Finite Machine Word Library

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

February 23, 2021

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 Arithmetic lemmas

theory More_Arithmetic
  imports Main "HOL-Library.Type_Length" "HOL-Library.Bit_Operations"
begin

declare iszero_0 [intro]

declare min.absorb1 [simp] min.absorb2 [simp]

lemma n_less_equal_power_2 [simp]:
  "n < 2 ^ n"
  ⟨*proof*⟩

lemma min_pm [simp]: "min a b + (a - b) = a"
  for a b :: nat
  ⟨*proof*⟩

lemma min_pm1 [simp]: "a - b + min a b = a"
  for a b :: nat
  ⟨*proof*⟩

lemma rev_min_pm [simp]: "min b a + (a - b) = a"
  for a b :: nat
  ⟨*proof*⟩

lemma rev_min_pm1 [simp]: "a - b + min b a = a"
  for a b :: nat
  ⟨*proof*⟩

lemma min_minus [simp]: "min m (m - k) = m - k"
  for m k :: nat
  ⟨*proof*⟩

lemma min_minus' [simp]: "min (m - k) m = m - k"
  for m k :: nat
  ⟨*proof*⟩

lemma nat_less_power_trans:
  fixes n :: nat
  assumes nv: "n < 2 ^ (m - k)"
  and kv: "k ≤ m"
  shows "2 ^ k * n < 2 ^ m"
⟨*proof*⟩

lemma nat_le_power_trans:
  fixes n :: nat
  shows "⟦n ≤ 2 ^ (m - k); k ≤ m⟧ ⟹ 2 ^ k * n ≤ 2 ^ m"
  ⟨*proof*⟩

lemma nat_add_offset_less:
  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 nat_power_less_diff:

assumes lt: "(2::nat) ^ n * q < 2 ^ m"
shows "q < 2 ^ (m - n)"
⟨*proof*⟩

lemma power_2_mult_step_le:
"⟦n' ≤ n; 2 ^ n' * k' < 2 ^ n * k⟧ ⟹ 2 ^ n' * (k' + 1) ≤ 2 ^ n * (k::nat)"
⟨*proof*⟩

lemma nat_mult_power_less_eq:
"b > 0 ⟹ (a * b ^ n < (b :: nat) ^ m) = (a < b ^ (m - n))"
⟨*proof*⟩

lemma diff_diff_less:
"(i < m - (m - (n :: nat))) = (i < m ∧ i < n)"
⟨*proof*⟩

lemma small_powers_of_2:
‹x < 2 ^ (x - 1)› **if** ‹x ≥ 3› **for** x :: nat
⟨*proof*⟩

end

2 Lemmas on division

theory More_Divides
imports
"HOL-Library.Word"
begin

declare div_eq_dividend_iff [simp]

lemma int_div_same_is_1 [simp]:
‹a div b = a ⟷ b = 1› **if** ‹0 < a› **for** a b :: int
⟨*proof*⟩

lemma int_div_minus_is_minus1 [simp]:
‹a div b = - a ⟷ b = - 1› **if** ‹0 > a› **for** a b :: int
⟨*proof*⟩

lemma nat_div_eq_Suc_0_iff: "n div m = Suc 0 ⟷ m ≤ n ∧ n < 2 * m"
⟨*proof*⟩

lemma diff_mod_le:
‹a - a mod b ≤ d - b› **if** ‹a < d› ‹b dvd d› **for** a b d :: nat
⟨*proof*⟩

lemma one_mod_exp_eq_one [simp]:
"1 mod (2 * 2 ^ n) = (1::int)"
⟨*proof*⟩

lemma int_mod_lem: "0 < n ⟹ 0 ≤ b ∧ b < n ⟷ b mod n = b"
for b n :: int
⟨*proof*⟩

lemma int_mod_ge': "b < 0 ⟹ 0 < n ⟹ b + n ≤ b mod n"
for b n :: int
⟨*proof*⟩

lemma int_mod_le': "0 ≤ b - n ⟹ b mod n ≤ b - n"
for b n :: int
⟨*proof*⟩

lemma emep1: "even n ⟹ even d ⟹ 0 ≤ d ⟹ (n + 1) mod d = (n mod d) + 1"
for n d :: int
⟨*proof*⟩

lemma m1mod2k: "- 1 mod 2 ^ n = (2 ^ n - 1 :: int)"
⟨*proof*⟩

lemma sb_inc_lem: "a + 2^k < 0 ⟹ a + 2^k + 2^(Suc k) ≤ (a + 2^k) mod 2^(Suc k)"
for a :: int
⟨*proof*⟩

lemma sb_inc_lem': "a < - (2^k) ⟹ a + 2^k + 2^(Suc k) ≤ (a + 2^k) mod 2^(Suc k)"
for a :: int
⟨*proof*⟩

lemma sb_dec_lem: "0 ≤ - (2 ^ k) + a ⟹ (a + 2 ^ k) mod (2 * 2 ^ k) ≤ - (2 ^ k) + a"
for a :: int
⟨*proof*⟩

lemma sb_dec_lem': "2 ^ k ≤ a ⟹ (a + 2 ^ k) mod (2 * 2 ^ k) ≤ - (2 ^ k) + a"
for a :: int
⟨*proof*⟩

lemma mod_2_neq_1_eq_eq_0: "k mod 2 ≠ 1 ⟷ k mod 2 = 0"
for k :: int
⟨*proof*⟩

lemma z1pmod2: "(2 * b + 1) mod 2 = (1::int)"
for b :: int
⟨*proof*⟩

lemma p1mod22k': "(1 + 2 * b) mod (2 * 2 ^ n) = 1 + 2 * (b mod 2 ^ n)"
for b :: int
⟨*proof*⟩

lemma p1mod22k: "(2 * b + 1) mod (2 * 2 ^ n) = 2 * (b mod 2 ^ n) + 1"
for b :: int
⟨*proof*⟩

lemma pos_mod_sign2:
⟨0 ≤ a mod 2⟩ **for** a :: int
⟨*proof*⟩

lemma pos_mod_bound2:
⟨a mod 2 < 2⟩ **for** a :: int
⟨*proof*⟩

lemma nmod2: "n mod 2 = 0 ∨ n mod 2 = 1"
for n :: int
⟨*proof*⟩

lemma eme1p:
"even n ⟹ even d ⟹ 0 ≤ d ⟹ (1 + n) mod d = 1 + n mod d" **for** n
d :: int
⟨*proof*⟩

lemma m1mod22k:
⟨- 1 mod (2 * 2 ^ n) = 2 * 2 ^ n - (1::int)⟩
⟨*proof*⟩

lemma z1pdiv2: "(2 * b + 1) div 2 = b"
for b :: int
⟨*proof*⟩

lemma zdiv_le_dividend:
⟨0 ≤ a ⟹ 0 < b ⟹ a div b ≤ a⟩ **for** a b :: int
⟨*proof*⟩

lemma axxmod2: "(1 + x + x) mod 2 = 1 ∧ (0 + x + x) mod 2 = 0"
for x :: int
⟨*proof*⟩

lemma axxdiv2: "(1 + x + x) div 2 = x ∧ (0 + x + x) div 2 = x"
for x :: int
⟨*proof*⟩

lemmas rdmods =
mod_minus_eq [symmetric]
mod_diff_left_eq [symmetric]
mod_diff_right_eq [symmetric]

```
  mod_add_left_eq [symmetric]
  mod_add_right_eq [symmetric]
  mod_mult_right_eq [symmetric]
  mod_mult_left_eq [symmetric]

lemma mod_plus_right: "(a + x) mod m = (b + x) mod m ⟷ a mod m = b
mod m"
  for a b m x :: nat
  ⟨proof⟩

lemma nat_minus_mod: "(n - n mod m) mod m = 0"
  for m n :: nat
  ⟨proof⟩

lemmas nat_minus_mod_plus_right =
  trans [OF nat_minus_mod mod_0 [symmetric],
    THEN mod_plus_right [THEN iffD2], simplified]

lemmas push_mods' = mod_add_eq
  mod_mult_eq mod_diff_eq
  mod_minus_eq

lemmas push_mods = push_mods' [THEN eq_reflection]
lemmas pull_mods = push_mods [symmetric] rdmods [THEN eq_reflection]

lemma nat_mod_eq: "b < n ⟹ a mod n = b mod n ⟹ a mod n = b"
  for a b n :: nat
  ⟨proof⟩

lemmas nat_mod_eq' = refl [THEN [2] nat_mod_eq]

lemma nat_mod_lem: "0 < n ⟹ b < n ⟷ b mod n = b"
  for b n :: nat
  ⟨proof⟩

lemma mod_nat_add: "x < z ⟹ y < z ⟹ (x + y) mod z = (if x + y < z
then x + y else x + y - z)"
  for x y z :: nat
  ⟨proof⟩

lemma mod_nat_sub: "x < z ⟹ (x - y) mod z = x - y"
  for x y :: nat
  ⟨proof⟩

lemma int_mod_eq: "0 ≤ b ⟹ b < n ⟹ a mod n = b mod n ⟹ a mod n
= b"
  for a b n :: int
  ⟨proof⟩
```

lemma zmde:
  ‹b * (a div b) = a - a mod b› **for** a b :: ‹'a::{group_add,semiring_modulo}›
  ⟨*proof*⟩

lemma zdiv_mult_self: "m ≠ 0 ⟹ (a + m * n) div m = a div m + n"
  for a m n :: int
  ⟨*proof*⟩

lemma mod_power_lem: "a > 1 ⟹ a ^ n mod a ^ m = (if m ≤ n then 0 else a ^ n)"
  for a :: int
  ⟨*proof*⟩

lemma nonneg_mod_div: "0 ≤ a ⟹ 0 ≤ b ⟹ 0 ≤ (a mod b) ∧ 0 ≤ a div b"
  for a b :: int
  ⟨*proof*⟩

lemma mod_exp_less_eq_exp:
  ‹a mod 2 ^ n < 2 ^ n› **for** a :: int
  ⟨*proof*⟩

lemma div_mult_le:
  ‹a div b * b ≤ a› **for** a b :: nat
  ⟨*proof*⟩

lemma power_sub:
  fixes a :: nat
  assumes lt: "n ≤ m"
  and av: "0 < a"
  shows "a ^ (m - n) = a ^ m div a ^ n"
⟨*proof*⟩

lemma mod_lemma: "[| (0::nat) < c; r < b |] ==> b * (q mod c) + r < b * c"
  ⟨*proof*⟩

lemma less_two_pow_divD:
  "⟦ (x :: nat) < 2 ^ n div 2 ^ m ⟧
    ⟹ n ≥ m ∧ (x < 2 ^ (n - m))"
  ⟨*proof*⟩

lemma less_two_pow_divI:
  "⟦ (x :: nat) < 2 ^ (n - m); m ≤ n ⟧ ⟹ x < 2 ^ n div 2 ^ m"
  ⟨*proof*⟩

lemmas m2pths = pos_mod_sign mod_exp_less_eq_exp

lemmas int_mod_eq' = mod_pos_pos_trivial

lemmas int_mod_le = zmod_le_nonneg_dividend

lemma power_mod_div:
  fixes x :: "nat"
  shows "x mod 2 ^ n div 2 ^ m = x div 2 ^ m mod 2 ^ (n - m)" (**is** "?LHS = ?RHS")
⟨*proof*⟩

lemma mod_mod_power:
  fixes k :: nat
  shows "k mod 2 ^ m mod 2 ^ n = k mod 2 ^ (min m n)"
⟨*proof*⟩

lemma mod_div_equality_div_eq:
  "a div b * b = (a - (a mod b) :: int)"
  ⟨*proof*⟩

lemma zmod_helper:
  "n mod m = k ⟹ ((n :: int) + a) mod m = (k + a) mod m"
  ⟨*proof*⟩

lemma int_div_sub_1:
  "⟦ m ≥ 1 ⟧ ⟹ (n - (1 :: int)) div m = (if m dvd n then (n div m) - 1 else n div m)"
  ⟨*proof*⟩

lemma power_minus_is_div:
  "b ≤ a ⟹ (2 :: nat) ^ (a - b) = 2 ^ a div 2 ^ b"
  ⟨*proof*⟩

lemma two_pow_div_gt_le:
  "v < 2 ^ n div (2 ^ m :: nat) ⟹ m ≤ n"
  ⟨*proof*⟩

lemma td_gal_lt:
  ‹0 < c ⟹ a < b * c ⟷ a div c < b›
  for a b c :: nat
  ⟨*proof*⟩

lemma td_gal:
  ‹0 < c ⟹ b * c ≤ a ⟷ b ≤ a div c›
  for a b c :: nat
  ⟨*proof*⟩

end

3 Lemmas on words

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

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 shiftl_power:
  "(shiftl1 ^^ x) (y::'a::len word) = 2 ^ x * y"
  ⟨*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 le_shiftr1:
‹shiftr1 u ≤ shiftr1 v› **if** ‹u ≤ v›
⟨*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 = max_word ⟹ a AND x = a"
⟨*proof*⟩

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

lemma of_int_uint:
  "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 nat_mask_eq:
  ‹nat (mask n) = mask n›
  ⟨*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 nv: "n < 2 ^ (m - k)"
  and     kv: "k ≤ m"
  and     mv: "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 yx: "y ≤ x"
  and     y0: "0 < y"
  shows "(x - y) div y = x div y - 1"
  ⟨proof⟩

lemma word_mult_less_mono1:
  fixes i :: "'a :: len word"
  assumes ij: "i < j"
  and    knz: "0 < k"
  and    ujk: "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 ij: "i ≤ j"
and ujk: "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 ij: "i < j"
and ujk: "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"
and knz: "0 < k"
and ujk: "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 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_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 push_bit_word_eq_nonzero:
⟨push_bit n w ≠ 0⟩ **if** ⟨w < 2 ^ m⟩ ⟨m + n < LENGTH('a)⟩ ⟨w ≠ 0⟩
for w :: ⟨'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::len)" "y < LENGTH('a::len)"
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:
fixes x :: "'a :: len word" **shows**
"x ≠ max_word ⟹ (y < (x + 1)) = (y < x ∨ y = x)"
⟨*proof*⟩

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

lemma word_Suc_le:
fixes k::"'a::len word" **shows** "x ≠ max_word ⟹ 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 ≠ max_word" **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:
"x < 2 ^ (LENGTH('a) - 1) ⟹ 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:
"((b::'a::len word) ≤ 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':
"(unat x + unat y < 2 ^ LENGTH('a)) ⟹
(unat (x + y :: 'a :: len word) = 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:
"⟦ (x :: 'a::len word) < 2 ^ n div 2 ^ m ⟧

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

lemma of_nat_less_two_pow_div_set:
```
  "⟦ n < LENGTH('a) ⟧ ⟹
   {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:
```
  "⟦2 ^ n * q < (2::'a::len word) ^ m; q < 2 ^ (LENGTH('a) - n)⟧ ⟹ 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:
```
  "⟦ {x .. x + r - 1} ⊆ {y .. y + s - 1};
     x ≤ x + r - 1; y ≤ y + (s :: 'a :: len word) - 1;
     s ≠ 0 ⟧
     ⟹ 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':
fixes x :: "'a :: len word"
shows "⟦x + v ≤ x + w; x ≤ x + v⟧ ⟹ v ≤ w"
⟨*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_div_less:
"m < n ⟹ m div n = 0" **for** m :: "'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*⟩

end

4 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*⟩

type_synonym 'a sword = "'a signed word"

end

5 Operation variants with traditional syntax

theory Traditional_Infix_Syntax
imports "HOL-Library.Word" More_Word Signed_Words
begin

class semiring_bit_syntax = semiring_bit_shifts
begin

definition test_bit :: ‹'a ⇒ nat ⇒ bool› (**infixl** "!!" 100)
where test_bit_eq_bit: ‹test_bit = bit›

definition shiftl :: ‹'a ⇒ nat ⇒ 'a› (**infixl** "<<" 55)
where shiftl_eq_push_bit: ‹a << n = push_bit n a›

definition shiftr :: ‹'a ⇒ nat ⇒ 'a› (**infixl** ">>" 55)
where shiftr_eq_drop_bit: ‹a >> n = drop_bit n a›

end

instance word :: (len) semiring_bit_syntax ⟨*proof*⟩

context
includes lifting_syntax
begin

lemma test_bit_word_transfer [transfer_rule]:
‹(pcr_word ===> (=)) (λk n. n < LENGTH('a) ∧ bit k n) (test_bit :: 'a::len word ⇒ _)›
⟨*proof*⟩

lemma shiftl_word_transfer [transfer_rule]:

```
  ‹(pcr_word ===> (=) ===> pcr_word) (λk n. push_bit n k) shiftl›
```
⟨*proof*⟩

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

end

lemma test_bit_word_eq:
```
  ‹test_bit = (bit :: 'a::len word ⇒ _)›
```
⟨*proof*⟩

lemma shiftl_word_eq:
```
  ‹w << n = push_bit n w› for w :: ‹'a::len word›
```
⟨*proof*⟩

lemma shiftr_word_eq:
```
  ‹w >> n = drop_bit n w› for w :: ‹'a::len word›
```
⟨*proof*⟩

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

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

lemma word_eq_iff: "x = y ⟷ (∀n<LENGTH('a). x !! n = y !! n)" (**is**
‹?P ⟷ ?Q›)
  for x y :: "'a::len word"
⟨*proof*⟩

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

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

lemma test_bit_bin': "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:
  ‹test_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) (test_bit :: 'a::len word ⇒ _)"
  ⟨*proof*⟩

lemma test_bit_wi [simp]:
  "(word_of_int x :: 'a::len word) !! n ⟷ n < LENGTH('a) ∧ bit x n"
  ⟨*proof*⟩

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

lemma word_ao_nth:
  "(x OR y) !! n = (x !! n | y !! n) ∧
    (x AND y) !! n = (x !! n ∧ y !! n)"
  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 test_bit_numeral [simp]:
  "(numeral w :: 'a::len word) !! n ⟷
    n < LENGTH('a) ∧ bit (numeral w :: int) n"
  ⟨*proof*⟩

lemma test_bit_neg_numeral [simp]:
  "(- numeral w :: 'a::len word) !! n ⟷
    n < LENGTH('a) ∧ bit (- numeral w :: int) n"
  ⟨*proof*⟩

lemma test_bit_1 [iff]: "(1 :: 'a::len word) !! n ⟷ n = 0"
  ⟨*proof*⟩

lemma nth_0 [simp]: "¬ (0 :: 'a::len word) !! n"
  ⟨*proof*⟩

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

lemma shiftl1_code [code]:
⟨shiftl1 w = push_bit 1 w⟩
⟨*proof*⟩

lemma uint_shiftr_eq:
⟨uint (w >> n) = uint w div 2 ^ n⟩
⟨*proof*⟩

lemma shiftr1_code [code]:
⟨shiftr1 w = drop_bit 1 w⟩
⟨*proof*⟩

lemma shiftl_def:
⟨w << n = (shiftl1 ^^ n) w⟩
⟨*proof*⟩

lemma shiftr_def:
⟨w >> n = (shiftr1 ^^ n) w⟩
⟨*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_simps]:
⟨bit (w >> m) n ⟷ bit w (m + n)⟩
for w :: ⟨'a::len word⟩
⟨*proof*⟩

lift_definition sshiftr :: ⟨'a::len word ⇒ nat ⇒ 'a word⟩ (**infixl** ⟨>>>⟩ 55)
is ⟨λk n. take_bit LENGTH('a) (drop_bit n (signed_take_bit (LENGTH('a) - Suc 0) k))⟩
⟨*proof*⟩

lemma sshiftr_eq [code]:
⟨w >>> n = signed_drop_bit n w⟩
⟨*proof*⟩

lemma sshiftr_eq_funpow_sshiftr1:
⟨w >>> n = (sshiftr1 ^^ n) w⟩
⟨*proof*⟩

lemma uint_sshiftr_eq:

```
  ‹uint (w >>> n) = take_bit LENGTH('a) (sint w div 2 ^  n)›
  for w :: ‹'a::len word›
  ⟨proof⟩

lemma sshift1_code [code]:
  ‹sshiftr1 w = signed_drop_bit 1 w›
  ⟨proof⟩

lemma sshiftr_0 [simp]: "0 >>> n = 0"
  ⟨proof⟩

lemma sshiftr_n1 [simp]: "-1 >>> n = -1"
  ⟨proof⟩

lemma bit_sshiftr_word_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⟩

lemma nth_sshiftr :
  "(w >>> m) !! n =
    (n < size w ∧ (if n + m ≥ size w then w !! (size w - 1) else w !!
(n + m)))"
  ⟨proof⟩

lemma sshiftr_numeral [simp]:
  ‹(numeral k >>> numeral n :: 'a::len word) =
    word_of_int (drop_bit (numeral n) (signed_take_bit (LENGTH('a) - 1)
(numeral k)))›
  ⟨proof⟩

⟨ML⟩

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

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

lemma sshiftr_div_2n: "sint (w >>> n) = sint w div 2 ^ n"
  ⟨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 w !! n else w !! (l - 1))"
  ⟨*proof*⟩

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

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

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

lemma mask_eq:
  ‹mask n = (1 << n) - (1 :: 'a::len word)›
  ⟨*proof*⟩

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

lemma shiftl_0 [simp]: "(0::'a::len word) << n = 0"
  ⟨*proof*⟩

lemma shiftr_0 [simp]: "(0::'a::len word) >> n = 0"
  ⟨*proof*⟩

lemma nth_shiftl1: "shiftl1 w !! n ⟷ n < size w ∧ n > 0 ∧ w !! (n
- 1)"
  ⟨*proof*⟩

lemma nth_shiftl': "(w << m) !! n ⟷ n < size w ∧ n >= m ∧ w !! (n
- m)"
  for w :: "'a::len word"
  ⟨*proof*⟩

lemmas nth_shiftl = nth_shiftl' [unfolded word_size]

lemma nth_shiftr1: "shiftr1 w !! n = w !! Suc n"
  ⟨*proof*⟩

lemma nth_shiftr: "(w >> m) !! n = w !! (n + m)"
  for w :: "'a::len word"

⟨*proof*⟩

lemma nth_sshiftr1: "sshiftr1 w !! n = (if n = size w - 1 then w !! n else w !! Suc n)"
⟨*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*⟩

lemma shiftl_numeral [simp]:
‹numeral k << numeral l = (push_bit (numeral l) (numeral k) :: 'a::len word)›
⟨*proof*⟩

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 shiftr_numeral [simp]:
‹(numeral k >> numeral n :: 'a::len word) = drop_bit (numeral n) (numeral k)›
⟨*proof*⟩

lemma shiftr_numeral_Suc [simp]:
‹(numeral k >> Suc 0 :: 'a::len word) = drop_bit (Suc 0) (numeral k)›
⟨*proof*⟩

lemma drop_bit_numeral_bit0_1 [simp]:
‹drop_bit (Suc 0) (numeral k) =
(word_of_int (drop_bit (Suc 0) (take_bit LENGTH('a) (numeral k)))
:: 'a::len word)›
⟨*proof*⟩

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

lemma slice_shiftr: "slice n w = ucast (w >> n)"
  ⟨*proof*⟩

lemma nth_slice: "(slice n w :: 'a::len word) !! m = (w !! (m + n) ∧
m < LENGTH('a))"
  ⟨*proof*⟩

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

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

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

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

lemmas ucast_up =
  rc1 [simplified rev_shiftr [symmetric] revcast_ucast [symmetric]]
lemmas ucast_down =
  rc2 [simplified rev_shiftr revcast_ucast [symmetric]]

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

lemma sofl_test:

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

lemma shiftr_zero_size: "size x ≤ n ⟹ x >> n = 0"
  for x :: "'a :: len word"
  ⟨proof⟩

lemma test_bit_cat [OF refl]:
  "wc = word_cat a b ⟹ wc !! n = (n < size wc ∧
    (if n < size b then b !! n else a !! (n - size b)))"
  ⟨proof⟩
lemma test_bit_split':
  "word_split c = (a, b) ⟶
    (∀n m.
      b !! n = (n < size b ∧ c !! n) ∧
      a !! m = (m < size a ∧ c !! (m + size b)))"
  ⟨proof⟩

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

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

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

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

lemma max_test_bit: "(max_word::'a::len word) !! n ⟷ n < LENGTH('a)"
  ⟨proof⟩

lemma shiftr_x_0 [iff]: "x >> 0 = x"
  for x :: "'a::len word"
  ⟨proof⟩

lemma shiftl_x_0 [simp]: "x << 0 = x"
```

for x :: "'a::len word"
⟨*proof*⟩

lemma shiftl_1 [simp]: "(1::'a::len word) << n = 2^n"
⟨*proof*⟩

lemma shiftr_1[simp]: "(1::'a::len word) >> n = (if n = 0 then 1 else 0)"
⟨*proof*⟩

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

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

lemma test_bit_1' [simp]:
"(1 :: 'a :: len word) !! n ⟷ 0 < LENGTH('a) ∧ n = 0"
⟨*proof*⟩

lemma shiftl0:
"x << 0 = (x :: 'a :: len word)"
⟨*proof*⟩

lemma word_ops_nth [simp]:
fixes x y :: ‹'a::len word›
shows
word_or_nth: "(x OR y) !! n = (x !! n ∨ y !! n)" **and**
word_and_nth: "(x AND y) !! n = (x !! n ∧ y !! n)" **and**
word_xor_nth: "(x XOR y) !! n = (x !! n ≠ y !! n)"
⟨*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 nth_w2p_same:
"(2^n :: 'a :: len word) !! n = (n < LENGTH('a))"
⟨*proof*⟩

lemma shiftr_div_2n_w: "n < size 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 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 word_leI:
```
  "(⋀n.  ⟦n < size (u::'a::len word); u !! n ⟧ ⟹ (v::'a::len word) !!
n) ⟹ u <= v"
```
⟨*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 bang_eq:
```
  fixes x :: "'a::len word"
  shows "(x = y) = (∀n. x !! n = y !! n)"
```
⟨*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:
assumes nv: "n < LENGTH('a)"
and xv: "x < 2 ^ (LENGTH('a) - n)"
shows "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:
"⟦ (x::'a::len word) ≤ 2 ^ m; m + n < LENGTH('a::len); x ≠ 0⟧
⟹ x << n ≠ 0"
⟨*proof*⟩

lemma word_shiftr_lt:
fixes w :: "'a::len word"
shows "unat (w >> n) < (2 ^ (LENGTH('a) - n))"
⟨*proof*⟩

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

lemma upper_bits_unset_is_l2p:
‹(∀n' ≥ n. n' < LENGTH('a) ⟶ ¬ 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' < LENGTH('a) ⟶ ¬ p !! n')" **for** p :: "'a :: len word"
⟨*proof*⟩

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

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

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

lemma neg_test_bit:
"(NOT x) !! n = (¬ x !! n ∧ n < LENGTH('a))" **for** x :: "'a::len word"
⟨*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 shiftr_not_mask_0:
"n+m ≥ LENGTH('a :: len) ⟹ ((w::'a::len word) >> n) AND NOT (mask

```
m) = 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 nth_w2p_scast [simp]:
```
  "((scast ((2::'a::len signed word) ^ n) :: 'a word) !! m)
        ⟷ ((((2::'a::len  word) ^ n) :: 'a word) !! m)"
```
⟨*proof*⟩

lemma scast_bit_test [simp]:
```
    "scast ((1 :: 'a::len signed word) << n) = (1 :: 'a word) << n"
```
⟨*proof*⟩

lemma signed_shift_guard_to_word:
```
  "⟦ n < len_of TYPE ('a); n > 0 ⟧
    ⟹ (unat (x :: 'a :: len word) * 2 ^ y < 2 ^ n)
    = (x = 0 ∨ x < (1 << n >> y))"
```
⟨*proof*⟩

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

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

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

end

6 Solving Word Equalities

theory Word_EqI
  imports
    More_Word
    Traditional_Infix_Syntax
    "HOL-Eisbach.Eisbach_Tools"
begin

Some word equalities can be solved by considering the problem bitwise for all `n < LENGTH('a)`, which is different to running `word_bitwise` and expanding into an explicit list of bits.

named_theorems word_eqI_simps

lemmas [word_eqI_simps] =
  word_ops_nth_size
  word_size
  word_or_zero
  neg_mask_test_bit
  nth_ucast
  nth_w2p nth_shiftl
  nth_shiftr
  less_2p_is_upper_bits_unset

```
  le_mask_high_bits
  bang_eq
  neg_test_bit
  is_up
  is_down

lemmas word_eqI_rule = word_eqI [rule_format]

lemma test_bit_lenD:
  "x !! n ⟹ n < LENGTH('a) ∧ x !! n" for x :: "'a :: len word"
  ⟨proof⟩

method word_eqI uses simp simp_del split split_del cong flip =
  (
   rule word_eqI_rule,

   (clarsimp simp: simp simp del: simp_del simp flip: flip split: split
split del: split_del cong: cong)?,

   ((drule less_mask_eq)+)?,

   (clarsimp simp: word_eqI_simps simp simp del: simp_del simp flip: flip
             split: split split del: split_del cong: cong)?,

   ((drule test_bit_lenD)+)?,

   (clarsimp simp: word_eqI_simps 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 word_eqI_solve uses simp simp_del split split_del cong flip =
  solves ‹word_eqI simp: simp simp_del: simp_del split: split split_del:
split_del
                   cong: cong simp flip: flip;
          (fastforce dest: test_bit_size simp: word_eqI_simps simp flip:
flip
                     simp: simp simp del: simp_del split: split split
del: split_del cong: cong)?›

end
```

7 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 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⟩

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 (set_bits P :: 'a::len word) n ⟷ n < LENGTH('a) ∧ P n›
  ⟨proof⟩

lemma set_bits_K_False [simp]:
  ‹set_bits (λ_. False) = (0 :: 'a :: len word)›
  ⟨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

8 Bitwise Operations on integers

theory Bits_Int
imports
"HOL-Library.Word"
Traditional_Infix_Syntax
begin

8.1 Implicit bit representation of int

abbreviation (input) bin_last :: "int ⇒ bool"
where "bin_last ≡ odd"

lemma bin_last_def:
"bin_last w ⟷ w mod 2 = 1"
⟨*proof*⟩

abbreviation (input) bin_rest :: "int ⇒ int"
where "bin_rest w ≡ w div 2"

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]:
"bin_rest 0 = 0"
"bin_rest 1 = 0"
"bin_rest (- 1) = - 1"
"bin_rest Numeral1 = 0"
"bin_rest (numeral (Num.Bit0 w)) = numeral w"
"bin_rest (numeral (Num.Bit1 w)) = numeral w"
"bin_rest (- numeral (Num.Bit0 w)) = - numeral w"

"bin_rest (- numeral (Num.Bit1 w)) = - numeral (w + Num.One)"
⟨*proof*⟩

lemma bin_rl_eqI: "⟦bin_rest x = bin_rest y; odd x = odd y⟧ ⟹ x = y"
⟨*proof*⟩

lemma [simp]:
shows bin_rest_lt0: "bin_rest i < 0 ⟷ i < 0"
and bin_rest_ge_0: "bin_rest i ≥ 0 ⟷ i ≥ 0"
⟨*proof*⟩

lemma bin_rest_gt_0 [simp]: "bin_rest x > 0 ⟷ x > 1"
⟨*proof*⟩

8.2 Bit projection

abbreviation (input) bin_nth :: ‹int ⇒ nat ⇒ bool›
where ‹bin_nth ≡ bit›

lemma bin_nth_eq_iff: "bin_nth x = bin_nth y ⟷ x = y"
⟨*proof*⟩

lemma bin_eqI:
"x = y" **if** "⋀n. bin_nth x n ⟷ bin_nth y n"
⟨*proof*⟩

lemma bin_eq_iff: "x = y ⟷ (∀n. bin_nth x n = bin_nth y n)"
⟨*proof*⟩

lemma bin_nth_zero [simp]: "¬ bin_nth 0 n"
⟨*proof*⟩

lemma bin_nth_1 [simp]: "bin_nth 1 n ⟷ n = 0"
⟨*proof*⟩

lemma bin_nth_minus1 [simp]: "bin_nth (- 1) n"
⟨*proof*⟩

lemma bin_nth_numeral: "bin_rest x = y ⟹ bin_nth x (numeral n) = bin_nth 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: "bin_nth (2 ^ n) m = (m = n)"` — for use when simplifying with `bin_nth_Bit`
⟨*proof*⟩

lemma `nth_rest_power_bin: "bin_nth ((bin_rest ^^ k) w) n = bin_nth w (n + k)"`
⟨*proof*⟩

lemma `bin_nth_numeral_unfold:`
`"bin_nth (numeral (num.Bit0 x)) n ⟷ n > 0 ∧ bin_nth (numeral x) (n - 1)"`
`"bin_nth (numeral (num.Bit1 x)) n ⟷ (n > 0 ⟶ bin_nth (numeral x) (n - 1))"`
⟨*proof*⟩

8.3 Truncating

definition `bin_sign :: "int ⇒ int"`
where `"bin_sign k = (if k ≥ 0 then 0 else - 1)"`

lemma `bin_sign_simps [simp]:`
`"bin_sign 0 = 0"`
`"bin_sign 1 = 0"`
`"bin_sign (- 1) = - 1"`
`"bin_sign (numeral k) = 0"`
`"bin_sign (- numeral k) = -1"`
⟨*proof*⟩

lemma `bin_sign_rest [simp]: "bin_sign (bin_rest w) = bin_sign w"`
⟨*proof*⟩

abbreviation (input) `bintrunc :: ⟨nat ⇒ int ⇒ int⟩`
where `⟨bintrunc ≡ take_bit⟩`

lemma `bintrunc_mod2p: "bintrunc n w = w mod 2 ^ n"`
⟨*proof*⟩

abbreviation (input) `sbintrunc :: ⟨nat ⇒ int ⇒ int⟩`
where `⟨sbintrunc ≡ signed_take_bit⟩`

abbreviation (input) `norm_sint :: ⟨nat ⇒ int ⇒ int⟩`
where `⟨norm_sint n ≡ signed_take_bit (n - 1)⟩`

lemma `sbintrunc_mod2p: "sbintrunc n w = (w + 2 ^ n) mod 2 ^ Suc n - 2 ^ n"`
⟨*proof*⟩

lemma `sbintrunc_eq_take_bit:`
`⟨sbintrunc n k = take_bit (Suc n) (k + 2 ^ n) - 2 ^ n⟩`

⟨*proof*⟩

lemma sign_bintr: "bin_sign (bintrunc n w) = 0"
⟨*proof*⟩

lemma bintrunc_n_0: "bintrunc n 0 = 0"
⟨*proof*⟩

lemma sbintrunc_n_0: "sbintrunc n 0 = 0"
⟨*proof*⟩

lemma sbintrunc_n_minus1: "sbintrunc n (- 1) = -1"
⟨*proof*⟩

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

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

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

lemma bin_sign_lem: "(bin_sign (sbintrunc n bin) = -1) = bit bin n"
⟨*proof*⟩

lemma nth_bintr: "bin_nth (bintrunc m w) n ⟷ n < m ∧ bin_nth w n"
⟨*proof*⟩

lemma nth_sbintr: "bin_nth (sbintrunc m w) n = (if n < m then bin_nth w n else bin_nth w m)"
⟨*proof*⟩

lemma bin_nth_Bit0:
"bin_nth (numeral (Num.Bit0 w)) n ⟷
(∃m. n = Suc m ∧ bin_nth (numeral w) m)"
⟨*proof*⟩

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

lemma bintrunc_bintrunc_l: "n ≤ m ⟹ bintrunc m (bintrunc n w) = bintrunc n w"
⟨*proof*⟩

lemma sbintrunc_sbintrunc_l: "n ≤ m ⟹ sbintrunc m (sbintrunc n w) = sbintrunc n w"
⟨*proof*⟩

lemma bintrunc_bintrunc_ge: "n ≤ m ⟹ bintrunc n (bintrunc m w) = bintrunc n w"
⟨*proof*⟩

lemma bintrunc_bintrunc_min [simp]: "bintrunc m (bintrunc n w) = bintrunc (min m n) w"
⟨*proof*⟩

lemma sbintrunc_sbintrunc_min [simp]: "sbintrunc m (sbintrunc n w) = sbintrunc (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 ⟹ bintrunc (Suc (n - 1)) w = bintrunc
n w"
  ⟨proof⟩

lemma sbintrunc_minus: "0 < n ⟹ sbintrunc (Suc (n - 1)) w = sbintrunc
n w"
  ⟨proof⟩

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

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

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

lemma sbintrunc_Suc_lem: "sbintrunc (Suc n) x = y ⟹ m = Suc n ⟹ sbintrunc
m x = y"
  ⟨proof⟩

lemmas sbintrunc_Suc_Ialts =
  sbintrunc_Suc_Is [THEN sbintrunc_Suc_lem]

lemma sbintrunc_bintrunc_lt: "m > n ⟹ sbintrunc n (bintrunc m w) =
sbintrunc n w"
  ⟨proof⟩

lemma bintrunc_sbintrunc_le: "m ≤ Suc n ⟹ bintrunc m (sbintrunc n
w) = bintrunc 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 ⟹ bintrunc n (sbintrunc (n
- 1) w) = bintrunc n w"
⟨*proof*⟩

lemma sbintrunc_bintrunc' [simp]: "0 < n ⟹ sbintrunc (n - 1) (bintrunc
n w) = sbintrunc (n - 1) w"
⟨*proof*⟩

lemma bin_sbin_eq_iff: "bintrunc (Suc n) x = bintrunc (Suc n) y ⟷
sbintrunc n x = sbintrunc n y"
⟨*proof*⟩

lemma bin_sbin_eq_iff':
"0 < n ⟹ bintrunc n x = bintrunc n y ⟷ sbintrunc (n - 1) x = sbintrunc
(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:
"bintrunc (numeral k) x = of_bool (odd x) + 2 * bintrunc (pred_numeral
k) (x div 2)"
⟨*proof*⟩

lemma sbintrunc_numeral:
"sbintrunc (numeral k) x = of_bool (odd x) + 2 * sbintrunc (pred_numeral
k) (x div 2)"
⟨*proof*⟩

lemma bintrunc_numeral_simps [simp]:
"bintrunc (numeral k) (numeral (Num.Bit0 w)) =
2 * bintrunc (pred_numeral k) (numeral w)"
"bintrunc (numeral k) (numeral (Num.Bit1 w)) =
1 + 2 * bintrunc (pred_numeral k) (numeral w)"
"bintrunc (numeral k) (- numeral (Num.Bit0 w)) =

```
    2 * bintrunc (pred_numeral k) (- numeral w)"
  "bintrunc (numeral k) (- numeral (Num.Bit1 w)) =
    1 + 2 * bintrunc (pred_numeral k) (- numeral (w + Num.One))"
  "bintrunc (numeral k) 1 = 1"
```
⟨*proof*⟩

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

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

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

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

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

lemma sbintrunc_inc:
⟨k + 2 ^ Suc n ≤ sbintrunc n k⟩ **if** ⟨k < - (2 ^ n)⟩
⟨*proof*⟩

lemma sbintrunc_dec:
⟨sbintrunc n k ≤ k - 2 ^ (Suc n)⟩ **if** ⟨k ≥ 2 ^ n⟩
⟨*proof*⟩

lemma bintr_ge0: "0 ≤ bintrunc n w"
⟨*proof*⟩

lemma bintr_lt2p: "bintrunc n w < 2 ^ n"
⟨*proof*⟩

lemma bintr_Min: "bintrunc n (- 1) = 2 ^ n - 1"
⟨*proof*⟩

lemma sbintr_ge: "- (2 ^ n) ≤ sbintrunc n w"
⟨*proof*⟩

lemma sbintr_lt: "sbintrunc n w < 2 ^ n"
⟨*proof*⟩

lemma sign_Pls_ge_0: "bin_sign bin = 0 ⟷ bin ≥ 0"
for bin :: int
⟨*proof*⟩

lemma sign_Min_lt_0: "bin_sign bin = -1 ⟷ bin < 0"
for bin :: int
⟨*proof*⟩

lemma bin_rest_trunc: "bin_rest (bintrunc n bin) = bintrunc (n - 1) (bin_rest bin)"
⟨*proof*⟩

lemma bin_rest_power_trunc:
"(bin_rest ^^ k) (bintrunc n bin) = bintrunc (n - k) ((bin_rest ^^ k) bin)"
⟨*proof*⟩

lemma bin_rest_trunc_i: "bintrunc n (bin_rest bin) = bin_rest (bintrunc (Suc n) bin)"
⟨*proof*⟩

lemma bin_rest_strunc: "bin_rest (sbintrunc (Suc n) bin) = sbintrunc n (bin_rest bin)"
⟨*proof*⟩

lemma bintrunc_rest [simp]: "bintrunc n (bin_rest (bintrunc n bin)) = bin_rest (bintrunc n bin)"
⟨*proof*⟩

lemma sbintrunc_rest [simp]: "sbintrunc n (bin_rest (sbintrunc n bin)) = bin_rest (sbintrunc n bin)"
⟨*proof*⟩

lemma bintrunc_rest': "bintrunc n ∘ bin_rest ∘ bintrunc n = bin_rest ∘ bintrunc n"
⟨*proof*⟩

lemma sbintrunc_rest': "sbintrunc n ∘ bin_rest ∘ sbintrunc n = bin_rest ∘ sbintrunc n"
⟨*proof*⟩

lemma rco_lem: "f ∘ g ∘ f = g ∘ f ⟹ 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]

8.4 Splitting and concatenation

definition bin_split :: ‹nat ⇒ int ⇒ int × int›
  where [simp]: ‹bin_split n k = (drop_bit n k, take_bit n k)›

lemma [code]:
  "bin_split (Suc n) w = (let (w1, w2) = bin_split n (w div 2) in (w1,
of_bool (odd w) + 2 * w2))"
  "bin_split 0 w = (w, 0)"
  ⟨*proof*⟩

abbreviation (input) bin_cat :: ‹int ⇒ nat ⇒ int ⇒ int›
  where ‹bin_cat k n l ≡ concat_bit n l k›

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

lemma bin_sign_cat: "bin_sign (bin_cat x n y) = bin_sign x"
⟨*proof*⟩

lemma bin_cat_assoc: "bin_cat (bin_cat x m y) n z = bin_cat x (m + n)
(bin_cat y n z)"
  ⟨*proof*⟩

lemma bin_cat_assoc_sym: "bin_cat x m (bin_cat y n z) = bin_cat (bin_cat
x (m - n) y) (min m n) z"
  ⟨*proof*⟩

definition bin_rcat :: ‹nat ⇒ int list ⇒ int›
  where ‹bin_rcat n = horner_sum (take_bit n) (2 ^ n) ∘ rev›

lemma bin_rcat_eq_foldl:
  ‹bin_rcat n = foldl (λu v. bin_cat u n v) 0›
⟨*proof*⟩

fun bin_rsplit_aux :: "nat ⇒ nat ⇒ int ⇒ int list ⇒ int list"
  where "bin_rsplit_aux n m c bs =
    (if m = 0 ∨ n = 0 then bs
     else
      let (a, b) = bin_split n c
      in bin_rsplit_aux n (m - n) a (b # bs))"

```
definition bin_rsplit :: "nat ⇒ nat × int ⇒ int list"
  where "bin_rsplit n w = bin_rsplit_aux n (fst w) (snd w) []"

value ‹bin_rsplit 1705 (3, 88)›

fun bin_rsplitl_aux :: "nat ⇒ nat ⇒ int ⇒ int list ⇒ int list"
  where "bin_rsplitl_aux n m c bs =
    (if m = 0 ∨ n = 0 then bs
     else
      let (a, b) = bin_split (min m n) c
      in bin_rsplitl_aux n (m - n) a (b # bs))"

definition bin_rsplitl :: "nat ⇒ nat × int ⇒ int list"
  where "bin_rsplitl n w = bin_rsplitl_aux n (fst w) (snd w) []"

declare bin_rsplit_aux.simps [simp del]
declare bin_rsplitl_aux.simps [simp del]

lemma bin_nth_cat:
  "bin_nth (bin_cat x k y) n =
    (if n < k then bin_nth y n else bin_nth x (n - k))"
  ⟨proof⟩

lemma bin_nth_drop_bit_iff:
  ‹bin_nth (drop_bit n c) k ⟷ bin_nth c (n + k)›
  ⟨proof⟩

lemma bin_nth_take_bit_iff:
  ‹bin_nth (take_bit n c) k ⟷ k < n ∧ bin_nth c k›
  ⟨proof⟩

lemma bin_nth_split:
  "bin_split n c = (a, b) ⟹
    (∀k. bin_nth a k = bin_nth c (n + k)) ∧
    (∀k. bin_nth b k = (k < n ∧ bin_nth c k))"
  ⟨proof⟩

lemma bin_cat_zero [simp]: "bin_cat 0 n w = bintrunc n w"
  ⟨proof⟩

lemma bintr_cat1: "bintrunc (k + n) (bin_cat a n b) = bin_cat (bintrunc
k a) n b"
  ⟨proof⟩

lemma bintr_cat: "bintrunc m (bin_cat a n b) =
    bin_cat (bintrunc (m - n) a) n (bintrunc (min m n) b)"

  ⟨proof⟩
```

lemma bintr_cat_same [simp]: "bintrunc n (bin_cat a n b) = bintrunc n b"
⟨*proof*⟩

lemma cat_bintr [simp]: "bin_cat a n (bintrunc n b) = bin_cat a n b"
⟨*proof*⟩

lemma split_bintrunc: "bin_split n c = (a, b) ⟹ b = bintrunc n c"
⟨*proof*⟩

lemma bin_cat_split: "bin_split n w = (u, v) ⟹ w = bin_cat u n v"
⟨*proof*⟩

lemma drop_bit_bin_cat_eq:
‹drop_bit n (bin_cat v n w) = v›
⟨*proof*⟩

lemma take_bit_bin_cat_eq:
‹take_bit n (bin_cat v n w) = take_bit n w›
⟨*proof*⟩

lemma bin_split_cat: "bin_split n (bin_cat v n w) = (v, bintrunc n w)"
⟨*proof*⟩

lemma bin_split_zero [simp]: "bin_split n 0 = (0, 0)"
⟨*proof*⟩

lemma bin_split_minus1 [simp]:
"bin_split n (- 1) = (- 1, bintrunc n (- 1))"
⟨*proof*⟩

lemma bin_split_trunc:
"bin_split (min m n) c = (a, b) ⟹
bin_split n (bintrunc m c) = (bintrunc (m - n) a, b)"
⟨*proof*⟩

lemma bin_split_trunc1:
"bin_split n c = (a, b) ⟹
bin_split n (bintrunc m c) = (bintrunc (m - n) a, bintrunc m b)"
⟨*proof*⟩

lemma bin_cat_num: "bin_cat a n b = a * 2 ^ n + bintrunc n b"
⟨*proof*⟩

lemma bin_split_num: "bin_split n b = (b div 2 ^ n, b mod 2 ^ n)"
⟨*proof*⟩

lemmas bin_rsplit_aux_simps = bin_rsplit_aux.simps bin_rsplitl_aux.simps
lemmas rsplit_aux_simps = bin_rsplit_aux_simps

```
lemmas th_if_simp1 = if_split [where P = "(=) l", THEN iffD1, THEN conjunct1,
THEN mp] for l
lemmas th_if_simp2 = if_split [where P = "(=) l", THEN iffD1, THEN conjunct2,
THEN mp] for l

lemmas rsplit_aux_simp1s = rsplit_aux_simps [THEN th_if_simp1]

lemmas rsplit_aux_simp2ls = rsplit_aux_simps [THEN th_if_simp2]
```
— these safe to `[simp add]` as require calculating `m - n`
```
lemmas bin_rsplit_aux_simp2s [simp] = rsplit_aux_simp2ls [unfolded Let_def]
lemmas rbscl = bin_rsplit_aux_simp2s (2)

lemmas rsplit_aux_0_simps [simp] =
  rsplit_aux_simp1s [OF disjI1] rsplit_aux_simp1s [OF disjI2]

lemma bin_rsplit_aux_append: "bin_rsplit_aux n m c (bs @ cs) = bin_rsplit_aux
n m c bs @ cs"
  ⟨proof⟩

lemma bin_rsplitl_aux_append: "bin_rsplitl_aux n m c (bs @ cs) = bin_rsplitl_aux
n m c bs @ cs"
  ⟨proof⟩

lemmas rsplit_aux_apps [where bs = "[]"] =
  bin_rsplit_aux_append bin_rsplitl_aux_append

lemmas rsplit_def_auxs = bin_rsplit_def bin_rsplitl_def

lemmas rsplit_aux_alts = rsplit_aux_apps
  [unfolded append_Nil rsplit_def_auxs [symmetric]]

lemma bin_split_minus: "0 < n ⟹ bin_split (Suc (n - 1)) w = bin_split
n w"
  ⟨proof⟩

lemma bin_split_pred_simp [simp]:
  "(0::nat) < numeral bin ⟹
    bin_split (numeral bin) w =
      (let (w1, w2) = bin_split (numeral bin - 1) (bin_rest w)
       in (w1, of_bool (odd w) + 2 * w2))"
  ⟨proof⟩

lemma bin_rsplit_aux_simp_alt:
  "bin_rsplit_aux n m c bs =
    (if m = 0 ∨ n = 0 then bs
     else let (a, b) = bin_split n c in bin_rsplit n (m - n, a) @ b #
bs)"
  ⟨proof⟩
```

lemmas bin_rsplit_simp_alt =
  trans [OF bin_rsplit_def bin_rsplit_aux_simp_alt]

lemmas bthrs = bin_rsplit_simp_alt [THEN [2] trans]

lemma bin_rsplit_size_sign' [rule_format]:
  "n > 0 ⟹ rev sw = bin_rsplit n (nw, w) ⟹ ∀v∈set sw. bintrunc n
v = v"
  ⟨*proof*⟩

lemmas bin_rsplit_size_sign = bin_rsplit_size_sign' [OF asm_rl
  rev_rev_ident [THEN trans] set_rev [THEN equalityD2 [THEN subsetD]]]

lemma bin_nth_rsplit [rule_format] :
  "n > 0 ⟹ m < n ⟹
    ∀w k nw.
      rev sw = bin_rsplit n (nw, w) ⟶
      k < size sw ⟶ bin_nth (sw ! k) m = bin_nth w (k * n + m)"
  ⟨*proof*⟩

lemma bin_rsplit_all: "0 < nw ⟹ nw ≤ n ⟹ bin_rsplit n (nw, w) =
[bintrunc n w]"
  ⟨*proof*⟩

lemma bin_rsplit_l [rule_format]:
  "∀bin. bin_rsplitl n (m, bin) = bin_rsplit n (m, bintrunc m bin)"
  ⟨*proof*⟩

lemma bin_rsplit_rcat [rule_format]:
  "n > 0 ⟶ bin_rsplit n (n * size ws, bin_rcat n ws) = map (bintrunc
n) ws"
  ⟨*proof*⟩

lemma bin_rsplit_aux_len_le [rule_format] :
  "∀ws m. n ≠ 0 ⟶ ws = bin_rsplit_aux n nw w bs ⟶
    length ws ≤ m ⟷ nw + length bs * n ≤ m * n"
⟨*proof*⟩

lemma bin_rsplit_len_le: "n ≠ 0 ⟶ ws = bin_rsplit n (nw, w) ⟶ length
ws ≤ m ⟷ nw ≤ m * n"
  ⟨*proof*⟩

lemma bin_rsplit_aux_len:
  "n ≠ 0 ⟹ length (bin_rsplit_aux n nw w cs) = (nw + n - 1) div n +
length cs"
  ⟨*proof*⟩

lemma bin_rsplit_len: "n ≠ 0 ⟹ length (bin_rsplit n (nw, w)) = (nw

```
+ n - 1) div n"
```
⟨*proof*⟩

lemma bin_rsplit_aux_len_indep:
```
  "n ≠ 0 ⟹ length bs = length cs ⟹
    length (bin_rsplit_aux n nw v bs) =
    length (bin_rsplit_aux n nw w cs)"
```
⟨*proof*⟩

lemma bin_rsplit_len_indep:
```
  "n ≠ 0 ⟹ length (bin_rsplit n (nw, v)) = length (bin_rsplit n (nw,
w))"
```
⟨*proof*⟩

8.5 Logical operations

primrec bin_sc :: "nat ⇒ bool ⇒ int ⇒ int"
  where
```
    Z: "bin_sc 0 b w = of_bool b + 2 * bin_rest w"
  | Suc: "bin_sc (Suc n) b w = of_bool (odd w) + 2 * bin_sc n b (w div
2)"
```

lemma bin_nth_sc [simp]: "bit (bin_sc n b w) n ⟷ b"
⟨*proof*⟩

lemma bin_sc_sc_same [simp]: "bin_sc n c (bin_sc n b w) = bin_sc n c
w"
⟨*proof*⟩

lemma bin_sc_sc_diff: "m ≠ n ⟹ bin_sc m c (bin_sc n b w) = bin_sc
n b (bin_sc m c w)"
⟨*proof*⟩

lemma bin_nth_sc_gen: "bin_nth (bin_sc n b w) m = (if m = n then b else
bin_nth w m)"
⟨*proof*⟩

lemma bin_sc_eq:
```
  ‹bin_sc n False = unset_bit n›
  ‹bin_sc n True = Bit_Operations.set_bit n›
```
⟨*proof*⟩

lemma bin_sc_nth [simp]: "bin_sc n (bin_nth w n) w = w"
⟨*proof*⟩

lemma bin_sign_sc [simp]: "bin_sign (bin_sc n b w) = bin_sign w"
⟨*proof*⟩

lemma bin_sc_bintr [simp]:

```
  "bintrunc m (bin_sc n x (bintrunc m w)) = bintrunc m (bin_sc n x w)"
  ⟨proof⟩

lemma bin_clr_le: "bin_sc n False w ≤ w"
  ⟨proof⟩

lemma bin_set_ge: "bin_sc n True w ≥ w"
  ⟨proof⟩

lemma bintr_bin_clr_le: "bintrunc n (bin_sc m False w) ≤ bintrunc n
w"
  ⟨proof⟩

lemma bintr_bin_set_ge: "bintrunc n (bin_sc m True w) ≥ bintrunc n w"
  ⟨proof⟩

lemma bin_sc_FP [simp]: "bin_sc n False 0 = 0"
  ⟨proof⟩

lemma bin_sc_TM [simp]: "bin_sc n True (- 1) = - 1"
  ⟨proof⟩

lemmas bin_sc_simps = bin_sc.Z bin_sc.Suc bin_sc_TM bin_sc_FP

lemma bin_sc_minus: "0 < n ⟹ bin_sc (Suc (n - 1)) b w = bin_sc n b
w"
  ⟨proof⟩

lemmas bin_sc_Suc_minus =
  trans [OF bin_sc_minus [symmetric] bin_sc.Suc]

lemma bin_sc_numeral [simp]:
  "bin_sc (numeral k) b w =
    of_bool (odd w) + 2 * bin_sc (pred_numeral k) b (w div 2)"
  ⟨proof⟩

lemmas bin_sc_minus_simps =
  bin_sc_simps (2,3,4) [THEN [2] trans, OF bin_sc_minus [THEN sym]]

instance int :: semiring_bit_syntax ⟨proof⟩

lemma test_bit_int_def [iff]:
  "i !! n ⟷ bin_nth i n"
  ⟨proof⟩

lemma shiftl_int_def:
  "shiftl x n = x * 2 ^ n" for x :: int
  ⟨proof⟩
```

lemma shiftr_int_def:
"shiftr x n = x div 2 ^ n" **for** x :: int
⟨*proof*⟩

8.5.1 Basic simplification rules

lemmas int_not_def = not_int_def

lemma int_not_simps [simp]:
"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*⟩

8.5.2 Binary destructors

lemma bin_rest_NOT [simp]: "bin_rest (NOT x) = NOT (bin_rest x)"
⟨*proof*⟩

lemma bin_last_NOT [simp]: "bin_last (NOT x) ⟷ ¬ bin_last x"
⟨*proof*⟩

lemma bin_rest_AND [simp]: "bin_rest (x AND y) = bin_rest x AND bin_rest y"
⟨*proof*⟩

lemma bin_last_AND [simp]: "bin_last (x AND y) ⟷ bin_last x ∧ bin_last y"
⟨*proof*⟩

lemma bin_rest_OR [simp]: "bin_rest (x OR y) = bin_rest x OR bin_rest y"
⟨*proof*⟩

lemma bin_last_OR [simp]: "bin_last (x OR y) ⟷ bin_last x ∨ bin_last y"
⟨*proof*⟩

lemma bin_rest_XOR [simp]: "bin_rest (x XOR y) = bin_rest x XOR bin_rest y"
⟨*proof*⟩

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

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

8.5.3 Derived properties

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

8.5.4 Basic properties of logical (bit-wise) operations

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*⟩

8.5.5 Simplification with numerals

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

lemma bin_rest_neg_numeral_BitM [simp]:
  "bin_rest (- numeral (Num.BitM w)) = - numeral w"
  ⟨*proof*⟩

lemma bin_last_neg_numeral_BitM [simp]:
  "bin_last (- numeral (Num.BitM w))"
  ⟨*proof*⟩

8.5.6 Interactions with arithmetic

lemma le_int_or: "bin_sign y = 0 ⟹ x ≤ x OR y"
  for x y :: int
  ⟨*proof*⟩

lemmas int_and_le =
  xtrans(3) [OF bbw_ao_absorbs (2) [THEN conjunct2, symmetric] le_int_or]

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*⟩

8.5.7 Truncating results of bit-wise operations

lemma bin_trunc_ao:
  "bintrunc n x AND bintrunc n y = bintrunc n (x AND y)"
  "bintrunc n x OR bintrunc n y = bintrunc n (x OR y)"
  ⟨*proof*⟩

lemma bin_trunc_xor: "bintrunc n (bintrunc n x XOR bintrunc n y) = bintrunc n (x XOR y)"
  ⟨*proof*⟩

lemma bin_trunc_not: "bintrunc n (NOT (bintrunc n x)) = bintrunc n (NOT x)"
  ⟨*proof*⟩

Want theorems of the form of bin_trunc_xor.

lemma bintr_bintr_i: "x = bintrunc n y ⟹ bintrunc n x = bintrunc 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]

8.5.8 More lemmas

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:
"bin_last i ⟷ i AND 1 ≠ 0"
⟨*proof*⟩

lemma bitval_bin_last:
"of_bool (bin_last i) = i AND 1"
⟨*proof*⟩

lemma bin_sign_and:
"bin_sign (i AND j) = - (bin_sign i * bin_sign j)"
⟨*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*⟩

8.6 Setting and clearing bits

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

lemma int_0_shiftl [simp]: "0 << n = (0 :: int)"
⟨*proof*⟩

lemma bin_last_shiftl: "bin_last (x << n) ⟷ n = 0 ∧ bin_last x"
⟨*proof*⟩

lemma bin_rest_shiftl: "bin_rest (x << n) = (if n > 0 then x << (n - 1) else bin_rest x)"
⟨*proof*⟩

lemma bin_nth_shiftl [simp]: "bin_nth (x << n) m ⟷ n ≤ m ∧ bin_nth x (m - n)"
⟨*proof*⟩

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

lemma bin_rest_shiftr [simp]: "bin_rest (x >> n) = x >> Suc n"
⟨*proof*⟩

lemma bin_nth_shiftr [simp]: "bin_nth (x >> n) m = bin_nth x (n + m)"
⟨*proof*⟩

lemma bin_nth_conv_AND:
fixes x :: int **shows**
"bin_nth x n ⟷ x AND (1 << n) ≠ 0"

⟨*proof*⟩

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

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

lemma shiftl_ge_0 [simp]: **fixes** i :: int **shows** "i << n ≥ 0 ⟷ i ≥ 0"
⟨*proof*⟩

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

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

lemma int_0shiftr [simp]: "(0 :: int) >> x = 0"
⟨*proof*⟩

lemma int_minus1_shiftr [simp]: "(-1 :: int) >> x = -1"
⟨*proof*⟩

lemma int_shiftr_ge_0 [simp]: **fixes** i :: int **shows** "i >> n ≥ 0 ⟷ i ≥ 0"
⟨*proof*⟩

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

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

⟨*proof*⟩

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

lemma bin_nth_minus_p2:
assumes sign: "bin_sign x = 0"
and y: "y = 1 << n"
and m: "m < n"
and x: "x < y"
shows "bin_nth (x - y) m = bin_nth x m"
⟨*proof*⟩

lemma bin_clr_conv_NAND:
"bin_sc n False i = i AND NOT (1 << n)"
⟨*proof*⟩

lemma bin_set_conv_OR:
"bin_sc n True i = i OR (1 << n)"
⟨*proof*⟩

8.7 More lemmas on words

lemma word_rcat_eq:
‹word_rcat ws = word_of_int (bin_rcat (LENGTH('a::len)) (map uint ws))›
for ws :: ‹'a::len word list›
⟨*proof*⟩

lemma sign_uint_Pls [simp]: "bin_sign (uint x) = 0"
⟨*proof*⟩

lemmas bin_log_bintrs = bin_trunc_not bin_trunc_xor bin_trunc_and bin_trunc_or

— following definitions require both arithmetic and bit-wise word operations

— to get word_no_log_defs from word_log_defs, using bin_log_bintrs
lemmas wils1 = bin_log_bintrs [THEN word_of_int_eq_iff [THEN iffD2],
folded uint_word_of_int_eq, THEN eq_reflection]

— the binary operations only
lemmas word_log_binary_defs =
word_and_def word_or_def word_xor_def

lemma setBit_no [simp]: "setBit (numeral bin) n = word_of_int (bin_sc
n True (numeral bin))"
⟨*proof*⟩

lemma clearBit_no [simp]:
"clearBit (numeral bin) n = word_of_int (bin_sc n False (numeral bin))"
⟨*proof*⟩

lemma eq_mod_iff: "0 < n ⟹ b = b mod n ⟷ 0 ≤ b ∧ b < n"
for b n :: int
⟨*proof*⟩

lemma split_uint_lem: "bin_split n (uint w) = (a, b) ⟹
a = take_bit (LENGTH('a) - n) a ∧ b = take_bit (LENGTH('a)) b"
for w :: "'a::len word"
⟨*proof*⟩
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 (bin_cat w (size b) (uint b))"
⟨*proof*⟩

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

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

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

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

lemma and_bin_mask_conv_mod: "x AND mask n = x mod 2 ^ n"
for x :: int
⟨*proof*⟩

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

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

lemma bin_sign_mask [simp]: "bin_sign (mask n) = 0"
⟨*proof*⟩

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

lemma sbintrunc_eq_in_range:
"(sbintrunc n x = x) = (x ∈ range (sbintrunc n))"
"(x = sbintrunc n x) = (x ∈ range (sbintrunc n))"
⟨*proof*⟩

lemma sbintrunc_If:
"- 3 * (2 ^ n) ≤ x ∧ x < 3 * (2 ^ n)
⟹ sbintrunc n x = (if x < - (2 ^ n) then x + 2 * (2 ^ n)
else if x ≥ 2 ^ n then x - 2 * (2 ^ n) else x)"
⟨*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))"
"(sint a - sint b = sint (a - b))
= ((0 <=s a - b) = (b <=s a))"
"(- sint a = sint (- a)) = (0 <=s (- a) = (a <=s 0))"
⟨*proof*⟩

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

code_identifier
code_module Bits_Int ⇀
(SML) Bit_Operations **and** (OCaml) Bit_Operations **and** (Haskell) Bit_Operations
and (Scala) Bit_Operations

end

9 Type Definition Theorems

theory Typedef_Morphisms
  imports Main "HOL-Library.Word" Bit_Comprehension Bits_Int
begin

9.1 More lemmas about normal type definitions

lemma tdD1: "type_definition Rep Abs A ⟹ ∀x. Rep x ∈ A"
  and tdD2: "type_definition Rep Abs A ⟹ ∀x. Abs (Rep x) = x"
  and tdD3: "type_definition Rep Abs A ⟹ ∀y. y ∈ A ⟶ Rep (Abs y)
= y"
  ⟨*proof*⟩

lemma td_nat_int: "type_definition int nat (Collect ((≤) 0))"
  ⟨*proof*⟩

context type_definition
begin

declare Rep [iff] Rep_inverse [simp] Rep_inject [simp]

lemma Abs_eqD: "Abs x = Abs y ⟹ x ∈ A ⟹ y ∈ A ⟹ x = y"
  ⟨*proof*⟩

lemma Abs_inverse': "r ∈ A ⟹ Abs r = a ⟹ Rep a = r"
  ⟨*proof*⟩

lemma Rep_comp_inverse: "Rep ∘ f = g ⟹ Abs ∘ g = f"
  ⟨*proof*⟩

lemma Rep_eqD [elim!]: "Rep x = Rep y ⟹ x = y"
  ⟨*proof*⟩

lemma Rep_inverse': "Rep a = r ⟹ Abs r = a"
  ⟨*proof*⟩

lemma comp_Abs_inverse: "f ∘ Abs = g ⟹ g ∘ Rep = f"
  ⟨*proof*⟩

lemma set_Rep: "A = range Rep"
⟨*proof*⟩

lemma set_Rep_Abs: "A = range (Rep ∘ Abs)"
⟨*proof*⟩

lemma Abs_inj_on: "inj_on Abs A"
  ⟨*proof*⟩

lemma image: "Abs ` A = UNIV"

⟨*proof*⟩

lemmas td_thm = type_definition_axioms

lemma fns1: "Rep ∘ fa = fr ∘ Rep ∨ fa ∘ Abs = Abs ∘ fr ⟹ Abs ∘ fr ∘ Rep = fa"
⟨*proof*⟩

lemmas fns1a = disjI1 [THEN fns1]
lemmas fns1b = disjI2 [THEN fns1]

lemma fns4: "Rep ∘ fa ∘ Abs = fr ⟹ Rep ∘ fa = fr ∘ Rep ∧ fa ∘ Abs = Abs ∘ fr"
⟨*proof*⟩

end

interpretation nat_int: type_definition int nat "Collect ((≤) 0)"
⟨*proof*⟩

declare
nat_int.Rep_cases [cases del]
nat_int.Abs_cases [cases del]
nat_int.Rep_induct [induct del]
nat_int.Abs_induct [induct del]

9.2 Extended form of type definition predicate

lemma td_conds:
"norm ∘ norm = norm ⟹
fr ∘ norm = norm ∘ fr ⟷ norm ∘ fr ∘ norm = fr ∘ norm ∧ norm ∘ fr ∘ norm = norm ∘ fr"
⟨*proof*⟩

lemma fn_comm_power: "fa ∘ tr = tr ∘ fr ⟹ fa ^^ n ∘ tr = tr ∘ fr ^^ n"
⟨*proof*⟩

lemmas fn_comm_power' =
ext [THEN fn_comm_power, THEN fun_cong, unfolded o_def]

locale td_ext = type_definition +
fixes norm
assumes eq_norm: "⋀x. Rep (Abs x) = norm x"
begin

lemma Abs_norm [simp]: "Abs (norm x) = Abs x"
⟨*proof*⟩

lemma td_th: "g ∘ Abs = f ⟹ f (Rep x) = g x"
⟨*proof*⟩

lemma eq_norm': "Rep ∘ Abs = norm"
⟨*proof*⟩

lemma norm_Rep [simp]: "norm (Rep x) = Rep x"
⟨*proof*⟩

lemmas td = td_thm

lemma set_iff_norm: "w ∈ A ⟷ w = norm w"
⟨*proof*⟩

lemma inverse_norm: "Abs n = w ⟷ Rep w = norm n"
⟨*proof*⟩

lemma norm_eq_iff: "norm x = norm y ⟷ Abs x = Abs y"
⟨*proof*⟩

lemma norm_comps:
"Abs ∘ norm = Abs"
"norm ∘ Rep = Rep"
"norm ∘ norm = norm"
⟨*proof*⟩

lemmas norm_norm [simp] = norm_comps

lemma fns5: "Rep ∘ fa ∘ Abs = fr ⟹ fr ∘ norm = fr ∧ norm ∘ fr = fr"
⟨*proof*⟩

following give conditions for converses to td_fns1

- the condition norm ∘ fr ∘ norm = fr ∘ norm says that fr takes normalised arguments to normalised results
- norm ∘ fr ∘ norm = norm ∘ fr says that fr takes norm-equivalent arguments to norm-equivalent results
- fr ∘ norm = fr says that fr takes norm-equivalent arguments to the same result
- norm ∘ fr = fr says that fr takes any argument to a normalised result

lemma fns2: "Abs ∘ fr ∘ Rep = fa ⟹ norm ∘ fr ∘ norm = fr ∘ norm ⟷ Rep ∘ fa = fr ∘ Rep"
⟨*proof*⟩

lemma fns3: "Abs ∘ fr ∘ Rep = fa ⟹ norm ∘ fr ∘ norm = norm ∘ fr ⟷ fa ∘ Abs = Abs ∘ fr"
⟨*proof*⟩

lemma fns: "fr ∘ norm = norm ∘ fr ⟹ fa ∘ Abs = Abs ∘ fr ⟷ Rep ∘ fa = fr ∘ Rep"
⟨*proof*⟩

lemma range_norm: "range (Rep ∘ Abs) = A"
⟨*proof*⟩

end

lemmas td_ext_def' =
td_ext_def [unfolded type_definition_def td_ext_axioms_def]

9.3 Type-definition locale instantiations

definition uints :: "nat ⇒ int set"
— the sets of integers representing the words
where "uints n = range (take_bit n)"

definition sints :: "nat ⇒ int set"
where "sints n = range (signed_take_bit (n - 1))"

lemma uints_num: "uints n = {i. 0 ≤ i ∧ i < 2 ^ n}"
⟨*proof*⟩

lemma sints_num: "sints n = {i. - (2 ^ (n - 1)) ≤ i ∧ i < 2 ^ (n - 1)}"
⟨*proof*⟩

definition unats :: "nat ⇒ nat set"
where "unats n = {i. i < 2 ^ n}"

— naturals
lemma uints_unats: "uints n = int ` unats n"
⟨*proof*⟩

lemma unats_uints: "unats n = nat ` uints n"
⟨*proof*⟩

lemma td_ext_uint:
"td_ext (uint :: 'a word ⇒ int) word_of_int (uints (LENGTH('a::len)))
(λw::int. w mod 2 ^ LENGTH('a))"
⟨*proof*⟩

interpretation word_uint:
td_ext
"uint::'a::len word ⇒ int"

```
    word_of_int
    "uints (LENGTH('a::len))"
    "λw. w mod 2 ^ LENGTH('a::len)"
  ⟨proof⟩

lemmas td_uint = word_uint.td_thm
lemmas int_word_uint = word_uint.eq_norm

lemma td_ext_ubin:
  "td_ext (uint :: 'a word ⇒ int) word_of_int (uints (LENGTH('a::len)))
    (take_bit (LENGTH('a)))"
  ⟨proof⟩

interpretation word_ubin:
  td_ext
    "uint::'a::len word ⇒ int"
    word_of_int
    "uints (LENGTH('a::len))"
    "take_bit (LENGTH('a::len))"
  ⟨proof⟩

lemma td_ext_unat [OF refl]:
  "n = LENGTH('a::len) ⟹
    td_ext (unat :: 'a word ⇒ nat) of_nat (unats n) (λi. i mod 2 ^ n)"
  ⟨proof⟩

lemmas unat_of_nat = td_ext_unat [THEN td_ext.eq_norm]

interpretation word_unat:
  td_ext
    "unat::'a::len word ⇒ nat"
    of_nat
    "unats (LENGTH('a::len))"
    "λi. i mod 2 ^ LENGTH('a::len)"
  ⟨proof⟩

lemmas td_unat = word_unat.td_thm

lemma unat_le: "y ≤ unat z ⟹ y ∈ unats (LENGTH('a))"
  for z :: "'a::len word"
  ⟨proof⟩

lemma td_ext_sbin:
  "td_ext (sint :: 'a word ⇒ int) word_of_int (sints (LENGTH('a::len)))
    (signed_take_bit (LENGTH('a) - 1))"
  ⟨proof⟩

lemma td_ext_sint:
  "td_ext (sint :: 'a word ⇒ int) word_of_int (sints (LENGTH('a::len)))
```

```
    (λw. (w + 2 ^ (LENGTH('a) - 1)) mod 2 ^ LENGTH('a) -
       2 ^ (LENGTH('a) - 1))"
```
⟨*proof*⟩

We do `sint` before `sbin`, before `sint` is the user version and interpretations do not produce thm duplicates. I.e. we get the name `word_sint.Rep_eqD`, but not `word_sbin.Req_eqD`, because the latter is the same thm as the former.

interpretation `word_sint`:
```
  td_ext
    "sint ::'a::len word ⇒ int"
    word_of_int
    "sints (LENGTH('a::len))"
    "λw. (w + 2^(LENGTH('a::len) - 1)) mod 2^LENGTH('a::len) -
      2 ^ (LENGTH('a::len) - 1)"
```
⟨*proof*⟩

interpretation `word_sbin`:
```
  td_ext
    "sint ::'a::len word ⇒ int"
    word_of_int
    "sints (LENGTH('a::len))"
    "signed_take_bit (LENGTH('a::len) - 1)"
```
⟨*proof*⟩

lemmas `int_word_sint = td_ext_sint [THEN td_ext.eq_norm]`

lemmas `td_sint = word_sint.td`

lemma `uints_mod: "uints n = range (λw. w mod 2 ^ n)"`
⟨*proof*⟩

lemmas `bintr_num =`
`  word_ubin.norm_eq_iff [of "numeral a" "numeral b", symmetric, folded word_numeral_alt]` **for** `a b`
lemmas `sbintr_num =`
`  word_sbin.norm_eq_iff [of "numeral a" "numeral b", symmetric, folded word_numeral_alt]` **for** `a b`

lemmas `uint_div_alt = word_div_def [THEN trans [OF uint_cong int_word_uint]]`
lemmas `uint_mod_alt = word_mod_def [THEN trans [OF uint_cong int_word_uint]]`

interpretation `test_bit`:
```
  td_ext
    "(!!) :: 'a::len word ⇒ nat ⇒ bool"
    set_bits
    "{f. ∀i. f i ⟶ i < LENGTH('a::len)}"
    "(λh i. h i ∧ i < LENGTH('a::len))"
```
⟨*proof*⟩

lemmas td_nth = test_bit.td_thm

lemma sints_subset:
"m ≤ n ⟹ sints m ⊆ sints n"
⟨*proof*⟩

end

10 Word Alignment

theory Aligned
imports
"HOL-Library.Word"
More_Word
Word_EqI
Typedef_Morphisms
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 word_mod_by_0: "k mod (0::'a::len word) = k"
⟨*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"
shows "⟦ is_aligned p n; ⟦ is_aligned p n; n < LENGTH('a) ⟧ ⟹ P;
⟦ p = 0; n ≥ LENGTH('a) ⟧ ⟹ P ⟧ ⟹ P"
⟨*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:
"⟦is_aligned x n; is_aligned y n; x < y⟧ ⟹ 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_nth [word_eqI_simps]:
"is_aligned p m = (∀n < m. ¬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:
"⟦is_aligned (x :: 'a :: len word) n;
∀n'. n' ≤ n. n' < LENGTH('a) ⟶ ¬ p !! n'⟧ ⟹ 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:
"is_aligned x n ⟹ ((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 [word_eqI_simps]:
"NOT(mask n) ≤ w ⟷ (∀i ∈ {n ..< size w}. w !! i)"
for w :: ‹'a::len word›
⟨*proof*⟩

lemma is_aligned_add_less_t2n:
"⟦is_aligned (p::'a::len word) n; d < 2^n; n ≤ m; p < 2^m⟧ ⟹ 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:
"⟦ n' ≤ n; p' ≤ p + mask n; is_aligned p n; is_aligned p' n' ⟧ ⟹
(p'::'a::len word) + 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:
"⟦ is_aligned x n; n > 0 ⟧ ⟹ x < x + 1"
⟨*proof*⟩

lemma aligned_add_offset_less:
"⟦is_aligned x n; is_aligned y n; x < y; z < 2 ^ n⟧ ⟹ 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"
and s10: "0 < s1"
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 "%x. 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:
"⟦x < 2 ^ n; is_aligned (y :: 'a :: len word) n;n ≤ LENGTH('a)⟧
⟹ 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:
"⟦is_aligned ptr m; m ≤ n; a < 2 ^ m⟧
⟹ ptr + a AND mask n = (ptr AND mask n) + a"
⟨*proof*⟩

end

11 Operation variant for the least significant bit

theory Least_significant_bit
imports
"HOL-Library.Word"
Bits_Int
begin

class lsb = semiring_bits +
fixes lsb :: ‹'a ⇒ bool›
assumes lsb_odd: ‹lsb = odd›

instantiation int :: lsb
begin

definition lsb_int :: ‹int ⇒ bool›

where ‹lsb i = i !! 0› **for** i :: int

instance
⟨*proof*⟩

end

lemma bin_last_conv_lsb: "bin_last = lsb"
⟨*proof*⟩

lemma int_lsb_numeral [simp]:
"lsb (0 :: int) = False"
"lsb (1 :: int) = True"
"lsb (Numeral1 :: int) = True"
"lsb (- 1 :: int) = True"
"lsb (- Numeral1 :: int) = True"
"lsb (numeral (num.Bit0 w) :: int) = False"
"lsb (numeral (num.Bit1 w) :: int) = True"
"lsb (- numeral (num.Bit0 w) :: int) = False"
"lsb (- numeral (num.Bit1 w) :: int) = True"
⟨*proof*⟩

instantiation word :: (len) lsb
begin

definition lsb_word :: ‹'a word ⇒ bool›
where word_lsb_def: ‹lsb a ⟷ odd (uint a)› **for** a :: ‹'a word›

instance
⟨*proof*⟩

end

lemma lsb_word_eq:
‹lsb = (odd :: 'a word ⇒ bool)› **for** w :: ‹'a::len word›
⟨*proof*⟩

lemma word_lsb_alt: "lsb w = test_bit w 0"
for w :: "'a::len word"
⟨*proof*⟩

lemma word_lsb_1_0 [simp]: "lsb (1::'a::len word) ∧ ¬ lsb (0::'b::len word)"
⟨*proof*⟩

lemma word_lsb_int: "lsb w ⟷ uint w mod 2 = 1"
⟨*proof*⟩

lemmas word_ops_lsb = lsb0 [unfolded word_lsb_alt]

lemma word_lsb_numeral [simp]:
"lsb (numeral bin :: 'a::len word) ⟷ bin_last (numeral bin)"
⟨*proof*⟩

lemma word_lsb_neg_numeral [simp]:
"lsb (- numeral bin :: 'a::len word) ⟷ bin_last (- numeral bin)"
⟨*proof*⟩

lemma word_lsb_nat:"lsb w = (unat w mod 2 = 1)"
⟨*proof*⟩

end

12 Dedicated operation for the most significant bit

theory Most_significant_bit
imports
"HOL-Library.Word"
Bits_Int
Traditional_Infix_Syntax
More_Arithmetic
begin

class msb =
fixes msb :: ‹'a ⇒ bool›

instantiation int :: msb
begin

definition ‹msb x ⟷ x < 0› **for** x :: int

instance ⟨*proof*⟩

end

lemma msb_conv_bin_sign: "msb x ⟷ bin_sign x = -1"
⟨*proof*⟩

lemma msb_bin_rest [simp]: "msb (x div 2) = msb x"
for x :: int
⟨*proof*⟩

lemma int_msb_and [simp]: "msb ((x :: int) AND y) ⟷ msb x ∧ msb y"
⟨*proof*⟩

lemma int_msb_or [simp]: "msb ((x :: int) OR y) ⟷ msb x ∨ msb y"
⟨*proof*⟩

lemma int_msb_xor [simp]: "msb ((x :: int) XOR y) ⟷ msb x ≠ msb y"
⟨*proof*⟩

lemma int_msb_not [simp]: "msb (NOT (x :: int)) ⟷ ¬ msb x"
⟨*proof*⟩

lemma msb_shiftl [simp]: "msb ((x :: int) << n) ⟷ msb x"
⟨*proof*⟩

lemma msb_shiftr [simp]: "msb ((x :: int) >> r) ⟷ msb x"
⟨*proof*⟩

lemma msb_bin_sc [simp]: "msb (bin_sc n b x) ⟷ msb x"
⟨*proof*⟩

lemma msb_0 [simp]: "msb (0 :: int) = False"
⟨*proof*⟩

lemma msb_1 [simp]: "msb (1 :: int) = False"
⟨*proof*⟩

lemma msb_numeral [simp]:
"msb (numeral n :: int) = False"
"msb (- numeral n :: int) = True"
⟨*proof*⟩

instantiation word :: (len) msb
begin

definition msb_word :: ⟨'a word ⇒ bool⟩
where ⟨msb a ⟷ bin_sign (sbintrunc (LENGTH('a) - 1) (uint a)) = -
1⟩

lemma msb_word_eq:
⟨msb w ⟷ bit w (LENGTH('a) - 1)⟩ **for** w :: ⟨'a::len word⟩
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma msb_word_iff_bit:
⟨msb w ⟷ bit w (LENGTH('a) - Suc 0)⟩
for w :: ⟨'a::len word⟩
⟨*proof*⟩

lemma word_msb_def:
"msb a ⟷ bin_sign (sint a) = - 1"
⟨*proof*⟩

lemma word_msb_sint: "msb w ⟷ sint w < 0"
⟨*proof*⟩

lemma msb_word_iff_sless_0:
‹msb w ⟷ w <s 0›
⟨*proof*⟩

lemma msb_word_of_int: "msb (word_of_int x::'a::len word) = bin_nth x
(LENGTH('a) - 1)"
⟨*proof*⟩

lemma word_msb_numeral [simp]:
"msb (numeral w::'a::len word) = bin_nth (numeral w) (LENGTH('a) - 1)"
⟨*proof*⟩

lemma word_msb_neg_numeral [simp]:
"msb (- numeral w::'a::len word) = bin_nth (- numeral w) (LENGTH('a)
- 1)"
⟨*proof*⟩

lemma word_msb_0 [simp]: "¬ msb (0::'a::len word)"
⟨*proof*⟩

lemma word_msb_1 [simp]: "msb (1::'a::len word) ⟷ LENGTH('a) = 1"
⟨*proof*⟩

lemma word_msb_nth: "msb w = bin_nth (uint w) (LENGTH('a) - 1)"
for w :: "'a::len word"
⟨*proof*⟩

lemma msb_nth: "msb w = w !! (LENGTH('a) - 1)"
for w :: "'a::len word"
⟨*proof*⟩

lemma word_msb_n1 [simp]: "msb (-1::'a::len word)"
⟨*proof*⟩

lemma msb_shift: "msb w ⟷ w >> (LENGTH('a) - 1) ≠ 0"
for w :: "'a::len word"
⟨*proof*⟩

lemmas word_ops_msb = msb1 [unfolded msb_nth [symmetric, unfolded One_nat_def]]

lemma word_sint_msb_eq: "sint x = uint x - (if msb x then 2 ^ size x
else 0)"
⟨*proof*⟩

lemma word_sle_msb_le: "x <=s y ⟷ (msb y ⟶ msb x) ∧ ((msb x ∧ ¬

```
msb y) ∨ x ≤ y)"
```
⟨*proof*⟩

lemma word_sless_msb_less: "x <s y ⟷ (msb y ⟶ msb x) ∧ ((msb x ∧ ¬ msb y) ∨ x < y)"
⟨*proof*⟩

lemma not_msb_from_less:
"(v :: 'a word) < 2 ^ (LENGTH('a :: len) - 1) ⟹ ¬ msb v"
⟨*proof*⟩

lemma sint_eq_uint:
"¬ msb x ⟹ sint x = uint x"
⟨*proof*⟩

lemma scast_eq_ucast:
"¬ msb x ⟹ scast x = ucast x"
⟨*proof*⟩

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

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

end

13 Lemmas on list operations

theory Even_More_List
imports Main
begin

lemma upt_add_eq_append':
assumes "i ≤ j" **and** "j ≤ k"
shows "[i..<k] = [i..<j] @ [j..<k]"
⟨*proof*⟩

lemma map_idem_upt_eq:
‹map f [m..<n] = [m..<n]› **if** ‹⋀q. m ≤ q ⟹ q < n ⟹ f q = q›
⟨*proof*⟩

lemma upt_zero_numeral_unfold:
‹[0..<numeral n] = [0..<pred_numeral n] @ [pred_numeral n]›
⟨*proof*⟩

lemma length_takeWhile_less:
"∃x∈set xs. ¬ P x ⟹ length (takeWhile P xs) < length xs"
⟨*proof*⟩

lemma Min_eq_length_takeWhile:
⟨Min {m. P m} = length (takeWhile (Not ∘ P) ([0..<n]))⟩
if ∗: ⟨⋀m. P m ⟹ m < n⟩ **and** ⟨∃m. P m⟩
⟨*proof*⟩

lemma Max_eq_length_takeWhile:
⟨Max {m. P m} = n - Suc (length (takeWhile (Not ∘ P) (rev [0..<n])))⟩
if ∗: ⟨⋀m. P m ⟹ m < n⟩ **and** ⟨∃m. P m⟩
⟨*proof*⟩

end

14 Bit values as reversed lists of bools

theory Reversed_Bit_Lists
imports
"HOL-Library.Word"
Typedef_Morphisms
Least_significant_bit
Most_significant_bit
Even_More_List
"HOL-Library.Sublist"
Aligned
begin

lemma horner_sum_of_bool_2_concat:
⟨horner_sum of_bool 2 (concat (map (λx. map (bit x) [0..<LENGTH('a)]) ws)) = horner_sum uint (2 ^ LENGTH('a)) ws⟩
for ws :: ⟨'a::len word list⟩
⟨*proof*⟩

14.1 Implicit augmentation of list prefixes

primrec takefill :: "'a ⇒ nat ⇒ 'a list ⇒ 'a list"
where
Z: "takefill fill 0 xs = []"
| Suc: "takefill fill (Suc n) xs =
(case xs of
[] ⇒ fill # takefill fill n xs
| y # ys ⇒ y # takefill fill n ys)"

lemma nth_takefill: "m < n ⟹ takefill fill n l ! m = (if m < length l then l ! m else fill)"
⟨*proof*⟩

lemma takefill_alt: "takefill fill n l = take n l @ replicate (n - length l) fill"
⟨*proof*⟩

lemma takefill_replicate [simp]: "takefill fill n (replicate m fill) = replicate n fill"
⟨*proof*⟩

lemma takefill_le': "n = m + k ⟹ takefill x m (takefill x n l) = takefill x m l"
⟨*proof*⟩

lemma length_takefill [simp]: "length (takefill fill n l) = n"
⟨*proof*⟩

lemma take_takefill': "n = k + m ⟹ take k (takefill fill n w) = takefill fill k w"
⟨*proof*⟩

lemma drop_takefill: "drop k (takefill fill (m + k) w) = takefill fill m (drop k w)"
⟨*proof*⟩

lemma takefill_le [simp]: "m ≤ n ⟹ takefill x m (takefill x n l) = takefill x m l"
⟨*proof*⟩

lemma take_takefill [simp]: "m ≤ n ⟹ take m (takefill fill n w) = takefill fill m w"
⟨*proof*⟩

lemma takefill_append: "takefill fill (m + length xs) (xs @ w) = xs @ (takefill fill m w)"
⟨*proof*⟩

lemma takefill_same': "l = length xs ⟹ takefill fill l xs = xs"
⟨*proof*⟩

lemmas takefill_same [simp] = takefill_same' [OF refl]

lemma tf_rev:
"n + k = m + length bl ⟹ takefill x m (rev (takefill y n bl)) =
rev (takefill y m (rev (takefill x k (rev bl))))"
⟨*proof*⟩

lemma takefill_minus: "0 < n ⟹ takefill fill (Suc (n - 1)) w = takefill fill n w"
⟨*proof*⟩

lemmas takefill_Suc_cases =
  list.cases [THEN takefill.Suc [THEN trans]]

lemmas takefill_Suc_Nil = takefill_Suc_cases (1)
lemmas takefill_Suc_Cons = takefill_Suc_cases (2)

lemmas takefill_minus_simps = takefill_Suc_cases [THEN [2]
  takefill_minus [symmetric, THEN trans]]

lemma takefill_numeral_Nil [simp]:
  "takefill fill (numeral k) [] = fill # takefill fill (pred_numeral k)
[]"
  ⟨*proof*⟩

lemma takefill_numeral_Cons [simp]:
  "takefill fill (numeral k) (x # xs) = x # takefill fill (pred_numeral
k) xs"
  ⟨*proof*⟩

14.2 Range projection

definition bl_of_nth :: "nat ⇒ (nat ⇒ 'a) ⇒ 'a list"
  where "bl_of_nth n f = map f (rev [0..<n])"

lemma bl_of_nth_simps [simp, code]:
  "bl_of_nth 0 f = []"
  "bl_of_nth (Suc n) f = f n # bl_of_nth n f"
  ⟨*proof*⟩

lemma length_bl_of_nth [simp]: "length (bl_of_nth n f) = n"
  ⟨*proof*⟩

lemma nth_bl_of_nth [simp]: "m < n ⟹ rev (bl_of_nth n f) ! m = f m"
  ⟨*proof*⟩

lemma bl_of_nth_inj: "(⋀k. k < n ⟹ f k = g k) ⟹ bl_of_nth n f =
bl_of_nth n g"
  ⟨*proof*⟩

lemma bl_of_nth_nth_le: "n ≤ length xs ⟹ bl_of_nth n (nth (rev xs))
= drop (length xs - n) xs"
  ⟨*proof*⟩

lemma bl_of_nth_nth [simp]: "bl_of_nth (length xs) ((!) (rev xs)) = xs"
  ⟨*proof*⟩

14.3 More

definition rotater1 :: "'a list ⇒ 'a list"
  where "rotater1 ys =

```
    (case ys of [] ⇒ [] | x # xs ⇒ last ys # butlast ys)"

definition rotater :: "nat ⇒ 'a list ⇒ 'a list"
  where "rotater n = rotater1 ^^ n"

lemmas rotater_0' [simp] = rotater_def [where n = "0", simplified]

lemma rotate1_rl': "rotater1 (l @ [a]) = a # l"
  ⟨proof⟩

lemma rotate1_rl [simp] : "rotater1 (rotate1 l) = l"
  ⟨proof⟩

lemma rotate1_lr [simp] : "rotate1 (rotater1 l) = l"
  ⟨proof⟩

lemma rotater1_rev': "rotater1 (rev xs) = rev (rotate1 xs)"
  ⟨proof⟩

lemma rotater_rev': "rotater n (rev xs) = rev (rotate n xs)"
  ⟨proof⟩

lemma rotater_rev: "rotater n ys = rev (rotate n (rev ys))"
  ⟨proof⟩

lemma rotater_drop_take:
  "rotater n xs =
    drop (length xs - n mod length xs) xs @
    take (length xs - n mod length xs) xs"
  ⟨proof⟩

lemma rotater_Suc [simp]: "rotater (Suc n) xs = rotater1 (rotater n xs)"
  ⟨proof⟩

lemma nth_rotater:
  ‹rotater m xs ! n = xs ! ((n + (length xs - m mod length xs)) mod length
xs)› if ‹n < length xs›
  ⟨proof⟩

lemma nth_rotater1:
  ‹rotater1 xs ! n = xs ! ((n + (length xs - 1)) mod length xs)› if ‹n <
length xs›
  ⟨proof⟩

lemma rotate_inv_plus [rule_format]:
  "∀k. k = m + n ⟶ rotater k (rotate n xs) = rotater m xs ∧
    rotate k (rotater n xs) = rotate m xs ∧
    rotater n (rotate k xs) = rotate m xs ∧
    rotate n (rotater k xs) = rotater m xs"
```

⟨*proof*⟩

lemmas rotate_inv_rel = le_add_diff_inverse2 [symmetric, THEN rotate_inv_plus]

lemmas rotate_inv_eq = order_refl [THEN rotate_inv_rel, simplified]

lemmas rotate_lr [simp] = rotate_inv_eq [THEN conjunct1]
lemmas rotate_rl [simp] = rotate_inv_eq [THEN conjunct2, THEN conjunct1]

lemma rotate_gal: "rotater n xs = ys ⟷ rotate n ys = xs"
⟨*proof*⟩

lemma rotate_gal': "ys = rotater n xs ⟷ xs = rotate n ys"
⟨*proof*⟩

lemma length_rotater [simp]: "length (rotater n xs) = length xs"
⟨*proof*⟩

lemma rotate_eq_mod: "m mod length xs = n mod length xs ⟹ rotate m
xs = rotate n xs"
⟨*proof*⟩

lemma restrict_to_left: "x = y ⟹ x = z ⟷ y = z"
⟨*proof*⟩

lemmas rotate_eqs =
trans [OF rotate0 [THEN fun_cong] id_apply]
rotate_rotate [symmetric]
rotate_id
rotate_conv_mod
rotate_eq_mod

lemmas rrs0 = rotate_eqs [THEN restrict_to_left,
simplified rotate_gal [symmetric] rotate_gal' [symmetric]]
lemmas rrs1 = rrs0 [THEN refl [THEN rev_iffD1]]
lemmas rotater_eqs = rrs1 [simplified length_rotater]
lemmas rotater_0 = rotater_eqs (1)
lemmas rotater_add = rotater_eqs (2)

lemma butlast_map: "xs ≠ [] ⟹ butlast (map f xs) = map f (butlast
xs)"
⟨*proof*⟩

lemma rotater1_map: "rotater1 (map f xs) = map f (rotater1 xs)"
⟨*proof*⟩

lemma rotater_map: "rotater n (map f xs) = map f (rotater n xs)"
⟨*proof*⟩

lemma but_last_zip [rule_format] :
"∀ys. length xs = length ys ⟶ xs ≠ [] ⟶
last (zip xs ys) = (last xs, last ys) ∧
butlast (zip xs ys) = zip (butlast xs) (butlast ys)"
⟨*proof*⟩

lemma but_last_map2 [rule_format] :
"∀ys. length xs = length ys ⟶ xs ≠ [] ⟶
last (map2 f xs ys) = f (last xs) (last ys) ∧
butlast (map2 f xs ys) = map2 f (butlast xs) (butlast ys)"
⟨*proof*⟩

lemma rotater1_zip:
"length xs = length ys ⟹
rotater1 (zip xs ys) = zip (rotater1 xs) (rotater1 ys)"
⟨*proof*⟩

lemma rotater1_map2:
"length xs = length ys ⟹
rotater1 (map2 f xs ys) = map2 f (rotater1 xs) (rotater1 ys)"
⟨*proof*⟩

lemmas lrth =
box_equals [OF asm_rl length_rotater [symmetric]
length_rotater [symmetric],
THEN rotater1_map2]

lemma rotater_map2:
"length xs = length ys ⟹
rotater n (map2 f xs ys) = map2 f (rotater n xs) (rotater n ys)"
⟨*proof*⟩

lemma rotate1_map2:
"length xs = length ys ⟹
rotate1 (map2 f xs ys) = map2 f (rotate1 xs) (rotate1 ys)"
⟨*proof*⟩

lemmas lth = box_equals [OF asm_rl length_rotate [symmetric]
length_rotate [symmetric], THEN rotate1_map2]

lemma rotate_map2:
"length xs = length ys ⟹
rotate n (map2 f xs ys) = map2 f (rotate n xs) (rotate n ys)"
⟨*proof*⟩

14.4 Explicit bit representation of int

primrec bl_to_bin_aux :: "bool list ⇒ int ⇒ int"
where

```
    Nil: "bl_to_bin_aux [] w = w"
  | Cons: "bl_to_bin_aux (b # bs) w = bl_to_bin_aux bs (of_bool b + 2
* w)"

definition bl_to_bin :: "bool list ⇒ int"
  where "bl_to_bin bs = bl_to_bin_aux bs 0"

primrec bin_to_bl_aux :: "nat ⇒ int ⇒ bool list ⇒ bool list"
  where
    Z: "bin_to_bl_aux 0 w bl = bl"
  | Suc: "bin_to_bl_aux (Suc n) w bl = bin_to_bl_aux n (bin_rest w) ((bin_last
w) # bl)"

definition bin_to_bl :: "nat ⇒ int ⇒ bool list"
  where "bin_to_bl n w = bin_to_bl_aux n w []"

lemma bin_to_bl_aux_zero_minus_simp [simp]:
  "0 < n ⟹ bin_to_bl_aux n 0 bl = bin_to_bl_aux (n - 1) 0 (False # bl)"
  ⟨proof⟩

lemma bin_to_bl_aux_minus1_minus_simp [simp]:
  "0 < n ⟹ bin_to_bl_aux n (- 1) bl = bin_to_bl_aux (n - 1) (- 1) (True
# bl)"
  ⟨proof⟩

lemma bin_to_bl_aux_one_minus_simp [simp]:
  "0 < n ⟹ bin_to_bl_aux n 1 bl = bin_to_bl_aux (n - 1) 0 (True # bl)"
  ⟨proof⟩

lemma bin_to_bl_aux_Bit0_minus_simp [simp]:
  "0 < n ⟹
    bin_to_bl_aux n (numeral (Num.Bit0 w)) bl = bin_to_bl_aux (n - 1)
(numeral w) (False # bl)"
  ⟨proof⟩

lemma bin_to_bl_aux_Bit1_minus_simp [simp]:
  "0 < n ⟹
    bin_to_bl_aux n (numeral (Num.Bit1 w)) bl = bin_to_bl_aux (n - 1)
(numeral w) (True # bl)"
  ⟨proof⟩

lemma bl_to_bin_aux_append: "bl_to_bin_aux (bs @ cs) w = bl_to_bin_aux
cs (bl_to_bin_aux bs w)"
  ⟨proof⟩

lemma bin_to_bl_aux_append: "bin_to_bl_aux n w bs @ cs = bin_to_bl_aux
n w (bs @ cs)"
  ⟨proof⟩
```

lemma bl_to_bin_append: "bl_to_bin (bs @ cs) = bl_to_bin_aux cs (bl_to_bin bs)"
⟨*proof*⟩

lemma bin_to_bl_aux_alt: "bin_to_bl_aux n w bs = bin_to_bl n w @ bs"
⟨*proof*⟩

lemma bin_to_bl_0 [simp]: "bin_to_bl 0 bs = []"
⟨*proof*⟩

lemma size_bin_to_bl_aux: "length (bin_to_bl_aux n w bs) = n + length bs"
⟨*proof*⟩

lemma size_bin_to_bl [simp]: "length (bin_to_bl n w) = n"
⟨*proof*⟩

lemma bl_bin_bl': "bin_to_bl (n + length bs) (bl_to_bin_aux bs w) = bin_to_bl_aux n w bs"
⟨*proof*⟩

lemma bl_bin_bl [simp]: "bin_to_bl (length bs) (bl_to_bin bs) = bs"
⟨*proof*⟩

lemma bl_to_bin_inj: "bl_to_bin bs = bl_to_bin cs ⟹ length bs = length cs ⟹ bs = cs"
⟨*proof*⟩

lemma bl_to_bin_False [simp]: "bl_to_bin (False # bl) = bl_to_bin bl"
⟨*proof*⟩

lemma bl_to_bin_Nil [simp]: "bl_to_bin [] = 0"
⟨*proof*⟩

lemma bin_to_bl_zero_aux: "bin_to_bl_aux n 0 bl = replicate n False @ bl"
⟨*proof*⟩

lemma bin_to_bl_zero: "bin_to_bl n 0 = replicate n False"
⟨*proof*⟩

lemma bin_to_bl_minus1_aux: "bin_to_bl_aux n (- 1) bl = replicate n True @ bl"
⟨*proof*⟩

lemma bin_to_bl_minus1: "bin_to_bl n (- 1) = replicate n True"
⟨*proof*⟩

14.5 Semantic interpretation of `bool list` as `int`

lemma `bin_bl_bin': "bl_to_bin (bin_to_bl_aux n w bs) = bl_to_bin_aux bs (bintrunc n w)"`
⟨*proof*⟩

lemma `bin_bl_bin [simp]: "bl_to_bin (bin_to_bl n w) = bintrunc n w"`
⟨*proof*⟩

lemma `bl_to_bin_rep_F: "bl_to_bin (replicate n False @ bl) = bl_to_bin bl"`
⟨*proof*⟩

lemma `bin_to_bl_trunc [simp]: "n ≤ m ⟹ bin_to_bl n (bintrunc m w) = bin_to_bl n w"`
⟨*proof*⟩

lemma `bin_to_bl_aux_bintr:`
```
"bin_to_bl_aux n (bintrunc m bin) bl =
  replicate (n - m) False @ bin_to_bl_aux (min n m) bin bl"
```
⟨*proof*⟩

lemma `bin_to_bl_bintr:`
`"bin_to_bl n (bintrunc m bin) = replicate (n - m) False @ bin_to_bl (min n m) bin"`
⟨*proof*⟩

lemma `bl_to_bin_rep_False: "bl_to_bin (replicate n False) = 0"`
⟨*proof*⟩

lemma `len_bin_to_bl_aux: "length (bin_to_bl_aux n w bs) = n + length bs"`
⟨*proof*⟩

lemma `len_bin_to_bl: "length (bin_to_bl n w) = n"`
⟨*proof*⟩

lemma `sign_bl_bin': "bin_sign (bl_to_bin_aux bs w) = bin_sign w"`
⟨*proof*⟩

lemma `sign_bl_bin: "bin_sign (bl_to_bin bs) = 0"`
⟨*proof*⟩

lemma `bl_sbin_sign_aux: "hd (bin_to_bl_aux (Suc n) w bs) = (bin_sign (sbintrunc n w) = -1)"`
⟨*proof*⟩

lemma `bl_sbin_sign: "hd (bin_to_bl (Suc n) w) = (bin_sign (sbintrunc n w) = -1)"`
⟨*proof*⟩

lemma bin_nth_of_bl_aux:
"bin_nth (bl_to_bin_aux bl w) n =
(n < size bl ∧ rev bl ! n ∨ n ≥ length bl ∧ bin_nth w (n - size bl))"
⟨*proof*⟩

lemma bin_nth_of_bl: "bin_nth (bl_to_bin bl) n = (n < length bl ∧ rev bl ! n)"
⟨*proof*⟩

lemma bin_nth_bl: "n < m ⟹ bin_nth w n = nth (rev (bin_to_bl m w)) n"
⟨*proof*⟩

lemma nth_bin_to_bl_aux:
"n < m + length bl ⟹ (bin_to_bl_aux m w bl) ! n =
(if n < m then bit w (m - 1 - n) else bl ! (n - m))"
⟨*proof*⟩

lemma nth_bin_to_bl: "n < m ⟹ (bin_to_bl m w) ! n = bin_nth w (m - Suc n)"
⟨*proof*⟩

lemma takefill_bintrunc: "takefill False n bl = rev (bin_to_bl n (bl_to_bin (rev bl)))"
⟨*proof*⟩

lemma bl_bin_bl_rtf: "bin_to_bl n (bl_to_bin bl) = rev (takefill False n (rev bl))"
⟨*proof*⟩

lemma bl_to_bin_lt2p_aux: "bl_to_bin_aux bs w < (w + 1) * (2 ^ length bs)"
⟨*proof*⟩

lemma bl_to_bin_lt2p_drop: "bl_to_bin bs < 2 ^ length (dropWhile Not bs)"
⟨*proof*⟩

lemma bl_to_bin_lt2p: "bl_to_bin bs < 2 ^ length bs"
⟨*proof*⟩

lemma bl_to_bin_ge2p_aux: "bl_to_bin_aux bs w ≥ w * (2 ^ length bs)"
⟨*proof*⟩

lemma bl_to_bin_ge0: "bl_to_bin bs ≥ 0"
⟨*proof*⟩

lemma butlast_rest_bin: "butlast (bin_to_bl n w) = bin_to_bl (n - 1)

```
(bin_rest w)"
```
⟨*proof*⟩

lemma `butlast_bin_rest: "butlast bl = bin_to_bl (length bl - Suc 0) (bin_rest (bl_to_bin bl))"`
⟨*proof*⟩

lemma `butlast_rest_bl2bin_aux:`
`"bl ≠ [] ⟹ bl_to_bin_aux (butlast bl) w = bin_rest (bl_to_bin_aux bl w)"`
⟨*proof*⟩

lemma `butlast_rest_bl2bin: "bl_to_bin (butlast bl) = bin_rest (bl_to_bin bl)"`
⟨*proof*⟩

lemma `trunc_bl2bin_aux:`
`"bintrunc m (bl_to_bin_aux bl w) =`
`bl_to_bin_aux (drop (length bl - m) bl) (bintrunc (m - length bl) w)"`
⟨*proof*⟩

lemma `trunc_bl2bin: "bintrunc m (bl_to_bin bl) = bl_to_bin (drop (length bl - m) bl)"`
⟨*proof*⟩

lemma `trunc_bl2bin_len [simp]: "bintrunc (length bl) (bl_to_bin bl) = bl_to_bin bl"`
⟨*proof*⟩

lemma `bl2bin_drop: "bl_to_bin (drop k bl) = bintrunc (length bl - k) (bl_to_bin bl)"`
⟨*proof*⟩

lemma `take_rest_power_bin: "m ≤ n ⟹ take m (bin_to_bl n w) = bin_to_bl m ((bin_rest ^^ (n - m)) w)"`
⟨*proof*⟩

lemma `last_bin_last': "size xs > 0 ⟹ last xs ⟷ bin_last (bl_to_bin_aux xs w)"`
⟨*proof*⟩

lemma `last_bin_last: "size xs > 0 ⟹ last xs ⟷ bin_last (bl_to_bin xs)"`
⟨*proof*⟩

lemma `bin_last_last: "bin_last w ⟷ last (bin_to_bl (Suc n) w)"`
⟨*proof*⟩

lemma drop_bin2bl_aux:
"drop m (bin_to_bl_aux n bin bs) =
bin_to_bl_aux (n - m) bin (drop (m - n) bs)"
⟨*proof*⟩

lemma drop_bin2bl: "drop m (bin_to_bl n bin) = bin_to_bl (n - m) bin"
⟨*proof*⟩

lemma take_bin2bl_lem1: "take m (bin_to_bl_aux m w bs) = bin_to_bl m
w"
⟨*proof*⟩

lemma take_bin2bl_lem: "take m (bin_to_bl_aux (m + n) w bs) = take m
(bin_to_bl (m + n) w)"
⟨*proof*⟩

lemma bin_split_take: "bin_split n c = (a, b) ⟹ bin_to_bl m a = take
m (bin_to_bl (m + n) c)"
⟨*proof*⟩

lemma bin_to_bl_drop_bit:
"k = m + n ⟹ bin_to_bl m (drop_bit n c) = take m (bin_to_bl k c)"
⟨*proof*⟩

lemma bin_split_take1:
"k = m + n ⟹ bin_split n c = (a, b) ⟹ bin_to_bl m a = take m (bin_to_bl
k c)"
⟨*proof*⟩

lemma bl_bin_bl_rep_drop:
"bin_to_bl n (bl_to_bin bl) =
replicate (n - length bl) False @ drop (length bl - n) bl"
⟨*proof*⟩

lemma bl_to_bin_aux_cat:
"bl_to_bin_aux bs (bin_cat w nv v) =
bin_cat w (nv + length bs) (bl_to_bin_aux bs v)"
⟨*proof*⟩

lemma bin_to_bl_aux_cat:
"bin_to_bl_aux (nv + nw) (bin_cat v nw w) bs =
bin_to_bl_aux nv v (bin_to_bl_aux nw w bs)"
⟨*proof*⟩

lemma bl_to_bin_aux_alt: "bl_to_bin_aux bs w = bin_cat w (length bs)
(bl_to_bin bs)"
⟨*proof*⟩

lemma bin_to_bl_cat:

```
  "bin_to_bl (nv + nw) (bin_cat v nw w) =
    bin_to_bl_aux nv v (bin_to_bl nw w)"
```
⟨*proof*⟩

lemmas bl_to_bin_aux_app_cat =
  trans [OF bl_to_bin_aux_append bl_to_bin_aux_alt]

lemmas bin_to_bl_aux_cat_app =
  trans [OF bin_to_bl_aux_cat bin_to_bl_aux_alt]

lemma bl_to_bin_app_cat:
  "bl_to_bin (bsa @ bs) = bin_cat (bl_to_bin bsa) (length bs) (bl_to_bin
bs)"
⟨*proof*⟩

lemma bin_to_bl_cat_app:
  "bin_to_bl (n + nw) (bin_cat w nw wa) = bin_to_bl n w @ bin_to_bl nw
wa"
⟨*proof*⟩

bl_to_bin_app_cat_alt and bl_to_bin_app_cat are easily interderivable.

lemma bl_to_bin_app_cat_alt: "bin_cat (bl_to_bin cs) n w = bl_to_bin
(cs @ bin_to_bl n w)"
⟨*proof*⟩

lemma mask_lem: "(bl_to_bin (True # replicate n False)) = bl_to_bin (replicate
n True) + 1"
⟨*proof*⟩

lemma bin_exhaust:
  "(⋀x b. bin = of_bool b + 2 * x ⟹ Q) ⟹ Q" **for** bin :: int
⟨*proof*⟩

primrec rbl_succ :: "bool list ⇒ bool list"
  where
    Nil: "rbl_succ Nil = Nil"
  | Cons: "rbl_succ (x # xs) = (if x then False # rbl_succ xs else True
xs)"

primrec rbl_pred :: "bool list ⇒ bool list"
  where
    Nil: "rbl_pred Nil = Nil"
  | Cons: "rbl_pred (x # xs) = (if x then False # xs else True # rbl_pred
xs)"

primrec rbl_add :: "bool list ⇒ bool list ⇒ bool list"
  where — result is length of first arg, second arg may be longer
    Nil: "rbl_add Nil x = Nil"
  | Cons: "rbl_add (y # ys) x =

```
      (let ws = rbl_add ys (tl x)
       in (y ≠ hd x) # (if hd x ∧ y then rbl_succ ws else ws))"

primrec rbl_mult :: "bool list ⇒ bool list ⇒ bool list"
  where — result is length of first arg, second arg may be longer
    Nil: "rbl_mult Nil x = Nil"
  | Cons: "rbl_mult (y # ys) x =
      (let ws = False # rbl_mult ys x
       in if y then rbl_add ws x else ws)"

lemma size_rbl_pred: "length (rbl_pred bl) = length bl"
  ⟨proof⟩

lemma size_rbl_succ: "length (rbl_succ bl) = length bl"
  ⟨proof⟩

lemma size_rbl_add: "length (rbl_add bl cl) = length bl"
  ⟨proof⟩

lemma size_rbl_mult: "length (rbl_mult bl cl) = length bl"
  ⟨proof⟩

lemmas rbl_sizes [simp] =
  size_rbl_pred size_rbl_succ size_rbl_add size_rbl_mult

lemmas rbl_Nils =
  rbl_pred.Nil rbl_succ.Nil rbl_add.Nil rbl_mult.Nil

lemma rbl_add_app2: "length blb ≥ length bla ⟹ rbl_add bla (blb @
blc) = rbl_add bla blb"
  ⟨proof⟩

lemma rbl_add_take2:
  "length blb ≥ length bla ⟹ rbl_add bla (take (length bla) blb) = rbl_add
bla blb"
  ⟨proof⟩

lemma rbl_mult_app2: "length blb ≥ length bla ⟹ rbl_mult bla (blb
@ blc) = rbl_mult bla blb"
  ⟨proof⟩

lemma rbl_mult_take2:
  "length blb ≥ length bla ⟹ rbl_mult bla (take (length bla) blb) =
rbl_mult bla blb"
  ⟨proof⟩

lemma rbl_add_split:
  "P (rbl_add (y # ys) (x # xs)) =
    (∀ws. length ws = length ys ⟶ ws = rbl_add ys xs ⟶
```

```
      (y ⟶ ((x ⟶ P (False # rbl_succ ws)) ∧ (¬ x ⟶ P (True # ws))))
∧
      (¬ y ⟶ P (x # ws)))"
  ⟨proof⟩

lemma rbl_mult_split:
  "P (rbl_mult (y # ys) xs) =
    (∀ws. length ws = Suc (length ys) ⟶ ws = False # rbl_mult ys xs
⟶
      (y ⟶ P (rbl_add ws xs)) ∧ (¬ y ⟶ P ws))"
  ⟨proof⟩

lemma rbl_pred: "rbl_pred (rev (bin_to_bl n bin)) = rev (bin_to_bl n
(bin - 1))"
⟨proof⟩

lemma rbl_succ: "rbl_succ (rev (bin_to_bl n bin)) = rev (bin_to_bl n
(bin + 1))"
  ⟨proof⟩

lemma rbl_add:
  "⋀bina binb. rbl_add (rev (bin_to_bl n bina)) (rev (bin_to_bl n binb))
=
    rev (bin_to_bl n (bina + binb))"
  ⟨proof⟩

lemma rbl_add_long:
  "m ≥ n ⟹ rbl_add (rev (bin_to_bl n bina)) (rev (bin_to_bl m binb))
=
    rev (bin_to_bl n (bina + binb))"
  ⟨proof⟩

lemma rbl_mult_gt1:
  "m ≥ length bl ⟹
    rbl_mult bl (rev (bin_to_bl m binb)) =
    rbl_mult bl (rev (bin_to_bl (length bl) binb))"
  ⟨proof⟩

lemma rbl_mult_gt:
  "m > n ⟹
    rbl_mult (rev (bin_to_bl n bina)) (rev (bin_to_bl m binb)) =
    rbl_mult (rev (bin_to_bl n bina)) (rev (bin_to_bl n binb))"
  ⟨proof⟩

lemmas rbl_mult_Suc = lessI [THEN rbl_mult_gt]

lemma rbbl_Cons: "b # rev (bin_to_bl n x) = rev (bin_to_bl (Suc n) (of_bool
b + 2 * x))"
  ⟨proof⟩
```

lemma rbl_mult:
"rbl_mult (rev (bin_to_bl n bina)) (rev (bin_to_bl n binb)) =
rev (bin_to_bl n (bina * binb))"
⟨*proof*⟩

lemma sclem: "size (concat (map (bin_to_bl n) xs)) = length xs * n"
⟨*proof*⟩

lemma bin_cat_foldl_lem:
"foldl (λu. bin_cat u n) x xs =
bin_cat x (size xs * n) (foldl (λu. bin_cat u n) y xs)"
⟨*proof*⟩

lemma bin_rcat_bl: "bin_rcat n wl = bl_to_bin (concat (map (bin_to_bl n) wl))"
⟨*proof*⟩

lemma bin_last_bl_to_bin: "bin_last (bl_to_bin bs) ⟷ bs ≠ [] ∧ last bs"
⟨*proof*⟩

lemma bin_rest_bl_to_bin: "bin_rest (bl_to_bin bs) = bl_to_bin (butlast bs)"
⟨*proof*⟩

lemma bl_xor_aux_bin:
"map2 (λx y. x ≠ y) (bin_to_bl_aux n v bs) (bin_to_bl_aux n w cs) =
bin_to_bl_aux n (v XOR w) (map2 (λx y. x ≠ y) bs cs)"
⟨*proof*⟩

lemma bl_or_aux_bin:
"map2 (∨) (bin_to_bl_aux n v bs) (bin_to_bl_aux n w cs) =
bin_to_bl_aux n (v OR w) (map2 (∨) bs cs)"
⟨*proof*⟩

lemma bl_and_aux_bin:
"map2 (∧) (bin_to_bl_aux n v bs) (bin_to_bl_aux n w cs) =
bin_to_bl_aux n (v AND w) (map2 (∧) bs cs)"
⟨*proof*⟩

lemma bl_not_aux_bin: "map Not (bin_to_bl_aux n w cs) = bin_to_bl_aux n (NOT w) (map Not cs)"
⟨*proof*⟩

lemma bl_not_bin: "map Not (bin_to_bl n w) = bin_to_bl n (NOT w)"
⟨*proof*⟩

lemma bl_and_bin: "map2 (∧) (bin_to_bl n v) (bin_to_bl n w) = bin_to_bl

```
n (v AND w)"
```
⟨*proof*⟩

lemma bl_or_bin: "map2 (∨) (bin_to_bl n v) (bin_to_bl n w) = bin_to_bl n (v OR w)"
⟨*proof*⟩

lemma bl_xor_bin: "map2 (≠) (bin_to_bl n v) (bin_to_bl n w) = bin_to_bl n (v XOR w)"
⟨*proof*⟩

14.6 Type 'a word

lift_definition of_bl :: ‹bool list ⇒ 'a::len word›
is bl_to_bin ⟨*proof*⟩

lift_definition to_bl :: ‹'a::len word ⇒ bool list›
is ‹bin_to_bl LENGTH('a)›
⟨*proof*⟩

lemma to_bl_eq:
‹to_bl w = bin_to_bl (LENGTH('a)) (uint w)›
for w :: ‹'a::len word›
⟨*proof*⟩

lemma bit_of_bl_iff [bit_simps]:
‹bit (of_bl bs :: 'a word) n ⟷ rev bs ! n ∧ n < LENGTH('a::len) ∧ n < length bs›
⟨*proof*⟩

lemma rev_to_bl_eq:
‹rev (to_bl w) = map (bit w) [0..<LENGTH('a)]›
for w :: ‹'a::len word›
⟨*proof*⟩

lemma to_bl_eq_rev:
‹to_bl w = map (bit w) (rev [0..<LENGTH('a)])›
for w :: ‹'a::len word›
⟨*proof*⟩

lemma of_bl_rev_eq:
‹of_bl (rev bs) = horner_sum of_bool 2 bs›
⟨*proof*⟩

lemma of_bl_eq:
‹of_bl bs = horner_sum of_bool 2 (rev bs)›
⟨*proof*⟩

lemma bshiftr1_eq:

```
  ‹bshiftr1 b w = of_bl (b # butlast (to_bl w))›
  ⟨proof⟩

lemma length_to_bl_eq:
  ‹length (to_bl w) = LENGTH('a)›
  for w :: ‹'a::len word›
  ⟨proof⟩

lemma word_rotr_eq:
  ‹word_rotr n w = of_bl (rotater n (to_bl w))›
  ⟨proof⟩

lemma word_rotl_eq:
  ‹word_rotl n w = of_bl (rotate n (to_bl w))›
⟨proof⟩

lemma to_bl_def': "(to_bl :: 'a::len word ⇒ bool list) = bin_to_bl (LENGTH('a))
∘ uint"
  ⟨proof⟩
lemma td_bl:
  "type_definition
    (to_bl :: 'a::len word ⇒ bool list)
    of_bl
    {bl. length bl = LENGTH('a)}"
  ⟨proof⟩

interpretation word_bl:
  type_definition
    "to_bl :: 'a::len word ⇒ bool list"
    of_bl
    "{bl. length bl = LENGTH('a::len)}"
  ⟨proof⟩

lemmas word_bl_Rep' = word_bl.Rep [unfolded mem_Collect_eq, iff]

lemma word_size_bl: "size w = size (to_bl w)"
  ⟨proof⟩

lemma to_bl_use_of_bl: "to_bl w = bl ⟷ w = of_bl bl ∧ length bl =
length (to_bl w)"
  ⟨proof⟩

lemma length_bl_gt_0 [iff]: "0 < length (to_bl x)"
  for x :: "'a::len word"
  ⟨proof⟩

lemma bl_not_Nil [iff]: "to_bl x ≠ []"
  for x :: "'a::len word"
  ⟨proof⟩
```

lemma length_bl_neq_0 [iff]: "length (to_bl x) ≠ 0"
  for x :: "'a::len word"
  ⟨*proof*⟩

lemma hd_bl_sign_sint: "hd (to_bl w) = (bin_sign (sint w) = -1)"
  ⟨*proof*⟩

lemma of_bl_drop':
  "lend = length bl - LENGTH('a::len) ⟹
    of_bl (drop lend bl) = (of_bl bl :: 'a word)"
  ⟨*proof*⟩

lemma test_bit_of_bl:
  "(of_bl bl::'a::len word) !! n = (rev bl ! n ∧ n < LENGTH('a) ∧ n <
length bl)"
  ⟨*proof*⟩

lemma no_of_bl: "(numeral bin ::'a::len word) = of_bl (bin_to_bl (LENGTH('a))
(numeral bin))"
  ⟨*proof*⟩

lemma uint_bl: "to_bl w = bin_to_bl (size w) (uint w)"
  ⟨*proof*⟩

lemma to_bl_bin: "bl_to_bin (to_bl w) = uint w"
  ⟨*proof*⟩

lemma to_bl_of_bin: "to_bl (word_of_int bin::'a::len word) = bin_to_bl
(LENGTH('a)) bin"
  ⟨*proof*⟩

lemma to_bl_numeral [simp]:
  "to_bl (numeral bin::'a::len word) =
    bin_to_bl (LENGTH('a)) (numeral bin)"
  ⟨*proof*⟩

lemma to_bl_neg_numeral [simp]:
  "to_bl (- numeral bin::'a::len word) =
    bin_to_bl (LENGTH('a)) (- numeral bin)"
  ⟨*proof*⟩

lemma to_bl_to_bin [simp] : "bl_to_bin (to_bl w) = uint w"
  ⟨*proof*⟩

lemma uint_bl_bin: "bl_to_bin (bin_to_bl (LENGTH('a)) (uint x)) = uint
x"
  for x :: "'a::len word"
  ⟨*proof*⟩

lemma ucast_bl: "ucast w = of_bl (to_bl w)"
⟨*proof*⟩

lemma ucast_down_bl:
⟨(ucast :: 'a::len word ⇒ 'b::len word) (of_bl bl) = of_bl bl⟩
if ⟨is_down (ucast :: 'a::len word ⇒ 'b::len word)⟩
⟨*proof*⟩

lemma of_bl_append_same: "of_bl (X @ to_bl w) = w"
⟨*proof*⟩

lemma ucast_of_bl_up:
⟨ucast (of_bl bl :: 'a::len word) = of_bl bl⟩
if ⟨size bl ≤ size (of_bl bl :: 'a::len word)⟩
⟨*proof*⟩

lemma word_rev_tf:
"to_bl (of_bl bl::'a::len word) =
rev (takefill False (LENGTH('a)) (rev bl))"
⟨*proof*⟩

lemma word_rep_drop:
"to_bl (of_bl bl::'a::len word) =
replicate (LENGTH('a) - length bl) False @
drop (length bl - LENGTH('a)) bl"
⟨*proof*⟩

lemma to_bl_ucast:
"to_bl (ucast (w::'b::len word) ::'a::len word) =
replicate (LENGTH('a) - LENGTH('b)) False @
drop (LENGTH('b) - LENGTH('a)) (to_bl w)"
⟨*proof*⟩

lemma ucast_up_app:
⟨to_bl (ucast w :: 'b::len word) = replicate n False @ (to_bl w)⟩
if ⟨source_size (ucast :: 'a word ⇒ 'b word) + n = target_size (ucast
:: 'a word ⇒ 'b word)⟩
for w :: ⟨'a::len word⟩
⟨*proof*⟩

lemma ucast_down_drop [OF refl]:
"uc = ucast ⟹ source_size uc = target_size uc + n ⟹
to_bl (uc w) = drop n (to_bl w)"
⟨*proof*⟩

lemma scast_down_drop [OF refl]:
"sc = scast ⟹ source_size sc = target_size sc + n ⟹
to_bl (sc w) = drop n (to_bl w)"

⟨*proof*⟩

lemma word_0_bl [simp]: "of_bl [] = 0"
⟨*proof*⟩

lemma word_1_bl: "of_bl [True] = 1"
⟨*proof*⟩

lemma of_bl_0 [simp]: "of_bl (replicate n False) = 0"
⟨*proof*⟩

lemma to_bl_0 [simp]: "to_bl (0::'a::len word) = replicate (LENGTH('a)) False"
⟨*proof*⟩

lemma word_succ_rbl: "to_bl w = bl ⟹ to_bl (word_succ w) = rev (rbl_succ (rev bl))"
⟨*proof*⟩

lemma word_pred_rbl: "to_bl w = bl ⟹ to_bl (word_pred w) = rev (rbl_pred (rev bl))"
⟨*proof*⟩

lemma word_add_rbl:
"to_bl v = vbl ⟹ to_bl w = wbl ⟹
to_bl (v + w) = rev (rbl_add (rev vbl) (rev wbl))"
⟨*proof*⟩

lemma word_mult_rbl:
"to_bl v = vbl ⟹ to_bl w = wbl ⟹
to_bl (v * w) = rev (rbl_mult (rev vbl) (rev wbl))"
⟨*proof*⟩

lemma rtb_rbl_ariths:
"rev (to_bl w) = ys ⟹ rev (to_bl (word_succ w)) = rbl_succ ys"
"rev (to_bl w) = ys ⟹ rev (to_bl (word_pred w)) = rbl_pred ys"
"rev (to_bl v) = ys ⟹ rev (to_bl w) = xs ⟹ rev (to_bl (v * w)) = rbl_mult ys xs"
"rev (to_bl v) = ys ⟹ rev (to_bl w) = xs ⟹ rev (to_bl (v + w)) = rbl_add ys xs"
⟨*proof*⟩

lemma of_bl_length_less:
‹(of_bl x :: 'a::len word) < 2 ^ k›
if ‹length x = k› ‹k < LENGTH('a)›
⟨*proof*⟩

lemma word_eq_rbl_eq: "x = y ⟷ rev (to_bl x) = rev (to_bl y)"
⟨*proof*⟩

lemma bl_word_not: "to_bl (NOT w) = map Not (to_bl w)"
⟨*proof*⟩

lemma bl_word_xor: "to_bl (v XOR w) = map2 (≠) (to_bl v) (to_bl w)"
⟨*proof*⟩

lemma bl_word_or: "to_bl (v OR w) = map2 (∨) (to_bl v) (to_bl w)"
⟨*proof*⟩

lemma bl_word_and: "to_bl (v AND w) = map2 (∧) (to_bl v) (to_bl w)"
⟨*proof*⟩

lemma bin_nth_uint': "bin_nth (uint w) n ⟷ rev (bin_to_bl (size w)
(uint w)) ! n ∧ n < size w"
⟨*proof*⟩

lemmas bin_nth_uint = bin_nth_uint' [unfolded word_size]

lemma test_bit_bl: "w !! n ⟷ rev (to_bl w) ! n ∧ n < size w"
⟨*proof*⟩

lemma to_bl_nth: "n < size w ⟹ to_bl w ! n = w !! (size w - Suc n)"
⟨*proof*⟩

lemma map_bit_interval_eq:
⟨map (bit w) [0..<n] = takefill False n (rev (to_bl w))⟩ **for** w :: ⟨'a::len
word⟩
⟨*proof*⟩

lemma to_bl_unfold:
⟨to_bl w = rev (map (bit w) [0..<LENGTH('a)])⟩ **for** w :: ⟨'a::len word⟩
⟨*proof*⟩

lemma nth_rev_to_bl:
⟨rev (to_bl w) ! n ⟷ bit w n⟩
if ⟨n < LENGTH('a)⟩ **for** w :: ⟨'a::len word⟩
⟨*proof*⟩

lemma nth_to_bl:
⟨to_bl w ! n ⟷ bit w (LENGTH('a) - Suc n)⟩
if ⟨n < LENGTH('a)⟩ **for** w :: ⟨'a::len word⟩
⟨*proof*⟩

lemma of_bl_rep_False: "of_bl (replicate n False @ bs) = of_bl bs"
⟨*proof*⟩

lemma [code abstract]:
⟨Word.the_int (of_bl bs :: 'a word) = horner_sum of_bool 2 (take LENGTH('a::len)
(rev bs))⟩

⟨*proof*⟩

lemma [code]:
⟨to_bl w = map (bit w) (rev [0..<LENGTH('a::len)])⟩
for w :: ⟨'a::len word⟩
⟨*proof*⟩

lemma word_reverse_eq_of_bl_rev_to_bl:
⟨word_reverse w = of_bl (rev (to_bl w))⟩
⟨*proof*⟩

lemmas word_reverse_no_def [simp] =
word_reverse_eq_of_bl_rev_to_bl [of "numeral w"] **for** w

lemma to_bl_word_rev: "to_bl (word_reverse w) = rev (to_bl w)"
⟨*proof*⟩

lemma to_bl_n1 [simp]: "to_bl (-1::'a::len word) = replicate (LENGTH('a))
True"
⟨*proof*⟩

lemma rbl_word_or: "rev (to_bl (x OR y)) = map2 (∨) (rev (to_bl x))
(rev (to_bl y))"
⟨*proof*⟩

lemma rbl_word_and: "rev (to_bl (x AND y)) = map2 (∧) (rev (to_bl x))
(rev (to_bl y))"
⟨*proof*⟩

lemma rbl_word_xor: "rev (to_bl (x XOR y)) = map2 (≠) (rev (to_bl x))
(rev (to_bl y))"
⟨*proof*⟩

lemma rbl_word_not: "rev (to_bl (NOT x)) = map Not (rev (to_bl x))"
⟨*proof*⟩

lemma bshiftr1_numeral [simp]:
⟨bshiftr1 b (numeral w :: 'a word) = of_bl (b # butlast (bin_to_bl LENGTH('a::len)
(numeral w)))⟩
⟨*proof*⟩

lemma bshiftr1_bl: "to_bl (bshiftr1 b w) = b # butlast (to_bl w)"
⟨*proof*⟩

lemma shiftl1_of_bl: "shiftl1 (of_bl bl) = of_bl (bl @ [False])"
⟨*proof*⟩

lemma shiftl1_bl: "shiftl1 w = of_bl (to_bl w @ [False])"
for w :: "'a::len word"

⟨*proof*⟩

lemma bl_shiftl1: "to_bl (shiftl1 w) = tl (to_bl w) @ [False]"
  for w :: "'a::len word"
  ⟨*proof*⟩
lemma bl_shiftl1': "to_bl (shiftl1 w) = tl (to_bl w @ [False])"
  ⟨*proof*⟩

lemma shiftr1_bl:
  ‹shiftr1 w = of_bl (butlast (to_bl w))›
⟨*proof*⟩

lemma bl_shiftr1: "to_bl (shiftr1 w) = False # butlast (to_bl w)"
  for w :: "'a::len word"
  ⟨*proof*⟩
lemma bl_shiftr1': "to_bl (shiftr1 w) = butlast (False # to_bl w)"
  ⟨*proof*⟩

lemma bl_sshiftr1: "to_bl (sshiftr1 w) = hd (to_bl w) # butlast (to_bl w)"
  for w :: "'a::len word"
⟨*proof*⟩

lemma drop_shiftr: "drop n (to_bl (w >> n)) = take (size w - n) (to_bl w)"
  for w :: "'a::len word"
  ⟨*proof*⟩

lemma drop_sshiftr: "drop n (to_bl (w >>> n)) = take (size w - n) (to_bl w)"
  for w :: "'a::len word"
   ⟨*proof*⟩

lemma take_shiftr: "n ≤ size w ⟹ take n (to_bl (w >> n)) = replicate n False"
  ⟨*proof*⟩

lemma take_sshiftr':
  "n ≤ size w ⟹ hd (to_bl (w >>> n)) = hd (to_bl w) ∧
    take n (to_bl (w >>> n)) = replicate n (hd (to_bl w))"
  for w :: "'a::len word"
  ⟨*proof*⟩

lemmas hd_sshiftr = take_sshiftr' [THEN conjunct1]
lemmas take_sshiftr = take_sshiftr' [THEN conjunct2]

lemma atd_lem: "take n xs = t ⟹ drop n xs = d ⟹ xs = t @ d"
  ⟨*proof*⟩

lemmas bl_shiftr = atd_lem [OF take_shiftr drop_shiftr]
lemmas bl_sshiftr = atd_lem [OF take_sshiftr drop_sshiftr]

lemma shiftl_of_bl: "of_bl bl << n = of_bl (bl @ replicate n False)"
⟨*proof*⟩

lemma shiftl_bl: "w << n = of_bl (to_bl w @ replicate n False)"
for w :: "'a::len word"
⟨*proof*⟩

lemma bl_shiftl: "to_bl (w << n) = drop n (to_bl w) @ replicate (min (size w) n) False"
⟨*proof*⟩

lemma shiftr1_bl_of:
"length bl ≤ LENGTH('a) ⟹
shiftr1 (of_bl bl::'a::len word) = of_bl (butlast bl)"
⟨*proof*⟩

lemma shiftr_bl_of:
"length bl ≤ LENGTH('a) ⟹
(of_bl bl::'a::len word) >> n = of_bl (take (length bl - n) bl)"
⟨*proof*⟩

lemma shiftr_bl: "x >> n ≡ of_bl (take (LENGTH('a) - n) (to_bl x))"
for x :: "'a::len word"
⟨*proof*⟩

lemma aligned_bl_add_size [OF refl]:
"size x - n = m ⟹ n ≤ size x ⟹ drop m (to_bl x) = replicate n False ⟹
take m (to_bl y) = replicate m False ⟹
to_bl (x + y) = take m (to_bl x) @ drop m (to_bl y)" **for** x :: ‹'a::len word›
⟨*proof*⟩

lemma mask_bl: "mask n = of_bl (replicate n True)"
⟨*proof*⟩

lemma bl_and_mask':
"to_bl (w AND mask n :: 'a::len word) =
replicate (LENGTH('a) - n) False @
drop (LENGTH('a) - n) (to_bl w)"
⟨*proof*⟩

lemma slice1_eq_of_bl:
‹(slice1 n w :: 'b::len word) = of_bl (takefill False n (to_bl w))›
for w :: ‹'a::len word›
⟨*proof*⟩

lemma slice1_no_bin [simp]:
"slice1 n (numeral w :: 'b word) = of_bl (takefill False n (bin_to_bl (LENGTH('b::len)) (numeral w)))"
⟨*proof*⟩

lemma slice_no_bin [simp]:
"slice n (numeral w :: 'b word) = of_bl (takefill False (LENGTH('b::len) - n)
(bin_to_bl (LENGTH('b::len)) (numeral w)))"
⟨*proof*⟩

lemma slice_take': "slice n w = of_bl (take (size w - n) (to_bl w))"
⟨*proof*⟩

lemmas slice_take = slice_take' [unfolded word_size]

— shiftr to a word of the same size is just slice, slice is just shiftr then ucast
lemmas shiftr_slice = trans [OF shiftr_bl [THEN meta_eq_to_obj_eq] slice_take [symmetric]]

lemma slice1_down_alt':
"sl = slice1 n w ⟹ fs = size sl ⟹ fs + k = n ⟹
to_bl sl = takefill False fs (drop k (to_bl w))"
⟨*proof*⟩

lemma slice1_up_alt':
"sl = slice1 n w ⟹ fs = size sl ⟹ fs = n + k ⟹
to_bl sl = takefill False fs (replicate k False @ (to_bl w))"
⟨*proof*⟩

lemmas sd1 = slice1_down_alt' [OF refl refl, unfolded word_size]
lemmas su1 = slice1_up_alt' [OF refl refl, unfolded word_size]
lemmas slice1_down_alt = le_add_diff_inverse [THEN sd1]
lemmas slice1_up_alts =
le_add_diff_inverse [symmetric, THEN su1]
le_add_diff_inverse2 [symmetric, THEN su1]

lemma slice1_tf_tf':
"to_bl (slice1 n w :: 'a::len word) =
rev (takefill False (LENGTH('a)) (rev (takefill False n (to_bl w))))"
⟨*proof*⟩

lemmas slice1_tf_tf = slice1_tf_tf' [THEN word_bl.Rep_inverse', symmetric]

lemma revcast_eq_of_bl:
‹(revcast w :: 'b::len word) = of_bl (takefill False (LENGTH('b)) (to_bl w))›
for w :: ‹'a::len word›

⟨*proof*⟩

lemmas revcast_no_def [simp] = revcast_eq_of_bl [**where** w="numeral w", unfolded word_size] **for** w

lemma to_bl_revcast:
"to_bl (revcast w :: 'a::len word) = takefill False (LENGTH('a)) (to_bl w)"
⟨*proof*⟩

lemma word_cat_bl: "word_cat a b = of_bl (to_bl a @ to_bl b)"
⟨*proof*⟩

lemma of_bl_append:
"(of_bl (xs @ ys) :: 'a::len word) = of_bl xs * 2^(length ys) + of_bl ys"
⟨*proof*⟩

lemma of_bl_False [simp]: "of_bl (False#xs) = of_bl xs"
⟨*proof*⟩

lemma of_bl_True [simp]: "(of_bl (True # xs) :: 'a::len word) = 2^length xs + of_bl xs"
⟨*proof*⟩

lemma of_bl_Cons: "of_bl (x#xs) = of_bool x * 2^length xs + of_bl xs"
⟨*proof*⟩

lemma word_split_bl':
"std = size c - size b ⟹ (word_split c = (a, b)) ⟹
(a = of_bl (take std (to_bl c)) ∧ b = of_bl (drop std (to_bl c)))"
⟨*proof*⟩

lemma word_split_bl: "std = size c - size b ⟹
(a = of_bl (take std (to_bl c)) ∧ b = of_bl (drop std (to_bl c)))
⟷
word_split c = (a, b)"
⟨*proof*⟩

lemma word_split_bl_eq:
"(word_split c :: ('c::len word × 'd::len word)) =
(of_bl (take (LENGTH('a::len) - LENGTH('d::len)) (to_bl c)),
of_bl (drop (LENGTH('a) - LENGTH('d)) (to_bl c)))"
for c :: "'a::len word"
⟨*proof*⟩

lemma word_rcat_bl:
‹word_rcat wl = of_bl (concat (map to_bl wl))›
⟨*proof*⟩

```
lemma size_rcat_lem': "size (concat (map to_bl wl)) = length wl * size
(hd wl)"
  ⟨proof⟩

lemmas size_rcat_lem = size_rcat_lem' [unfolded word_size]

lemma nth_rcat_lem:
  "n < length (wl::'a word list) * LENGTH('a::len) ⟹
    rev (concat (map to_bl wl)) ! n =
    rev (to_bl (rev wl ! (n div LENGTH('a)))) ! (n mod LENGTH('a))"
  ⟨proof⟩

lemma foldl_eq_foldr: "foldl (+) x xs = foldr (+) (x # xs) 0"
  for x :: "'a::comm_monoid_add"
  ⟨proof⟩

lemmas word_cat_bl_no_bin [simp] =
  word_cat_bl [where a="numeral a" and b="numeral b", unfolded to_bl_numeral]
  for a b

lemmas word_split_bl_no_bin [simp] =
  word_split_bl_eq [where c="numeral c", unfolded to_bl_numeral] for c

lemmas word_rot_defs = word_roti_eq_word_rotr_word_rotl word_rotr_eq
word_rotl_eq

lemma to_bl_rotl: "to_bl (word_rotl n w) = rotate n (to_bl w)"
  ⟨proof⟩

lemmas blrs0 = rotate_eqs [THEN to_bl_rotl [THEN trans]]

lemmas word_rotl_eqs =
  blrs0 [simplified word_bl_Rep' word_bl.Rep_inject to_bl_rotl [symmetric]]

lemma to_bl_rotr: "to_bl (word_rotr n w) = rotater n (to_bl w)"
  ⟨proof⟩

lemmas brrs0 = rotater_eqs [THEN to_bl_rotr [THEN trans]]

lemmas word_rotr_eqs =
  brrs0 [simplified word_bl_Rep' word_bl.Rep_inject to_bl_rotr [symmetric]]

declare word_rotr_eqs (1) [simp]
declare word_rotl_eqs (1) [simp]

lemmas abl_cong = arg_cong [where f = "of_bl"]

locale word_rotate
```

begin

lemmas word_rot_defs' = to_bl_rotl to_bl_rotr

lemmas blwl_syms [symmetric] = bl_word_not bl_word_and bl_word_or bl_word_xor

lemmas lbl_lbl = trans [OF word_bl_Rep' word_bl_Rep' [symmetric]]

lemmas ths_map2 [OF lbl_lbl] = rotate_map2 rotater_map2

lemmas ths_map [**where** xs = "to_bl v"] = rotate_map rotater_map **for** v

lemmas th1s [simplified word_rot_defs' [symmetric]] = ths_map2 ths_map

end

lemmas bl_word_rotl_dt = trans [OF to_bl_rotl rotate_drop_take,
  simplified word_bl_Rep']

lemmas bl_word_rotr_dt = trans [OF to_bl_rotr rotater_drop_take,
  simplified word_bl_Rep']

lemma bl_word_roti_dt':
  "n = nat ((- i) mod int (size (w :: 'a::len word))) ⟹
    to_bl (word_roti i w) = drop n (to_bl w) @ take n (to_bl w)"
  ⟨*proof*⟩

lemmas bl_word_roti_dt = bl_word_roti_dt' [unfolded word_size]

lemmas word_rotl_dt = bl_word_rotl_dt [THEN word_bl.Rep_inverse' [symmetric]]
lemmas word_rotr_dt = bl_word_rotr_dt [THEN word_bl.Rep_inverse' [symmetric]]
lemmas word_roti_dt = bl_word_roti_dt [THEN word_bl.Rep_inverse' [symmetric]]

lemmas word_rotr_dt_no_bin' [simp] =
  word_rotr_dt [**where** w="numeral w", unfolded to_bl_numeral] **for** w

lemmas word_rotl_dt_no_bin' [simp] =
  word_rotl_dt [**where** w="numeral w", unfolded to_bl_numeral] **for** w

lemma max_word_bl: "to_bl (max_word::'a::len word) = replicate LENGTH('a)
True"
  ⟨*proof*⟩

lemma to_bl_mask:
  "to_bl (mask n :: 'a::len word) =
  replicate (LENGTH('a) - n) False @
    replicate (min (LENGTH('a)) n) True"

⟨*proof*⟩

lemma map_replicate_True:
"n = length xs ⟹
map (λ(x,y). x ∧ y) (zip xs (replicate n True)) = xs"
⟨*proof*⟩

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

lemma bl_and_mask:
fixes w :: "'a::len word"
and n :: nat
defines "n' ≡ LENGTH('a) - n"
shows "to_bl (w AND mask n) = replicate n' False @ drop n' (to_bl w)"
⟨*proof*⟩

lemma drop_rev_takefill:
"length xs ≤ n ⟹
drop (n - length xs) (rev (takefill False n (rev xs))) = xs"
⟨*proof*⟩

declare bin_to_bl_def [simp]

lemmas of_bl_reasoning = to_bl_use_of_bl of_bl_append

lemma uint_of_bl_is_bl_to_bin_drop:
"length (dropWhile Not l) ≤ LENGTH('a) ⟹ uint (of_bl l :: 'a::len
word) = bl_to_bin l"
⟨*proof*⟩

corollary uint_of_bl_is_bl_to_bin:
"length l≤LENGTH('a) ⟹ uint ((of_bl::bool list⇒ ('a :: len) word)
l) = bl_to_bin l"
⟨*proof*⟩

lemma bin_to_bl_or:
"bin_to_bl n (a OR b) = map2 (∨) (bin_to_bl n a) (bin_to_bl n b)"
⟨*proof*⟩

lemma word_and_1_bl:
fixes x::"'a::len word"
shows "(x AND 1) = of_bl [x !! 0]"
⟨*proof*⟩

lemma word_1_and_bl:
fixes x::"'a::len word"

shows "(1 AND x) = of_bl [x !! 0]"
⟨*proof*⟩

lemma of_bl_drop:
"of_bl (drop n xs) = (of_bl xs AND mask (length xs - n))"
⟨*proof*⟩

lemma to_bl_1:
"to_bl (1::'a::len word) = replicate (LENGTH('a) - 1) False @ [True]"
⟨*proof*⟩

lemma eq_zero_set_bl:
"(w = 0) = (True ∉ set (to_bl w))"
⟨*proof*⟩

lemma of_drop_to_bl:
"of_bl (drop n (to_bl x)) = (x AND mask (size x - n))"
⟨*proof*⟩

lemma unat_of_bl_length:
"unat (of_bl xs :: 'a::len word) < 2 ^ (length xs)"
⟨*proof*⟩

lemma word_msb_alt: "msb w ⟷ hd (to_bl w)"
for w :: "'a::len word"
⟨*proof*⟩

lemma word_lsb_last:
‹lsb w ⟷ last (to_bl w)›
for w :: ‹'a::len word›
⟨*proof*⟩

lemma is_aligned_to_bl:
"is_aligned (w :: 'a :: len word) n = (True ∉ set (drop (size w - n)
(to_bl w)))"
⟨*proof*⟩

lemma is_aligned_replicate:
fixes w::"'a::len word"
assumes aligned: "is_aligned w n"
and nv: "n ≤ LENGTH('a)"
shows "to_bl w = (take (LENGTH('a) - n) (to_bl w)) @ replicate n False"
⟨*proof*⟩

lemma is_aligned_drop:
fixes w::"'a::len word"
assumes "is_aligned w n" "n ≤ LENGTH('a)"
shows "drop (LENGTH('a) - n) (to_bl w) = replicate n False"
⟨*proof*⟩

lemma less_is_drop_replicate:
fixes x::"'a::len word"
assumes lt: "x < 2 ^ n"
shows "to_bl x = replicate (LENGTH('a) - n) False @ drop (LENGTH('a) - n) (to_bl x)"
⟨*proof*⟩

lemma is_aligned_add_conv:
fixes off::"'a::len word"
assumes aligned: "is_aligned w n"
and offv: "off < 2 ^ n"
shows "to_bl (w + off) =
(take (LENGTH('a) - n) (to_bl w)) @ (drop (LENGTH('a) - n) (to_bl off))"
⟨*proof*⟩

lemma is_aligned_replicateI:
"to_bl p = addr @ replicate n False ⟹ is_aligned (p::'a::len word) n"
⟨*proof*⟩

lemma to_bl_2p:
"n < LENGTH('a) ⟹
to_bl ((2::'a::len word) ^ n) =
replicate (LENGTH('a) - Suc n) False @ True # replicate n False"
⟨*proof*⟩

lemma xor_2p_to_bl:
fixes x::"'a::len word"
shows "to_bl (x XOR 2^n) =
(if n < LENGTH('a)
then take (LENGTH('a)-Suc n) (to_bl x) @ (¬rev (to_bl x)!n) # drop (LENGTH('a)-n) (to_bl x)
else to_bl x)"
⟨*proof*⟩

lemma is_aligned_replicateD:
"⟦ is_aligned (w::'a::len word) n; n ≤ LENGTH('a) ⟧
⟹ ∃xs. to_bl w = xs @ replicate n False
∧ length xs = size w - n"
⟨*proof*⟩

right-padding a word to a certain length

definition
"bl_pad_to bl sz ≡ bl @ (replicate (sz - length bl) False)"

lemma bl_pad_to_length:
assumes lbl: "length bl ≤ sz"
shows "length (bl_pad_to bl sz) = sz"

⟨*proof*⟩

lemma bl_pad_to_prefix:
"prefix bl (bl_pad_to bl sz)"
⟨*proof*⟩

lemma of_bl_length:
"length xs < LENGTH('a) ⟹ of_bl xs < (2 :: 'a::len word) ^ length xs"
⟨*proof*⟩

lemma of_bl_mult_and_not_mask_eq:
"⟦is_aligned (a :: 'a::len word) n; length b + m ≤ n⟧
⟹ a + of_bl b * (2^m) AND NOT(mask n) = a"
⟨*proof*⟩

lemma bin_to_bl_of_bl_eq:
"⟦is_aligned (a::'a::len word) n; length b + c ≤ n; length b + c < LENGTH('a)⟧
⟹ bin_to_bl (length b) (uint ((a + of_bl b * 2^c) >> c)) = b"
⟨*proof*⟩

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

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

lemma bl_cast_long_short_long_ingoreLeadingZero_generic:
"⟦ length (dropWhile Not (to_bl w)) ≤ LENGTH('s); LENGTH('s) ≤ LENGTH('l) ⟧ ⟹
(of_bl :: _ ⇒ 'l::len word) (to_bl ((of_bl::_ ⇒ 's::len word) (to_bl w))) = w"
⟨*proof*⟩

corollary ucast_short_ucast_long_ingoreLeadingZero:
"⟦ length (dropWhile Not (to_bl w)) ≤ LENGTH('s); LENGTH('s) ≤ LENGTH('l) ⟧ ⟹
(ucast:: 's::len word ⇒ 'l::len word) ((ucast:: 'l::len word ⇒ 's::len word) w) = w"
⟨*proof*⟩

lemma length_drop_mask:
  fixes w::"'a::len word"
  shows "length (dropWhile Not (to_bl (w AND mask n))) ≤ n"
⟨*proof*⟩

lemma map_bits_rev_to_bl:
  "map ((!!) x) [0..<size x] = rev (to_bl x)"
  ⟨*proof*⟩

lemma of_bl_length2:
  "length xs + c < LENGTH('a) ⟹ of_bl xs * 2^c < (2::'a::len word) ^
(length xs + c)"
  ⟨*proof*⟩

lemma of_bl_max:
  "(of_bl xs :: 'a::len word) ≤ mask (length xs)"
⟨*proof*⟩

end

theory Ancient_Numeral
  imports Main Reversed_Bit_Lists
begin

definition Bit :: "int ⇒ bool ⇒ int" (**infixl** "BIT" 90)
  where "k BIT b = (if b then 1 else 0) + k + k"

lemma Bit_B0: "k BIT False = k + k"
  ⟨*proof*⟩

lemma Bit_B1: "k BIT True = k + k + 1"
  ⟨*proof*⟩

lemma Bit_B0_2t: "k BIT False = 2 * k"
  ⟨*proof*⟩

lemma Bit_B1_2t: "k BIT True = 2 * k + 1"
  ⟨*proof*⟩

lemma uminus_Bit_eq:
  "- k BIT b = (- k - of_bool b) BIT b"
  ⟨*proof*⟩

lemma power_BIT: "2 ^ Suc n - 1 = (2 ^ n - 1) BIT True"
  ⟨*proof*⟩

lemma bin_rl_simp [simp]: "bin_rest w BIT bin_last w = w"
  ⟨*proof*⟩

lemma bin_rest_BIT [simp]: "bin_rest (x BIT b) = x"
⟨*proof*⟩

lemma even_BIT [simp]: "even (x BIT b) ⟷ ¬ b"
⟨*proof*⟩

lemma bin_last_BIT [simp]: "bin_last (x BIT b) = b"
⟨*proof*⟩

lemma BIT_eq_iff [iff]: "u BIT b = v BIT c ⟷ u = v ∧ b = c"
⟨*proof*⟩

lemma BIT_bin_simps [simp]:
"numeral k BIT False = numeral (Num.Bit0 k)"
"numeral k BIT True = numeral (Num.Bit1 k)"
"(- numeral k) BIT False = - numeral (Num.Bit0 k)"
"(- numeral k) BIT True = - numeral (Num.BitM k)"
⟨*proof*⟩

lemma BIT_special_simps [simp]:
shows "0 BIT False = 0"
and "0 BIT True = 1"
and "1 BIT False = 2"
and "1 BIT True = 3"
and "(- 1) BIT False = - 2"
and "(- 1) BIT True = - 1"
⟨*proof*⟩

lemma Bit_eq_0_iff: "w BIT b = 0 ⟷ w = 0 ∧ ¬ b"
⟨*proof*⟩

lemma Bit_eq_m1_iff: "w BIT b = -1 ⟷ w = -1 ∧ b"
⟨*proof*⟩

lemma expand_BIT:
"numeral (Num.Bit0 w) = numeral w BIT False"
"numeral (Num.Bit1 w) = numeral w BIT True"
"- numeral (Num.Bit0 w) = (- numeral w) BIT False"
"- numeral (Num.Bit1 w) = (- numeral (w + Num.One)) BIT True"
⟨*proof*⟩

lemma less_Bits: "v BIT b < w BIT c ⟷ v < w ∨ v ≤ w ∧ ¬ b ∧ c"
⟨*proof*⟩

lemma le_Bits: "v BIT b ≤ w BIT c ⟷ v < w ∨ v ≤ w ∧ (¬ b ∨ c)"
⟨*proof*⟩

lemma pred_BIT_simps [simp]:

```
  "x BIT False - 1 = (x - 1) BIT True"
  "x BIT True - 1 = x BIT False"
```
⟨*proof*⟩

lemma succ_BIT_simps [simp]:
```
  "x BIT False + 1 = x BIT True"
  "x BIT True + 1 = (x + 1) BIT False"
```
⟨*proof*⟩

lemma add_BIT_simps [simp]:
```
  "x BIT False + y BIT False = (x + y) BIT False"
  "x BIT False + y BIT True = (x + y) BIT True"
  "x BIT True + y BIT False = (x + y) BIT True"
  "x BIT True + y BIT True = (x + y + 1) BIT False"
```
⟨*proof*⟩

lemma mult_BIT_simps [simp]:
```
  "x BIT False * y = (x * y) BIT False"
  "x * y BIT False = (x * y) BIT False"
  "x BIT True * y = (x * y) BIT False + y"
```
⟨*proof*⟩

lemma B_mod_2': "X = 2 ⟹ (w BIT True) mod X = 1 ∧ (w BIT False) mod X = 0"
⟨*proof*⟩

lemma bin_ex_rl: "∃w b. w BIT b = bin"
⟨*proof*⟩

lemma bin_exhaust: "(⋀x b. bin = x BIT b ⟹ Q) ⟹ Q"
⟨*proof*⟩

lemma bin_abs_lem: "bin = (w BIT b) ⟹ bin ≠ -1 ⟶ bin ≠ 0 ⟶ nat |w| < nat |bin|"
⟨*proof*⟩

lemma bin_induct:
```
  assumes PPls: "P 0"
    and PMin: "P (- 1)"
    and PBit: "⋀bin bit. P bin ⟹ P (bin BIT bit)"
  shows "P bin"
```
⟨*proof*⟩

lemma Bit_div2: "(w BIT b) div 2 = w"
⟨*proof*⟩

lemma twice_conv_BIT: "2 * x = x BIT False"
⟨*proof*⟩

lemma BIT_lt0 [simp]: "x BIT b < 0 ⟷ x < 0"
⟨*proof*⟩

lemma BIT_ge0 [simp]: "x BIT b ≥ 0 ⟷ x ≥ 0"
⟨*proof*⟩

lemma bin_to_bl_aux_Bit_minus_simp [simp]:
"0 < n ⟹ bin_to_bl_aux n (w BIT b) bl = bin_to_bl_aux (n - 1) w (b
bl)"
⟨*proof*⟩

lemma bl_to_bin_BIT:
"bl_to_bin bs BIT b = bl_to_bin (bs @ [b])"
⟨*proof*⟩

lemma bin_nth_0_BIT: "bin_nth (w BIT b) 0 ⟷ b"
⟨*proof*⟩

lemma bin_nth_Suc_BIT: "bin_nth (w BIT b) (Suc n) = bin_nth w n"
⟨*proof*⟩

lemma bin_nth_minus [simp]: "0 < n ⟹ bin_nth (w BIT b) n = bin_nth
w (n - 1)"
⟨*proof*⟩

lemma bin_sign_simps [simp]:
"bin_sign (w BIT b) = bin_sign w"
⟨*proof*⟩

lemma bin_nth_Bit: "bin_nth (w BIT b) n ⟷ n = 0 ∧ b ∨ (∃m. n = Suc
m ∧ bin_nth w m)"
⟨*proof*⟩

lemmas sbintrunc_Suc_BIT [simp] =
signed_take_bit_Suc [**where** a="w BIT b", simplified bin_last_BIT bin_rest_BIT]
for w b

lemmas sbintrunc_0_BIT_B0 [simp] =
signed_take_bit_0 [**where** a="w BIT False", simplified bin_last_numeral_simps
bin_rest_numeral_simps]
for w

lemmas sbintrunc_0_BIT_B1 [simp] =
signed_take_bit_0 [**where** a="w BIT True", simplified bin_last_BIT bin_rest_numeral_simps]
for w

lemma sbintrunc_Suc_minus_Is:
‹0 < n ⟹
sbintrunc (n - 1) w = y ⟹

sbintrunc n (w BIT b) = y BIT b⟩
⟨*proof*⟩

lemma bin_cat_Suc_Bit: "bin_cat w (Suc n) (v BIT b) = bin_cat w n v BIT b"
⟨*proof*⟩

lemma int_not_BIT [simp]: "NOT (w BIT b) = (NOT w) BIT (¬ b)"
⟨*proof*⟩

lemma int_and_Bits [simp]: "(x BIT b) AND (y BIT c) = (x AND y) BIT (b ∧ c)"
⟨*proof*⟩

lemma int_or_Bits [simp]: "(x BIT b) OR (y BIT c) = (x OR y) BIT (b ∨ c)"
⟨*proof*⟩

lemma int_xor_Bits [simp]: "(x BIT b) XOR (y BIT c) = (x XOR y) BIT ((b ∨ c) ∧ ¬ (b ∧ c))"
⟨*proof*⟩

lemma mod_BIT:
"bin BIT bit mod 2 ^ Suc n = (bin mod 2 ^ n) BIT bit" **for** bit
⟨*proof*⟩

lemma minus_BIT_0: **fixes** x y :: int **shows** "x BIT b - y BIT False = (x - y) BIT b"
⟨*proof*⟩

lemma int_lsb_BIT [simp]: **fixes** x :: int **shows**
"lsb (x BIT b) ⟷ b"
⟨*proof*⟩

lemma int_shiftr_BIT [simp]: **fixes** x :: int
shows int_shiftr0: "x >> 0 = x"
and int_shiftr_Suc: "x BIT b >> Suc n = x >> n"
⟨*proof*⟩

lemma msb_BIT [simp]: "msb (x BIT b) = msb x"
⟨*proof*⟩

end

theory Bitwise
imports
"HOL-Library.Word"
More_Arithmetic

```
    Reversed_Bit_Lists
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:
```
  "(word_of_int x :: 'a::len word) !! n = (n < LENGTH('a) ∧ bin_nth x
n)"
```
⟨*proof*⟩

lemma nth_scast:
```
  "(scast (x :: 'a::len word) :: 'b::len word) !! n =
    (n < LENGTH('b) ∧
    (if n < LENGTH('a) - 1 then x !! n
     else 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 ((!!) x) [0 ..< size x])"
  ⟨proof⟩

lemma rev_to_bl_upt: "rev (to_bl x) = map ((!!) 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⟩
```

14.7 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
```

15 Bitwise tactic for Signed Words

```
theory Bitwise_Signed
imports
  "HOL-Library.Word"
  Bitwise
  Signed_Words
begin
```

⟨*ML*⟩

end

16 Enumeration extensions and alternative definition

theory Enumeration
imports Main
begin

abbreviation
  "enum ≡ enum_class.enum"
abbreviation
  "enum_all ≡ enum_class.enum_all"
abbreviation
  "enum_ex ≡ enum_class.enum_ex"

primrec (nonexhaustive)
  the_index :: "'a list ⇒ 'a ⇒ nat"
where
  "the_index (x # xs) y = (if x = y then 0 else Suc (the_index xs y))"

lemma the_index_bounded:
  "x ∈ set xs ⟹ the_index xs x < length xs"
  ⟨*proof*⟩

lemma nth_the_index:
  "x ∈ set xs ⟹ xs ! the_index xs x = x"
  ⟨*proof*⟩

lemma distinct_the_index_is_index[simp]:
  "⟦ distinct xs ; n < length xs ⟧ ⟹ the_index xs (xs ! n) = n"
  ⟨*proof*⟩

lemma the_index_last_distinct:
  "distinct xs ∧ xs ≠ [] ⟹ the_index xs (last xs) = length xs - 1"
  ⟨*proof*⟩

context enum **begin**

lemmas enum_surj[simp] = enum_UNIV
declare enum_distinct[simp]

lemma enum_nonempty[simp]: "(enum :: 'a list) ≠ []"
  ⟨*proof*⟩

definition
```
  maxBound :: 'a where
  "maxBound ≡ last enum"
```

definition
```
  minBound :: 'a where
  "minBound ≡ hd enum"
```

definition
```
  toEnum :: "nat ⇒ 'a" where
  "toEnum n ≡ if n < length (enum :: 'a list) then enum ! n else the None"
```

definition
```
  fromEnum :: "'a ⇒ nat" where
  "fromEnum x ≡ the_index enum x"
```

lemma maxBound_is_length:
```
  "fromEnum maxBound = length (enum :: 'a list) - 1"
```
⟨*proof*⟩

lemma maxBound_less_length:
```
  "(x ≤ fromEnum maxBound) = (x < length (enum :: 'a list))"
```
⟨*proof*⟩

lemma maxBound_is_bound [simp]:
```
  "fromEnum x ≤ fromEnum maxBound"
```
⟨*proof*⟩

lemma to_from_enum [simp]:
```
  fixes x :: 'a
  shows "toEnum (fromEnum x) = x"
```
⟨*proof*⟩

lemma from_to_enum [simp]:
```
  "x ≤ fromEnum maxBound ⟹ fromEnum (toEnum x) = x"
```
⟨*proof*⟩

lemma map_enum:
```
  fixes x :: 'a
  shows "map f enum ! fromEnum x = f x"
```
⟨*proof*⟩

definition
```
  assocs :: "('a ⇒ 'b) ⇒ ('a × 'b) list" where
 "assocs f ≡ map (λx. (x, f x)) enum"
```

end

lemmas enum_bool = enum_bool_def

lemma fromEnumTrue [simp]: "fromEnum True = 1"
⟨*proof*⟩

lemma fromEnumFalse [simp]: "fromEnum False = 0"
⟨*proof*⟩

class enum_alt =
fixes enum_alt :: "nat ⇒ 'a option"

class enumeration_alt = enum_alt +
assumes enum_alt_one_bound:
"enum_alt x = (None :: 'a option) ⟹ enum_alt (Suc x) = (None :: 'a option)"
assumes enum_alt_surj:
"range enum_alt ∪ {None} = UNIV"
assumes enum_alt_inj:
"(enum_alt x :: 'a option) = enum_alt y ⟹ (x = y) ∨ (enum_alt x = (None :: 'a option))"
begin

lemma enum_alt_inj_2:
assumes "enum_alt x = (enum_alt y :: 'a option)"
"enum_alt x ≠ (None :: 'a option)"
shows "x = y"
⟨*proof*⟩

lemma enum_alt_surj_2:
"∃x. enum_alt x = Some y"
⟨*proof*⟩

end

definition
alt_from_ord :: "'a list ⇒ nat ⇒ 'a option"
where
"alt_from_ord L ≡ λn. if (n < length L) then Some (L ! n) else None"

lemma handy_if_lemma: "((if P then Some A else None) = Some B) = (P ∧ (A = B))"
⟨*proof*⟩

class enumeration_both = enum_alt + enum +
assumes enum_alt_rel: "enum_alt = alt_from_ord enum"

instance enumeration_both < enumeration_alt

⟨*proof*⟩

instantiation bool :: enumeration_both
begin
  definition enum_alt_bool: "enum_alt ≡ alt_from_ord [False, True]"
  instance ⟨*proof*⟩
end

definition
  toEnumAlt :: "nat ⇒ ('a :: enum_alt)" **where**
 "toEnumAlt n ≡ the (enum_alt n)"

definition
  fromEnumAlt :: "('a :: enum_alt) ⇒ nat" **where**
 "fromEnumAlt x ≡ THE n. enum_alt n = Some x"

definition
  upto_enum :: "('a :: enumeration_alt) ⇒ 'a ⇒ 'a list" ("(1[_ .e. _])")
where
 "upto_enum n m ≡ map toEnumAlt [fromEnumAlt n ..< Suc (fromEnumAlt m)]"

lemma fromEnum_alt_red[simp]:
  "fromEnumAlt = (fromEnum :: ('a :: enumeration_both) ⇒ nat)"
  ⟨*proof*⟩

lemma toEnum_alt_red[simp]:
  "toEnumAlt = (toEnum :: nat ⇒ 'a :: enumeration_both)"
  ⟨*proof*⟩

lemma upto_enum_red:
  "[(n :: ('a :: enumeration_both)) .e. m] = map toEnum [fromEnum n ..<
Suc (fromEnum m)]"
  ⟨*proof*⟩

instantiation nat :: enumeration_alt
begin
  definition enum_alt_nat: "enum_alt ≡ Some"
  instance ⟨*proof*⟩
end

lemma toEnumAlt_nat[simp]: "toEnumAlt = id"
  ⟨*proof*⟩

lemma fromEnumAlt_nat[simp]: "fromEnumAlt = id"
  ⟨*proof*⟩

lemma upto_enum_nat[simp]: "[n .e. m] = [n ..< Suc m]"
  ⟨*proof*⟩

definition
```
  zipE1 :: "'a :: enum_alt ⇒ 'b list ⇒ ('a × 'b) list"
```
where
```
  "zipE1 x L ≡ zip (map toEnumAlt [fromEnumAlt x ..< fromEnumAlt x + length
L]) L"
```

definition
```
  zipE2 :: "'a :: enum_alt ⇒ 'a ⇒ 'b list ⇒ ('a × 'b) list"
```
where
```
  "zipE2 x xn L ≡ zip (map (λn. toEnumAlt (fromEnumAlt x + (fromEnumAlt
xn - fromEnumAlt x) * n))
                      [0 ..< length L]) L"
```

definition
```
  zipE3 :: "'a list ⇒ 'b :: enum_alt ⇒ ('a × 'b) list"
```
where
```
  "zipE3 L x ≡ zip L (map toEnumAlt [fromEnumAlt x ..< fromEnumAlt x +
length L])"
```

definition
```
  zipE4 :: "'a list ⇒ 'b :: enum_alt ⇒ 'b ⇒ ('a × 'b) list"
```
where
```
  "zipE4 L x xn ≡ zip L (map (λn. toEnumAlt (fromEnumAlt x + (fromEnumAlt
xn - fromEnumAlt x) * n))
                         [0 ..< length L])"
```

lemma `to_from_enum_alt[simp]:`
```
  "toEnumAlt (fromEnumAlt x) = (x :: 'a :: enumeration_alt)"
```
⟨*proof*⟩

lemma `upto_enum_triv [simp]: "[x .e. x] = [x]"`
⟨*proof*⟩

lemma `toEnum_eq_to_fromEnum_eq:`
fixes `v :: "'a :: enum"`
shows `"n ≤ fromEnum (maxBound :: 'a) ⟹ (toEnum n = v) = (n = fromEnum
v)"`
⟨*proof*⟩

lemma `le_imp_diff_le:`
```
  "(j::nat) ≤ k ⟹ j - n ≤ k"
```
⟨*proof*⟩

lemma `fromEnum_upto_nth:`
fixes `start :: "'a :: enumeration_both"`
assumes `"n < length [start .e. end]"`
shows `"fromEnum ([start .e. end] ! n) = fromEnum start + n"`
⟨*proof*⟩

lemma length_upto_enum_le_maxBound:
fixes start :: "'a :: enumeration_both"
shows "length [start .e. end] ≤ Suc (fromEnum (maxBound :: 'a))"
⟨*proof*⟩

lemma less_length_upto_enum_maxBoundD:
fixes start :: "'a :: enumeration_both"
assumes "n < length [start .e. end]"
shows "n ≤ fromEnum (maxBound :: 'a)"
⟨*proof*⟩

lemma fromEnum_eq_iff:
"(fromEnum e = fromEnum f) = (e = f)"
⟨*proof*⟩

lemma maxBound_is_bound':
"i = fromEnum (e::('a::enum)) ⟹ i ≤ fromEnum (maxBound::('a::enum))"
⟨*proof*⟩

end

17 Enumeration Instances for Words

theory Enumeration_Word
imports
"HOL-Library.Word"
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" ("[_ , _ .e. _]")
```
where
```
  "upto_enum_step a b 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) = max_word"
```
⟨*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 xin: "x ∈ set [(a::'a::len word).e.2 ^ n - 1]"
and nv: "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*⟩

end

18 Operation variant for setting and unsetting bits

theory Generic_set_bit
imports
"HOL-Library.Word"
Bits_Int
Most_significant_bit
begin

class set_bit = semiring_bits +
fixes set_bit :: ‹'a ⇒ nat ⇒ bool ⇒ 'a›
assumes bit_set_bit_iff [bit_simps]:
‹bit (set_bit a m b) n ⟷
(if m = n then b else bit a n) ∧ 2 ^ n ≠ 0›

lemma set_bit_eq:
‹set_bit a n b = (if b then Bit_Operations.set_bit else unset_bit) n a›

for a :: ‹'a::{ring_bit_operations, set_bit}›
⟨*proof*⟩

instantiation int :: set_bit
begin

definition set_bit_int :: ‹int ⇒ nat ⇒ bool ⇒ int›
where ‹set_bit i n b = bin_sc n b i›

instance
⟨*proof*⟩

end

lemma int_set_bit_0 [simp]: **fixes** x :: int **shows**
"set_bit x 0 b = of_bool b + 2 * (x div 2)"
⟨*proof*⟩

lemma int_set_bit_Suc: **fixes** x :: int **shows**
"set_bit x (Suc n) b = of_bool (odd x) + 2 * set_bit (x div 2) n b"
⟨*proof*⟩

lemma bin_last_set_bit:
"bin_last (set_bit x n b) = (if n > 0 then bin_last x else b)"
⟨*proof*⟩

lemma bin_rest_set_bit:
"bin_rest (set_bit x n b) = (if n > 0 then set_bit (x div 2) (n - 1)
b else x div 2)"
⟨*proof*⟩

lemma int_set_bit_numeral: **fixes** x :: int **shows**
"set_bit x (numeral w) b = of_bool (odd x) + 2 * set_bit (x div 2) (pred_numeral
w) b"
⟨*proof*⟩

lemmas int_set_bit_numerals [simp] =
int_set_bit_numeral[**where** x="numeral w'"]
int_set_bit_numeral[**where** x="- numeral w'"]
int_set_bit_numeral[**where** x="Numeral1"]
int_set_bit_numeral[**where** x="1"]
int_set_bit_numeral[**where** x="0"]
int_set_bit_Suc[**where** x="numeral w'"]
int_set_bit_Suc[**where** x="- numeral w'"]
int_set_bit_Suc[**where** x="Numeral1"]
int_set_bit_Suc[**where** x="1"]
int_set_bit_Suc[**where** x="0"]
for w'

lemma msb_set_bit [simp]: "msb (set_bit (x :: int) n b) ⟷ msb x"
⟨*proof*⟩

instantiation word :: (len) set_bit
begin

definition set_bit_word :: ‹'a word ⇒ nat ⇒ bool ⇒ 'a word›
  where word_set_bit_def: ‹set_bit a n x = word_of_int (bin_sc n x (uint a))›

instance
  ⟨*proof*⟩

end

lemma set_bit_unfold:
  ‹set_bit w n b = (if b then Bit_Operations.set_bit n w else unset_bit n w)›
  for w :: ‹'a::len word›
  ⟨*proof*⟩

lemma bit_set_bit_word_iff [bit_simps]:
  ‹bit (set_bit w m b) n ⟷ (if m = n then n < LENGTH('a) ∧ b else bit w n)›
  for w :: ‹'a::len word›
  ⟨*proof*⟩

lemma word_set_nth [simp]: "set_bit w n (test_bit w n) = w"
  for w :: "'a::len word"
  ⟨*proof*⟩

lemma test_bit_set: "(set_bit w n x) !! n ⟷ n < size w ∧ x"
  for w :: "'a::len word"
  ⟨*proof*⟩

lemma test_bit_set_gen:
  "test_bit (set_bit w n x) m = (if m = n then n < size w ∧ x else test_bit w m)"
  for w :: "'a::len word"
  ⟨*proof*⟩

lemma word_set_set_same [simp]: "set_bit (set_bit w n x) n y = set_bit w n y"
  for w :: "'a::len word"
  ⟨*proof*⟩

lemma word_set_set_diff:
  fixes w :: "'a::len word"
  assumes "m ≠ n"

shows "set_bit (set_bit w m x) n y = set_bit (set_bit w n y) m x"
⟨*proof*⟩

lemma set_bit_word_of_int: "set_bit (word_of_int x) n b = word_of_int (bin_sc n b x)"
⟨*proof*⟩

lemma word_set_numeral [simp]:
"set_bit (numeral bin::'a::len word) n b =
word_of_int (bin_sc n b (numeral bin))"
⟨*proof*⟩

lemma word_set_neg_numeral [simp]:
"set_bit (- numeral bin::'a::len word) n b =
word_of_int (bin_sc n b (- numeral bin))"
⟨*proof*⟩

lemma word_set_bit_0 [simp]: "set_bit 0 n b = word_of_int (bin_sc n b 0)"
⟨*proof*⟩

lemma word_set_bit_1 [simp]: "set_bit 1 n b = word_of_int (bin_sc n b 1)"
⟨*proof*⟩

lemma word_set_nth_iff: "set_bit w n b = w ⟷ w !! n = b ∨ n ≥ size w"
for w :: "'a::len word"
⟨*proof*⟩

lemma word_clr_le: "w ≥ set_bit w n False"
for w :: "'a::len word"
⟨*proof*⟩

lemma word_set_ge: "w ≤ set_bit w n True"
for w :: "'a::len word"
⟨*proof*⟩

lemma set_bit_beyond:
"size x ≤ n ⟹ set_bit x n b = x" **for** x :: "'a :: len word"
⟨*proof*⟩

lemma one_bit_shiftl: "set_bit 0 n True = (1 :: 'a :: len word) << n"
⟨*proof*⟩

lemmas one_bit_pow = trans [OF one_bit_shiftl shiftl_1]

end

19 Print Words in Hex

theory Hex_Words
imports
  "HOL-Library.Word"
begin

Print words in hex.

⟨*ML*⟩

end

20 Lemmas on sublists

theory More_Sublist
  imports
    "HOL-Library.Sublist"
begin

lemma same_length_is_parallel:
  assumes len: "∀y ∈ set as. length y = x"
  shows "∀x ∈ set as. ∀y ∈ set as - {x}. x ∥ y"
⟨*proof*⟩

lemma sublist_equal_part:
  "prefix xs ys ⟹ take (length xs) ys = xs"
  ⟨*proof*⟩

lemma prefix_length_less:
  "strict_prefix xs ys ⟹ length xs < length ys"
  ⟨*proof*⟩

lemmas take_less = take_strict_prefix

lemma not_prefix_longer:
  "⟦ length xs > length ys ⟧ ⟹ ¬ prefix xs ys"
  ⟨*proof*⟩

lemma map_prefixI:
  "prefix xs ys ⟹ prefix (map f xs) (map f ys)"
  ⟨*proof*⟩

lemma list_all2_induct_suffixeq [consumes 1, case_names Nil Cons]:
  assumes lall: "list_all2 Q as bs"
  and   nilr: "P [] []"
  and   consr: "⋀x xs y ys.
  ⟦list_all2 Q xs ys; Q x y; P xs ys; suffix (x # xs) as; suffix (y # ys) bs⟧
  ⟹ P (x # xs) (y # ys)"

shows "P as bs"
⟨*proof*⟩

lemma take_prefix:
"(take (length xs) ys = xs) = prefix xs ys"
⟨*proof*⟩

end

21 Miscellaneous lemmas

theory More_Misc
imports Main
begin

lemmas ls_splits = prod.split prod.split_asm if_split_asm

end

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

definition
strict_part_mono :: "'a set ⇒ ('a :: order ⇒ 'b :: order) ⇒ bool"
where
"strict_part_mono S f ≡ ∀A∈S. ∀B∈S. A < B ⟶ f A < f B"

lemma strict_part_mono_by_steps:
"strict_part_mono {..n :: nat} f = (n ≠ 0 ⟶ f (n - 1) < f n ∧ strict_part_mono {.. n - 1} f)"
⟨*proof*⟩

lemma strict_part_mono_singleton[simp]:
"strict_part_mono {x} f"
⟨*proof*⟩

lemma strict_part_mono_lt:
"⟦ x < f 0; strict_part_mono {.. n :: nat} f ⟧ ⟹ ∀m ≤ n. x < f m"
⟨*proof*⟩

lemma strict_part_mono_reverseE:
"⟦ f n ≤ f m; strict_part_mono {.. N :: nat} f; n ≤ N ⟧ ⟹ n ≤ m"
⟨*proof*⟩

lemma two_power_strict_part_mono:
"strict_part_mono {..LENGTH('a) - 1} (λx. (2 :: 'a :: len word) ^ x)"
⟨*proof*⟩

end

22 Legacy aliases

theory Legacy_Aliases
  imports "HOL-Library.Word"
begin

definition
  complement :: "'a :: len word ⇒ 'a word" **where**
 "complement x ≡ NOT x"

lemma complement_mask:
  "complement (2 ^ n - 1) = NOT (mask n)"
  ⟨*proof*⟩

lemmas less_def = less_eq [symmetric]
lemmas le_def = not_less [symmetric, **where** ?'a = nat]

end

23 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

24 Normalising Word Numerals

theory Norm_Words
  imports Bits_Int Signed_Words
begin

Normalise word numerals, including negative ones apart from `- (1::'a)`, to the interval `[0..2^len_of 'a)`. Only for concrete word lengths.

lemma neg_num_bintr:
```
  "(- numeral x :: 'a::len word) =
  word_of_int (bintrunc (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"]
```

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::'a) 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

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

theory Rsplit
  imports "HOL-Library.Word" Bits_Int
begin

definition word_rsplit :: "'a::len word ⇒ 'b::len word list"
  where "word_rsplit w = map word_of_int (bin_rsplit (LENGTH('b)) (LENGTH('a), uint w))"

lemma word_rsplit_no:
  "(word_rsplit (numeral bin :: 'b::len word) :: 'a word list) =
    map word_of_int (bin_rsplit (LENGTH('a::len))
      (LENGTH('b), take_bit (LENGTH('b)) (numeral bin)))"
  ⟨*proof*⟩

lemmas word_rsplit_no_cl [simp] = word_rsplit_no
  [unfolded bin_rsplitl_def bin_rsplit_l [symmetric]]

This odd result arises from the fact that the statement of the result implies that the decoded words are of the same type, and therefore of the same length, as the original word.

lemma word_rsplit_same: "word_rsplit w = [w]"
⟨*proof*⟩

lemma word_rsplit_empty_iff_size: "word_rsplit w = [] ⟷ size w = 0"
⟨*proof*⟩

lemma test_bit_rsplit:
"sw = word_rsplit w ⟹ m < size (hd sw) ⟹
k < length sw ⟹ (rev sw ! k) !! m = w !! (k * size (hd sw) + m)"
for sw :: "'a::len word list"
⟨*proof*⟩

lemma test_bit_rsplit_alt:
‹(word_rsplit w :: 'b::len word list) ! i !! m ⟷
w !! ((length (word_rsplit w :: 'b::len word list) - Suc i) * size
(hd (word_rsplit w :: 'b::len word list)) + m)›
if ‹i < length (word_rsplit w :: 'b::len word list)› ‹m < size (hd (word_rsplit
w :: 'b::len word list))› ‹0 < length (word_rsplit w :: 'b::len word list)›
for w :: ‹'a::len word›
⟨*proof*⟩

lemma word_rsplit_len_indep [OF refl refl refl refl]:
"[u,v] = p ⟹ [su,sv] = q ⟹ word_rsplit u = su ⟹
word_rsplit v = sv ⟹ length su = length sv"
⟨*proof*⟩

lemma length_word_rsplit_size:
"n = LENGTH('a::len) ⟹
length (word_rsplit w :: 'a word list) ≤ m ⟷ size w ≤ m * n"
⟨*proof*⟩

lemmas length_word_rsplit_lt_size =
length_word_rsplit_size [unfolded Not_eq_iff linorder_not_less [symmetric]]

lemma length_word_rsplit_exp_size:
"n = LENGTH('a::len) ⟹
length (word_rsplit w :: 'a word list) = (size w + n - 1) div n"
⟨*proof*⟩

lemma length_word_rsplit_even_size:
"n = LENGTH('a::len) ⟹ size w = m * n ⟹
length (word_rsplit w :: 'a word list) = m"
⟨*proof*⟩

lemmas length_word_rsplit_exp_size' = refl [THEN length_word_rsplit_exp_size]

— alternative proof of word_rcat_rsplit
lemmas tdle = times_div_less_eq_dividend
lemmas dtle = xtrans(4) [OF tdle mult.commute]

lemma word_rcat_rsplit: "word_rcat (word_rsplit w) = w"
⟨*proof*⟩

lemma size_word_rsplit_rcat_size:
"word_rcat ws = frcw ⟹ size frcw = length ws * LENGTH('a)
⟹ length (word_rsplit frcw::'a word list) = length ws"
for ws :: "'a::len word list" **and** frcw :: "'b::len word"
⟨*proof*⟩

lemma msrevs:
"0 < n ⟹ (k * n + m) div n = m div n + k"
"(k * n + m) mod n = m mod n"
for n :: nat
⟨*proof*⟩

lemma word_rsplit_rcat_size [OF refl]:
"word_rcat ws = frcw ⟹
size frcw = length ws * LENGTH('a) ⟹ word_rsplit frcw = ws"
for ws :: "'a::len word list"
⟨*proof*⟩

lemma word_rsplit_upt:
"⟦ size x = LENGTH('a :: len) * n; n ≠ 0 ⟧
⟹ word_rsplit x = map (λi. ucast (x >> i * len_of TYPE ('a)) ::
'a word) (rev [0 ..< n])"
⟨*proof*⟩

end

25 Displaying Phantom Types for Word Operations

theory Type_Syntax
imports "HOL-Library.Word"
begin

⟨*ML*⟩

syntax
"_Ucast" :: "type ⇒ type ⇒ logic" ("(1UCAST/(1'(_ → _')))")
translations
"UCAST('s → 't)" => "CONST ucast :: ('s word ⇒ 't word)"
⟨*ML*⟩

syntax

```
  "_Scast" :: "type ⇒ type ⇒ logic" ("(1SCAST/(1'(_ → _')))")
translations
  "SCAST('s → 't)" => "CONST scast :: ('s word ⇒ 't word)"
```
⟨*ML*⟩

```
syntax
  "_Revcast" :: "type ⇒ type ⇒ logic" ("(1REVCAST/(1'(_ → _')))")
translations
  "REVCAST('s → 't)" => "CONST revcast :: ('s word ⇒ 't word)"
```
⟨*ML*⟩

end

26 Signed division on word

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

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*⟩

instance ⟨*proof*⟩

end

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

lemma smod_word_def [code]:
  ‹v smod w = word_of_int (sint v smod sint w)›
  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_div1 [simp]:
    "(a :: ('a::len) word) sdiv 1 = a"
  ⟨*proof*⟩

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

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

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*⟩

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 [simp]:
  "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*⟩

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

27 Lemmas with Generic Word Length

theory Word_Lemmas
imports
Type_Syntax
Signed_Division_Word
Signed_Words
More_Word
Most_significant_bit
Enumeration_Word
Aligned
begin

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 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 scast_nop1 [simp]:
"((scast ((of_int x)::('a::len) word))::'a sword) = of_int x"
⟨*proof*⟩

lemma scast_nop2 [simp]:
"((scast ((of_int x)::('a::len) sword))::'a word) = of_int x"
⟨*proof*⟩

lemmas scast_nop = scast_nop1 scast_nop2 scast_id

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 bintrunc_id:
"⟦m ≤ of_nat n; 0 < m⟧ ⟹ bintrunc n m = m"
⟨*proof*⟩

lemma shiftr1_unfold: "shiftr1 x = 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 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_by_0_word:"(x::('a::len) word) div 0 = 0"
⟨*proof*⟩

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

lemma shiftr1_lt:"x ≠ 0 ⟹ (x::('a::len) word) >> 1 < 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 shiftr1_0_or_1:"(x::('a::len) word) >> 1 = 0 ⟹ x = 0 ∨ x = 1"
⟨*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 ⟹ x !! 0"
⟨*proof*⟩

lemma odd_iff_lsb:"odd (unat (x::('a::len) word)) = 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 shiftr1_irrelevant_lsb:"(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':"¬((x::('a::len) word) !! 0) ⟹ x >> 1 = (x + 1) >> 1"
⟨*proof*⟩

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

lemma cast_chunk_assemble_id:
"⟦n = LENGTH('a::len); m = LENGTH('b::len); n * 2 = m⟧ ⟹
(((ucast ((ucast (x::'b word))::'a word))::'b word) OR (((ucast ((ucast (x >> n))::'a word))::'b word) << n)) = x"
⟨*proof*⟩

lemma cast_chunk_scast_assemble_id:
"⟦n = LENGTH('a::len); m = LENGTH('b::len); n * 2 = m⟧ ⟹
(((ucast ((scast (x::'b word))::'a word))::'b word) OR

```
   (((ucast ((scast (x >> n))::'a word))::'b word) << n)) = x"
  ⟨proof⟩

lemma mask_or_not_mask:
  "x AND mask n OR x AND NOT (mask n) = x"
  for x :: ‹'a::len word›
  ⟨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 word_gr0_conv_Suc: "(m::'a::len word) > 0 ⟹ ∃n. m = n + 1"
  ⟨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_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 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 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 complement_nth_w2p:
shows "n' < LENGTH('a) ⟹ (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 unat_shiftr_less_t2n:
fixes x :: "'a :: len word"
shows "unat x < 2 ^ (n + m) ⟹ unat (x >> n) < 2 ^ m"
⟨*proof*⟩

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

lemma le_shiftr1':
"⟦ shiftr1 u ≤ shiftr1 v ; shiftr1 u ≠ shiftr1 v ⟧ ⟹ u ≤ v"
⟨*proof*⟩

lemma le_shiftr':
"⟦ u >> n ≤ v >> n ; u >> n ≠ v >> n ⟧ ⟹ (u::'a::len word) ≤ v"
⟨*proof*⟩

lemma word_add_no_overflow:"(x::'a::len word) < max_word ⟹ 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:
assumes no_overflow: "unat b < (2 :: nat) ^ LENGTH('a)"
and upward_cast: "LENGTH('a) < LENGTH('b)"
shows "(ucast (f::'a::len word) < (b :: 'b :: len word)) = (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 mask_exceed:
"n ≥ LENGTH('a) ⟹ (x::'a::len word) AND NOT (mask n) = 0"
⟨*proof*⟩

lemma word_shift_by_2:
"x * 4 = (x::'a::len word) << 2"
⟨*proof*⟩

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

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

lemma max_word_mask:
"(max_word :: '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 NOT_mask_shifted_lenword:
"NOT (mask len << (LENGTH('a) - len) ::'a::len word) = mask (LENGTH('a) - len)"
⟨*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_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)"
⟨*proof*⟩

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 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 word_exists_nth:
"(w::'a::len word) ≠ 0 ⟹ ∃i. w !! i"
⟨*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 max_word_not_0 [simp]:

"- 1 ≠ (0 :: 'a::len word)"
⟨*proof*⟩

lemma ucast_zero_is_aligned:
"UCAST('a::len → 'b::len) w = 0 ⟹ n ≤ LENGTH('b) ⟹ is_aligned w n"
⟨*proof*⟩

lemma unat_ucast_eq_unat_and_mask:
"unat (UCAST('b::len → 'a::len) w) = unat (w AND mask LENGTH('a))"
⟨*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 bounded: "x < 2 ^ LENGTH('a)" "y < 2 ^ LENGTH('a)"
assumes of_nats: "of_nat x = (of_nat y :: 'a::len word)"
shows "x = y"
⟨*proof*⟩

lemma uints_mono_iff: "uints l ⊆ uints m ⟷ l ≤ m"
⟨*proof*⟩

lemmas uints_monoI = uints_mono_iff[THEN iffD2]

lemma Bit_in_uints_Suc: "of_bool c + 2 * w ∈ uints (Suc m)" **if** "w ∈ uints m"
⟨*proof*⟩

lemma Bit_in_uintsI: "of_bool c + 2 * w ∈ uints m" **if** "w ∈ uints (m - 1)" "m > 0"
⟨*proof*⟩

lemma bin_cat_in_uintsI:
‹bin_cat a n b ∈ uints m› **if** ‹a ∈ uints l› ‹m ≥ l + n›
⟨*proof*⟩

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

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

lemma bin_cat_eqD2: "bin_cat a n b = bin_cat c n d ⟹ bintrunc n b =
bintrunc n d"
  ⟨*proof*⟩

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

lemma word_of_int_bin_cat_eq_iff:
  "(word_of_int (bin_cat (uint a) LENGTH('b) (uint b))::'c::len word)
=
  word_of_int (bin_cat (uint c) LENGTH('b) (uint d)) ⟷ b = d ∧ a =
c"
  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*⟩

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 << n = y << n; x ≤ mask (LENGTH('a)-n); y ≤ mask (LENGTH('a)-n)
⟧ ⟹
x = (y :: 'a :: len word)"
⟨*proof*⟩

lemma distinct_word_add_ucast_shift_inj:
"⟦ p + (UCAST('a::len → 'b::len) off << n) = p' + (ucast off' << n);
is_aligned p n'; is_aligned p' n'; n' = n + LENGTH('a); n' < LENGTH('b)
⟧
⟹ p' = p ∧ off' = off"
⟨*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 max_word_not_less[simp]:
"¬ max_word < x"
⟨*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:
"⟦ i ≤ j; k ≤ unat (j - i) ⟧ ⟹ [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:
"a < - 1 ⟹ [(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:
"⟦ x AND mask n = 0; y AND mask n = 0; x ≤ x + y ⟧
⟹ (x + y :: ('a :: len) word) >> 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 [simp]:
"(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' < len ⟶ ¬ 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 :: 'a :: len word) < 2 ^ (m - n); m < LENGTH('a) ⟧ ⟹ unat (x << n) < 2 ^ m"
⟨*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:
"2 ^ m ≠ (0 :: 'a word) ⟹ 2 ^ n ≤ - (2 ^ m :: 'a :: len word)"
⟨*proof*⟩

lemma unat_shiftl_absorb:
"⟦ x ≤ 2 ^ p; p + k < LENGTH('a) ⟧ ⟹ unat (x :: 'a :: len word) *
2 ^ k = unat (x * 2 ^ k)"
⟨*proof*⟩

lemma word_plus_mono_right_split:
"⟦ unat ((x :: 'a :: len word) AND mask sz) + unat z < 2 ^ sz; sz < LENGTH('a) ⟧
⟹ 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*⟩

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 (max_word
:: '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:
assumes "Suc n = LENGTH('b::len)"
shows "sbintrunc n (uint (ucast w :: 'b word)) = sbintrunc n (uint w)"
⟨*proof*⟩ **lemma** test_bit_sbintrunc:
assumes "i < LENGTH('a)"
shows "(word_of_int (sbintrunc n (uint w)) :: 'a word) !! i
= (if n < i then w !! n else w !! i)"
⟨*proof*⟩ **lemma** test_bit_sbintrunc_ucast:

assumes len_a: "i < LENGTH('a)"
shows "(word_of_int (sbintrunc (LENGTH('b) - 1) (uint (ucast w :: 'b word))) :: 'a word) !! i
= (if LENGTH('b::len) ≤ i then w !! (LENGTH('b) - 1) else w !! i)"
⟨*proof*⟩

lemma scast_ucast_high_bits:
‹scast (ucast w :: 'b::len word) = w
⟷ (∀ i ∈ {LENGTH('b) ..< size w}. w !! i = 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:
"⟦ is_aligned p n ⟧ ⟹
([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*⟩

end

28 Words of Length 8

theory Word_8
imports
  More_Word
  Enumeration_Word
  Even_More_List
  Signed_Words
  Word_Lemmas
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

29 Words of Length 16

theory Word_16
imports
  More_Word
  Signed_Words
begin

lemma len16: "len_of (x :: 16 itself) = 16" ⟨*proof*⟩

lemma word16_and_max_simp:
  ‹x AND 0xFFFF = x› **for** x :: ‹16 word›
  ⟨*proof*⟩

end

30 Additional Syntax for Word Bit Operations

theory Word_Syntax
imports
  "HOL-Library.Word"
begin

Additional bit and type syntax that forces word types.

abbreviation
  wordNOT :: "'a::len word ⇒ 'a word"   ("~~ _" [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
```

31 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
```

32 Misc word operations

```
theory More_Word_Operations
  imports
    "HOL-Library.Word"
    Aligned
    Reversed_Bit_Lists
    More_Misc
    Signed_Words
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_le:
  "word_ctz (w :: ('a::len word)) ≤ LENGTH('a)"
  ⟨proof⟩

lemma word_ctz_less:
  "w ≠ 0 ⟹ 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 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 ⟶ w !! i = 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_same:
  "w ≠ 0 ⟹ w !! word_log2 w"
  ⟨*proof*⟩

lemma word_log2_nth_not_set:
  "⟦ word_log2 w < i ; i < size w ⟧ ⟹ ¬ w !! i"
  ⟨*proof*⟩

lemma word_log2_highest:
  assumes a: "w !! i"
  shows "i ≤ word_log2 w"
  ⟨*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:
"is_aligned p sz ⟹ (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 sign_extend_bitwise_if:
"i < size w ⟹ sign_extend e w !! i ⟷ (if i < e then w !! i else w !! e)"
⟨*proof*⟩

lemma sign_extend_bitwise_if' [word_eqI_simps]:
‹i < LENGTH('a) ⟹ sign_extend e w !! i ⟷ (if i < e then w !! i else w !! e)›
for w :: ‹'a::len word›
⟨*proof*⟩

lemma sign_extend_bitwise_disj:
"i < size w ⟹ sign_extend e w !! i ⟷ i ≤ e ∧ w !! i ∨ e ≤ i ∧ w !! e"
⟨*proof*⟩

lemma sign_extend_bitwise_cases:
"i < size w ⟹ sign_extend e w !! i ⟷ (i ≤ e ⟶ w !! i) ∧ (e ≤ i ⟶ 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 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 ⟧ ⟹ p !! i = 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 shiftl_shiftr1[unfolded word_size mask_eq_decr_exp]
shiftl_shiftr2[unfolded word_size mask_eq_decr_exp]

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) = (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:
```
  "is_aligned ptr bits ⟹ (ptr' AND NOT(mask bits) = ptr) = (ptr' ∈ mask_range
ptr bits)"
```
⟨*proof*⟩

lemma aligned_offset_in_range:
```
  "⟦ is_aligned (x :: 'a :: len word) m; y < 2 ^ m; is_aligned p n; n
≥ m; n < LENGTH('a) ⟧
   ⟹ (x + y ∈ {p .. p + mask n}) = (x ∈ mask_range p n)"
```
⟨*proof*⟩

lemma mask_range_to_bl':
```
  "⟦ is_aligned (ptr :: 'a :: len word) bits; bits < LENGTH('a) ⟧
   ⟹ 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:
  "⟦ is_aligned p n; is_aligned ptr bits; n ≥ m; n < size p; m ≤ bits;
     (∀y < 2 ^ (n - m). p + (y << m) ∉ mask_range ptr bits) ⟧
   ⟹ 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]:
  "LENGTH('a) ≥ 3
   ⟹ - 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 nasty_split_lt:
  "⟦ (x :: 'a:: len word) < 2 ^ (m - n); n ≤ m; m < LENGTH('a::len) ⟧
     ⟹ x * 2 ^ n + (2 ^ n - 1) ≤ 2 ^ m - 1"
  ⟨proof⟩

lemma nasty_split_less:
  "⟦m ≤ n; n ≤ nm; nm < LENGTH('a::len); x < 2 ^ (nm - n)⟧
```

```
   ⟹ (x :: 'a word) * 2 ^ n + (2 ^ m - 1) < 2 ^ nm"
```
⟨*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:
```
  "LENGTH('a) > 2
   ⟹ 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))"
```
⟨*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*⟩

end

33 Words of Length 32

theory Word_32
  imports
    Word_Lemmas
    Word_Syntax
    Word_Names
    Rsplit
    More_Word_Operations
    Bitwise
begin

type_synonym word32 = "32 word"
lemma len32: "len_of (x :: 32 itself) = 32" ⟨*proof*⟩

type_synonym sword32 = "32 sword"

type_synonym machine_word_len = 32
type_synonym machine_word = "machine_word_len word"

definition word_bits :: nat
where
  "word_bits = LENGTH(machine_word_len)"

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 word_bits_conv[code]:
  "word_bits = 32"
  ⟨*proof*⟩

lemma word_size_word_size_bits:
  "(word_size::nat) = 2 ^ word_size_bits"
  ⟨*proof*⟩

lemma word_bits_word_size_conv:
  "word_bits = word_size * 8"
  ⟨*proof*⟩

lemma ucast_8_32_inj:
  "inj (ucast :: 8 word ⇒ 32 word)"
  ⟨*proof*⟩

lemma upto_2_helper:
  "{0..<2 :: 32 word} = {0, 1}"
  ⟨*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]

lemma word_bits_len_of:
  "len_of TYPE (32) = word_bits"
  ⟨*proof*⟩

lemmas unat_power_lower32' = unat_power_lower[**where** 'a=32]

```
lemmas unat_power_lower32 [simp] = unat_power_lower32'[unfolded word_bits_len_of]

lemmas word32_less_sub_le' = word_less_sub_le[where 'a = 32]
lemmas word32_less_sub_le[simp] = word32_less_sub_le' [folded word_bits_def]

lemma word_bits_size:
  "size (w::word32) = word_bits"
  ⟨proof⟩

lemmas word32_power_less_1' = word_power_less_1[where 'a = 32]
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⟩

lemma unat_mask_2_less_4:
  "unat (p && mask 2 :: word32) < 4"
  ⟨proof⟩

lemmas unat_of_nat32' = unat_of_nat_eq[where 'a=32]
lemmas unat_of_nat32 = unat_of_nat32'[unfolded word_bits_len_of]

lemmas word_power_nonzero_32 = word_power_nonzero [where 'a=32, folded
word_bits_def]

lemmas unat_mult_simple = iffD1 [OF unat_mult_lem [where 'a = 32, unfolded
word_bits_len_of]]

lemmas div_power_helper_32 = div_power_helper [where 'a=32, folded word_bits_def]

lemma n_less_word_bits:
  "(n < word_bits) = (n < 32)"
  ⟨proof⟩

lemmas of_nat_less_pow_32 = of_nat_power [where 'a=32, folded word_bits_def]

lemma lt_word_bits_lt_pow:
  "sz < word_bits ⟹ sz < 2 ^ word_bits"
  ⟨proof⟩

lemma unat_less_word_bits:
  fixes y :: word32
  shows "x < unat y ⟹ x < 2 ^ word_bits"
  ⟨proof⟩

lemmas unat_mask_word32' = unat_mask[where 'a=32]
lemmas unat_mask_word32 = unat_mask_word32'[folded word_bits_def]
```

lemma unat_less_2p_word_bits:
"unat (x :: 32 word) < 2 ^ word_bits"
⟨*proof*⟩

lemma Suc_unat_mask_div:
"Suc (unat (mask sz div word_size::word32)) = 2 ^ (min sz word_bits
- 2)"
⟨*proof*⟩

lemmas word32_minus_one_le' = word_minus_one_le[**where** 'a=32]
lemmas word32_minus_one_le = word32_minus_one_le'[simplified]

lemma ucast_not_helper:
fixes a::"8 word"
assumes a: "a ≠ 0xFF"
shows "ucast a ≠ (0xFF::word32)"
⟨*proof*⟩

lemma less_4_cases:
"(x::word32) < 4 ⟹ x=0 ∨ x=1 ∨ x=2 ∨ x=3"
⟨*proof*⟩

lemma unat_ucast_8_32:
fixes x :: "8 word"
shows "unat (ucast x :: word32) = unat x"
⟨*proof*⟩

lemma if_then_1_else_0:
"((if P then 1 else 0) = (0 :: word32)) = (¬ P)"
⟨*proof*⟩

lemma if_then_0_else_1:
"((if P then 0 else 1) = (0 :: word32)) = (P)"
⟨*proof*⟩

lemmas if_then_simps = if_then_0_else_1 if_then_1_else_0

lemma ucast_le_ucast_8_32:
"(ucast x ≤ (ucast y :: word32)) = (x ≤ (y :: 8 word))"
⟨*proof*⟩

lemma in_16_range:
"0 ∈ S ⟹ r ∈ (λx. r + x * (16 :: word32)) ` S"
"n - 1 ∈ S ⟹ (r + (16 * n - 16)) ∈ (λx :: word32. r + x * 16) ` S"
⟨*proof*⟩

lemma eq_2_32_0:
"(2 ^ 32 :: word32) = 0"
⟨*proof*⟩

lemma x_less_2_0_1:
  fixes x :: word32 **shows**
  "x < 2 ⟹ x = 0 ∨ x = 1"
  ⟨*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 word_rsplit_0:
  "word_rsplit (0 :: word32) = [0, 0, 0, 0 :: 8 word]"
  ⟨*proof*⟩

lemma unat_ucast_10_32 :
  fixes x :: "10 word"
  shows "unat (ucast x :: word32) = unat x"
  ⟨*proof*⟩

lemma bool_mask [simp]:
  fixes x :: word32
  shows "(0 < x && 1) = (x && 1 = 1)"
  ⟨*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*⟩

lemma word_ge_min:"sint (x::32 word) ≥ -2147483648"
  ⟨*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 le_step_down_word_3:
  fixes x :: "32 word"
  shows "⟦x ≤  y; x ≠ y; y < 2 ^ 32 - 1⟧ ⟹ x ≤ y - 1"
  ⟨proof⟩

lemma shiftr_1:
  "(x::word32) >> 1 = 0 ⟹ x < 2"
  ⟨proof⟩

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

theory Many_More
  imports
    Main
    "HOL-Library.Word"
    More_Word
    Even_More_List
begin

lemma nat_less_mult_monoish: "⟦ a < b; c < (d :: nat) ⟧ ⟹ (a + 1) *
(c + 1) <= b * d"

⟨*proof*⟩

lemma if_and_helper:
"(If x v v') AND v'' = If x (v AND v'') (v' AND v'')"
⟨*proof*⟩

lemma eq_eqI:
"a = b ⟹ (a = x) = (b = x)"
⟨*proof*⟩

lemma map2_Cons_2_3:
"(map2 f xs (y # ys) = (z # zs)) = (∃x xs'. xs = x # xs' ∧ f x y = z ∧ map2 f xs' ys = zs)"
⟨*proof*⟩

lemma map2_xor_replicate_False:
"map2 (λx y. x ⟷ ¬ y) xs (replicate n False) = take n xs"
⟨*proof*⟩

lemma plus_Collect_helper:
"(+) x ` {xa. P (xa :: 'a :: len word)} = {xa. P (xa - x)}"
⟨*proof*⟩

lemma plus_Collect_helper2:
"(+) (- x) ` {xa. P (xa :: 'a :: len word)} = {xa. P (x + xa)}"
⟨*proof*⟩

lemma range_subset_eq2:
"{a :: 'a :: len word .. b} ≠ {} ⟹ ({a .. b} ⊆ {c .. d}) = (c ≤ a ∧ b ≤ d)"
⟨*proof*⟩

lemma nat_mod_power_lem:
fixes a :: nat
shows "1 < a ⟹ a ^ n mod a ^ m = (if m ≤ n then 0 else a ^ n)"
⟨*proof*⟩

lemma i_hate_words_helper:
"i ≤ (j - k :: nat) ⟹ i ≤ j"
⟨*proof*⟩

lemma i_hate_words:
"unat (a :: 'a word) ≤ unat (b :: 'a :: len word) - Suc 0
⟹ a ≠ -1"
⟨*proof*⟩

lemma If_eq_obvious:
"x ≠ z ⟹ ((if P then x else y) = z) = (¬ P ∧ y = z)"
⟨*proof*⟩

lemma Some_to_the:
"v = Some x ⟹ x = the v"
⟨*proof*⟩

lemma dom_if_Some:
"dom (λx. if P x then Some (f x) else g x) = {x. P x} ∪ dom g"
⟨*proof*⟩

lemma dom_insert_absorb:
"x ∈ dom f ⟹ insert x (dom f) = dom f"
⟨*proof*⟩

lemma emptyE2:
"⟦ S = {}; x ∈ S ⟧ ⟹ P"
⟨*proof*⟩

lemma ptr_add_image_multI:
"⟦ ⋀x y. (x * val = y * val') = (x * val'' = y); x * val'' ∈ S ⟧ ⟹
ptr_add ptr (x * val) ∈ (λp. ptr_add ptr (p * val')) ` S"
⟨*proof*⟩

lemmas map_prod_split_imageI'
= map_prod_imageI[**where** f="case_prod f" **and** g="case_prod g"
and a="(a, b)" **and** b="(c, d)" **for** a b c d f g]
lemmas map_prod_split_imageI = map_prod_split_imageI'[simplified]

lemma dom_if:
"dom (λa. if a ∈ addrs then Some (f a) else g a) = addrs ∪ dom g"
⟨*proof*⟩

lemmas arg_cong_Not = arg_cong [**where** f=Not]

lemma drop_append_miracle:
"n = length xs ⟹ drop n (xs @ ys) = ys"
⟨*proof*⟩

lemma foldr_does_nothing_to_xf:
"⟦ ⋀x s. x ∈ set xs ⟹ xf (f x s) = xf s ⟧ ⟹ xf (foldr f xs s) =
xf s"
⟨*proof*⟩

lemma mod_mod_power_int:
fixes k :: int
shows "k mod 2 ^ m mod 2 ^ n = k mod 2 ^ (min m n)"
⟨*proof*⟩

lemma le_step_down_nat:"⟦(i::nat) ≤ n; i = n ⟶ P; i ≤ n - 1 ⟶ P⟧
⟹ P"

⟨*proof*⟩

lemma le_step_down_int:"⟦(i::int) ≤ n; i = n ⟶ P; i ≤ n - 1 ⟶ P⟧
⟹ P"
⟨*proof*⟩

lemma replicate_numeral [simp]: "replicate (numeral k) x = x # replicate
(pred_numeral k) x"
⟨*proof*⟩

lemma list_exhaust_size_gt0:
assumes "⋀a list. y = a # list ⟹ P"
shows "0 < length y ⟹ P"
⟨*proof*⟩

lemma list_exhaust_size_eq0:
assumes "y = [] ⟹ P"
shows "length y = 0 ⟹ P"
⟨*proof*⟩

lemma size_Cons_lem_eq: "y = xa # list ⟹ size y = Suc k ⟹ size list
= k"
⟨*proof*⟩

lemma takeWhile_take_has_property:
"n ≤ length (takeWhile P xs) ⟹ ∀x ∈ set (take n xs). P x"
⟨*proof*⟩

lemma takeWhile_take_has_property_nth:
"⟦ n < length (takeWhile P xs) ⟧ ⟹ P (xs ! n)"
⟨*proof*⟩

lemma takeWhile_replicate:
"takeWhile f (replicate len x) = (if f x then replicate len x else [])"
⟨*proof*⟩

lemma takeWhile_replicate_empty:
"¬ f x ⟹ takeWhile f (replicate len x) = []"
⟨*proof*⟩

lemma takeWhile_replicate_id:
"f x ⟹ takeWhile f (replicate len x) = replicate len x"
⟨*proof*⟩

lemma nth_rev: "n < length xs ⟹ rev xs ! n = xs ! (length xs - 1 -
n)"
⟨*proof*⟩

lemma nth_rev_alt: "n < length ys ⟹ ys ! n = rev ys ! (length ys -

```
Suc n)"
```
⟨*proof*⟩

lemma hd_butlast: "length xs > 1 ⟹ hd (butlast xs) = hd xs"
⟨*proof*⟩

lemma split_upt_on_n:
"n < m ⟹ [0 ..< m] = [0 ..< n] @ [n] @ [Suc n ..< m]"
⟨*proof*⟩

lemma drop_eq_mono:
assumes le: "m ≤ n"
assumes drop: "drop m xs = drop m ys"
shows "drop n xs = drop n ys"
⟨*proof*⟩

lemma drop_Suc_nth:
"n < length xs ⟹ drop n xs = xs!n # drop (Suc n) xs"
⟨*proof*⟩

lemma and_len: "xs = ys ⟹ xs = ys ∧ length xs = length ys"
⟨*proof*⟩

lemma tl_if: "tl (if p then xs else ys) = (if p then tl xs else tl ys)"
⟨*proof*⟩

lemma hd_if: "hd (if p then xs else ys) = (if p then hd xs else hd ys)"
⟨*proof*⟩

lemma if_single: "(if xc then [xab] else [an]) = [if xc then xab else
an]"
⟨*proof*⟩
lemma if_Cons: "(if p then x # xs else y # ys) = If p x y # If p xs ys"
⟨*proof*⟩

lemma list_of_false:
"True ∉ set xs ⟹ xs = replicate (length xs) False"
⟨*proof*⟩

lemma list_all2_induct [consumes 1, case_names Nil Cons]:
assumes lall: "list_all2 Q xs ys"
and nilr: "P [] []"
and consr: "⋀x xs y ys. ⟦list_all2 Q xs ys; Q x y; P xs ys⟧ ⟹
P (x # xs) (y # ys)"
shows "P xs ys"
⟨*proof*⟩

lemma replicate_minus:
"k < n ⟹ replicate n False = replicate (n - k) False @ replicate k

```
False"
```
⟨*proof*⟩

lemma cart_singleton_empty:
```
  "(S × {e} = {}) = (S = {})"
```
⟨*proof*⟩

lemma MinI:
```
  assumes fa: "finite A"
  and     ne: "A ≠ {}"
  and     xv: "m ∈ A"
  and    min: "∀y ∈ A. m ≤ y"
  shows "Min A = m"
```
⟨*proof*⟩

lemma power_numeral: "a ^ numeral k = a * a ^ (pred_numeral k)"
⟨*proof*⟩

lemma funpow_numeral [simp]: "f ^^ numeral k = f ∘ f ^^ (pred_numeral k)"
⟨*proof*⟩

lemma funpow_minus_simp: "0 < n ⟹ f ^^ n = f ∘ f ^^ (n - 1)"
⟨*proof*⟩

lemma rco_alt: "(f ∘ g) ^^ n ∘ f = f ∘ (g ∘ f) ^^ n"
⟨*proof*⟩

lemma union_sub:
```
  "⟦B ⊆ A; C ⊆ B⟧ ⟹ (A - B) ∪ (B - C) = (A - C)"
```
⟨*proof*⟩

lemma insert_sub:
```
  "x ∈ xs ⟹ (insert x (xs - ys)) = (xs - (ys - {x}))"
```
⟨*proof*⟩

lemma ran_upd:
```
  "⟦ inj_on f (dom f); f y = Some z ⟧ ⟹ ran (λx. if x = y then None
else f x) = ran f - {z}"
```
⟨*proof*⟩

lemma if_apply_def2:
```
  "(if P then F else G) = (λx. (P ⟶ F x) ∧ (¬ P ⟶ G x))"
```
⟨*proof*⟩

lemma case_bool_If:
```
  "case_bool P Q b = (if b then P else Q)"
```
⟨*proof*⟩

lemma if_f:

```
  "(if a then f b else f c) = f (if a then b else c)"
```
⟨*proof*⟩

lemma size_if: "size (if p then xs else ys) = (if p then size xs else size ys)"
⟨*proof*⟩

lemma if_Not_x: "(if p then ¬ x else x) = (p = (¬ x))"
⟨*proof*⟩

lemma if_x_Not: "(if p then x else ¬ x) = (p = x)"
⟨*proof*⟩

lemma if_same_and: "(If p x y ∧ If p u v) = (if p then x ∧ u else y ∧ v)"
⟨*proof*⟩

lemma if_same_eq: "(If p x y = (If p u v)) = (if p then x = u else y = v)"
⟨*proof*⟩

lemma if_same_eq_not: "(If p x y = (¬ If p u v)) = (if p then x = (¬ u) else y = (¬ v))"
⟨*proof*⟩

lemma the_elemI: "y = {x} ⟹ the_elem y = x"
⟨*proof*⟩

lemma nonemptyE: "S ≠ {} ⟹ (⋀x. x ∈ S ⟹ R) ⟹ R"
⟨*proof*⟩

lemmas xtr1 = xtrans(1)
lemmas xtr2 = xtrans(2)
lemmas xtr3 = xtrans(3)
lemmas xtr4 = xtrans(4)
lemmas xtr5 = xtrans(5)
lemmas xtr6 = xtrans(6)
lemmas xtr7 = xtrans(7)
lemmas xtr8 = xtrans(8)

lemmas if_fun_split = if_apply_def2

lemma not_empty_eq:
```
  "(S ≠ {}) = (∃x. x ∈ S)"
```
⟨*proof*⟩

lemma range_subset_lower:
```
  fixes c :: "'a ::linorder"
  shows "⟦ {a..b} ⊆ {c..d}; x ∈ {a..b} ⟧ ⟹ c ≤ a"
```

⟨*proof*⟩

lemma range_subset_upper:
fixes c :: "'a ::linorder"
shows "⟦ {a..b} ⊆ {c..d}; x ∈ {a..b} ⟧ ⟹ b ≤ d"
⟨*proof*⟩

lemma range_subset_eq:
fixes a::"'a::linorder"
assumes non_empty: "a ≤ b"
shows "({a..b} ⊆ {c..d}) = (c ≤ a ∧ b ≤ d)"
⟨*proof*⟩

lemma range_eq:
fixes a::"'a::linorder"
assumes non_empty: "a ≤ b"
shows "({a..b} = {c..d}) = (a = c ∧ b = d)"
⟨*proof*⟩

lemma range_strict_subset_eq:
fixes a::"'a::linorder"
assumes non_empty: "a ≤ b"
shows "({a..b} ⊂ {c..d}) = (c ≤ a ∧ b ≤ d ∧ (a = c ⟶ b ≠ d))"
⟨*proof*⟩

lemma range_subsetI:
fixes x :: "'a :: order"
assumes xX: "X ≤ x"
and yY: "y ≤ Y"
shows "{x .. y} ⊆ {X .. Y}"
⟨*proof*⟩

lemma set_False [simp]:
"(set bs ⊆ {False}) = (True ∉ set bs)" ⟨*proof*⟩

lemma int_not_emptyD:
"A ∩ B ≠ {} ⟹ ∃x. x ∈ A ∧ x ∈ B"
⟨*proof*⟩

definition
sum_map :: "('a ⇒ 'b) ⇒ ('c ⇒ 'd) ⇒ 'a + 'c ⇒ 'b + 'd" **where**
"sum_map f g x ≡ case x of Inl v ⇒ Inl (f v) | Inr v' ⇒ Inr (g v')"

lemma sum_map_simps[simp]:
"sum_map f g (Inl v) = Inl (f v)"
"sum_map f g (Inr w) = Inr (g w)"
⟨*proof*⟩

lemma if_Some_None_eq_None:

```
  "((if P then Some v else None) = None) = (¬ P)"
```
⟨*proof*⟩

lemma CollectPairFalse [iff]:
```
  "{(a,b). False} = {}"
```
⟨*proof*⟩

lemma if_conj_dist:
```
  "((if b then w else x) ∧ (if b then y else z) ∧ X) =
  ((if b then w ∧ y else x ∧ z) ∧ X)"
```
⟨*proof*⟩

lemma if_P_True1:
```
  "Q ⟹ (if P then True else Q)"
```
⟨*proof*⟩

lemma if_P_True2:
```
  "Q ⟹ (if P then Q else True)"
```
⟨*proof*⟩

lemmas nat_simps = diff_add_inverse2 diff_add_inverse

lemmas nat_iffs = le_add1 le_add2

lemma nat_min_simps:
```
  "(a::nat) ≤ b ⟹ min b a = a"
  "a ≤ b ⟹ min a b = a"
```
⟨*proof*⟩

lemmas zadd_diff_inverse =
```
  trans [OF diff_add_cancel [symmetric] add.commute]
```

lemmas add_diff_cancel2 =
```
  add.commute [THEN diff_eq_eq [THEN iffD2]]
```

lemmas mcl = mult_cancel_left [THEN iffD1, THEN make_pos_rule]

lemma pl_pl_rels: "a + b = c + d ⟹ a ≥ c ∧ b ≤ d ∨ a ≤ c ∧ b ≥ d"
```
  for a b c d :: nat
```
⟨*proof*⟩

lemmas pl_pl_rels' = add.commute [THEN [2] trans, THEN pl_pl_rels]

lemma iszero_minus:
```
  ‹iszero (- z) ⟷ iszero z›
```
⟨*proof*⟩

lemma diff_le_eq': "a - b ≤ c ⟷ a ≤ b + c"
```
  for a b c :: int
```

⟨*proof*⟩

lemma zless2: "0 < (2 :: int)"
⟨*proof*⟩

lemma zless2p: "0 < (2 ^ n :: int)"
⟨*proof*⟩

lemma zle2p: "0 ≤ (2 ^ n :: int)"
⟨*proof*⟩

lemma ex_eq_or: "(∃m. n = Suc m ∧ (m = k ∨ P m)) ⟷ n = Suc k ∨ (∃m. n = Suc m ∧ P m)"
⟨*proof*⟩

lemma power_minus_simp: "0 < n ⟹ a ^ n = a * a ^ (n - 1)"
⟨*proof*⟩

lemma n2s_ths:
‹2 + n = Suc (Suc n)›
‹n + 2 = Suc (Suc n)›
⟨*proof*⟩

lemma s2n_ths:
‹Suc (Suc n) = 2 + n›
‹Suc (Suc n) = n + 2›
⟨*proof*⟩

lemma gt_or_eq_0: "0 < y ∨ 0 = y"
for y :: nat
⟨*proof*⟩

lemma sum_imp_diff: "j = k + i ⟹ j - i = k"
for k :: nat
⟨*proof*⟩

lemma le_diff_eq': "a ≤ c - b ⟷ b + a ≤ c"
for a b c :: int
⟨*proof*⟩

lemma less_diff_eq': "a < c - b ⟷ b + a < c"
for a b c :: int
⟨*proof*⟩

lemma diff_less_eq': "a - b < c ⟷ a < b + c"
for a b c :: int
⟨*proof*⟩

lemma axxbyy: "a + m + m = b + n + n ⟹ a = 0 ∨ a = 1 ⟹ b = 0 ∨ b

```
= 1 ⟹ a = b ∧ m = n"
  for a b m n :: int
  ⟨proof⟩

lemma minus_eq: "m - k = m ⟷ k = 0 ∨ m = 0"
  for k m :: nat
  ⟨proof⟩

lemma pl_pl_mm: "a + b = c + d ⟹ a - c = d - b"
  for a b c d :: nat
  ⟨proof⟩

lemmas pl_pl_mm' = add.commute [THEN [2] trans, THEN pl_pl_mm]

lemma less_le_mult': "w * c < b * c ⟹ 0 ≤ c ⟹ (w + 1) * c ≤ b *
c"
  for b c w :: int
  ⟨proof⟩

lemma less_le_mult: "w * c < b * c ⟹ 0 ≤ c ⟹ w * c + c ≤ b * c"
  for b c w :: int
  ⟨proof⟩

lemmas less_le_mult_minus = iffD2 [OF le_diff_eq less_le_mult,
  simplified left_diff_distrib]

lemma gen_minus: "0 < n ⟹ f n = f (Suc (n - 1))"
  ⟨proof⟩

lemma mpl_lem: "j ≤ i ⟹ k < j ⟹ i - j + k < i"
  for i j k :: nat
  ⟨proof⟩

lemmas dme = div_mult_mod_eq
lemmas dtle = div_times_less_eq_dividend
lemmas th2 = order_trans [OF order_refl [THEN [2] mult_le_mono] div_times_less_eq_dividend

lemmas sdl = div_nat_eqI

lemma given_quot: "f > 0 ⟹ (f * l + (f - 1)) div f = l"
  for f l :: nat
  ⟨proof⟩

lemma given_quot_alt: "f > 0 ⟹ (l * f + f - Suc 0) div f = l"
  for f l :: nat
  ⟨proof⟩

lemma x_power_minus_1:
  fixes x :: "'a :: {ab_group_add, power, numeral, one}"
```

shows "x + (2::'a) ^ n - (1::'a) = x + (2 ^ n - 1)" ⟨*proof*⟩

lemma nat_diff_add:
fixes i :: nat
shows "⟦ i + j = k ⟧ ⟹ i = k - j"
⟨*proof*⟩

lemma pow_2_gt: "n ≥ 2 ⟹ (2::int) < 2 ^ n"
⟨*proof*⟩

lemma sum_to_zero:
"(a :: 'a :: ring) + b = 0 ⟹ a = (- b)"
⟨*proof*⟩

lemma arith_is_1:
"⟦ x ≤ Suc 0; x > 0 ⟧ ⟹ x = 1"
⟨*proof*⟩

lemma suc_le_pow_2:
"1 < (n::nat) ⟹ Suc n < 2 ^ n"
⟨*proof*⟩

lemma nat_le_Suc_less_imp:
"x < y ⟹ x ≤ y - Suc 0"
⟨*proof*⟩

lemma power_sub_int:
"⟦ m ≤ n; 0 < b ⟧ ⟹ b ^ n div b ^ m = (b ^ (n - m) :: int)"
⟨*proof*⟩

lemma nat_Suc_less_le_imp:
"(k::nat) < Suc n ⟹ k ≤ n"
⟨*proof*⟩

lemma nat_add_less_by_max:
"⟦ (x::nat) ≤ xmax ; y < k - xmax ⟧ ⟹ x + y < k"
⟨*proof*⟩

lemma nat_le_Suc_less:
"0 < y ⟹ (x ≤ y - Suc 0) = (x < y)"
⟨*proof*⟩

lemma nat_power_minus_less:
"a < 2 ^ (x - n) ⟹ (a :: nat) < 2 ^ x"
⟨*proof*⟩

lemma less_le_mult_nat':
"w * c < b * c ==> 0 ≤ c ==> Suc w * c ≤ b * (c::nat)"
⟨*proof*⟩

lemma less_le_mult_nat:
  ‹0 < c ∧ w < b ⟹ c + w * c ≤ b * c› **for** b c w :: nat
  ⟨*proof*⟩

lemma p_assoc_help:
  fixes p :: "'a::{ring,power,numeral,one}"
  shows "p + 2^sz - 1 = p + (2^sz - 1)"
  ⟨*proof*⟩

lemma pow_mono_leq_imp_lt:
  "x ≤ y ⟹ x < 2 ^ y"
  ⟨*proof*⟩

lemma small_powers_of_2:
  "x ≥ 3 ⟹ x < 2 ^ (x - 1)"
  ⟨*proof*⟩

lemma nat_less_power_trans2:
  fixes n :: nat
  shows "⟦n < 2 ^ (m - k); k ≤ m⟧ ⟹ n * 2 ^ k < 2 ^ m"
  ⟨*proof*⟩

lemma nat_move_sub_le: "(a::nat) + b ≤ c ⟹ a ≤ c - b"
  ⟨*proof*⟩

lemma plus_minus_one_rewrite:
  "v + (- 1 :: ('a :: {ring, one, uminus})) ≡ v - 1"
  ⟨*proof*⟩

lemma Suc_0_lt_2p_len_of: "Suc 0 < 2 ^ LENGTH('a :: len)"
  ⟨*proof*⟩

end

34 Ancient comprehensive Word Library

theory Word_Lib_Sumo
imports
  "HOL-Library.Word"
  Aligned
  Ancient_Numeral
  Bit_Comprehension
  Bits_Int
  Bitwise_Signed
  Bitwise
  Enumeration_Word
  Generic_set_bit
  Hex_Words

```
  Least_significant_bit
  More_Arithmetic
  More_Divides
  More_Sublist
  Even_More_List
  More_Misc
  Strict_part_mono
  Legacy_Aliases
  Most_significant_bit
  Next_and_Prev
  Norm_Words
  Reversed_Bit_Lists
  Rsplit
  Signed_Words
  Traditional_Infix_Syntax
  Typedef_Morphisms
  Type_Syntax
  Word_EqI
  Word_Lemmas
  Word_8
  Word_16
  Word_32
  Word_Syntax
  Signed_Division_Word
  More_Word_Operations
  Many_More
begin

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

lemma shiftl_transfer [transfer_rule]:
  includes lifting_syntax
  shows "(pcr_word ===> (=) ===> pcr_word) (<<) (<<)"
  ⟨proof⟩

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)
  test_bit ("testBit")

lemmas cast_simps = cast_simps ucast_down_bl


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

end
```

35 Words of Length 64

```
theory Word_64
  imports
    Word_Lemmas
    Word_Names
    Word_Syntax
    Rsplit
    More_Word_Operations
begin

lemma len64: "len_of (x :: 64 itself) = 64" ⟨proof⟩

type_synonym machine_word_len = 64
type_synonym machine_word = "machine_word_len word"

definition word_bits :: nat
where
  "word_bits = LENGTH(machine_word_len)"
```

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 word_bits_conv[code]:
"word_bits = 64"
⟨*proof*⟩

lemma word_size_word_size_bits:
"(word_size::nat) = 2 ^ word_size_bits"
⟨*proof*⟩

lemma word_bits_word_size_conv:
"word_bits = word_size * 8"
⟨*proof*⟩

lemma ucast_8_64_inj:
"inj (ucast :: 8 word ⇒ 64 word)"
⟨*proof*⟩

lemma upto_2_helper:
"{0..<2 :: 64 word} = {0, 1}"
⟨*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]

lemma word_bits_len_of:
"len_of TYPE (64) = word_bits"
⟨*proof*⟩

lemmas unat_power_lower64' = unat_power_lower[**where** 'a=64]
lemmas unat_power_lower64 [simp] = unat_power_lower64'[unfolded word_bits_len_of]

lemmas word64_less_sub_le' = word_less_sub_le[**where** 'a = 64]
lemmas word64_less_sub_le[simp] = word64_less_sub_le' [folded word_bits_def]

lemma word_bits_size:
"size (w::word64) = word_bits"
⟨*proof*⟩

lemmas word64_power_less_1' = word_power_less_1[**where** 'a = 64]
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*⟩

lemma unat_mask_2_less_4:
"unat (p && mask 2 :: word64) < 4"

⟨*proof*⟩

lemmas unat_of_nat64' = unat_of_nat_eq[**where** 'a=64]
lemmas unat_of_nat64 = unat_of_nat64'[unfolded word_bits_len_of]

lemmas word_power_nonzero_64 = word_power_nonzero [**where** 'a=64, folded word_bits_def]

lemmas unat_mult_simple = iffD1 [OF unat_mult_lem [**where** 'a = 64, unfolded word_bits_len_of]]

lemmas div_power_helper_64 = div_power_helper [**where** 'a=64, folded word_bits_def]

lemma n_less_word_bits:
  "(n < word_bits) = (n < 64)"
  ⟨*proof*⟩

lemmas of_nat_less_pow_64 = of_nat_power [**where** 'a=64, folded word_bits_def]

lemma lt_word_bits_lt_pow:
  "sz < word_bits ⟹ sz < 2 ^ word_bits"
  ⟨*proof*⟩

lemma unat_less_word_bits:
  fixes y :: word64
  shows "x < unat y ⟹ x < 2 ^ word_bits"
  ⟨*proof*⟩

lemmas unat_mask_word64' = unat_mask[**where** 'a=64]
lemmas unat_mask_word64 = unat_mask_word64'[folded word_bits_def]

lemma unat_less_2p_word_bits:
  "unat (x :: 64 word) < 2 ^ word_bits"
  ⟨*proof*⟩

lemma Suc_unat_mask_div:
  "Suc (unat (mask sz div word_size::word64)) = 2 ^ (min sz word_bits - 3)"
  ⟨*proof*⟩

lemmas word64_minus_one_le' = word_minus_one_le[**where** 'a=64]
lemmas word64_minus_one_le = word64_minus_one_le'[simplified]

lemma ucast_not_helper:
  fixes a::"8 word"
  assumes a: "a ≠ 0xFF"
  shows "ucast a ≠ (0xFF::word64)"
⟨*proof*⟩

lemma less_4_cases:
"(x::word64) < 4 ⟹ x=0 ∨ x=1 ∨ x=2 ∨ x=3"
⟨*proof*⟩

lemma if_then_1_else_0:
"((if P then 1 else 0) = (0 :: word64)) = (¬ P)"
⟨*proof*⟩

lemma if_then_0_else_1:
"((if P then 0 else 1) = (0 :: word64)) = (P)"
⟨*proof*⟩

lemmas if_then_simps = if_then_0_else_1 if_then_1_else_0

lemma ucast_le_ucast_8_64:
"(ucast x ≤ (ucast y :: word64)) = (x ≤ (y :: 8 word))"
⟨*proof*⟩

lemma in_16_range:
"0 ∈ S ⟹ r ∈ (λx. r + x * (16 :: word64)) ` S"
"n - 1 ∈ S ⟹ (r + (16 * n - 16)) ∈ (λx :: word64. r + x * 16) ` S"
⟨*proof*⟩

lemma eq_2_64_0:
"(2 ^ 64 :: word64) = 0"
⟨*proof*⟩

lemma x_less_2_0_1:
fixes x :: word64 **shows**
"x < 2 ⟹ x = 0 ∨ x = 1"
⟨*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 word_rsplit_0:
"word_rsplit (0 :: word64) = [0, 0, 0, 0, 0, 0, 0, 0 :: 8 word]"
⟨*proof*⟩

lemma unat_ucast_10_64 :
fixes x :: "10 word"
shows "unat (ucast x :: word64) = unat x"
⟨*proof*⟩

lemma bool_mask [simp]:
fixes x :: word64

```
  shows "(0 < x && 1) = (x && 1 = 1)"
  ⟨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⟩

lemma word_ge_min:"sint (x::64 word) ≥ -9223372036854775808"
  ⟨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 le_step_down_word_3:
fixes x :: "64 word"
shows "⟦x ≤ y; x ≠ y; y < 2 ^ 64 - 1⟧ ⟹ x ≤ y - 1"
⟨*proof*⟩

lemma shiftr_1:
"(x::word64) >> 1 = 0 ⟹ x < 2"
⟨*proof*⟩

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

36 A short overview over bit operations and word types

36.1 Basic theories and key ideas

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 alge-

braic 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 most fundamental ideas are developed in theory `HOL.Parity` (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. 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 = P x n` are available in fact collection `bit_simps`.

On top of this, the following generic operations are provided after import of theory `HOL-Library.Bit_Operations`:

- 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` $2^{\texttt{n}}$
- Negation: `bit (NOT a) n` $\longleftrightarrow$ $2^{\texttt{n}} \neq 0 \wedge \neg$ `bit a n`
- And: `bit (a AND b) n` $\longleftrightarrow$ `bit a n` $\wedge$ `bit b n`
- Or: `bit (a OR b) n` $\longleftrightarrow$ `bit a n` $\vee$ `bit b n`
- Xor: `bit (a XOR b) n` $\longleftrightarrow$ `bit a n` $\neq$ `bit b n`
- Set a single bit: `set_bit n a = a OR push_bit n 1`
- Unset a single bit: `unset_bit n a = a AND NOT (push_bit n 1)`
- Flip 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
  ```

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_eqI`; fact collection `bit_simps`).
- Using the `transfer` method.

36.2 More library theories

Note: currently, the 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.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.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 an alias: `lsb = odd`

`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.Traditional_Infix_Syntax` Clones of existing operations decorated with traditional syntax:

- `(!!) = bit`
- `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.

36.3 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.

36.4 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_32` or `Word_Lib.Word_64` separately.

`Word_Lib.Generic_set_bit` Kind of an alias: `set_bit_class.set_bit a n b = (if b then 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; difficult to handle for `int`.

`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.

theory `Examples`
  imports `Bitwise Next_and_Prev Generic_set_bit Word_Syntax Signed_Division_Word`
begin

modulus

lemma `"(27 :: 4 word) = -5"` ⟨*proof*⟩

lemma `"(27 :: 4 word) = 11"` ⟨*proof*⟩

lemma `"27 ≠ (11 :: 6 word)"` ⟨*proof*⟩

signed

lemma `"(127 :: 6 word) = -1"` ⟨*proof*⟩

number ring simps

lemma
```
  "27 + 11 = (38::'a::len word)"
  "27 + 11 = (6::5 word)"
  "7 * 3 = (21::'a::len word)"
  "11 - 27 = (-16::'a::len word)"
  "- (- 11) = (11::'a::len word)"
  "-40 + 1 = (-39::'a::len word)"
```
  ⟨*proof*⟩

lemma `"word_pred 2 = 1"` ⟨*proof*⟩

lemma `"word_succ (- 3) = -2"` ⟨*proof*⟩

lemma "23 < (27::8 word)" ⟨*proof*⟩
lemma "23 ≤ (27::8 word)" ⟨*proof*⟩
lemma "¬ 23 < (27::2 word)" ⟨*proof*⟩
lemma "0 < (4::3 word)" ⟨*proof*⟩
lemma "1 < (4::3 word)" ⟨*proof*⟩
lemma "0 < (1::3 word)" ⟨*proof*⟩

ring operations

lemma "a + 2 * b + c - b = (b + c) + (a :: 32 word)" ⟨*proof*⟩

casting

lemma "uint (234567 :: 10 word) = 71" ⟨*proof*⟩
lemma "uint (-234567 :: 10 word) = 953" ⟨*proof*⟩
lemma "sint (234567 :: 10 word) = 71" ⟨*proof*⟩
lemma "sint (-234567 :: 10 word) = -71" ⟨*proof*⟩
lemma "uint (1 :: 10 word) = 1" ⟨*proof*⟩

lemma "unat (-234567 :: 10 word) = 953" ⟨*proof*⟩
lemma "unat (1 :: 10 word) = 1" ⟨*proof*⟩

lemma "ucast (0b1010 :: 4 word) = (0b10 :: 2 word)" ⟨*proof*⟩
lemma "ucast (0b1010 :: 4 word) = (0b1010 :: 10 word)" ⟨*proof*⟩
lemma "scast (0b1010 :: 4 word) = (0b111010 :: 6 word)" ⟨*proof*⟩
lemma "ucast (1 :: 4 word) = (1 :: 2 word)" ⟨*proof*⟩

reducing goals to nat or int and arith:

lemma "i < x ⟹ i < i + 1" **for** i x :: "'a::len word"
⟨*proof*⟩
lemma "i < x ⟹ i < i + 1" **for** i x :: "'a::len word"
⟨*proof*⟩

bool lists

lemma "of_bl [True, False, True, True] = (0b1011::'a::len word)" ⟨*proof*⟩

lemma "to_bl (0b110::4 word) = [False, True, True, False]" ⟨*proof*⟩

lemma "of_bl (replicate 32 True) = (0xFFFFFFFF::32 word)"
⟨*proof*⟩

bit operations

lemma "0b110 AND 0b101 = (0b100 :: 32 word)" ⟨*proof*⟩
lemma "0b110 OR 0b011 = (0b111 :: 8 word)" ⟨*proof*⟩
lemma "0xF0 XOR 0xFF = (0x0F :: 8 word)" ⟨*proof*⟩
lemma "NOT (0xF0 :: 16 word) = 0xFF0F" ⟨*proof*⟩
lemma "0 AND 5 = (0 :: 8 word)" ⟨*proof*⟩
lemma "1 AND 1 = (1 :: 8 word)" ⟨*proof*⟩
lemma "1 AND 0 = (0 :: 8 word)" ⟨*proof*⟩

lemma "1 AND 5 = (1 :: 8 word)" ⟨*proof*⟩
lemma "1 OR 6 = (7 :: 8 word)" ⟨*proof*⟩
lemma "1 OR 1 = (1 :: 8 word)" ⟨*proof*⟩
lemma "1 XOR 7 = (6 :: 8 word)" ⟨*proof*⟩
lemma "1 XOR 1 = (0 :: 8 word)" ⟨*proof*⟩
lemma "NOT 1 = (254 :: 8 word)" ⟨*proof*⟩
lemma "NOT 0 = (255 :: 8 word)" ⟨*proof*⟩

lemma "(-1 :: 32 word) = 0xFFFFFFFF" ⟨*proof*⟩

lemma "(0b0010 :: 4 word) !! 1" ⟨*proof*⟩
lemma "¬ (0b0010 :: 4 word) !! 0" ⟨*proof*⟩
lemma "¬ (0b1000 :: 3 word) !! 4" ⟨*proof*⟩
lemma "¬ (1 :: 3 word) !! 2" ⟨*proof*⟩

lemma "(0b11000 :: 10 word) !! n = (n = 4 ∨ n = 3)"
⟨*proof*⟩

lemma "set_bit 55 7 True = (183::'a::len word)" ⟨*proof*⟩
lemma "set_bit 0b0010 7 True = (0b10000010::'a::len word)" ⟨*proof*⟩
lemma "set_bit 0b0010 1 False = (0::'a::len word)" ⟨*proof*⟩
lemma "set_bit 1 3 True = (0b1001::'a::len word)" ⟨*proof*⟩
lemma "set_bit 1 0 False = (0::'a::len word)" ⟨*proof*⟩
lemma "set_bit 0 3 True = (0b1000::'a::len word)" ⟨*proof*⟩
lemma "set_bit 0 3 False = (0::'a::len word)" ⟨*proof*⟩

lemma "odd (0b0101::'a::len word)" ⟨*proof*⟩
lemma "even (0b1000::'a::len word)" ⟨*proof*⟩
lemma "odd (1::'a::len word)" ⟨*proof*⟩
lemma "even (0::'a::len word)" ⟨*proof*⟩

lemma "¬ msb (0b0101::4 word)" ⟨*proof*⟩
lemma "msb (0b1000::4 word)" ⟨*proof*⟩
lemma "¬ msb (1::4 word)" ⟨*proof*⟩
lemma "¬ msb (0::4 word)" ⟨*proof*⟩

lemma "word_cat (27::4 word) (27::8 word) = (2843::'a::len word)" ⟨*proof*⟩
lemma "word_cat (0b0011::4 word) (0b1111::6word) = (0b0011001111 :: 10
word)"
⟨*proof*⟩

lemma "0b1011 << 2 = (0b101100::'a::len word)" ⟨*proof*⟩
lemma "0b1011 >> 2 = (0b10::8 word)" ⟨*proof*⟩
lemma "0b1011 >>> 2 = (0b10::8 word)" ⟨*proof*⟩
lemma "1 << 2 = (0b100::'a::len word)" ⟨*proof*⟩

lemma "slice 3 (0b101111::6 word) = (0b101::3 word)" ⟨*proof*⟩
lemma "slice 3 (1::6 word) = (0::3 word)" ⟨*proof*⟩

lemma "word_rotr 2 0b0110 = (0b1001::4 word)" ⟨*proof*⟩
lemma "word_rotl 1 0b1110 = (0b1101::4 word)" ⟨*proof*⟩
lemma "word_roti 2 0b1110 = (0b1011::4 word)" ⟨*proof*⟩
lemma "word_roti (- 2) 0b0110 = (0b1001::4 word)" ⟨*proof*⟩
lemma "word_rotr 2 0 = (0::4 word)" ⟨*proof*⟩
lemma "word_rotr 2 1 = (0b0100::4 word)" ⟨*proof*⟩
lemma "word_rotl 2 1 = (0b0100::4 word)" ⟨*proof*⟩
lemma "word_roti (- 2) 1 = (0b0100::4 word)" ⟨*proof*⟩

lemma "(x AND 0xff00) OR (x AND 0x00ff) = (x::16 word)"
⟨*proof*⟩

lemma "word_next (2:: 8 word) = 3" ⟨*proof*⟩
lemma "word_next (255:: 8 word) = 255" ⟨*proof*⟩
lemma "word_prev (2:: 8 word) = 1" ⟨*proof*⟩
lemma "word_prev (0:: 8 word) = 0" ⟨*proof*⟩

proofs using bitwise expansion

lemma "(x AND 0xff00) OR (x AND 0x00ff) = (x::16 word)"
⟨*proof*⟩

lemma "(x AND NOT 3) >> 4 << 2 = ((x >> 2) AND NOT 3)"
for x :: "10 word"
⟨*proof*⟩

lemma "((x AND -8) >> 3) AND 7 = (x AND 56) >> 3"
for x :: "12 word"
⟨*proof*⟩

some problems require further reasoning after bit expansion

lemma "x ≤ 42 ⟹ x ≤ 89"
for x :: "8 word"
⟨*proof*⟩

lemma "(x AND 1023) = 0 ⟹ x ≤ -1024"
for x :: ‹32 word›
⟨*proof*⟩

operations like shifts by non-numerals will expose some internal list representations but may still be easy to solve

lemma shiftr_overflow: "32 ≤ a ⟹ b >> a = 0"
for b :: ‹32 word›
⟨*proof*⟩

lemma "((x :: 32 word) >> 3) AND 7 = (x AND 56) >> 3"
⟨*proof*⟩

lemma
```
"( 4 :: 32 word) sdiv  4 =  1"
"(-4 :: 32 word) sdiv  4 = -1"
"(-3 :: 32 word) sdiv  4 =  0"
"( 3 :: 32 word) sdiv -4 =  0"
"(-3 :: 32 word) sdiv -4 =  0"
"(-5 :: 32 word) sdiv -4 =  1"
"( 5 :: 32 word) sdiv -4 = -1"
```
⟨*proof*⟩

lemma
```
"( 4 :: 32 word) smod  4 =   0"
"( 3 :: 32 word) smod  4 =   3"
"(-3 :: 32 word) smod  4 =  -3"
"( 3 :: 32 word) smod -4 =   3"
"(-3 :: 32 word) smod -4 =  -3"
"(-5 :: 32 word) smod -4 =  -1"
"( 5 :: 32 word) smod -4 =   1"
```
⟨*proof*⟩

lemma `"1 < (1024::32 word) ∧ 1 ≤ (1024::32 word)"`
⟨*proof*⟩

end