ucast_up_eq:
"⟦ ucast x = (ucast y::'b::len word); LENGTH('a) ≤ LENGTH ('b) ⟧
⟹ ucast x = (ucast y::'a::len word)"
⟨*proof*⟩

lemma ucast_up_neq:
"⟦ ucast x ≠ (ucast y::'b::len word); LENGTH('b) ≤ LENGTH ('a) ⟧
⟹ ucast x ≠ (ucast y::'a::len word)"
⟨*proof*⟩

lemma mask_AND_less_0:
"⟦ x AND mask n = 0; m ≤ n ⟧ ⟹ x AND mask m = 0"
for x :: ‹'a::len word›
⟨*proof*⟩

lemma mask_len_id:
"(x :: 'a :: len word) AND mask LENGTH('a) = x"
⟨*proof*⟩

lemma scast_ucast_down_same:
"LENGTH('b) ≤ LENGTH('a) ⟹ SCAST('a → 'b) = UCAST('a::len → 'b::len)"
⟨*proof*⟩

lemma word_aligned_0_sum:
"⟦ a + b = 0; is_aligned (a :: 'a :: len word) n; b ≤ mask n; n < LENGTH('a)
⟧
⟹ a = 0 ∧ b = 0"
⟨*proof*⟩

lemma mask_eq1_nochoice:
"⟦ LENGTH('a) > 1; (x :: 'a :: len word) AND 1 = x ⟧ ⟹ x = 0 ∨ x =
1"
⟨*proof*⟩

lemma shiftr_and_eq_shiftl:
"(w >> n) AND x = y ⟹ w AND (x << n) = (y << n)" **for** y :: "'a:: len
word"
⟨*proof*⟩

lemma add_mask_lower_bits':
"⟦ len = LENGTH('a); is_aligned (x :: 'a :: len word) n;
∀n'. n' ≥ n. n' < len ⟶ ¬ bit p n' ⟧
⟹ x + p AND NOT(mask n) = x"
⟨*proof*⟩

lemma leq_mask_shift:
"(x :: 'a :: len word) ≤ mask (low_bits + high_bits) ⟹ (x >> low_bits)
≤ mask high_bits"

⟨*proof*⟩

lemma ucast_ucast_eq_mask_shift:
"(x :: 'a :: len word) ≤ mask (low_bits + LENGTH('b))
⟹ ucast((ucast (x >> low_bits)) :: 'b :: len word) = x >> low_bits"
⟨*proof*⟩

lemma const_le_unat:
"⟦ b < 2 ^ LENGTH('a); of_nat b ≤ a ⟧ ⟹ b ≤ unat (a :: 'a :: len word)"
⟨*proof*⟩

lemma upt_enum_offset_trivial:
"⟦ x < 2 ^ LENGTH('a) - 1 ; n ≤ unat x ⟧
⟹ ([(0 :: 'a :: len word) .e. x] ! n) = of_nat n"
⟨*proof*⟩

lemma word_le_mask_out_plus_2sz:
"x ≤ (x AND NOT(mask sz)) + 2 ^ sz - 1"
for x :: ‹'a::len word›
⟨*proof*⟩

lemma ucast_add:
"ucast (a + (b :: 'a :: len word)) = ucast a + (ucast b :: ('a signed word))"
⟨*proof*⟩

lemma ucast_minus:
"ucast (a - (b :: 'a :: len word)) = ucast a - (ucast b :: ('a signed word))"
⟨*proof*⟩

lemma scast_ucast_add_one [simp]:
"scast (ucast (x :: 'a::len word) + (1 :: 'a signed word)) = x + 1"
⟨*proof*⟩

lemma word_and_le_plus_one:
"a > 0 ⟹ (x :: 'a :: len word) AND (a - 1) < a"
⟨*proof*⟩

lemma unat_of_ucast_then_shift_eq_unat_of_shift[simp]:
"LENGTH('b) ≥ LENGTH('a)
⟹ unat ((ucast (x :: 'a :: len word) :: 'b :: len word) >> n) = unat (x >> n)"
⟨*proof*⟩

lemma unat_of_ucast_then_mask_eq_unat_of_mask[simp]:
"LENGTH('b) ≥ LENGTH('a)
⟹ unat ((ucast (x :: 'a :: len word) :: 'b :: len word) AND mask m) = unat (x AND mask m)"

⟨*proof*⟩

lemma shiftr_less_t2n3:
"⟦ (2 :: 'a word) ^ (n + m) = 0; m < LENGTH('a) ⟧
⟹ (x :: 'a :: len word) >> n < 2 ^ m"
⟨*proof*⟩

lemma unat_shiftr_le_bound:
"⟦ 2 ^ (LENGTH('a :: len) - n) - 1 ≤ bnd; 0 < n ⟧
⟹ unat ((x :: 'a word) >> n) ≤ bnd"
⟨*proof*⟩

lemma shiftr_eqD:
"⟦ x >> n = y >> n; is_aligned x n; is_aligned y n ⟧
⟹ x = y"
⟨*proof*⟩

lemma word_shiftr_shiftl_shiftr_eq_shiftr:
"a ≥ b ⟹ (x :: 'a :: len word) >> a << b >> b = x >> a"
⟨*proof*⟩

lemma of_int_uint_ucast:
"of_int (uint (x :: 'a::len word)) = (ucast x :: 'b::len word)"
⟨*proof*⟩

lemma mod_mask_drop:
"⟦ m = 2 ^ n; 0 < m; mask n AND msk = mask n ⟧
⟹ (x mod m) AND msk = x mod m"
for x :: ‹'a::len word›
⟨*proof*⟩

lemma mask_eq_ucast_eq:
"⟦ x AND mask LENGTH('a) = (x :: ('c :: len word));
LENGTH('a) ≤ LENGTH('b)⟧
⟹ ucast (ucast x :: ('a :: len word)) = (ucast x :: ('b :: len word))"
⟨*proof*⟩

lemma of_nat_less_t2n:
"of_nat i < (2 :: ('a :: len) word) ^ n ⟹ n < LENGTH('a) ∧ unat (of_nat
i :: 'a word) < 2 ^ n"
⟨*proof*⟩

lemma two_power_increasing_less_1:
"⟦ n ≤ m; m ≤ LENGTH('a) ⟧ ⟹ (2 :: 'a :: len word) ^ n - 1 ≤ 2 ^
m - 1"
⟨*proof*⟩

lemma word_sub_mono4:
"⟦ y + x ≤ z + x; y ≤ y + x; z ≤ z + x ⟧ ⟹ y ≤ z" **for** y :: "'a ::

```
len word"
```
⟨*proof*⟩

lemma eq_or_less_helperD:
```
  "⟦ n = unat (2 ^ m - 1 :: 'a :: len word) ∨ n < unat (2 ^ m - 1 :: 'a
word); m < LENGTH('a) ⟧
   ⟹ n < 2 ^ m"
```
⟨*proof*⟩

lemma mask_sub:
```
  "n ≤ m ⟹ mask m - mask n = mask m AND NOT(mask n :: 'a::len word)"
```
⟨*proof*⟩

lemma neg_mask_diff_bound:
```
  "sz'≤ sz ⟹ (ptr AND NOT(mask sz')) - (ptr AND NOT(mask sz)) ≤ 2 ^
sz - 2 ^ sz'"
  (is "_ ⟹ ?lhs ≤ ?rhs")
  for ptr :: ‹'a::len word›
```
⟨*proof*⟩

lemma mask_out_eq_0:
```
  "⟦ idx < 2 ^ sz; sz < LENGTH('a) ⟧ ⟹ (of_nat idx :: 'a :: len word)
AND NOT(mask sz) = 0"
```
⟨*proof*⟩

lemma is_aligned_neg_mask_eq':
```
  "is_aligned ptr sz = (ptr AND NOT(mask sz) = ptr)"
```
⟨*proof*⟩

lemma neg_mask_mask_unat:
```
  "sz < LENGTH('a)
   ⟹ unat ((ptr :: 'a :: len word) AND NOT(mask sz)) + unat (ptr AND
mask sz) = unat ptr"
```
⟨*proof*⟩

lemma unat_pow_le_intro:
```
  "LENGTH('a) ≤ n ⟹ unat (x :: 'a :: len word) < 2 ^ n"
```
⟨*proof*⟩

lemma unat_shiftl_less_t2n:
```
  ‹unat (x << n) < 2 ^ m›
  if ‹unat (x :: 'a :: len word) < 2 ^ (m - n)› ‹m < LENGTH('a)›
```
⟨*proof*⟩

lemma unat_is_aligned_add:
```
  "⟦ is_aligned p n; unat d < 2 ^ n ⟧
   ⟹ unat (p + d AND mask n) = unat d ∧ unat (p + d AND NOT(mask n))
= unat p"
```
⟨*proof*⟩

lemma unat_shiftr_shiftl_mask_zero:
"⟦ c + a ≥ LENGTH('a) + b ; c < LENGTH('a) ⟧
⟹ unat (((q :: 'a :: len word) >> a << b) AND NOT(mask c)) = 0"
⟨*proof*⟩

lemmas of_nat_ucast = ucast_of_nat[symmetric]

lemma shift_then_mask_eq_shift_low_bits:
"x ≤ mask (low_bits + high_bits) ⟹ (x >> low_bits) AND mask high_bits
= x >> low_bits"
for x :: ‹'a::len word›
⟨*proof*⟩

lemma leq_low_bits_iff_zero:
"⟦ x ≤ mask (low bits + high bits); x >> low_bits = 0 ⟧ ⟹ (x AND mask
low_bits = 0) = (x = 0)"
for x :: ‹'a::len word›
⟨*proof*⟩

lemma unat_less_iff:
"⟦ unat (a :: 'a :: len word) = b; c < 2 ^ LENGTH('a) ⟧ ⟹ (a < of_nat
c) = (b < c)"
⟨*proof*⟩

lemma is_aligned_no_overflow3:
"⟦ is_aligned (a :: 'a :: len word) n; n < LENGTH('a); b < 2 ^ n; c ≤
2 ^ n; b < c ⟧
⟹ a + b ≤ a + (c - 1)"
⟨*proof*⟩

lemma mask_add_aligned_right:
"is_aligned p n ⟹ (q + p) AND mask n = q AND mask n"
⟨*proof*⟩

lemma leq_high_bits_shiftr_low_bits_leq_bits_mask:
"x ≤ mask high_bits ⟹ (x :: 'a :: len word) << low_bits ≤ mask (low_bits
+ high_bits)"
⟨*proof*⟩

lemma word_two_power_neg_ineq:
assumes "2 ^ m ≠ (0::'a word)"
shows "2 ^ n ≤ - (2 ^ m :: 'a :: len word)"
⟨*proof*⟩

lemma unat_shiftl_absorb:
fixes x :: "'a :: len word"
shows "⟦ x ≤ 2 ^ p; p + k < LENGTH('a) ⟧ ⟹ unat x * 2 ^ k = unat
(x * 2 ^ k)"

⟨*proof*⟩

lemma word_plus_mono_right_split:
fixes x :: "'a :: len word"
assumes "unat (x AND mask sz) + unat z < 2 ^ sz" **and** "sz < LENGTH('a)"
shows "x ≤ x + z"
⟨*proof*⟩

lemma mul_not_mask_eq_neg_shiftl:
"NOT(mask n :: 'a::len word) = -1 << n"
⟨*proof*⟩

lemma shiftr_mul_not_mask_eq_and_not_mask:
"(x >> n) * NOT(mask n) = - (x AND NOT(mask n))"
for x :: ‹'a::len word›
⟨*proof*⟩

lemma mask_eq_n1_shiftr:
"n ≤ LENGTH('a) ⟹ (mask n :: 'a :: len word) = -1 >> (LENGTH('a) -
n)"
⟨*proof*⟩

lemma is_aligned_mask_out_add_eq:
"is_aligned p n ⟹ (p + x) AND NOT(mask n) = p + (x AND NOT(mask n))"
⟨*proof*⟩

lemmas is_aligned_mask_out_add_eq_sub
= is_aligned_mask_out_add_eq[**where** x="a - b" **for** a b, simplified field_simps]

lemma aligned_bump_down:
"is_aligned x n ⟹ (x - 1) AND NOT(mask n) = x - 2 ^ n"
⟨*proof*⟩

lemma unat_2tp_if:
"unat (2 ^ n :: ('a :: len) word) = (if n < LENGTH ('a) then 2 ^ n else
0)"
⟨*proof*⟩

lemma mask_of_mask:
"mask (n::nat) AND mask (m::nat) = (mask (min m n) :: 'a::len word)"
⟨*proof*⟩

lemma unat_signed_ucast_less_ucast:
"LENGTH('a) ≤ LENGTH('b) ⟹ unat (ucast (x :: 'a :: len word) :: 'b
:: len signed word) = unat x"
⟨*proof*⟩

lemma toEnum_of_ucast:
"LENGTH('b) ≤ LENGTH('a) ⟹

(toEnum (unat (b::'b :: len word))::'a :: len word) = of_nat (unat b)"
⟨*proof*⟩

lemma plus_mask_AND_NOT_mask_eq:
"x AND NOT(mask n) = x ⟹ (x + mask n) AND NOT(mask n) = x" **for** x::‹'a::len word›
⟨*proof*⟩

lemmas unat_ucast_mask = unat_ucast_eq_unat_and_mask[**where** w=a **for** a]

lemma t2n_mask_eq_if:
"2 ^ n AND mask m = (if n < m then 2 ^ n else (0 :: 'a::len word))"
⟨*proof*⟩

lemma unat_ucast_le:
"unat (ucast (x :: 'a :: len word) :: 'b :: len word) ≤ unat x"
⟨*proof*⟩

lemma ucast_le_up_down_iff:
"⟦ LENGTH('a) ≤ LENGTH('b); (x :: 'b :: len word) ≤ ucast (- 1 :: 'a :: len word) ⟧
⟹ (ucast x ≤ (y :: 'a word)) = (x ≤ ucast y)"
⟨*proof*⟩

lemma ucast_ucast_mask_shift:
"a ≤ LENGTH('a) + b
⟹ ucast (ucast (p AND mask a >> b) :: 'a :: len word) = p AND mask a >> b"
⟨*proof*⟩

lemma unat_ucast_mask_shift:
"a ≤ LENGTH('a) + b
⟹ unat (ucast (p AND mask a >> b) :: 'a :: len word) = unat (p AND mask a >> b)"
⟨*proof*⟩

lemma mask_overlap_zero:
"a ≤ b ⟹ (p AND mask a) AND NOT(mask b) = 0"
for p :: ‹'a::len word›
⟨*proof*⟩

lemma mask_shifl_overlap_zero:
"a + c ≤ b ⟹ (p AND mask a << c) AND NOT(mask b) = 0"
for p :: ‹'a::len word›
⟨*proof*⟩

lemma mask_overlap_zero':
"a ≥ b ⟹ (p AND NOT(mask a)) AND mask b = 0"

for p :: ‹'a::len word›
⟨*proof*⟩

lemma mask_rshift_mult_eq_rshift_lshift:
"((a :: 'a :: len word) >> b) * (1 << c) = (a >> b << c)"
⟨*proof*⟩

lemma shift_alignment:
"a ≥ b ⟹ is_aligned (p >> a << a) b"
⟨*proof*⟩

lemma mask_split_sum_twice:
"a ≥ b ⟹ (p AND NOT(mask a)) + ((p AND mask a) AND NOT(mask b)) +
(p AND mask b) = p"
for p :: ‹'a::len word›
⟨*proof*⟩

lemma mask_shift_eq_mask_mask:
"(p AND mask a >> b << b) = (p AND mask a) AND NOT(mask b)"
for p :: ‹'a::len word›
⟨*proof*⟩

lemma mask_shift_sum:
"⟦ a ≥ b; unat n = unat (p AND mask b) ⟧
⟹ (p AND NOT(mask a)) + (p AND mask a >> b) * (1 << b) + n = (p ::
'a :: len word)"
⟨*proof*⟩

lemma is_up_compose:
"⟦ is_up uc; is_up uc' ⟧ ⟹ is_up (uc' ∘ uc)"
⟨*proof*⟩

lemma of_int_sint_scast:
"of_int (sint (x :: 'a :: len word)) = (scast x :: 'b :: len word)"
⟨*proof*⟩

lemma scast_of_nat_to_signed [simp]:
"scast (of_nat x :: 'a :: len word) = (of_nat x :: 'a signed word)"
⟨*proof*⟩

lemma scast_of_nat_signed_to_unsigned_add:
"scast (of_nat x + of_nat y :: 'a :: len signed word) = (of_nat x +
of_nat y :: 'a :: len word)"
⟨*proof*⟩

lemma scast_of_nat_unsigned_to_signed_add:
"(scast (of_nat x + of_nat y :: 'a :: len word)) = (of_nat x + of_nat
y :: 'a :: len signed word)"
⟨*proof*⟩

lemma and_mask_cases:
fixes x :: "'a :: len word"
assumes len: "n < LENGTH('a)"
shows "x AND mask n ∈ of_nat ` set [0 ..< 2 ^ n]"
⟨*proof*⟩

lemma sint_eq_uint_2pl:
"⟦ (a :: 'a :: len word) < 2 ^ (LENGTH('a) - 1) ⟧
⟹ sint a = uint a"
⟨*proof*⟩

lemma pow_sub_less:
"⟦ a + b ≤ LENGTH('a); unat (x :: 'a :: len word) = 2 ^ a ⟧
⟹ unat (x * 2 ^ b - 1) < 2 ^ (a + b)"
⟨*proof*⟩

lemma sle_le_2pl:
"⟦ (b :: 'a :: len word) < 2 ^ (LENGTH('a) - 1); a ≤ b ⟧ ⟹ a <=s b"
⟨*proof*⟩

lemma sless_less_2pl:
"⟦ (b :: 'a :: len word) < 2 ^ (LENGTH('a) - 1); a < b ⟧ ⟹ a <s b"
⟨*proof*⟩

lemma and_mask2:
"w << n >> n = w AND mask (size w - n)"
for w :: ‹'a::len word›
⟨*proof*⟩

lemma aligned_sub_aligned_simple:
"⟦ is_aligned a n; is_aligned b n ⟧ ⟹ is_aligned (a - b) n"
⟨*proof*⟩

lemma minus_one_shift:
"- (1 << n) = (-1 << n :: 'a::len word)"
⟨*proof*⟩

lemma ucast_eq_mask:
"(UCAST('a::len → 'b::len) x = UCAST('a → 'b) y) =
(x AND mask LENGTH('b) = y AND mask LENGTH('b))"
⟨*proof*⟩

context
fixes w :: "'a::len word"
begin

private lemma sbintrunc_uint_ucast:
‹signed_take_bit n (uint (ucast w :: 'b word)) = signed_take_bit n (uint

```
w)› if ‹Suc n = LENGTH('b::len)›
  ⟨proof⟩ lemma test_bit_sbintrunc:
  assumes "i < LENGTH('a)"
  shows "bit (word_of_int (signed_take_bit n (uint w)) :: 'a word) i
           = (if n < i then bit w n else bit w i)"
  ⟨proof⟩ lemma test_bit_sbintrunc_ucast:
  assumes len_a: "i < LENGTH('a)"
  shows "bit (word_of_int (signed_take_bit (LENGTH('b) - 1) (uint (ucast
w :: 'b word))) :: 'a word) i
          = (if LENGTH('b::len) ≤ i then bit w (LENGTH('b) - 1) else bit
w i)"
  ⟨proof⟩

lemma scast_ucast_high_bits:
  ‹scast (ucast w :: 'b::len word) = w
     ⟷ (∀ i ∈ {LENGTH('b) ..< size w}. bit w i = bit w (LENGTH('b)
- 1))›
⟨proof⟩

lemma scast_ucast_mask_compare:
  "scast (ucast w :: 'b::len word) = w
   ⟷ (w ≤ mask (LENGTH('b) - 1) ∨ NOT(mask (LENGTH('b) - 1)) ≤ w)"
  ⟨proof⟩

lemma ucast_less_shiftl_helper':
  "⟦ LENGTH('b) + (a::nat) < LENGTH('a); 2 ^ (LENGTH('b) + a) ≤ n⟧
   ⟹ (ucast (x :: 'b::len word) << a) < (n :: 'a::len word)"
  ⟨proof⟩

end

lemma ucast_ucast_mask2:
  "is_down (UCAST ('a → 'b)) ⟹
   UCAST ('b::len → 'c::len) (UCAST ('a::len → 'b::len) x) = UCAST ('a
→ 'c) (x AND mask LENGTH('b))"
  ⟨proof⟩

lemma ucast_NOT:
  "ucast (NOT x) = NOT(ucast x) AND mask (LENGTH('a))" for x::"'a::len
word"
  ⟨proof⟩

lemma ucast_NOT_down:
  "is_down UCAST('a::len → 'b::len) ⟹ UCAST('a → 'b) (NOT x) = NOT(UCAST('a
→ 'b) x)"
  ⟨proof⟩

lemma upto_enum_step_shift:
  assumes "is_aligned p n"
```

shows "([p , p + 2 ^ m .e. p + 2 ^ n - 1]) = map ((+) p) [0, 2 ^ m .e. 2 ^ n - 1]"
⟨*proof*⟩

lemma upto_enum_step_shift_red:
"⟦ is_aligned p sz; sz < LENGTH('a); us ≤ sz ⟧
⟹ [p :: 'a :: len word, p + 2 ^ us .e. p + 2 ^ sz - 1]
= map (λx. p + of_nat x * 2 ^ us) [0 ..< 2 ^ (sz - us)]"
⟨*proof*⟩

lemma upto_enum_step_subset:
"set [x, y .e. z] ⊆ {x .. z}"
⟨*proof*⟩

lemma ucast_distrib:
fixes M :: "'a::len word ⇒ 'a::len word ⇒ 'a::len word"
fixes M' :: "'b::len word ⇒ 'b::len word ⇒ 'b::len word"
fixes L :: "int ⇒ int ⇒ int"
assumes lift_M: "⋀x y. uint (M x y) = L (uint x) (uint y) mod 2 ^ LENGTH('a)"
assumes lift_M': "⋀x y. uint (M' x y) = L (uint x) (uint y) mod 2 ^ LENGTH('b)"
assumes distrib: "⋀x y. (L (x mod (2 ^ LENGTH('b))) (y mod (2 ^ LENGTH('b)))) mod (2 ^ LENGTH('b))
= (L x y) mod (2 ^ LENGTH('b))"
assumes is_down: "is_down (ucast :: 'a word ⇒ 'b word)"
shows "ucast (M a b) = M' (ucast a) (ucast b)"
⟨*proof*⟩

lemma ucast_down_add:
"is_down (ucast:: 'a word ⇒ 'b word) ⟹ ucast ((a :: 'a::len word) + b) = (ucast a + ucast b :: 'b::len word)"
⟨*proof*⟩

lemma ucast_down_minus:
"is_down (ucast:: 'a word ⇒ 'b word) ⟹ ucast ((a :: 'a::len word) - b) = (ucast a - ucast b :: 'b::len word)"
⟨*proof*⟩

lemma ucast_down_mult:
"is_down (ucast:: 'a word ⇒ 'b word) ⟹ ucast ((a :: 'a::len word) * b) = (ucast a * ucast b :: 'b::len word)"
⟨*proof*⟩

lemma scast_distrib:
fixes M :: "'a::len word ⇒ 'a::len word ⇒ 'a::len word"
fixes M' :: "'b::len word ⇒ 'b::len word ⇒ 'b::len word"
fixes L :: "int ⇒ int ⇒ int"

assumes lift_M: "⋀x y. uint (M x y) = L (uint x) (uint y) mod 2 ^ LENGTH('a)"
assumes lift_M': "⋀x y. uint (M' x y) = L (uint x) (uint y) mod 2 ^ LENGTH('b)"
assumes distrib: "⋀x y. (L (x mod (2 ^ LENGTH('b))) (y mod (2 ^ LENGTH('b)))) mod (2 ^ LENGTH('b))
= (L x y) mod (2 ^ LENGTH('b))"
assumes is_down: "is_down (scast :: 'a word ⇒ 'b word)"
shows "scast (M a b) = M' (scast a) (scast b)"
⟨*proof*⟩

lemma scast_down_add:
"is_down (scast:: 'a word ⇒ 'b word) ⟹ scast ((a :: 'a::len word) + b) = (scast a + scast b :: 'b::len word)"
⟨*proof*⟩

lemma scast_down_minus:
"is_down (scast:: 'a word ⇒ 'b word) ⟹ scast ((a :: 'a::len word) - b) = (scast a - scast b :: 'b::len word)"
⟨*proof*⟩

lemma scast_down_mult:
"is_down (scast:: 'a word ⇒ 'b word) ⟹ scast ((a :: 'a::len word) * b) = (scast a * scast b :: 'b::len word)"
⟨*proof*⟩

lemma scast_ucast_1:
"⟦ is_down (ucast :: 'a word ⇒ 'b word); is_down (ucast :: 'b word ⇒ 'c word) ⟧ ⟹
(scast (ucast (a :: 'a::len word) :: 'b::len word) :: 'c::len word) = ucast a"
⟨*proof*⟩

lemma scast_ucast_3:
"⟦ is_down (ucast :: 'a word ⇒ 'c word); is_down (ucast :: 'b word ⇒ 'c word) ⟧ ⟹
(scast (ucast (a :: 'a::len word) :: 'b::len word) :: 'c::len word) = ucast a"
⟨*proof*⟩

lemma scast_ucast_4:
"⟦ is_up (ucast :: 'a word ⇒ 'b word); is_down (ucast :: 'b word ⇒ 'c word) ⟧ ⟹
(scast (ucast (a :: 'a::len word) :: 'b::len word) :: 'c::len word) = ucast a"
⟨*proof*⟩

lemma scast_scast_b:
"⟦ is_up (scast :: 'a word ⇒ 'b word) ⟧ ⟹

```
    (scast (scast (a :: 'a::len word) :: 'b::len word) :: 'c::len word)
= scast a"
```
⟨*proof*⟩

lemma ucast_scast_1:
```
  "⟦ is_down (scast :: 'a word ⇒ 'b word); is_down (ucast :: 'b word
⇒ 'c word) ⟧ ⟹
           (ucast (scast (a :: 'a::len word) :: 'b::len word) :: 'c::len
word) = scast a"
```
⟨*proof*⟩

lemma ucast_scast_3:
```
  "⟦ is_down (scast :: 'a word ⇒ 'c word); is_down (ucast :: 'b word
⇒ 'c word) ⟧ ⟹
    (ucast (scast (a :: 'a::len word) :: 'b::len word) :: 'c::len word)
= scast a"
```
⟨*proof*⟩

lemma ucast_scast_4:
```
  "⟦ is_up (scast :: 'a word ⇒ 'b word); is_down (ucast :: 'b word ⇒
'c word) ⟧ ⟹
    (ucast (scast (a :: 'a::len word) :: 'b::len word) :: 'c::len word)
= scast a"
```
⟨*proof*⟩

lemma ucast_ucast_a:
```
  "⟦ is_down (ucast :: 'b word ⇒ 'c word) ⟧ ⟹
       (ucast (ucast (a :: 'a::len word) :: 'b::len word) :: 'c::len
word) = ucast a"
```
⟨*proof*⟩

lemma ucast_ucast_b:
```
  "⟦ is_up (ucast :: 'a word ⇒ 'b word) ⟧ ⟹
    (ucast (ucast (a :: 'a::len word) :: 'b::len word) :: 'c::len word)
= ucast a"
```
⟨*proof*⟩

lemma scast_scast_a:
```
  "⟦ is_down (scast :: 'b word ⇒ 'c word) ⟧ ⟹
           (scast (scast (a :: 'a::len word) :: 'b::len word) :: 'c::len
word) = scast a"
```
⟨*proof*⟩

lemma scast_down_wi [OF refl]:
```
  "uc = scast ⟹ is_down uc ⟹ uc (word_of_int x) = word_of_int x"
```
⟨*proof*⟩

lemmas cast_simps =
```
  is_down is_up
```

```
  scast_down_add scast_down_minus scast_down_mult
  ucast_down_add ucast_down_minus ucast_down_mult
  scast_ucast_1 scast_ucast_3 scast_ucast_4
  ucast_scast_1 ucast_scast_3 ucast_scast_4
  ucast_ucast_a ucast_ucast_b
  scast_scast_a scast_scast_b
  ucast_down_wi scast_down_wi
  ucast_of_nat scast_of_nat
  uint_up_ucast sint_up_scast
  up_scast_surj up_ucast_surj
```

lemma sdiv_word_max:

```
    "(sint (a :: ('a::len) word) sdiv sint (b :: ('a::len) word) < (2
^ (size a - 1))) =
          ((a ≠ - (2 ^ (size a - 1)) ∨ (b ≠ -1)))"
    (is "?lhs = (¬ ?a_int_min ∨ ¬ ?b_minus1)")
```

⟨*proof*⟩

lemmas sdiv_word_min' = sdiv_word_min [simplified word_size, simplified]
lemmas sdiv_word_max' = sdiv_word_max [simplified word_size, simplified]

lemma signed_arith_ineq_checks_to_eq:

```
  "((- (2 ^ (size a - 1)) ≤ (sint a + sint b)) ∧ (sint a + sint b ≤ (2
^ (size a - 1) - 1)))
    = (sint a + sint b = sint (a + b ))"
  "((- (2 ^ (size a - 1)) ≤ (sint a - sint b)) ∧ (sint a - sint b ≤ (2
^ (size a - 1) - 1)))
    = (sint a - sint b = sint (a - b))"
  "((- (2 ^ (size a - 1)) ≤ (- sint a)) ∧ (- sint a) ≤ (2 ^ (size a -
1) - 1))
    = ((- sint a) = sint (- a))"
  "((- (2 ^ (size a - 1)) ≤ (sint a * sint b)) ∧ (sint a * sint b ≤ (2
^ (size a - 1) - 1)))
    = (sint a * sint b = sint (a * b))"
  "((- (2 ^ (size a - 1)) ≤ (sint a sdiv sint b)) ∧ (sint a sdiv sint
b ≤ (2 ^ (size a - 1) - 1)))
    = (sint a sdiv sint b = sint (a sdiv b))"
  "((- (2 ^ (size a - 1)) ≤ (sint a smod sint b)) ∧ (sint a smod sint
b ≤ (2 ^ (size a - 1) - 1)))
    = (sint a smod sint b = sint (a smod b))"
  ⟨proof⟩
```

lemma signed_arith_sint:

```
  "((- (2 ^ (size a - 1)) ≤ (sint a + sint b)) ∧ (sint a + sint b ≤ (2
^ (size a - 1) - 1)))
    ⟹ sint (a + b) = (sint a + sint b)"
  "((- (2 ^ (size a - 1)) ≤ (sint a - sint b)) ∧ (sint a - sint b ≤ (2
^ (size a - 1) - 1)))
    ⟹ sint (a - b) = (sint a - sint b)"
```

```
  "((- (2 ^ (size a - 1)) ≤ (- sint a)) ∧ (- sint a) ≤ (2 ^ (size a -
1) - 1))
    ⟹ sint (- a) = (- sint a)"
  "((- (2 ^ (size a - 1)) ≤ (sint a * sint b)) ∧ (sint a * sint b ≤ (2
^ (size a - 1) - 1)))
    ⟹ sint (a * b) = (sint a * sint b)"
  "((- (2 ^ (size a - 1)) ≤ (sint a sdiv sint b)) ∧ (sint a sdiv sint
b ≤ (2 ^ (size a - 1) - 1)))
    ⟹ sint (a sdiv b) = (sint a sdiv sint b)"
  "((- (2 ^ (size a - 1)) ≤ (sint a smod sint b)) ∧ (sint a smod sint
b ≤ (2 ^ (size a - 1) - 1)))
    ⟹ sint (a smod b) = (sint a smod sint b)"
  ⟨proof⟩

lemma nasty_split_lt:
  ‹x * 2 ^ n + (2 ^ n - 1) ≤ 2 ^ m - 1›
    if ‹x < 2 ^ (m - n)› ‹n ≤ m› ‹m < LENGTH('a::len)›
    for x :: ‹'a::len word›
⟨proof⟩

end

end
```

20 Words of Length 8

```
theory Word_8
imports
  More_Word
  Enumeration_Word
  Even_More_List
  Signed_Words
  Word_Lemmas
begin

context
  includes bit_operations_syntax
begin

lemma len8: "len_of (x :: 8 itself) = 8" ⟨proof⟩

lemma word8_and_max_simp:
  ‹x AND 0xFF = x› for x :: ‹8 word›
  ⟨proof⟩

lemma enum_word8_eq:
  ‹enum = [0 :: 8 word, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19,
                            20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
```

```
31, 32, 33, 34, 35, 36,
                              37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47,
48, 49, 50, 51, 52, 53,
                              54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64,
65, 66, 67, 68, 69, 70,
                              71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81,
82, 83, 84, 85, 86, 87,
                              88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98,
99, 100, 101, 102, 103,
                              104, 105, 106, 107, 108, 109, 110, 111, 112,
113, 114, 115, 116, 117,
                              118, 119, 120, 121, 122, 123, 124, 125, 126,
127, 128, 129, 130, 131,
                              132, 133, 134, 135, 136, 137, 138, 139, 140,
141, 142, 143, 144, 145,
                              146, 147, 148, 149, 150, 151, 152, 153, 154,
155, 156, 157, 158, 159,
                              160, 161, 162, 163, 164, 165, 166, 167, 168,
169, 170, 171, 172, 173,
                              174, 175, 176, 177, 178, 179, 180, 181, 182,
183, 184, 185, 186, 187,
                              188, 189, 190, 191, 192, 193, 194, 195, 196,
197, 198, 199, 200, 201,
                              202, 203, 204, 205, 206, 207, 208, 209, 210,
211, 212, 213, 214, 215,
                              216, 217, 218, 219, 220, 221, 222, 223, 224,
225, 226, 227, 228, 229,
                              230, 231, 232, 233, 234, 235, 236, 237, 238,
239, 240, 241, 242, 243,
                              244, 245, 246, 247, 248, 249, 250, 251, 252,
253, 254, 255]› (is ‹?lhs = ?rhs›)
```

⟨*proof*⟩

lemma `set_enum_word8_def:`

```
  "(set enum :: 8 word set) = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
13, 14, 15, 16, 17, 18, 19,
                              20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
31, 32, 33, 34, 35, 36,
                              37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47,
48, 49, 50, 51, 52, 53,
                              54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64,
65, 66, 67, 68, 69, 70,
                              71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81,
82, 83, 84, 85, 86, 87,
                              88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98,
99, 100, 101, 102, 103,
                              104, 105, 106, 107, 108, 109, 110, 111, 112,
113, 114, 115, 116, 117,
                              118, 119, 120, 121, 122, 123, 124, 125, 126,
```

```
127, 128, 129, 130, 131,
                                132, 133, 134, 135, 136, 137, 138, 139, 140,
141, 142, 143, 144, 145,
                                146, 147, 148, 149, 150, 151, 152, 153, 154,
155, 156, 157, 158, 159,
                                160, 161, 162, 163, 164, 165, 166, 167, 168,
169, 170, 171, 172, 173,
                                174, 175, 176, 177, 178, 179, 180, 181, 182,
183, 184, 185, 186, 187,
                                188, 189, 190, 191, 192, 193, 194, 195, 196,
197, 198, 199, 200, 201,
                                202, 203, 204, 205, 206, 207, 208, 209, 210,
211, 212, 213, 214, 215,
                                216, 217, 218, 219, 220, 221, 222, 223, 224,
225, 226, 227, 228, 229,
                                230, 231, 232, 233, 234, 235, 236, 237, 238,
239, 240, 241, 242, 243,
                                244, 245, 246, 247, 248, 249, 250, 251, 252,
253, 254, 255}"
  ⟨proof⟩

lemma set_strip_insert: "⟦ x ∈ insert a S; x ≠ a ⟧ ⟹ x ∈ S"
  ⟨proof⟩

lemma word8_exhaust:
  fixes x :: <8 word>
  shows "⟦x ≠ 0; x ≠ 1; x ≠ 2; x ≠ 3; x ≠ 4; x ≠ 5; x ≠ 6; x ≠ 7;
x ≠ 8; x ≠ 9; x ≠ 10; x ≠ 11; x ≠
         12; x ≠ 13; x ≠ 14; x ≠ 15; x ≠ 16; x ≠ 17; x ≠ 18; x ≠ 19;
x ≠ 20; x ≠ 21; x ≠ 22; x ≠
         23; x ≠ 24; x ≠ 25; x ≠ 26; x ≠ 27; x ≠ 28; x ≠ 29; x ≠ 30;
x ≠ 31; x ≠ 32; x ≠ 33; x ≠
         34; x ≠ 35; x ≠ 36; x ≠ 37; x ≠ 38; x ≠ 39; x ≠ 40; x ≠ 41;
x ≠ 42; x ≠ 43; x ≠ 44; x ≠
         45; x ≠ 46; x ≠ 47; x ≠ 48; x ≠ 49; x ≠ 50; x ≠ 51; x ≠ 52;
x ≠ 53; x ≠ 54; x ≠ 55; x ≠
         56; x ≠ 57; x ≠ 58; x ≠ 59; x ≠ 60; x ≠ 61; x ≠ 62; x ≠ 63;
x ≠ 64; x ≠ 65; x ≠ 66; x ≠
         67; x ≠ 68; x ≠ 69; x ≠ 70; x ≠ 71; x ≠ 72; x ≠ 73; x ≠ 74;
x ≠ 75; x ≠ 76; x ≠ 77; x ≠
         78; x ≠ 79; x ≠ 80; x ≠ 81; x ≠ 82; x ≠ 83; x ≠ 84; x ≠ 85;
x ≠ 86; x ≠ 87; x ≠ 88; x ≠
         89; x ≠ 90; x ≠ 91; x ≠ 92; x ≠ 93; x ≠ 94; x ≠ 95; x ≠ 96;
x ≠ 97; x ≠ 98; x ≠ 99; x ≠
         100; x ≠ 101; x ≠ 102; x ≠ 103; x ≠ 104; x ≠ 105; x ≠ 106;
x ≠ 107; x ≠ 108; x ≠ 109; x ≠
         110; x ≠ 111; x ≠ 112; x ≠ 113; x ≠ 114; x ≠ 115; x ≠ 116;
x ≠ 117; x ≠ 118; x ≠ 119; x ≠
         120; x ≠ 121; x ≠ 122; x ≠ 123; x ≠ 124; x ≠ 125; x ≠ 126;
```

```
x ≠ 127; x ≠ 128; x ≠ 129; x ≠
          130; x ≠ 131; x ≠ 132; x ≠ 133; x ≠ 134; x ≠ 135; x ≠ 136;
x ≠ 137; x ≠ 138; x ≠ 139; x ≠
          140; x ≠ 141; x ≠ 142; x ≠ 143; x ≠ 144; x ≠ 145; x ≠ 146;
x ≠ 147; x ≠ 148; x ≠ 149; x ≠
          150; x ≠ 151; x ≠ 152; x ≠ 153; x ≠ 154; x ≠ 155; x ≠ 156;
x ≠ 157; x ≠ 158; x ≠ 159; x ≠
          160; x ≠ 161; x ≠ 162; x ≠ 163; x ≠ 164; x ≠ 165; x ≠ 166;
x ≠ 167; x ≠ 168; x ≠ 169; x ≠
          170; x ≠ 171; x ≠ 172; x ≠ 173; x ≠ 174; x ≠ 175; x ≠ 176;
x ≠ 177; x ≠ 178; x ≠ 179; x ≠
          180; x ≠ 181; x ≠ 182; x ≠ 183; x ≠ 184; x ≠ 185; x ≠ 186;
x ≠ 187; x ≠ 188; x ≠ 189; x ≠
          190; x ≠ 191; x ≠ 192; x ≠ 193; x ≠ 194; x ≠ 195; x ≠ 196;
x ≠ 197; x ≠ 198; x ≠ 199; x ≠
          200; x ≠ 201; x ≠ 202; x ≠ 203; x ≠ 204; x ≠ 205; x ≠ 206;
x ≠ 207; x ≠ 208; x ≠ 209; x ≠
          210; x ≠ 211; x ≠ 212; x ≠ 213; x ≠ 214; x ≠ 215; x ≠ 216;
x ≠ 217; x ≠ 218; x ≠ 219; x ≠
          220; x ≠ 221; x ≠ 222; x ≠ 223; x ≠ 224; x ≠ 225; x ≠ 226;
x ≠ 227; x ≠ 228; x ≠ 229; x ≠
          230; x ≠ 231; x ≠ 232; x ≠ 233; x ≠ 234; x ≠ 235; x ≠ 236;
x ≠ 237; x ≠ 238; x ≠ 239; x ≠
          240; x ≠ 241; x ≠ 242; x ≠ 243; x ≠ 244; x ≠ 245; x ≠ 246;
x ≠ 247; x ≠ 248; x ≠ 249; x ≠
          250; x ≠ 251; x ≠ 252; x ≠ 253; x ≠ 254; x ≠ 255⟧ ⟹ P"
```
⟨*proof*⟩

end

end

21 Words of Length 16

theory Word_16
imports
More_Word
Signed_Words
begin

lemma len16: "len_of (x :: 16 itself) = 16" ⟨*proof*⟩

context
includes bit_operations_syntax
begin

lemma word16_and_max_simp:
‹x AND 0xFFFF = x› **for** x :: ‹16 word›
⟨*proof*⟩

end

end

22 Additional Syntax for Word Bit Operations

theory Word_Syntax
imports
"HOL-Library.Word"
begin

Additional bit and type syntax that forces word types.

context
includes bit_operations_syntax
begin

abbreviation
wordNOT :: "'a::len word ⇒ 'a word" (<(<open_block notation=<prefix ~~>>~~ _)> [70] 71)
where
"~~ x == NOT x"

abbreviation
wordAND :: "'a::len word ⇒ 'a word ⇒ 'a word" (**infixr** <&&> 64)
where
"a && b == a AND b"

abbreviation
wordOR :: "'a::len word ⇒ 'a word ⇒ 'a word" (**infixr** <||> 59)
where
"a || b == a OR b"

abbreviation
wordXOR :: "'a::len word ⇒ 'a word ⇒ 'a word" (**infixr** <xor> 59)
where
"a xor b == a XOR b"

end

end

23 Names of Specific Word Lengths

theory Word_Names
imports Signed_Words
begin

```
type_synonym word8 = "8 word"
type_synonym word16 = "16 word"
type_synonym word32 = "32 word"
type_synonym word64 = "64 word"

type_synonym sword8 =  "8 sword"
type_synonym sword16 = "16 sword"
type_synonym sword32 = "32 sword"
type_synonym sword64 = "64 sword"

end
```

24 Misc word operations

```
theory More_Word_Operations
  imports
    "HOL-Library.Word"
    Aligned Reversed_Bit_Lists More_Misc Signed_Words
    Word_Lemmas Many_More Word_EqI

begin

context
  includes bit_operations_syntax
begin

definition
  ptr_add :: "'a :: len word ⇒ nat ⇒ 'a word" where
  "ptr_add ptr n ≡ ptr + of_nat n"

definition
  alignUp :: "'a::len word ⇒ nat ⇒ 'a word" where
 "alignUp x n ≡ x + 2 ^ n - 1 AND NOT (2 ^ n - 1)"

lemma alignUp_unfold:
  ‹alignUp w n = (w + mask n) AND NOT (mask n)›
  ⟨proof⟩


abbreviation mask_range :: "'a::len word ⇒ nat ⇒ 'a word set" where
  "mask_range p n ≡ {p .. p + mask n}"

definition
  w2byte :: "'a :: len word ⇒ 8 word" where
  "w2byte ≡ ucast"


definition
  word_clz :: "'a::len word ⇒ nat"
```

where
"word_clz w ≡ length (takeWhile Not (to_bl w))"

definition
word_ctz :: "'a::len word ⇒ nat"
where
"word_ctz w ≡ length (takeWhile Not (rev (to_bl w)))"

lemma word_ctz_unfold:
‹word_ctz w = length (takeWhile (Not ∘ bit w) [0..<LENGTH('a)])› **for** w :: ‹'a::len word›
⟨*proof*⟩

lemma word_ctz_unfold':
‹word_ctz w = Min (insert LENGTH('a) {n. bit w n})› **for** w :: ‹'a::len word›
⟨*proof*⟩

lemma word_ctz_le: "word_ctz (w :: ('a::len word)) ≤ LENGTH('a)"
⟨*proof*⟩

lemma word_ctz_less:
assumes "w ≠ 0"
shows "word_ctz (w :: ('a::len word)) < LENGTH('a)"
⟨*proof*⟩

lemma take_bit_word_ctz_eq [simp]:
‹take_bit LENGTH('a) (word_ctz w) = word_ctz w›
for w :: ‹'a::len word›
⟨*proof*⟩

lemma word_ctz_not_minus_1:
‹word_of_nat (word_ctz (w :: 'a :: len word)) ≠ (- 1 :: 'a::len word)›
if ‹1 < LENGTH('a)›
⟨*proof*⟩

lemma unat_of_nat_ctz_mw:
"unat (of_nat (word_ctz (w :: 'a :: len word)) :: 'a :: len word) = word_ctz w"
⟨*proof*⟩

lemma unat_of_nat_ctz_smw:
"unat (of_nat (word_ctz (w :: 'a :: len word)) :: 'a :: len signed word) = word_ctz w"
⟨*proof*⟩

definition
word_log2 :: "'a::len word ⇒ nat"

where
"word_log2 (w::'a::len word) ≡ size w - 1 - word_clz w"

definition
pop_count :: "('a::len) word ⇒ nat"
where
"pop_count w ≡ length (filter id (to_bl w))"

definition
sign_extend :: "nat ⇒ 'a::len word ⇒ 'a word"
where
"sign_extend n w ≡ if bit w n then w OR NOT (mask n) else w AND mask n"

lemma sign_extend_eq_signed_take_bit:
‹sign_extend = signed_take_bit›
⟨*proof*⟩

definition
sign_extended :: "nat ⇒ 'a::len word ⇒ bool"
where
"sign_extended n w ≡ ∀i. n < i ⟶ i < size w ⟶ bit w i = bit w n"

lemma ptr_add_0 [simp]:
"ptr_add ref 0 = ref "
⟨*proof*⟩

lemma pop_count_0[simp]:
"pop_count 0 = 0"
⟨*proof*⟩

lemma pop_count_1[simp]:
"pop_count 1 = 1"
⟨*proof*⟩

lemma pop_count_0_imp_0:
"(pop_count w = 0) = (w = 0)"
⟨*proof*⟩

lemma word_log2_zero_eq [simp]:
‹word_log2 0 = 0›
⟨*proof*⟩

lemma word_log2_unfold:
‹word_log2 w = (if w = 0 then 0 else Max {n. bit w n})›
for w :: ‹'a::len word›
⟨*proof*⟩

lemma word_log2_eqI:
  ‹word_log2 w = n›
  if ‹w ≠ 0› ‹bit w n› ‹⋀m. bit w m ⟹ m ≤ n›
  for w :: ‹'a::len word›
⟨*proof*⟩

lemma bit_word_log2:
  ‹bit w (word_log2 w)› **if** ‹w ≠ 0›
⟨*proof*⟩

lemma word_log2_maximum:
  ‹n ≤ word_log2 w› **if** ‹bit w n›
⟨*proof*⟩

lemma word_log2_nth_not_set:
  "⟦ word_log2 w < i ; i < size w ⟧ ⟹ ¬ bit w i"
  ⟨*proof*⟩

lemma word_log2_max:
  "word_log2 w < size w"
  ⟨*proof*⟩

lemma word_clz_0[simp]:
  "word_clz (0::'a::len word) = LENGTH('a)"
  ⟨*proof*⟩

lemma word_clz_minus_one[simp]:
  "word_clz (-1::'a::len word) = 0"
  ⟨*proof*⟩

lemma is_aligned_alignUp[simp]:
  "is_aligned (alignUp p n) n"
  ⟨*proof*⟩

lemma alignUp_le[simp]:
  "alignUp p n ≤ p + 2 ^ n - 1"
  ⟨*proof*⟩

lemma alignUp_idem:
  fixes a :: "'a::len word"
  assumes "is_aligned a n" "n < LENGTH('a)"
  shows "alignUp a n = a"
  ⟨*proof*⟩

lemma alignUp_not_aligned_eq:
  fixes a :: "'a :: len word"
  assumes al: "¬ is_aligned a n"
  and   sz: "n < LENGTH('a)"

shows "alignUp a n = (a div 2 ^ n + 1) * 2 ^ n"
⟨*proof*⟩

lemma alignUp_ge:
fixes a :: "'a :: len word"
assumes sz: "n < LENGTH('a)"
and nowrap: "alignUp a n ≠ 0"
shows "a ≤ alignUp a n"
⟨*proof*⟩

lemma alignUp_le_greater_al:
fixes x :: "'a :: len word"
assumes le: "a ≤ x"
and sz: "n < LENGTH('a)"
and al: "is_aligned x n"
shows "alignUp a n ≤ x"
⟨*proof*⟩

lemma alignUp_is_aligned_nz:
fixes a :: "'a :: len word"
assumes al: "is_aligned x n"
and sz: "n < LENGTH('a)"
and ax: "a ≤ x"
and az: "a ≠ 0"
shows "alignUp (a::'a :: len word) n ≠ 0"
⟨*proof*⟩

lemma alignUp_ar_helper:
fixes a :: "'a :: len word"
assumes al: "is_aligned x n"
and sz: "n < LENGTH('a)"
and sub: "{x..x + 2 ^ n - 1} ⊆ {a..b}"
and anz: "a ≠ 0"
shows "a ≤ alignUp a n ∧ alignUp a n + 2 ^ n - 1 ≤ b"
⟨*proof*⟩

lemma alignUp_def2:
"alignUp a sz = a + 2 ^ sz - 1 AND NOT (mask sz)"
⟨*proof*⟩

lemma alignUp_def3:
"alignUp a sz = 2^ sz + (a - 1 AND NOT (mask sz))"
⟨*proof*⟩

lemma alignUp_plus:
"is_aligned w us ⟹ alignUp (w + a) us = w + alignUp a us"
⟨*proof*⟩

lemma alignUp_distance:

"alignUp (q :: 'a :: len word) sz - q ≤ mask sz"
⟨*proof*⟩

lemma is_aligned_diff_neg_mask:
assumes "is_aligned p sz"
shows "(p - q AND NOT (mask sz)) = (p - ((alignUp q sz) AND NOT (mask sz)))"
⟨*proof*⟩

lemma word_clz_max: "word_clz w ≤ size (w::'a::len word)"
⟨*proof*⟩

lemma word_clz_nonzero_max:
fixes w :: "'a::len word"
assumes nz: "w ≠ 0"
shows "word_clz w < size (w::'a::len word)"
⟨*proof*⟩

lemma bin_sign_extend_iff [bit_simps]:
‹bit (sign_extend e w) i ⟷ bit w (min e i)›
if ‹i < LENGTH('a)› **for** w :: ‹'a::len word›
⟨*proof*⟩

lemma sign_extend_bitwise_if:
"i < size w ⟹ bit (sign_extend e w) i ⟷ (if i < e then bit w i else bit w e)"
⟨*proof*⟩

lemma sign_extend_bitwise_if' [word_eqI_simps]:
‹i < LENGTH('a) ⟹ bit (sign_extend e w) i ⟷ (if i < e then bit w i else bit w e)›
for w :: ‹'a::len word›
⟨*proof*⟩

lemma sign_extend_bitwise_disj:
"i < size w ⟹ bit (sign_extend e w) i ⟷ i ≤ e ∧ bit w i ∨ e ≤ i ∧ bit w e"
⟨*proof*⟩

lemma sign_extend_bitwise_cases:
"i < size w ⟹ bit (sign_extend e w) i ⟷ (i ≤ e ⟶ bit w i) ∧ (e ≤ i ⟶ bit w e)"
⟨*proof*⟩

lemmas sign_extend_bitwise_disj' = sign_extend_bitwise_disj[simplified word_size]
lemmas sign_extend_bitwise_cases' = sign_extend_bitwise_cases[simplified

```
word_size]
```

lemma sign_extend_def':
```
  "sign_extend n w = (if bit w n then w OR NOT (mask (Suc n)) else w AND
mask (Suc n))"
```
⟨*proof*⟩

lemma sign_extended_sign_extend:
```
  "sign_extended n (sign_extend n w)"
```
⟨*proof*⟩

lemma sign_extended_iff_sign_extend:
```
  "sign_extended n w ⟷ sign_extend n w = w"
```
⟨*proof*⟩

lemma sign_extended_weaken:
```
  "sign_extended n w ⟹ n ≤ m ⟹ sign_extended m w"
```
⟨*proof*⟩

lemma sign_extend_sign_extend_eq:
```
  "sign_extend m (sign_extend n w) = sign_extend (min m n) w"
```
⟨*proof*⟩

lemma sign_extended_high_bits:
```
  "⟦ sign_extended e p; j < size p; e ≤ i; i < j ⟧ ⟹ bit p i = bit p
j"
```
⟨*proof*⟩

lemma sign_extend_eq:
```
  "w AND mask (Suc n) = v AND mask (Suc n) ⟹ sign_extend n w = sign_extend
n v"
```
⟨*proof*⟩

lemma sign_extended_add:
```
  assumes p: "is_aligned p n"
  assumes f: "f < 2 ^ n"
  assumes e: "n ≤ e"
  assumes "sign_extended e p"
  shows "sign_extended e (p + f)"
```
⟨*proof*⟩

lemma sign_extended_neq_mask:
```
  "⟦sign_extended n ptr; m ≤ n⟧ ⟹ sign_extended n (ptr AND NOT (mask
m))"
```
⟨*proof*⟩

definition
```
  "limited_and (x :: 'a :: len word) y ⟷ (x AND y = x)"
```

lemma limited_and_eq_0:
"⟦ limited_and x z; y AND NOT z = y ⟧ ⟹ x AND y = 0"
⟨*proof*⟩

lemma limited_and_eq_id:
"⟦ limited_and x z; y AND z = z ⟧ ⟹ x AND y = x"
⟨*proof*⟩

lemma lshift_limited_and:
"limited_and x z ⟹ limited_and (x << n) (z << n)"
⟨*proof*⟩

lemma rshift_limited_and:
"limited_and x z ⟹ limited_and (x >> n) (z >> n)"
⟨*proof*⟩

lemmas limited_and_simps1 = limited_and_eq_0 limited_and_eq_id

lemmas is_aligned_limited_and
= is_aligned_neg_mask_eq[unfolded mask_eq_decr_exp, folded limited_and_def]

lemmas limited_and_simps = limited_and_simps1
limited_and_simps1[OF is_aligned_limited_and]
limited_and_simps1[OF lshift_limited_and]
limited_and_simps1[OF rshift_limited_and]
limited_and_simps1[OF rshift_limited_and, OF is_aligned_limited_and]
not_one_eq

definition
from_bool :: "bool ⇒ 'a::len word" **where**
"from_bool b ≡ case b of True ⇒ of_nat 1
| False ⇒ of_nat 0"

lemma from_bool_eq: ‹from_bool = of_bool›
⟨*proof*⟩

lemma from_bool_0: "(from_bool x = 0) = (¬ x)"
⟨*proof*⟩

lemma from_bool_eq_if':
"((if P then 1 else 0) = from_bool Q) = (P = Q)"
⟨*proof*⟩

definition
to_bool :: "'a::len word ⇒ bool" **where**
"to_bool ≡ (≠) 0"

lemma to_bool_and_1:

```
  "to_bool (x AND 1) ⟷ bit x 0"
  ⟨proof⟩

lemma to_bool_from_bool [simp]: "to_bool (from_bool r) = r"
  ⟨proof⟩

lemma from_bool_neq_0 [simp]: "(from_bool b ≠ 0) = b"
  ⟨proof⟩

lemma from_bool_mask_simp [simp]:
  "(from_bool r :: 'a::len word) AND 1 = from_bool r"
  ⟨proof⟩

lemma from_bool_1 [simp]: "(from_bool P = 1) = P"
  ⟨proof⟩

lemma ge_0_from_bool [simp]: "(0 < from_bool P) = P"
  ⟨proof⟩

lemma limited_and_from_bool:
  "limited_and (from_bool b) 1"
  ⟨proof⟩

lemma to_bool_1 [simp]: "to_bool 1" ⟨proof⟩
lemma to_bool_0 [simp]: "¬to_bool 0" ⟨proof⟩

lemma from_bool_eq_if:
  "(from_bool Q = (if P then 1 else 0)) = (P = Q)"
  ⟨proof⟩

lemma to_bool_eq_0:
  "(¬ to_bool x) = (x = 0)"
  ⟨proof⟩

lemma to_bool_neq_0:
  "(to_bool x) = (x ≠ 0)"
  ⟨proof⟩

lemma from_bool_all_helper:
  "(∀bool. from_bool bool = val ⟶ P bool)
      = ((∃bool. from_bool bool = val) ⟶ P (val ≠ 0))"
  ⟨proof⟩

lemma fold_eq_0_to_bool:
  "(v = 0) = (¬ to_bool v)"
  ⟨proof⟩

lemma from_bool_to_bool_iff:
  "w = from_bool b ⟷ to_bool w = b ∧ (w = 0 ∨ w = 1)"
```

⟨*proof*⟩

lemma from_bool_eqI:
"from_bool x = from_bool y ⟹ x = y"
⟨*proof*⟩

lemma neg_mask_in_mask_range:
assumes "is_aligned ptr bits"
shows "(ptr' AND NOT(mask bits) = ptr) = (ptr' ∈ mask_range ptr bits)"
⟨*proof*⟩

lemma aligned_offset_in_range:
assumes "is_aligned x m"
and "y < 2 ^ m"
and "is_aligned p n"
and "m ≤ n"
shows "(x + y ∈ {p .. p + mask n}) = (x ∈ mask_range p n)"
⟨*proof*⟩

lemma mask_range_to_bl':
assumes "is_aligned (ptr :: 'a :: len word) bits" "bits < LENGTH('a)"
shows "mask_range ptr bits
= {x. take (LENGTH('a) - bits) (to_bl x) = take (LENGTH('a) - bits)
(to_bl ptr)}"
⟨*proof*⟩

lemma mask_range_to_bl:
"is_aligned (ptr :: 'a :: len word) bits
⟹ mask_range ptr bits
= {x. take (LENGTH('a) - bits) (to_bl x) = take (LENGTH('a) -
bits) (to_bl ptr)}"
⟨*proof*⟩

lemma aligned_mask_range_cases:
"⟦ is_aligned (p :: 'a :: len word) n; is_aligned (p' :: 'a :: len word)
n' ⟧
⟹ mask_range p n ∩ mask_range p' n' = {} ∨
mask_range p n ⊆ mask_range p' n' ∨
mask_range p n ⊇ mask_range p' n'"
⟨*proof*⟩

lemma aligned_mask_range_offset_subset:
assumes al: "is_aligned (ptr :: 'a :: len word) sz" **and** al': "is_aligned
x sz'"
and szv: "sz' ≤ sz"
and xsz: "x < 2 ^ sz"
shows "mask_range (ptr+x) sz' ⊆ mask_range ptr sz"
⟨*proof*⟩

lemma aligned_mask_ranges_disjoint:
"⟦ is_aligned (p :: 'a :: len word) n; is_aligned (p' :: 'a :: len word) n';
p AND NOT(mask n') ≠ p'; p' AND NOT(mask n) ≠ p ⟧
⟹ mask_range p n ∩ mask_range p' n' = {}"
⟨*proof*⟩

lemma aligned_mask_ranges_disjoint2:
assumes "is_aligned p n"
and "is_aligned ptr bits"
and "m ≤ n"
and "n < size p"
and "m ≤ bits"
and "∀y<2 ^ (n - m). p + (y << m) ∉ mask_range ptr bits"
shows "mask_range p n ∩ mask_range ptr bits = {}"
⟨*proof*⟩

lemma word_clz_sint_upper[simp]:
"LENGTH('a) ≥ 3 ⟹ sint (of_nat (word_clz (w :: 'a :: len word)) :: 'a sword) ≤ int (LENGTH('a))"
⟨*proof*⟩

lemma word_clz_sint_lower[simp]:
assumes "LENGTH('a) ≥ 3"
shows "- sint (of_nat (word_clz (w :: 'a :: len word)) :: 'a signed word) ≤ int (LENGTH('a))"
⟨*proof*⟩

lemma mask_range_subsetD:
"⟦ p' ∈ mask_range p n; x' ∈ mask_range p' n'; n' ≤ n; is_aligned p n; is_aligned p' n' ⟧ ⟹
x' ∈ mask_range p n"
⟨*proof*⟩

lemma add_mult_in_mask_range:
"⟦ is_aligned (base :: 'a :: len word) n; n < LENGTH('a); bits ≤ n; x < 2 ^ (n - bits) ⟧
⟹ base + x * 2^bits ∈ mask_range base n"
⟨*proof*⟩

lemma from_to_bool_last_bit:
"from_bool (to_bool (x AND 1)) = x AND 1"
⟨*proof*⟩

lemma sint_ctz:
‹0 ≤ sint (of_nat (word_ctz (x :: 'a :: len word)) :: 'a signed word)
∧ sint (of_nat (word_ctz x) :: 'a signed word) ≤ int (LENGTH('a))›
(**is** ‹?P ∧ ?Q›)
if ‹LENGTH('a) > 2›

⟨*proof*⟩

lemma unat_of_nat_word_log2:
"LENGTH('a) < 2 ^ LENGTH('b)
⟹ unat (of_nat (word_log2 (n :: 'a :: len word)) :: 'b :: len word)
= word_log2 n"
⟨*proof*⟩

lemma aligned_mask_diff:
"⟦ is_aligned (dest :: 'a :: len word) bits; is_aligned (ptr :: 'a ::
len word) sz;
bits ≤ sz; sz < LENGTH('a); dest < ptr ⟧
⟹ mask bits + dest < ptr"
⟨*proof*⟩

lemma Suc_mask_eq_mask:
"¬bit a n ⟹ a AND mask (Suc n) = a AND mask n" **for** a::"'a::len word"
⟨*proof*⟩

lemma word_less_high_bits:
fixes a::"'a::len word"
assumes high_bits: "∀i > n. bit a i = bit b i"
assumes less: "a AND mask (Suc n) < b AND mask (Suc n)"
shows "a < b"
⟨*proof*⟩

lemma word_less_bitI:
fixes a :: "'a::len word"
assumes hi_bits: "∀i > n. bit a i = bit b i"
assumes a_bits: "¬bit a n"
assumes b_bits: "bit b n" "n < LENGTH('a)"
shows "a < b"
⟨*proof*⟩

lemma word_less_bitD:
fixes a::"'a::len word"
assumes less: "a < b"
shows "∃n. (∀i > n. bit a i = bit b i) ∧ ¬bit a n ∧ bit b n"
⟨*proof*⟩

lemma word_less_bit_eq:
"(a < b) = (∃n < LENGTH('a). (∀i > n. bit a i = bit b i) ∧ ¬bit a n
∧ bit b n)" **for** a::"'a::len word"
⟨*proof*⟩

end

end

25 Words of Length 32

theory Word_32
  imports
    Word_Lemmas
    Word_Syntax
    Word_Names
    Rsplit
    More_Word_Operations
    Bitwise
begin

context
  includes bit_operations_syntax
begin

type_synonym word32 = "32 word"
lemma len32: "len_of (x :: 32 itself) = 32" ⟨*proof*⟩

type_synonym sword32 = "32 sword"

lemma ucast_8_32_inj:
  "inj (ucast :: 8 word ⇒ 32 word)"
  ⟨*proof*⟩

lemmas unat_power_lower32' = unat_power_lower[**where** 'a=32]

lemmas word32_less_sub_le' = word_less_sub_le[**where** 'a = 32]

lemmas word32_power_less_1' = word_power_less_1[**where** 'a = 32]

lemmas unat_of_nat32' = unat_of_nat_eq[**where** 'a=32]

lemmas unat_mask_word32' = unat_mask[**where** 'a=32]

lemmas word32_minus_one_le' = word_minus_one_le[**where** 'a=32]
lemmas word32_minus_one_le = word32_minus_one_le'[simplified]

lemma unat_ucast_8_32:
  fixes x :: "8 word"
  shows "unat (ucast x :: word32) = unat x"
  ⟨*proof*⟩

lemma ucast_le_ucast_8_32:
  "(ucast x ≤ (ucast y :: word32)) = (x ≤ (y :: 8 word))"
  ⟨*proof*⟩

lemma eq_2_32_0:
  "(2 ^ 32 :: word32) = 0"

⟨*proof*⟩

lemmas mask_32_max_word = max_word_mask [symmetric, **where** 'a=32, simplified]

lemma of_nat32_n_less_equal_power_2:
"n < 32 ⟹ ((of_nat n)::32 word) < 2 ^ n"
⟨*proof*⟩

lemma unat_ucast_10_32 :
fixes x :: "10 word"
shows "unat (ucast x :: word32) = unat x"
⟨*proof*⟩

lemma word32_bounds:
"- (2 ^ (size (x :: word32) - 1)) = (-2147483648 :: int)"
"((2 ^ (size (x :: word32) - 1)) - 1) = (2147483647 :: int)"
"- (2 ^ (size (y :: 32 signed word) - 1)) = (-2147483648 :: int)"
"((2 ^ (size (y :: 32 signed word) - 1)) - 1) = (2147483647 :: int)"
⟨*proof*⟩

lemmas signed_arith_ineq_checks_to_eq_word32'
= signed_arith_ineq_checks_to_eq[**where** 'a=32]
signed_arith_ineq_checks_to_eq[**where** 'a="32 signed"]

lemmas signed_arith_ineq_checks_to_eq_word32
= signed_arith_ineq_checks_to_eq_word32' [unfolded word32_bounds]

lemmas signed_mult_eq_checks32_to_64'
= signed_mult_eq_checks_double_size[**where** 'a=32 **and** 'b=64]
signed_mult_eq_checks_double_size[**where** 'a="32 signed" **and** 'b=64]

lemmas signed_mult_eq_checks32_to_64 = signed_mult_eq_checks32_to_64'[simplified]

lemmas sdiv_word32_max' = sdiv_word_max [**where** 'a=32] sdiv_word_max
[**where** 'a="32 signed"]
lemmas sdiv_word32_max = sdiv_word32_max'[simplified word_size, simplified]

lemmas sdiv_word32_min' = sdiv_word_min [**where** 'a=32] sdiv_word_min
[**where** 'a="32 signed"]
lemmas sdiv_word32_min = sdiv_word32_min' [simplified word_size, simplified]

lemmas sint32_of_int_eq' = sint_of_int_eq [**where** 'a=32]
lemmas sint32_of_int_eq = sint32_of_int_eq' [simplified]

lemma ucast_of_nats [simp]:
"(ucast (of_nat x :: word32) :: sword32) = (of_nat x)"
"(ucast (of_nat x :: word32) :: 16 sword) = (of_nat x)"
"(ucast (of_nat x :: word32) :: 8 sword) = (of_nat x)"
"(ucast (of_nat x :: 16 word) :: 16 sword) = (of_nat x)"

```
  "(ucast (of_nat x :: 16 word) :: 8 sword) = (of_nat x)"
  "(ucast (of_nat x :: 8 word) :: 8 sword) = (of_nat x)"
  ⟨proof⟩

lemmas signed_shift_guard_simpler_32'
    = power_strict_increasing_iff[where b="2 :: nat" and y=31]
lemmas signed_shift_guard_simpler_32 = signed_shift_guard_simpler_32'[simplified]

lemma word32_31_less:
  "31 < len_of TYPE (32 signed)" "31 > (0 :: nat)"
  "31 < len_of TYPE (32)" "31 > (0 :: nat)"
  ⟨proof⟩

lemmas signed_shift_guard_to_word_32
    = signed_shift_guard_to_word[OF word32_31_less(1-2)]
    signed_shift_guard_to_word[OF word32_31_less(3-4)]

lemma has_zero_byte:
  "~~ (((((v::word32) && 0x7f7f7f7f) + 0x7f7f7f7f) || v) || 0x7f7f7f7f)
≠ 0
    ⟹ v && 0xff000000 = 0 ∨ v && 0xff0000 = 0 ∨ v && 0xff00 = 0 ∨ v
&& 0xff = 0"
  ⟨proof⟩

lemma mask_step_down_32:
  ‹∃x. mask x = b› if ‹b && 1 = 1›
    and ‹∃x. x < 32 ∧ mask x = b >> 1› for b :: ‹32word›
⟨proof⟩

lemma unat_of_int_32:
  "⟦i ≥ 0; i ≤2 ^ 31⟧ ⟹ (unat ((of_int i)::sword32)) = nat i"
  ⟨proof⟩

lemmas word_ctz_not_minus_1_32 = word_ctz_not_minus_1[where 'a=32, simplified]


lemma cast_chunk_assemble_id_64[simp]:
  "(((ucast ((ucast (x::64 word))::32 word))::64 word) || (((ucast ((ucast
(x >> 32))::32 word))::64 word) << 32)) = x"
  ⟨proof⟩


lemma cast_chunk_assemble_id_64'[simp]:
  "(((ucast ((scast (x::64 word))::32 word))::64 word) || (((ucast ((scast
(x >> 32))::32 word))::64 word) << 32)) = x"
  ⟨proof⟩


lemma cast_down_u64: "(scast::64 word ⇒ 32 word) = (ucast::64 word ⇒
```

```
32 word)"
```
⟨*proof*⟩

lemma `cast_down_s64: "(scast::64 sword ⇒ 32 word) = (ucast::64 sword ⇒ 32 word)"`
⟨*proof*⟩

lemma `word32_and_max_simp:`
`‹x AND 0xFFFFFFFF = x›` **for** `x :: ‹32 word›`
⟨*proof*⟩

end

end

26 Ancient comprehensive Word Library

theory `Word_Lib_Sumo`
imports
`"HOL-Library.Word"`
`Aligned`
`Bit_Comprehension`
`Bit_Comprehension_Int`
`Bit_Shifts_Infix_Syntax`
`Bitwise_Signed`
`Bitwise`
`Enumeration_Word`
`Generic_set_bit`
`Hex_Words`
`Least_significant_bit`
`More_Arithmetic`
`More_Divides`
`More_Sublist`
`More_Int`
`Bin_sign`
`Even_More_List`
`More_Misc`
`Legacy_Aliases`
`Most_significant_bit`
`Next_and_Prev`
`Norm_Words`
`Reversed_Bit_Lists`
`Rsplit`
`Signed_Words`
`Syntax_Bundles`
`Sgn_Abs`
`Typedef_Morphisms`
`Type_Syntax`
`Word_EqI`

```
  Word_Lemmas
  Word_8
  Word_16
  Word_32
  Word_Syntax
  Signed_Division_Word
  Singleton_Bit_Shifts
  More_Word_Operations
  Many_More
begin

unbundle bit_operations_syntax
unbundle bit_projection_infix_syntax

declare word_induct2[induct type]
declare word_nat_cases[cases type]

declare signed_take_bit_Suc [simp]


lemmas of_int_and_nat = unsigned_of_nat unsigned_of_int signed_of_int
signed_of_nat

bundle no_take_bit
begin
declare of_int_and_nat[simp del]
end

lemmas bshiftr1_def = bshiftr1_eq
lemmas is_down_def = is_down_eq
lemmas is_up_def = is_up_eq
lemmas mask_def = mask_eq
lemmas scast_def = scast_eq
lemmas shiftl1_def = shiftl1_eq
lemmas shiftr1_def = shiftr1_eq
lemmas sshiftr1_def = sshiftr1_eq
lemmas sshiftr_def = sshiftr_eq_funpow_sshiftr1
lemmas to_bl_def = to_bl_eq
lemmas ucast_def = ucast_eq
lemmas unat_def = unat_eq_nat_uint
lemmas word_cat_def = word_cat_eq
lemmas word_reverse_def = word_reverse_eq_of_bl_rev_to_bl
lemmas word_roti_def = word_roti_eq_word_rotr_word_rotl
lemmas word_rotl_def = word_rotl_eq
lemmas word_rotr_def = word_rotr_eq
lemmas word_sle_def = word_sle_eq
lemmas word_sless_def = word_sless_eq

lemmas uint_0 = uint_nonnegative
```

```
lemmas uint_lt = uint_bounded
lemmas uint_mod_same = uint_idem
lemmas of_nth_def = word_set_bits_def

lemmas of_nat_word_eq_iff = word_of_nat_eq_iff
lemmas of_nat_word_eq_0_iff = word_of_nat_eq_0_iff
lemmas of_int_word_eq_iff = word_of_int_eq_iff
lemmas of_int_word_eq_0_iff = word_of_int_eq_0_iff

lemmas word_next_def = word_next_unfold

lemmas word_prev_def = word_prev_unfold

lemmas is_aligned_def = is_aligned_iff_dvd_nat

lemmas word_and_max_simps =
  word8_and_max_simp
  word16_and_max_simp
  word32_and_max_simp

lemma distinct_lemma: "f x ≠ f y ⟹ x ≠ y" ⟨proof⟩

lemmas and_bang = word_and_nth

lemmas sdiv_int_def = signed_divide_int_def
lemmas smod_int_def = signed_modulo_int_def


lemma word_fixed_sint_1[simp]:
  "sint (1::8 word) = 1"
  "sint (1::16 word) = 1"
  "sint (1::32 word) = 1"
  "sint (1::64 word) = 1"
  ⟨proof⟩

declare of_nat_diff [simp]


notation (input)
  bit (<testBit>)

lemmas cast_simps = cast_simps ucast_down_bl

end
```

27 32 bit standard platform-specific word size and alignment.

theory Machine_Word_32_Basics
imports "HOL-Library.Word" Word_32
begin

type_synonym machine_word_len = 32

definition word_bits :: nat
where
  ‹word_bits = LENGTH(machine_word_len)›

lemma word_bits_conv [code]:
  ‹word_bits = 32›
  ⟨*proof*⟩

The following two are numerals so they can be used as nats and words.

definition word_size_bits :: ‹'a :: numeral›
where
  ‹word_size_bits = 2›

definition word_size :: ‹'a :: numeral›
where
  ‹word_size = 4›

lemma n_less_word_bits:
  "(n < word_bits) = (n < 32)"
  ⟨*proof*⟩

lemmas upper_bits_unset_is_l2p_32 = upper_bits_unset_is_l2p [**where** 'a=32, folded word_bits_def]

lemmas le_2p_upper_bits_32 = le_2p_upper_bits [**where** 'a=32, folded word_bits_def]
lemmas le2p_bits_unset_32 = le2p_bits_unset[**where** 'a=32, folded word_bits_def]

lemmas unat_power_lower32 [simp] = unat_power_lower32'[folded word_bits_def]

lemmas word32_less_sub_le[simp] = word32_less_sub_le' [folded word_bits_def]

lemmas word32_power_less_1[simp] = word32_power_less_1'[folded word_bits_def]

lemma of_nat32_0:
  "⟦of_nat n = (0::word32); n < 2 ^ word_bits⟧ ⟹ n = 0"
  ⟨*proof*⟩

lemmas unat_of_nat32 = unat_of_nat32'[folded word_bits_def]

lemmas word_power_nonzero_32 = word_power_nonzero [**where** 'a=32, folded

word_bits_def]

lemmas div_power_helper_32 = div_power_helper [**where** 'a=32, folded word_bits_def]

lemmas of_nat_less_pow_32 = of_nat_power [**where** 'a=32, folded word_bits_def]

lemmas unat_mask_word32 = unat_mask_word32'[folded word_bits_def]

end

28 32-Bit Machine Word Setup

theory Machine_Word_32
  imports Machine_Word_32_Basics More_Word Bit_Shifts_Infix_Syntax Rsplit
begin

context
  includes bit_operations_syntax
begin

type_synonym machine_word = ‹machine_word_len word›

lemma word_bits_len_of:
  ‹LENGTH(machine_word_len) = word_bits›
  ⟨*proof*⟩

lemma word_bits_size:
  "size (w :: machine_word) = word_bits"
  ⟨*proof*⟩

lemma word_bits_word_size_conv:
  ‹word_bits = word_size * 8›
  ⟨*proof*⟩

lemma word_size_word_size_bits:
  ‹word_size = (2 :: 'a :: semiring_1) ^ word_size_bits›
  ⟨*proof*⟩

lemma lt_word_bits_lt_pow:
  "sz < word_bits ⟹ sz < 2 ^ word_bits"
  ⟨*proof*⟩

lemma if_then_1_else_0:
  "((if P then 1 else 0) = (0 :: machine_word)) = (¬ P)"
  ⟨*proof*⟩

lemma if_then_0_else_1:
  "((if P then 0 else 1) = (0 :: machine_word)) = (P)"
  ⟨*proof*⟩

lemmas if_then_simps = if_then_0_else_1 if_then_1_else_0

lemma bool_mask [simp]:
  ‹0 < x AND 1 ⟷ x AND 1 = 1› **for** x :: machine_word
  ⟨*proof*⟩

lemma in_16_range:
  "0 ∈ S ⟹ r ∈ (λx. r + x * (16 :: machine_word)) ‘ S"
  "n - 1 ∈ S ⟹ (r + (16 * n - 16)) ∈ (λx :: machine_word. r + x * 16)
‘ S"
  ⟨*proof*⟩

lemma le_step_down_word_3:
  fixes x :: machine_word
  shows "⟦x ≤ y; x ≠ y⟧ ⟹ x ≤ y - 1"
  ⟨*proof*⟩

lemma shiftr_1:
  "(x::machine_word) >> 1 = 0 ⟹ x < 2"
  ⟨*proof*⟩

lemma Suc_unat_mask_div:
  "Suc (unat (mask sz div word_size :: machine_word)) = 2 ^ (min sz word_bits
- word_size_bits)"
  ⟨*proof*⟩

lemma ucast_not_helper:
  fixes a::"8 word"
  assumes a: "a ≠ 0xFF"
  shows "ucast a ≠ (0xFF::machine_word)"
⟨*proof*⟩

lemma unat_less_2p_word_bits:
  "unat (x :: machine_word) < 2 ^ word_bits"
  ⟨*proof*⟩

lemma unat_less_word_bits:
  fixes y :: machine_word
  shows "x < unat y ⟹ x < 2 ^ word_bits"
  ⟨*proof*⟩

lemma unat_mask_2_less_4:
  "unat (p AND mask 2 :: machine_word) < 4"
  ⟨*proof*⟩

lemma unat_mult_simple:
  ‹unat (x * y) = unat x * unat y›
    if ‹unat x * unat y < 2 ^ LENGTH(machine_word_len)›

for x y :: machine_word
⟨*proof*⟩

lemma upto_2_helper:
"{0..<2 :: machine_word} = {0, 1}"
⟨*proof*⟩

lemma word_ge_min:
‹- (2 ^ (word_bits - 1)) ≤ sint x› **for** x :: machine_word
⟨*proof*⟩

lemma word_rsplit_0:
"word_rsplit (0 :: machine_word) = replicate (word_bits div 8) (0 ::
8 word)"
⟨*proof*⟩

lemma x_less_2_0_1:
fixes x :: machine_word
shows "x < 2 ⟹ x = 0 ∨ x = 1"
⟨*proof*⟩

end

end

29 Words of Length 64

theory Word_64
imports
Word_Lemmas
Word_Names
Word_Syntax
Rsplit
More_Word_Operations
begin

context
includes bit_operations_syntax
begin

lemma len64: "len_of (x :: 64 itself) = 64" ⟨*proof*⟩

lemma ucast_8_64_inj:
"inj (ucast :: 8 word ⇒ 64 word)"
⟨*proof*⟩

lemmas unat_power_lower64' = unat_power_lower[**where** 'a=64]

lemmas word64_less_sub_le' = word_less_sub_le[**where** 'a = 64]

```
lemmas word64_power_less_1' = word_power_less_1[where 'a = 64]

lemmas unat_of_nat64' = unat_of_nat_eq[where 'a=64]

lemmas unat_mask_word64' = unat_mask[where 'a=64]

lemmas word64_minus_one_le' = word_minus_one_le[where 'a=64]
lemmas word64_minus_one_le = word64_minus_one_le'[simplified]

lemma less_4_cases:
  "(x::word64) < 4 ⟹ x=0 ∨ x=1 ∨ x=2 ∨ x=3"
  ⟨proof⟩

lemma ucast_le_ucast_8_64:
  "(ucast x ≤ (ucast y :: word64)) = (x ≤ (y :: 8 word))"
  ⟨proof⟩

lemma eq_2_64_0:
  "(2 ^ 64 :: word64) = 0"
  ⟨proof⟩

lemmas mask_64_max_word  = max_word_mask [symmetric, where 'a=64, simplified]

lemma of_nat64_n_less_equal_power_2:
 "n < 64 ⟹ ((of_nat n)::64 word) < 2 ^ n"
  ⟨proof⟩

lemma unat_ucast_10_64 :
  fixes x :: "10 word"
  shows "unat (ucast x :: word64) = unat x"
  ⟨proof⟩

lemma word64_bounds:
  "- (2 ^ (size (x :: word64) - 1)) = (-9223372036854775808 :: int)"
  "((2 ^ (size (x :: word64) - 1)) - 1) = (9223372036854775807 :: int)"
  "- (2 ^ (size (y :: 64 signed word) - 1)) = (-9223372036854775808 ::
int)"
  "((2 ^ (size (y :: 64 signed word) - 1)) - 1) = (9223372036854775807
:: int)"
  ⟨proof⟩

lemmas signed_arith_ineq_checks_to_eq_word64'
    = signed_arith_ineq_checks_to_eq[where 'a=64]
      signed_arith_ineq_checks_to_eq[where 'a="64 signed"]

lemmas signed_arith_ineq_checks_to_eq_word64
    = signed_arith_ineq_checks_to_eq_word64' [unfolded word64_bounds]
```

```
lemmas signed_mult_eq_checks64_to_64'
    = signed_mult_eq_checks_double_size[where 'a=64 and 'b=64]
      signed_mult_eq_checks_double_size[where 'a="64 signed" and 'b=64]

lemmas signed_mult_eq_checks64_to_64 = signed_mult_eq_checks64_to_64'[simplified]

lemmas sdiv_word64_max' = sdiv_word_max [where 'a=64] sdiv_word_max
[where 'a="64 signed"]
lemmas sdiv_word64_max = sdiv_word64_max'[simplified word_size, simplified]

lemmas sdiv_word64_min' = sdiv_word_min [where 'a=64] sdiv_word_min
[where 'a="64 signed"]
lemmas sdiv_word64_min = sdiv_word64_min' [simplified word_size, simplified]

lemmas sint64_of_int_eq' = sint_of_int_eq [where 'a=64]
lemmas sint64_of_int_eq = sint64_of_int_eq' [simplified]

lemma ucast_of_nats [simp]:
  "(ucast (of_nat x :: word64) :: sword64) = (of_nat x)"
  "(ucast (of_nat x :: word64) :: 16 sword) = (of_nat x)"
  "(ucast (of_nat x :: word64) :: 8 sword) = (of_nat x)"
  ⟨proof⟩

lemmas signed_shift_guard_simpler_64'
    = power_strict_increasing_iff[where b="2 :: nat" and y=31]
lemmas signed_shift_guard_simpler_64 = signed_shift_guard_simpler_64'[simplified]

lemma word64_31_less:
  "31 < len_of TYPE (64 signed)" "31 > (0 :: nat)"
  "31 < len_of TYPE (64)" "31 > (0 :: nat)"
  ⟨proof⟩

lemmas signed_shift_guard_to_word_64
    = signed_shift_guard_to_word[OF word64_31_less(1-2)]
    signed_shift_guard_to_word[OF word64_31_less(3-4)]

lemma mask_step_down_64:
  ‹∃x. mask x = b› if ‹b && 1 = 1›
    and ‹∃x. x < 64 ∧ mask x = b >> 1› for b :: ‹64word›
⟨proof⟩

lemma unat_of_int_64:
  "⟦i ≥ 0; i ≤ 2 ^ 63⟧ ⟹ (unat ((of_int i)::sword64)) = nat i"
  ⟨proof⟩

lemmas word_ctz_not_minus_1_64 = word_ctz_not_minus_1[where 'a=64, simplified]

lemma word64_and_max_simp:
  ‹x AND 0xFFFFFFFFFFFFFFFF = x› for x :: ‹64 word›
```

⟨*proof*⟩

end

end

30 64 bit standard platform-specific word size and alignment.

theory Machine_Word_64_Basics
imports "HOL-Library.Word" Word_64
begin

type_synonym machine_word_len = 64

definition word_bits :: nat
where
  ‹word_bits = LENGTH(machine_word_len)›

lemma word_bits_conv [code]:
  ‹word_bits = 64›
  ⟨*proof*⟩

The following two are numerals so they can be used as nats and words.

definition word_size_bits :: ‹'a :: numeral›
where
  ‹word_size_bits = 3›

definition word_size :: ‹'a :: numeral›
where
  ‹word_size = 8›

lemma n_less_word_bits:
  "(n < word_bits) = (n < 64)"
  ⟨*proof*⟩

lemmas upper_bits_unset_is_l2p_64 = upper_bits_unset_is_l2p [**where** 'a=64, folded word_bits_def]

lemmas le_2p_upper_bits_64 = le_2p_upper_bits [**where** 'a=64, folded word_bits_def]
lemmas le2p_bits_unset_64 = le2p_bits_unset[**where** 'a=64, folded word_bits_def]

lemmas unat_power_lower64 [simp] = unat_power_lower64'[folded word_bits_def]

lemmas word64_less_sub_le[simp] = word64_less_sub_le' [folded word_bits_def]

lemmas word64_power_less_1[simp] = word64_power_less_1'[folded word_bits_def]

lemma of_nat64_0:
"⟦of_nat n = (0::word64); n < 2 ^ word_bits⟧ ⟹ n = 0"
⟨*proof*⟩

lemmas unat_of_nat64 = unat_of_nat64'[folded word_bits_def]

lemmas word_power_nonzero_64 = word_power_nonzero [**where** 'a=64, folded word_bits_def]

lemmas div_power_helper_64 = div_power_helper [**where** 'a=64, folded word_bits_def]

lemmas of_nat_less_pow_64 = of_nat_power [**where** 'a=64, folded word_bits_def]

lemmas unat_mask_word64 = unat_mask_word64'[folded word_bits_def]

end

31 64-Bit Machine Word Setup

theory Machine_Word_64
imports Machine_Word_64_Basics More_Word Bit_Shifts_Infix_Syntax Rsplit
begin

context
includes bit_operations_syntax
begin

type_synonym machine_word = ‹machine_word_len word›

lemma word_bits_len_of:
‹LENGTH(machine_word_len) = word_bits›
⟨*proof*⟩

lemma word_bits_size:
"size (w :: machine_word) = word_bits"
⟨*proof*⟩

lemma word_bits_word_size_conv:
‹word_bits = word_size * 8›
⟨*proof*⟩

lemma word_size_word_size_bits:
‹word_size = (2 :: 'a :: semiring_1) ^ word_size_bits›
⟨*proof*⟩

lemma lt_word_bits_lt_pow:
"sz < word_bits ⟹ sz < 2 ^ word_bits"
⟨*proof*⟩

lemma if_then_1_else_0:
"((if P then 1 else 0) = (0 :: machine_word)) = (¬ P)"
⟨*proof*⟩

lemma if_then_0_else_1:
"((if P then 0 else 1) = (0 :: machine_word)) = (P)"
⟨*proof*⟩

lemmas if_then_simps = if_then_0_else_1 if_then_1_else_0

lemma bool_mask [simp]:
‹0 < x AND 1 ⟷ x AND 1 = 1› **for** x :: machine_word
⟨*proof*⟩

lemma in_16_range:
"0 ∈ S ⟹ r ∈ (λx. r + x * (16 :: machine_word)) ‘ S"
"n - 1 ∈ S ⟹ (r + (16 * n - 16)) ∈ (λx :: machine_word. r + x * 16)
‘ S"
⟨*proof*⟩

lemma le_step_down_word_3:
fixes x :: machine_word
shows "⟦x ≤ y; x ≠ y⟧ ⟹ x ≤ y - 1"
⟨*proof*⟩

lemma shiftr_1:
"(x::machine_word) >> 1 = 0 ⟹ x < 2"
⟨*proof*⟩

lemma Suc_unat_mask_div:
"Suc (unat (mask sz div word_size :: machine_word)) = 2 ^ (min sz word_bits
- word_size_bits)"
⟨*proof*⟩

lemma ucast_not_helper:
fixes a::"8 word"
assumes a: "a ≠ 0xFF"
shows "ucast a ≠ (0xFF::machine_word)"
⟨*proof*⟩

lemma unat_less_2p_word_bits:
"unat (x :: machine_word) < 2 ^ word_bits"
⟨*proof*⟩

lemma unat_less_word_bits:
fixes y :: machine_word
shows "x < unat y ⟹ x < 2 ^ word_bits"
⟨*proof*⟩

lemma unat_mask_2_less_4:
"unat (p AND mask 2 :: machine_word) < 4"
⟨*proof*⟩

lemma unat_mult_simple:
‹unat (x * y) = unat x * unat y›
if ‹unat x * unat y < 2 ^ LENGTH(machine_word_len)›
for x y :: machine_word
⟨*proof*⟩

lemma upto_2_helper:
"{0..<2 :: machine_word} = {0, 1}"
⟨*proof*⟩

lemma word_ge_min:
‹- (2 ^ (word_bits - 1)) ≤ sint x› **for** x :: machine_word
⟨*proof*⟩

lemma word_rsplit_0:
"word_rsplit (0 :: machine_word) = replicate (word_bits div 8) (0 ::
8 word)"
⟨*proof*⟩

lemma x_less_2_0_1:
fixes x :: machine_word
shows "x < 2 ⟹ x = 0 ∨ x = 1"
⟨*proof*⟩

end

end

⟨*proof*⟩

32 A short overview over bit operations and word types

32.1 Key principles

When formalizing bit operations, it is tempting to represent bit values as explicit lists over a binary type. This however is a bad idea, mainly due to the inherent ambiguities in representation concerning repeating leading bits.

Hence this approach avoids such explicit lists altogether following an algebraic path:

- Bit values are represented by numeric types: idealized unbounded bit

values can be represented by type `int`, bounded bit values by quotient types over `int`, aka `'a word`.

- (A special case are idealized unbounded bit values ending in `0` which can be represented by type `nat` but only support a restricted set of operations).

The fundamental principles are developed in theory `HOL.Bit_Operations` (which is part of `Main`):

- Multiplication by `2` is a bit shift to the left and
- Division by `2` is a bit shift to the right.
- Concerning bounded bit values, iterated shifts to the left may result in eliminating all bits by shifting them all beyond the boundary. The property $2^{n} \neq 0$ represents that `n` is *not* beyond that boundary.
- The projection on a single bit is then `bit a n` $\longleftrightarrow$ `odd (a div` 2^{n}`)`.
- This leads to the most fundamental properties of bit values:
  - Equality rule:

    ```
    a = b ⟷ (∀n. 2^n ≠ 0 ⟶ bit a n ⟷ bit b n)
    ```

  - Induction rule:

    ```
    ⟦⋀a. a div 2 = a ⟹ P a;
     ⋀a b. ⟦P a; (of_bool b + 2 * a) div 2 = a⟧ ⟹ P (of_bool b
    + 2 * a)⟧
    ⟹ P a
    ```

- Characteristic properties `bit (f x) n` $\longleftrightarrow$ `P x n` are available in fact collection `bit_simps`.

On top of this, the following generic operations are provided:

- Singleton `n`th bit: 2^{n}
- Bit mask upto bit `n`: `mask n =` 2^{n} `- 1`
- Left shift: `push_bit n a = a *` 2^{n}
- Right shift: `drop_bit n a = a div` 2^{n}
- Truncation: `take_bit n a = a mod` 2^{n}
- Bitwise negation: `bit (NOT a) n` $\longleftrightarrow$ $2^{n} \neq 0 \wedge \neg$ `bit a n`

- Bitwise conjunction: `bit (a AND b) n` ⟷ `bit a n` ∧ `bit b n`
- Bitwise disjunction: `bit (a OR b) n` ⟷ `bit a n` ∨ `bit b n`
- Bitwise exclusive disjunction: `bit (a XOR b) n` ⟷ `bit a n` ≠ `bit b n`
- Setting a single bit: `semiring_bit_operations_class.set_bit n a = a OR push_bit n 1`
- Unsetting a single bit: `unset_bit n a = a AND NOT (push_bit n 1)`
- Flipping a single bit: `flip_bit n a = a XOR push_bit n 1`
- Signed truncation, or modulus centered around `0`:

  ```
  signed_take_bit n a = take_bit n a OR of_bool (bit a n) * NOT (mask n)
  ```

- (Bounded) conversion from and to a list of bits:

  ```
  horner_sum of_bool 2 (map (bit a) [0..<n]) = take_bit n a
  ```

Bit concatenation on `int` as given by

```
concat_bit n k l = take_bit n k OR push_bit n l
```

appears quite technical but is the logical foundation for the quite natural bit concatenation on `'a word` (see below).

32.2 Core word theory

Proper word types are introduced in theory `HOL-Library.Word`, with the following specific operations:

- Standard arithmetic: `(+)`, `uminus`, `(-)`, `(*)`, `0`, `1`, numerals etc.
- Standard bit operations: see above.
- Conversion with unsigned interpretation of words:
  - `unsigned :: 'a::len word ⇒ 'b::semiring_1`
  - Important special cases as abbreviations:
    - `unat :: 'a::len word ⇒ nat`
    - `uint :: 'a::len word ⇒ int`
    - `ucast :: 'a::len word ⇒ 'b::len word`

- Conversion with signed interpretation of words:
  - `signed :: 'a::len word ⇒ 'b::ring_1`
  - Important special cases as abbreviations:
    - `sint :: 'a::len word ⇒ int`
    - `scast :: 'a::len word ⇒ 'b::len word`
- Operations with unsigned interpretation of words:
  - `a ≤ b ⟷ unat a ≤ unat b`
  - `a < b ⟷ unat a < unat b`
  - `unat (v div w) = unat v div unat w`
  - `unat (drop_bit n w) = drop_bit n (unat w)`
  - `unat (v mod w) = unat v mod unat w`
  - `x udvd y ⟷ unat x dvd unat y`
- Operations with signed interpretation of words:
  - `a ≤s b ⟷ sint a ≤ sint b`
  - `a <s b ⟷ sint a < sint b`
  - `sint (signed_drop_bit n w) = drop_bit n (sint w)`
- Rotation and reversal:
  - `word_rotl :: nat ⇒ 'a::len word ⇒ 'a word`
  - `word_rotr :: nat ⇒ 'a::len word ⇒ 'a word`
  - `word_roti :: int ⇒ 'a::len word ⇒ 'a word`
  - `word_reverse :: 'a::len word ⇒ 'a word`
- Concatenation:

  `word_cat :: 'a::len word ⇒ 'b::len word ⇒ 'c::len word`

For proofs about words the following default strategies are applicable:

- Using bit extensionality (facts `bit_eq_iff`, `bit_word_eqI`; fact collection `bit_simps`).
- Using the `transfer` method.

32.3 More library theories

Note: currently, most theories listed here are hardly separate entities since they import each other in various ways. Always inspect them to understand what you pull in if you want to import one.

Syntax `Word_Lib.Syntax_Bundles` Bundles to provide alternative syntax for various bit operations.

`Word_Lib.Hex_Words` Printing word numerals as hexadecimal numerals.

`Word_Lib.Type_Syntax` Pretty type-sensitive syntax for cast operations.

`Word_Lib.Word_Syntax` Specific ASCII syntax for prominent bit operations on word.

Proof tools `Word_Lib.Norm_Words` Rewriting word numerals to normal forms.

`Word_Lib.Bitwise` Method `word_bitwise` decomposes word equalities and inequalities into bit propositions.

`Word_Lib.Bitwise_Signed` Method `word_bitwise_signed` decomposes word equalities and inequalities into bit propositions.

`Word_Lib.Word_EqI` Method `word_eqI_solve` decomposes word equalities and inequalities into bit propositions.

Operations `Word_Lib.Signed_Division_Word` Signed division on word:

- `(sdiv) :: 'a::len word ⇒ 'a word ⇒ 'a word`
- `(smod) :: 'a::len word ⇒ 'a word ⇒ 'a word`

`Word_Lib.Aligned`

- `is_aligned w n ⟷ 2^n udvd w`

`Word_Lib.Least_significant_bit` The least significant bit as abbreviation `lsb ≡ λa. bit a 0`.

`Word_Lib.Most_significant_bit` The most significant bit:

- `msb k ⟷ k < 0`
- `msb w ⟷ sint w < 0`
- `msb w ⟷ w <s 0`
- `msb w ⟷ bit w (LENGTH('a) - Suc 0)`

`Word_Lib.Bit_Shifts_Infix_Syntax` Bit shifts decorated with infix syntax:

- `a << n = push_bit n a`
- `a >> n = drop_bit n a`
- `w >>> n = signed_drop_bit n w`

`Word_Lib.Next_and_Prev`

- `word_next w = (if w = - 1 then - 1 else w + 1)`
- `word_prev w = (if w = 0 then 0 else w - 1)`

`Word_Lib.Enumeration_Word` More on explicit enumeration of word types.

`Word_Lib.More_Word_Operations` Even more operations on word.

Types `Word_Lib.Signed_Words` Formal tagging of word types with a `signed` marker.

Lemmas `Word_Lib.More_Word` More lemmas on words.

`Word_Lib.Word_Lemmas` More lemmas on words, covering many other theories mentioned here.

Words of popular lengths .

`Word_Lib.Word_8` for 8-bit words.

`Word_Lib.Word_16` for 16-bit words.

`Word_Lib.Word_32` for 32-bit words.

`Word_Lib.Word_64` for 64-bit words. This theory is not part of `Word_Lib_Sumo`, because it shadows names from `Word_Lib.Word_32`. They can be used together, but then will have to use qualified names in applications.

`Word_Lib.Machine_Word_32` **and** `Word_Lib.Machine_Word_64` provide lemmas for 32-bit words and 64-bit words under the same name, which can help to organize applications relying on some form of genericity.

32.4 More library sessions

`Native_Word` Makes machine words and machine arithmetic available for code generation. It provides a common abstraction that hides the differences between the different target languages. The code generator maps these operations to the APIs of the target languages.

32.5 Legacy theories

The following theories contain material which has been factored out since it is not recommended to use it in new applications, mostly because matters can be expressed succinctly using already existing operations.

This section gives some indication how to migrate away from those theories. However theorem coverage may still be terse in some cases.

`Word_Lib.Word_Lib_Sumo` An entry point importing any relevant theory in that session. Intended for backward compatibility: start importing this theory when migrating applications to Isabelle2021, and later sort out what you really need. You may need to include `Word_Lib.Word_64` separately.

`Word_Lib.Generic_set_bit` A variant of a singleton bit operation: `Generic_set_bit.set_bit a n b = (if b then semiring_bit_operations_class.set_bit else unset_bit) n a`

`Word_Lib.Typedef_Morphisms` A low-level extension to HOL typedef providing conversions along type morphisms. The `transfer` method seems to be sufficient for most applications though.

`Word_Lib.Bit_Comprehension` Comprehension syntax for bit values over predicates `nat ⇒ bool`, for `'a word`; straightforward alternatives exist.

`Word_Lib.Bit_Comprehension_Int` Comprehension syntax for bit values over predicates `nat ⇒ bool`, for `int`; inherently non-computational.

`Word_Lib.Reversed_Bit_Lists` Representation of bit values as explicit list in *reversed* order.

This should rarely be necessary: the `bit` projection should be sufficient in most cases. In case explicit lists are needed, existing operations can be used:

```
horner_sum of_bool 2 (map (bit a) [0..<n]) = take_bit n a
```

`Word_Lib.Many_More` Collection of operations and theorems which are kept for backward compatibility and not used in other theories in session `Word_Lib`. They are used in applications of `Word_Lib`, but should be migrated to there.

33 Changelog

Changes since AFP 2025

- `Generic_set_bit.set_bit` is now a regular derived operation without any special treatment.

Changes since AFP 2024

- Theory `Strict_part_mono` is not part of text`Word_Lib_Sumo` any longer.

- Session Native_Word: Fact aliases word_sdiv_def and word_smod_def are gone, use sdiv_word_def and smod_word_def instead.
- Session Native_Word: Removed abbreviation word_of_integer.

Changes since AFP 2022

- Theory Word_Lib.Ancient_Numeral has been removed from session.
- Bit comprehension syntax for int moved to separate theory Word_Lib.Bit_Comprehension_In
- Operation lsb ≡ λa. bit a 0 turned into abbreviation or bit _ 0.

Changes since AFP 2021

- Theory Word_Lib.Ancient_Numeral is not part of Word_Lib.Word_Lib_Sumo any longer.
- Infix syntax for (AND), (OR), (XOR) organized in syntax bundle bit_operations_syntax.
- Abbreviation max_word ≡ - 1 moved from distribution into theory Word_Lib.Legacy_Aliases.
- Operation test_bit replaced by input abbreviation test_bit ≡ bit.
- Abbreviations bin_nth ≡ bit, bin_last ≡ odd, bin_rest ≡ λw. w div 2, bintrunc ≡ take_bit, sbintrunc ≡ signed_take_bit, norm_sint ≡ λn. signed_take_bit (n - 1), bin_cat ≡ λk n l. concat_bit n l k moved into theory Word_Lib.Legacy_Aliases.
- Operations bshiftr1 ≡ λb w. w div 2 OR push_bit (LENGTH('a) - Suc 0) (of_bool b), setBit ≡ λw n. semiring_bit_operations_class.set_bit n w, clearBit ≡ λw n. unset_bit n w moved from distribution into theory Word_Lib.Legacy_Aliases and replaced by input abbreviations.
- Operations shiftl1, shiftr1, sshiftr1 moved here from distribution.
- Operation complement replaced by input abbreviation complement ≡ NOT.

References

[1] D. Leijen. Division and modulus for computer scientists. 2001.